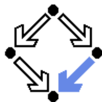
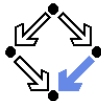
# Formal Methods for Distributed Systems
# An Introduction

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





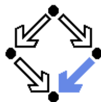

1. A Client/Server System

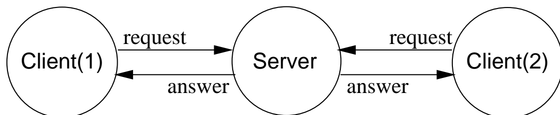
2. Modeling Concurrent Systems

3. Specifying System Properties

4. Verifying System Properties

# A Client/Server System

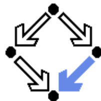





- System of one server and two clients.
  - Three concurrently executing system components.
- Server manages a resource.
  - An object that only one system component may use at any time.
- Clients request resource and, having received an answer, use it.
  - Server ensures that not both clients use resource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

Set of system requirements.

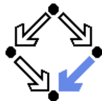
# System Implementation (Pseudo-Code)



```
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
    elsif 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
```

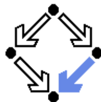
# Desired System Properties

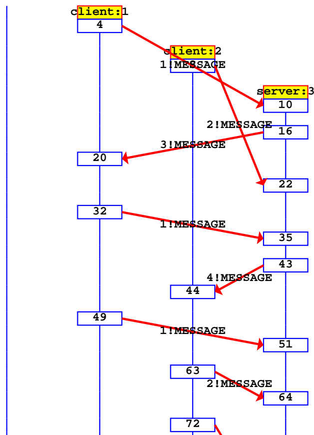


- Property: mutual exclusion.
  - At no time, both clients are in critical region.
    - Critical region: program region after receiving resource from server and before returning resource to server.
  - The system shall only reach states, in which mutual exclusion holds.
- Property: no starvation.
  - 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?

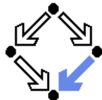
# Simulating the System Execution






Just one execution, infinitely many are possible!

# Specifying a System Property



Linear Time Temporal Logic Formulae

Formula: [] !(c1 && c2) Load...

Operators: [] <> U -> and or not

Property holds for: All Executions (desired behavior) No Executions (error behavior)

Notes [file clientServer-mutex.ltl]:

```
Use Load to open a file or a template.
```

Symbol Definitions:

```
#define c1 client[1]@C
#define c2 client[2]@C
```

Never Claim: Generate

```
        /*
         * Formula As Typed: [] !(c1 && c2)
         * The Never Claim Below Corresponds
         * To The Negated Formula !([] !(c1 && c2))
         * (formalizing violations of the original)
         */

never {    /* !([] !(c1 && c2)) */
```

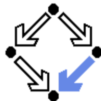
Verification Result: valid Run Verification

```
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 5 (line 68)
(Spin Version 4.3.0 -- 22 June 2007)
        + Partial Order Reduction

Full statespace search for:
```

Help Clear Close Save As..

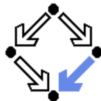
Formal specification of the “mutual exclusion” property.

# Verifying the System Property



```
(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)

...
```

*No* possible execution violates the “mutual exclusion” property.



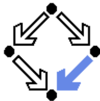
1. A Client/Server System

2. Modeling Concurrent Systems

3. Specifying System Properties

4. Verifying System Properties

# System States

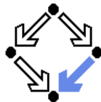


At each moment in time, a system is in a particular state.

- A state $s : Var \rightarrow Val$
  - A state $s$ is a mapping of every system variable $x$ to its value $s(x)$.
    - Typical notation: $s = [x = 0, y = 1, \ldots] = [0, 1, \ldots]$.
  - $Var$ ... 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 $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$ in the state space.

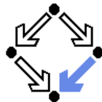
# Deterministic Systems



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.
  - $s_0 \rightarrow s_1 = F(s_0) \rightarrow s_2 = F(s_1) \rightarrow \ldots$
- 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 $F : State \stackrel{partial}{\rightarrow} State$.
- A run of a deterministic system $\langle I, F \rangle$ is a (finite or infinite) sequence $s_0 \rightarrow s_1 \rightarrow \ldots$ 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 $s_n$.

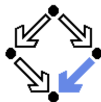
# Nondeterministic Systems



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 $s_0 \to s_1 \to \ldots$ and $s_0 \to s'_1 \to \ldots$ 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 $s_0 \to s_1 \to s_2 \ldots$ 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 $R(s_n, t)$.

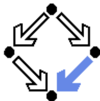
# Derived Notions



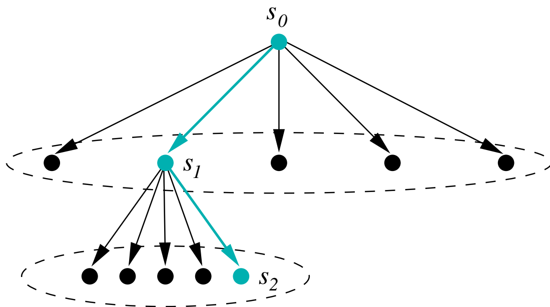
- Successor and predecessor:
  - State $t$ is a (direct) successor of state $s$, if $R(s,t)$.
  - State $s$ is then a predecessor of $t$.
    - A finite run $s_0 \rightarrow \ldots \rightarrow s_n$ ends in a state which has no successor.
- Reachability:
  - A state $t$ is reachable, if there exists some run $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$ such that $t = s_i$ (for some $i$).
  - A state $t$ is unreachable, if it is not reachable.

Not all states are reachable (typically most are unreachable).

# Reachability Graph

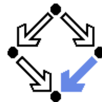


The transitions of a system can be visualized by a graph.




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

# Examples





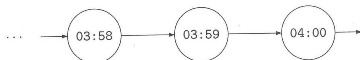



**Fig. 1.1.** A model of a watch

of $\mathcal{A}_{c3}$ correspond to the possible counter values. Its transitions reflect the possible actions on the counter. In this example we restrict our operations to increments (`inc`) and decrements (`dec`).

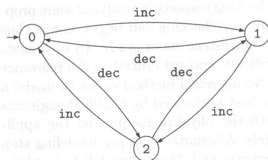
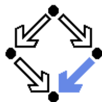



**Fig. 1.2.** $\mathcal{A}_{c3}$ : a modulo 3 counter

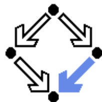
B.Berard et al: "Systems and Software Verification", 2001.

# Examples



- A deterministic system $W = (I_W, F_W)$ ("watch").
  - $State := \mathbb{N}_{24} \times \mathbb{N}_{60}$.
    - $\mathbb{N}_n := \{i \in \mathbb{N} : i < n\}$.
  - $I_W(h, m) :\Leftrightarrow h = 0 \wedge m = 0$.
    - $I_W := \{\langle h, m\rangle : h = 0 \wedge m = 0\} = \{\langle 0, 0\rangle\}$.
  - $F_W(h, m) :=$
    **if** $m < 59$ **then** $\langle h, m + 1\rangle$
    **else if** $h < 23$ **then** $\langle h + 1, 0\rangle$
    **else** $\langle 0, 0\rangle$.
- A nondeterministic system $C = (I_C, R_C)$ (modulo 3 "counter").
  - $State := \mathbb{N}_3$.
  - $I_C(i) :\Leftrightarrow i = 0$.
  - $R_C(i, i') :\Leftrightarrow inc(i, i') \vee dec(i, i')$.
    - $inc(i, i') :\Leftrightarrow$ **if** $i < 2$ **then** $i' = i + 1$ **else** $i' = 0$.
    - $dec(i, i') :\Leftrightarrow$ **if** $i > 0$ **then** $i' = i - 1$ **else** $i' = 2$.

# Composing Systems



Compose $n$ components $S_i$ to a concurrent system $S$.

- State space $State := State_0 \times \ldots \times State_{n-1}$.
  - $State_i$ is the state space of component $i$.
  - State space is Cartesian product of component state spaces.
  - Size of state space is product of the sizes of the component spaces.
- Example: three counters with state spaces $\mathbb{N}_2$ and $\mathbb{N}_3$ and $\mathbb{N}_4$.

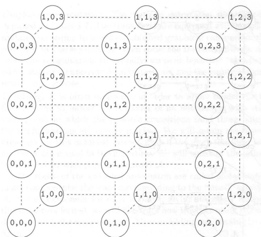
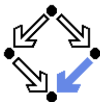



**Fig. 1.9.** The states of the product of the three counters

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

# Initial States of Composed System

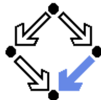


What are the initial states $I$ of the composed system?

- Set $I := I_0 \times \ldots \times I_{n-1}$.
  - $I_i$ is the set of initial states of component $i$.
  - Set of initial states is Cartesian product of the sets of initial states of the individual components.
- Predicate $I(s_0, \ldots, s_{n-1}) :\Leftrightarrow I_0(s_0) \wedge \ldots \wedge I_{n-1}(s_{n-1})$.
  - $I_i$ is the initial state condition of component $i$.
  - Initial state condition is conjunction of the initial state conditions of the components on the corresponding projection of the state.

Size of initial state set is the product of the sizes of the initial state sets of the individual components.

# Transitions of Composed System



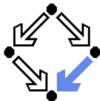
Which transitions can the composed system perform?

- Synchronized composition.
  - At each step, every component must perform a transition.
    - $R_i$ is the transition relation of component $i$.

      $R(\langle s_0, \ldots, s_{n-1}\rangle, \langle s'_0, \ldots, s'_{n-1}\rangle) :\Leftrightarrow$
      $R_0(s_0, s'_0) \wedge \ldots \wedge R_{n-1}(s_{n-1}, s'_{n-1})$.
- Asynchronous composition.
  - At each moment, every component may perform a transition.
    - At least one component performs a transition.
    - Multiple simultaneous transitions are possible
    - With $n$ components, $2^n - 1$ possibilities of (combined) transitions.

      $R(\langle s_0, \ldots, s_{n-1}\rangle, \langle s'_0, \ldots, s'_{n-1}\rangle) :\Leftrightarrow$
      $(R_0(s_0, s'_0) \wedge \ldots \wedge s_{n-1} = s'_{n-1}) \vee$
      $\ldots$
      $(s_0 = s'_0 \wedge \ldots \wedge R_{n-1}(s_{n-1}, s'_{n-1})) \vee$
      $\ldots$
      $(R_0(s_0, s'_0) \wedge \ldots \wedge R_{n-1}(s_{n-1}, s'_{n-1}))$.

# Example



System of three counters with state space $\mathbb{N}_2$ each.

- Synchronous composition:

$$[0,0,0] \leftrightarrows [1,1,1]$$

- Asynchronous composition:

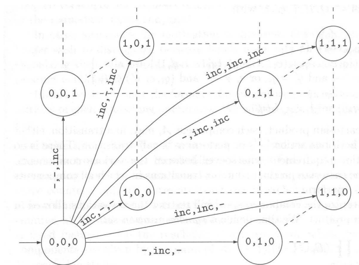
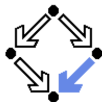



**Fig. 1.10.** A few transitions of the product of the three counters

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

# Interleaving Execution



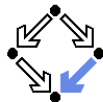
Simplified view of asynchronous execution.

- At each moment, only one component performs a transition.
  - Do not allow simultaneous transition $t_i|t_j$ of two components $i$ and $j$.
  - Transition sequences $t_i; t_j$ and $t_j; t_i$ are possible.
    - All possible interleavings of component transitions are considered.
    - Nondeterminism is used to simulate concurrency.
    - Essentially no change of system properties.
  - With $n$ components, only $n$ possibilities of a transition.

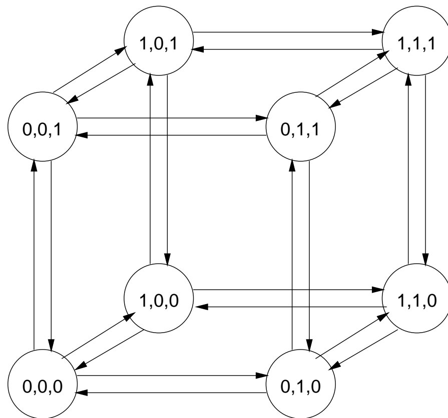
$$
\begin{aligned}
&R(\langle s_0, s_1, \ldots, s_{n-1}\rangle, \langle s'_0, s'_1, \ldots, s'_{n-1}\rangle) :\Leftrightarrow \\
&\quad (R_0(s_0, s'_0) \wedge s_1 = s'_1 \wedge \ldots \wedge s_{n-1} = s'_{n-1}) \vee \\
&\quad (s_0 = s'_0 \wedge R_1(s_1, s'_1) \wedge \ldots \wedge s_{n-1} = s'_{n-1}) \vee \\
&\quad \ldots \\
&\quad (s_0 = s'_0 \wedge s_1 = s'_1 \wedge \ldots \wedge R_{n-1}(s_{n-1}, s'_{n-1})).
\end{aligned}
$$

Interleaving model (respectively a variant of it) suffices in practice.

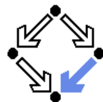
# Example



System of three counters with state space $\mathbb{N}_2$ each.



# Digital Circuits



Synchronous composition of hardware components.

- A modulo 8 counter $C = \langle I_C, R_C \rangle$.

$$State := \mathbb{N}_2 \times \mathbb{N}_2 \times \mathbb{N}_2.$$

$$I_C(v_0, v_1, v_2) :\Leftrightarrow v_0 = v_1 = v_2 = 0.$$

$$\begin{aligned} &R_C(\langle v_0, v_1, v_2\rangle, \langle v_0', v_1', v_2'\rangle) :\Leftrightarrow \\ &\quad R_0(v_0, v_0') \wedge \\ &\quad R_1(v_0, v_1, v_1') \wedge \\ &\quad R_2(v_0, v_1, v_2, v_2'). \end{aligned}$$

$$\begin{aligned} &R_0(v_0, v_0') :\Leftrightarrow v_0' = \neg v_0. \\ &R_1(v_0, v_1, v_1') :\Leftrightarrow v_1' = v_0 \oplus v_1. \\ &R_2(v_0, v_1, v_2, v_2') :\Leftrightarrow v_2' = (v_0 \wedge v_1) \oplus v_2. \end{aligned}$$

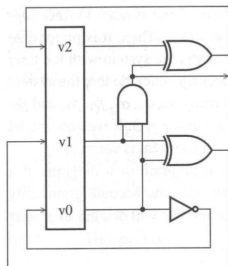
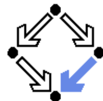



Figure 2.1
Synchronous modulo 8 counter.

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

# Concurrent Software



Asynchronous composition of software components with shared variables.

$$P :: l_0 : \textbf{while true do} \; NC_0 : \textbf{wait } turn = 0 \; CR_0 : turn := 1 \; \textbf{end} \quad || \quad Q :: l_1 : \textbf{while true do} \; NC_1 : \textbf{wait } turn = 1 \; CR_1 : turn := 0 \; \textbf{end}$$

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

$State := PC \times PC \times \mathbb{N}_2$. // shared variable

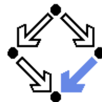
$I_M(p, q, turn) :\Leftrightarrow p = l_0 \land q = l_1$.

$R_M(\langle p, q, turn \rangle, \langle p', q', turn' \rangle) :\Leftrightarrow$
$(P(\langle p, turn \rangle, \langle p', turn' \rangle) \land q' = q) \lor (Q(\langle q, turn \rangle, \langle q', turn' \rangle) \land p' = p)$.

$P(\langle p, turn \rangle, \langle p', turn' \rangle) :\Leftrightarrow$
$(p = l_0 \land p' = NC_0 \land turn' = turn) \lor$
$(p = NC_0 \land p' = CR_0 \land turn = 0 \land turn' = turn) \lor$
$(p = CR_0 \land p' = l_0 \land turn' = 1)$.

$Q(\langle q, turn \rangle, \langle q', turn' \rangle) :\Leftrightarrow$
$(q = l_1 \land q' = NC_1 \land turn' = turn) \lor$
$(q = NC_1 \land q' = CR_1 \land turn = 1 \land turn' = turn) \lor$
$(q = CR_1 \land q' = l_1 \land turn' = 0)$.

# Concurrent Software



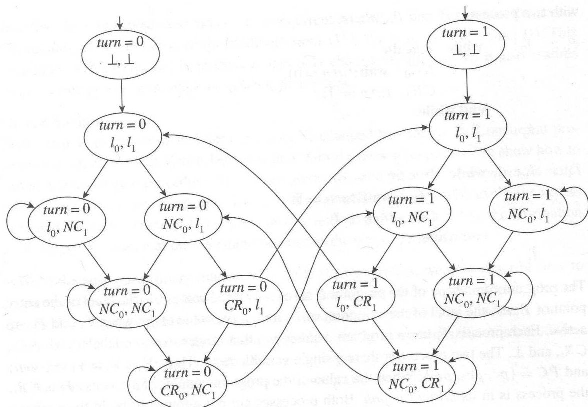
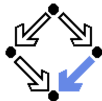



**Figure 2.2**
Reachable states of Kripke structure for mutual exclusion example.

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

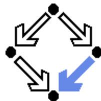
Model guarantees mutual exclusion.

# Summary



- A system is described by
  - its (finite or infinite) state space,
  - the initial state condition (set of input states),
  - the transition relation on states.
- State space of composed system is product of component spaces.
  - Variable shared among components occurs only once in product.
- System composition can be
  - synchronous: conjunction of individual transition relations.
    - Suitable for digital hardware.
  - asynchronous: disjunction of relations.
    - Interleaving model: each relation conjoins the transition relation of one component with the identity relations of all other components.
    - Suitable for concurrent software.

Next: how to specify system properties.



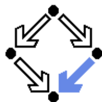

1. A Client/Server System

2. Modeling Concurrent Systems

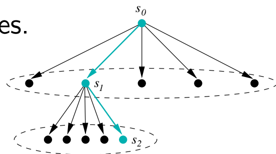
3. **Specifying System Properties**

4. Verifying System Properties

# Motivation



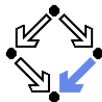
We need a language for specifying system properties.




- A system $S$ is a pair $\langle I, R \rangle$.
  - Initial states $I$, transition relation $R$.
  - More intuitive: reachability graph.
    - Starting from an initial state $s_0$, the system runs evolve.
- Consider the reachability graph as an infinite computation tree.
  - Different tree nodes may denote occurrences of the same state.
    - Each occurrence of a state has a unique predecessor in the tree.
  - Every path in this tree is infinite.
    - Every finite run $s_0 \rightarrow \ldots \rightarrow s_n$ is extended to an infinite run $s_0 \rightarrow \ldots \rightarrow s_n \rightarrow s_n \rightarrow s_n \rightarrow \ldots$
- Or simply consider the graph as a set of system runs.
  - Same state may occur multiple times (in one or in different runs).

We need to talk about such trees respectively sets of system runs.

# Computation Trees versus System Runs



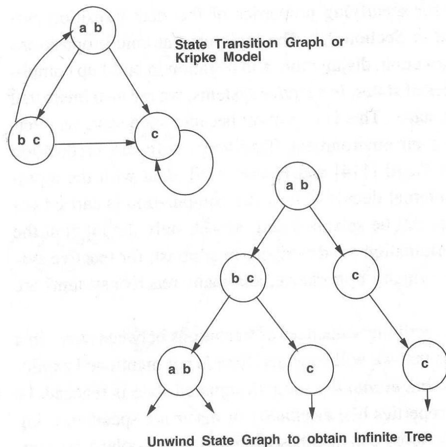


**Figure 3.1**
Computation trees.

Set of system runs:

$[a, b] \rightarrow c \rightarrow c \rightarrow \dots$

$[a, b] \rightarrow [b, c] \rightarrow c \rightarrow \dots$

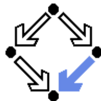
$[a, b] \rightarrow [b, c] \rightarrow [a, b] \rightarrow \dots$

$[a, b] \rightarrow [b, c] \rightarrow [a, b] \rightarrow \dots$

$\dots$

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

# State Formula

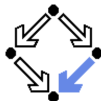


Temporal logic is based on classical logic.

- A state formula $F$ is evaluated on a state $s$.
  - Any predicate logic formula is a state formula:
    $p(x), \neg F, F_0 \wedge F_1, F_0 \vee F_1, F_0 \Rightarrow F_1, F_0 \Leftrightarrow F_1, \forall x : F, \exists x : F.$
  - In propositional temporal logic only propositional logic formulas are state formulas (no quantification):
    $p, \neg F, F_0 \wedge F_1, F_0 \vee F_1, F_0 \Rightarrow F_1, F_0 \Leftrightarrow F_1.$
- Semantics: $s \models F$ ("$F$ holds in state $s$").
  - Example: semantics of conjunction.
    - $(s \models F_0 \wedge F_1) \ :\Leftrightarrow\ (s \models F_0) \wedge (s \models F_1).$
    - "$F_0 \wedge F_1$ holds in $s$ if and only if $F_0$ holds in $s$ and $F_1$ holds in $s$".

Classical logic reasons on individual states.

# Temporal Logic



Extension of classical logic to reason about multiple states.

- Temporal logic is an instance of modal logic.
  - Logic of “multiple worlds (situations)” that are in some way related.
  - Relationship may e.g. be a temporal one.
  - Amir Pnueli, 1977: temporal logic is suited to system specifications.
  - Many variants, two fundamental classes.
- Branching Time Logic
  - Semantics defined over computation trees.

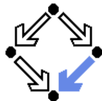
    At each moment, there are multiple possible futures.
  - Prominent variant: CTL.

    Computation tree logic; a propositional branching time logic.
- Linear Time Logic
  - Semantics defined over sets of system runs.

    At each moment, there is only one possible future.
  - Prominent variant: PLTL.

    A propositional linear time logic.

# Linear Time Logic (LTL)

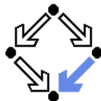


We use temporal logic to specify a system property $P$.

- Core question: $S \models P$ ("$P$ holds in system $S$").
  - System $S = \langle I, R \rangle$, temporal logic formula $P$.
- Linear time logic:
  - $S \models P \ :\Leftrightarrow\ r \models P$, 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



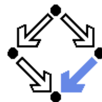
No path quantifiers; all formulas are path formulas.

- 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 $\Box F$,
    $\mathbf{F}\ F$ ("eventually $F$"), often written $\Diamond F$,
    $F\ \mathbf{U}\ G$ ("$F$ until $G$").
- Semantics: $p \models P$ ("$P$ holds in path $p$").
  - $p^i := \langle p_i, p_{i+1}, \ldots \rangle$.

    $p \models f :\Leftrightarrow p_0 \models f$.
    $p \models \mathbf{X}\ F\ :\Leftrightarrow p^1 \models F$.
    $p \models \mathbf{G}\ F\ :\Leftrightarrow \forall i \in \mathbb{N} : p^i \models F$.
    $p \models \mathbf{F}\ F\ :\Leftrightarrow \exists i \in \mathbb{N} : p^i \models F$.
    $p \models F\ \mathbf{U}\ G\ :\Leftrightarrow \exists i \in \mathbb{N} : p^i \models G \wedge \forall j \in \mathbb{N}_i : p^j \models F$.

# Formulas



$F\varphi$: ○ → ○ → ○ → $\varphi$ - - →

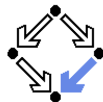
$G\varphi$: $\varphi$ → $\varphi$ → $\varphi$ → $\varphi$ - - →

$X\varphi$: ○ → $\varphi$ → ○ → ○ - - →

$\varphi U \psi$: $\varphi$ → $\varphi$ → $\varphi$ → $\psi$ - - →

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

# Frequently Used LTL Patterns

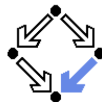


In practice, most temporal formulas are instances of particular patterns.

| 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 |

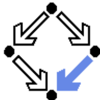
Typically, there are at most two levels of nesting of temporal operators.

# Examples



- Mutual exclusion: $\Box\neg(pc_1 = C \land pc_2 = C)$.
  - Alternatively: $\neg\Diamond(pc_1 = C \land pc_2 = C)$.
  - 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.

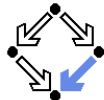
# Temporal Rules



Temporal operators obey a number of fairly intuitive rules.

- Extraction laws:
  - $\Box F \Leftrightarrow F \wedge \bigcirc\Box F$.
  - $\Diamond F \Leftrightarrow F \vee \bigcirc\Diamond F$.
  - $F \;\mathbf{U}\; G \Leftrightarrow G \vee (F \wedge \bigcirc(F \;\mathbf{U}\; G))$.
- Negation laws:
  - $\neg\Box F \Leftrightarrow \Diamond\neg F$.
  - $\neg\Diamond F \Leftrightarrow \Box\neg F$.
  - $\neg(F \;\mathbf{U}\; G) \Leftrightarrow (\neg G) \;\mathbf{U}\; (\neg F \wedge \neg G)$.
- Distributivity laws:
  - $\Box(F \wedge G) \Leftrightarrow (\Box F) \wedge (\Box G)$.
  - $\Diamond(F \vee G) \Leftrightarrow (\Diamond F) \vee (\Diamond G)$.
  - $(F \wedge G) \;\mathbf{U}\; H \Leftrightarrow (F \;\mathbf{U}\; H) \wedge (G \;\mathbf{U}\; H)$.
  - $F \;\mathbf{U}\; (G \vee H) \Leftrightarrow (F \;\mathbf{U}\; G) \vee (F \;\mathbf{U}\; H)$.
  - $\Box\Diamond(F \vee G) \Leftrightarrow (\Box\Diamond F) \vee (\Box\Diamond G)$.
  - $\Diamond\Box(F \wedge G) \Leftrightarrow (\Diamond\Box F) \wedge (\Diamond\Box G)$.

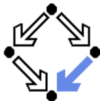
# Classifying System Properties



- Safety Properties:
  - A safety property is a property such that, if it is violated by a run, it is already violated by some finite prefix of the run.
    - This finite prefix cannot be extended in any way to a complete run satisfying the property.
  - Example: $\Box F$.
    - The violating run $F \rightarrow F \rightarrow \neg F \rightarrow \ldots$ has the prefix $F \rightarrow F \rightarrow \neg F$ that cannot be extended in any way to a run satisfying $\Box F$.
- Liveness Properties:
  - A liveness property is a property such that every finite prefix can be extended to a complete run satisfying this property.
    - Only a complete run itself can violate that property.
  - Example: $\Diamond F$.
    - Any finite prefix $p$ can be extended to a run $p \rightarrow F \rightarrow \ldots$ which satisfies $\Diamond F$.

Every system property $P$ is a conjunction $S \wedge L$ of some safety property $S$ and some liveness property $L$ (both may be just "true").

# Verifying Liveness



Example: verify that eventually some state property holds.

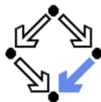
$$
\begin{array}{lcl}
\textbf{var } x := 0, y := 0 & & \\
\textbf{loop} & || & \textbf{loop} \\
\quad x := x + 1 & & \quad y := y + 1
\end{array}
$$

$State = \mathbb{N} \times \mathbb{N}$; $Label = \{p, q\}$.
$I(x, y) :\Leftrightarrow x = 0 \land y = 0$.
$R(l, \langle x, y \rangle, \langle x', y' \rangle) :\Leftrightarrow$
$\quad (l = p \land x' = x + 1 \land y' = y) \lor (l = q \land x' = x \land y' = y + 1)$.

- Prove $\langle I, R \rangle \models \Diamond x = 1$.
  - $[x = 0, y = 0] \rightarrow [x = 0, y = 1] \rightarrow [x = 0, y = 2] \rightarrow \ldots$
  - This run violates (as the only one) $\Diamond x = 1$.
  - Thus the system as a whole does not satisfy $\Diamond x = 1$.

For verifying liveness properties, "unfair" runs have to be ruled out.

# Weak Fairness



- Weak Fairness
  - A run $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \ldots$ is weakly fair to a transition $l$, if
    - if transition $l$ is eventually permanently enabled in the run,
    - then transition $l$ is executed infinitely often in the run.

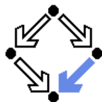
      $(\exists i : \forall j \geq i : Enabled_R(l, s_j)) \Rightarrow (\forall i : \exists j \geq i : l_j = l)$.
- LTL formulas may explicitly specify weak fairness constraints.
  - Let $E_l$ denote the enabling condition of transition $l$.
  - Let $X_l$ denote the predicate "transition $l$ is executed".
  - Define $WF_l :\Leftrightarrow (\Diamond\Box E_l) \Rightarrow (\Box\Diamond X_l)$.

    If $l$ is eventually enabled forever, it is executed infinitely often.
  - Prove $\langle I, S \rangle \models (WF_l \Rightarrow P)$.

    Property $P$ is only proved for runs that are weakly fair to $l$.

A weak requirement to the fairness of a system.

# Strong Fairness



- Strong Fairness
  - A run $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \ldots$ is strongly fair to a transition $l$, if
    - if $l$ is infinitely often enabled in the run,
    - then $l$ is also infinitely often executed the run.

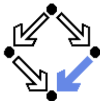
      $(\forall i : \exists j \geq i : Enabled_R(l, s_j)) \Rightarrow (\forall i : \exists j \geq i : l_j = l)$.
  - If $r$ is strongly fair to $l$, it is also weakly fair to $l$ (but not vice versa).
- LTL formulas may explicitly specify strong fairness constraints.
  - Let $E_l$ denote the enabling condition of transition $l$.
  - Let $X_l$ denote the predicate "transition $l$ is executed".
  - Define $SF_l :\Leftrightarrow (\Box\Diamond E_l) \Rightarrow (\Box\Diamond X_l)$.

    If $l$ is enabled infinitely often, it is executed infinitely often.
  - Prove $\langle I, S \rangle \models (SF_l \Rightarrow P)$.

    Property $P$ is only proved for runs that are strongly fair to $l$.

A stronger requirement to the fairness of a system.

# Example



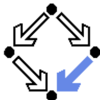
$$
\begin{array}{lcl}
\textbf{var } x := 0, y := 0 & & \\
\textbf{loop} & \| & \textbf{loop} \\
\quad x := x + 1 & & \quad y := y + 1
\end{array}
$$

$State = \mathbb{N} \times \mathbb{N}$; $Label = \{p, q\}$.
$I(x, y) :\Leftrightarrow x = 0 \wedge y = 0$.
$R(l, \langle x, y \rangle, \langle x', y' \rangle) :\Leftrightarrow$
$\quad (l = p \wedge x' = x + 1 \wedge y' = y) \vee (l = q \wedge x' = x \wedge y' = y + 1)$.

- Prove $\langle I, R \rangle \models \Diamond x = 1$.
  - Violating run $[x = 0, y = 0] \rightarrow [x = 0, y = 1] \rightarrow [x = 0, y = 2] \rightarrow \ldots$
  - $Enabled_R(p, \langle x, y \rangle) :\Leftrightarrow$ true.
  - Violating run is not weakly fair to $p$.

Violating run can be ruled out by demanding weak fairness.

# Example



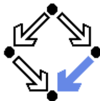
```
var x=0
loop
  a :  x := −x
  b :  choose x := 0 [] x := 1
```

$State := \{a, b\} \times \mathbb{Z}; Label = \{A, B_0, B_1\}$.
$I(p, x) :\Leftrightarrow p = a \wedge x = 0$.
$R(l, \langle p, x\rangle, \langle p', x'\rangle) :\Leftrightarrow$
$(l = A \wedge (p = a \wedge p' = b \wedge x' = -x)) \vee$
$(l = B_0 \wedge (p = b \wedge p' = a \wedge x' = 0)) \vee$
$(l = B_1 \wedge (p = b \wedge p' = a \wedge x' = 1))$.

- Prove: $\langle I, R\rangle \models \Diamond x = 1$.
  - Violating run $[a, 0] \xrightarrow{A} [b, 0] \xrightarrow{B_0} [a, 0] \xrightarrow{A} [b, 0] \xrightarrow{B_0} [a, 0] \xrightarrow{A} \ldots$
  - $Enabled_R(B_1, x) :\Leftrightarrow p = b$.
  - Violating run is weakly fair but not strongly fair to $B_1$.

Violating run can be ruled out by demanding strong fairness.

# Weak versus Strong Fairness

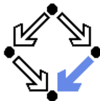


In which situations is which notion of fairness appropriate?

- Multiple processes compete for resource that is permanently available.
  - CPU time, communication bandwidth, etc.
  - Weak fairness suffices.
- Multiple processes compete for resource that becomes temporarily unavailable by being in use.
  - Critical region protected by lock variable (mutex/semaphore).
  - Strong fairness is required.
- Non-deterministic choices are repeatedly made in program.
  - Simultaneous listing on multiple communication channels.
  - Strong fairness is required.

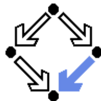
Many other notions or fairness exist.

# Summary



- We can use LTL to specify properties of systems.
  - Every possible system run must satisfy the properties.
- Typical LTL formula patterns for various kinds of properties.
  - Typically at most two levels of temporal operators.
- Properties can be decomposed into a safety and a liveness part.
  - Different verification strategies can be applied to both parts.
- For verifying safety properties, fairness conditions are needed.
  - Unfair system runs are ruled out.

Next: how to verify system properties.



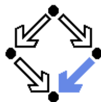
1. A Client/Server System

2. Modeling Concurrent Systems

3. Specifying System Properties

**4. Verifying System Properties**

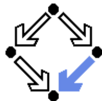
# The Model Checker Spin



- 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

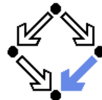


On-the-fly LTL model checking of finite state systems.

- System $S$ modeled by automaton $S_A$.
  - Explicit representation of automaton states.
- 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



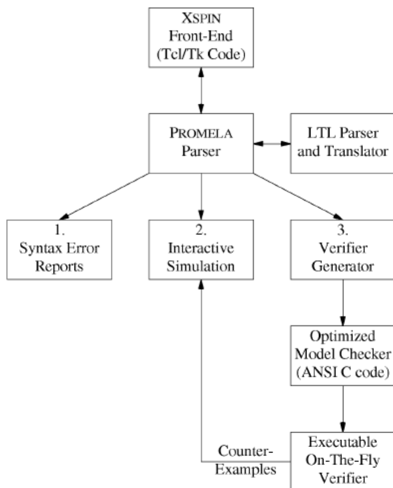
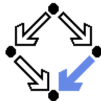



Fig. 1. The structure of SPIN simulation and verification.

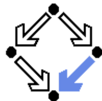
# Features of Spin



- 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.

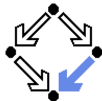
# The Client/Server System in Promela



```
/* 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 };

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

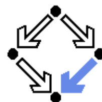
```
/* the client process type */
proctype client(byte id)
{
  do :: true ->
    request[id-1] ! MESSAGE;
    W: answer[id-1]  ? MESSAGE;
    C: skip; // the critical region
    request[id-1] ! MESSAGE
  od;
}
```

# 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;
}
```

# Spin Simulation Options

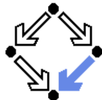


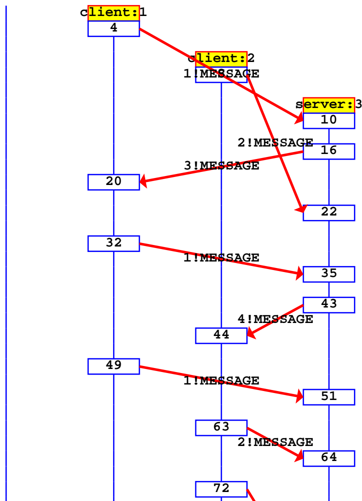
Simulation Options

| Display Mode | Simulation Style |
| --- | --- |
| MSC Panel - with: | Random (using seed) |
| Step Number Labels | Seed Value 1 |
| Source Text Labels | Guided |
| Normal Spacing | Using pan_in.trail |
| Condensed Spacing | Use Browse |
| Time Sequence Panel - with: | Steps Skipped 0 |
| Interleaved Steps | Interactive |
| One Window per Process | **A Full Queue** |
| One Trace per Process | Blocks New Msgs |
| Data Values Panel | Loses New Msgs |
| Track Buffered Channels | **Hide Queues in MSC** |
| Track Global Variables | Queue nr: |
| Track Local Variables | Queue nr: |
| Display vars marked 'show' in MSC | Queue nr: |
| Execution Bar Panel | |

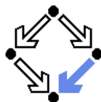
Help Cancel Start

# Simulating the System Execution in Spin





# Spin Verification Options



Basic Verification Options

**Correctness Properties**

- Safety (state properties)
  - Assertions
  - Invalid Endstates
- Liveness (cycles/sequences)
  - Non-Progress Cycles
  - Acceptance Cycles
  - With Weak Fairness
- Apply Never Claim (If Present)
- Report Unreachable Code
- Check xr/xs Assertions

**Search Mode**

- Exhaustive
- Supertrace/Bitstate
- Hash-Compact

**A Full Queue**

- Blocks New Msgs
- Loses New Msgs

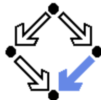
[Add Never Claim from File]

[Verify an LTL Property]

[Set Advanced Options]

Help | Cancel | Run

# Specifying a System Property in Spin



Linear Time Temporal Logic Formulae

Formula: [] !(c1 && c2) Load...

Operators: [] <> U -> and or not

Property holds for: All Executions (desired behavior) No Executions (error behavior)

Notes [file clientServer-mutex.ltl]:

```
Use Load to open a file or a template.
```

Symbol Definitions:

```
#define c1 client[1]@C
#define c2 client[2]@C
```

Never Claim: Generate

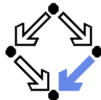
```
        /*
         * Formula As Typed: [] !(c1 && c2)
         * The Never Claim Below Corresponds
         * To The Negated Formula !([] !(c1 && c2))
         * (formalizing violations of the original)
         */

never {    /* !([] !(c1 && c2)) */
```

Verification Result: valid Run Verification

```
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 5 (line 68)
(Spin Version 4.3.0 -- 22 June 2007)
        + Partial Order Reduction

Full statespace search for:
```

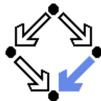
Help Clear Close Save As..

# Spin LTL



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

Operands (opd):
true, false, and user-defined names starting with a lower-case letter

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)
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

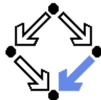


```
#define p (a > b)
#define q (len(q) < 5)
#define r (process@Label)
#define s (process[pid]@Label)
```

- PROMELA conditions with references to *global* system variables.
  - 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*.
    - First instantiated process receives process identifier 1.

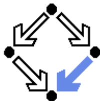
Atomic predicates can describe arbitrary state conditions.

# 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
```

# The Implementation of Spin

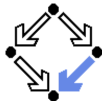


Translation of the original problem to a problem in automata theory.

- 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

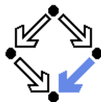


A (variant of a) labeled transition system in a finite state space.

- 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 $I \subseteq State$.
  - A labeled transition relation $R \subseteq Label \times State \times State$.
  - A set of final states $F \subseteq State$.
    - Büchi automata: $F$ is called the set of accepting states.

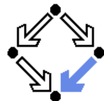
We will only consider infinite runs of Büchi automata.

# Runs and Languages



- An infinite run $r = s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \ldots$ 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 $w(r) := \langle l_0, l_1, l_2, \ldots \rangle$.
- $A = \langle I, R, F \rangle$ 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 $A$.

# A Finite State System as an Automaton

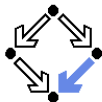


The automaton $S_A = \langle I, R, F \rangle$ for a finite state system $S = \langle I_S, R_S \rangle$:

- $State := States_S \cup \{\iota\}$.
  - The state space $States_S$ 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 $\iota$.
- $R(l, s, s') :\Leftrightarrow l = L(s') \wedge (R_S(s, s') \vee (s = \iota \wedge I_S(s')))$.
  - $L(s) := \{p \in AP : s \models p\}$.
  - Each transition is labeled by the set of atomic propositions satisfied by the successor state.
  - Thus all atomic propositions are evaluated on the successor state.
- $F := State$.
  - Every state is accepting.

# A Finite State System as an Automaton



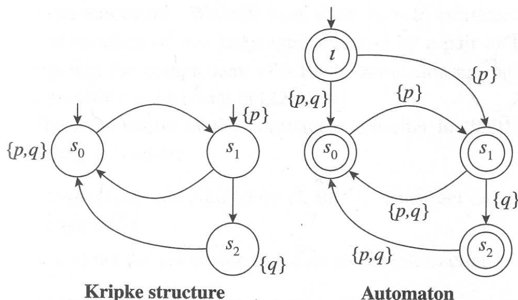
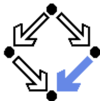


Figure 9.2
Transforming a Kripke structure into an automaton.

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

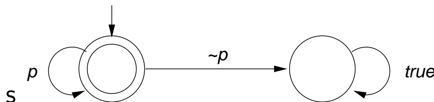
If $r = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$ is a run of $S$, then $S_A$ accepts the labelled version $r_l := \iota \xrightarrow{L(s_0)} s_0 \xrightarrow{L(s_1)} s_1 \xrightarrow{L(s_2)} s_2 \xrightarrow{L(s_3)} \ldots$ of $r$.

# A System Property as an Automaton

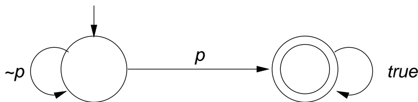


Also an PLTL formula can be translated to a finite state automaton.

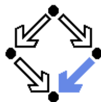
- We need the automaton $P_A$ for a PLTL property $P$.
  - Requirement: $r \models P \Leftrightarrow P_A$ accepts $r_l$.
  - A run satisfies property $P$ if and only if automaton $A_P$ accepts the labeled version of the run.
- Example: $\Box p$.



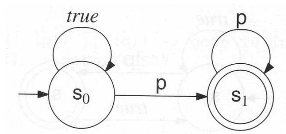

- Example: $\Diamond p$.



# Further Examples

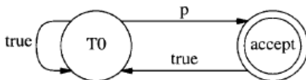


- Example: $\Diamond\Box p$.




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

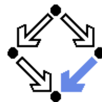
- Example: $\Box\Diamond p$.




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

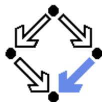
Arbitrary PLTL formulas can be converted to automata.

# 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 $\overline{L} := \{w : w \notin L\}$.
  - Equivalent to: $\mathcal{L}(S_A) \cap \mathcal{L}(\neg P_A) = \emptyset$.
    - $\overline{\mathcal{L}(A)} = \mathcal{L}(\neg A)$.
- 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 $w = w(r_l)$ exists, then $r$ is a counterexample, i.e. a run of $S$ such that $r \not\models P$.

# Synchronized Product of Two Automata

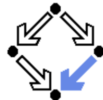


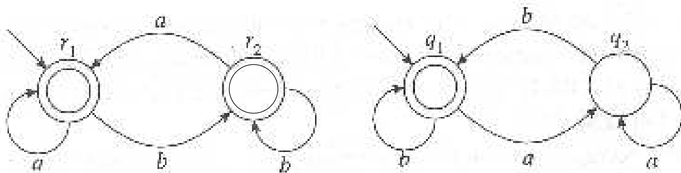
Given two finite automata $A = \langle I_A, R_A, State_A \rangle$ and $B = \langle I_B, R_B, F_B \rangle$.

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

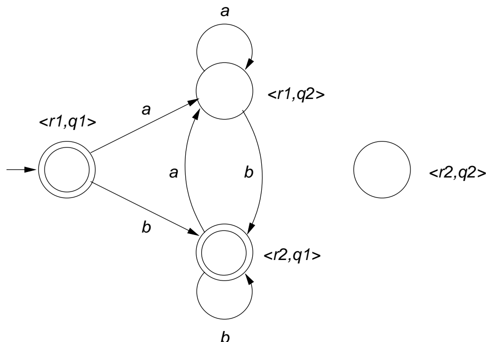
Special case where all states of automaton $A$ are accepting.

# Synchronized Product of Two Automata

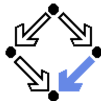





Edmund Clarke: "Model Checking", 1999.



# Checking Emptiness

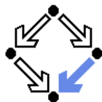


How to check whether $\mathcal{L}(A)$ is non-empty?

- Search for a run $r$ accepted by $A$.
  - $r$ must contain infinitely many occurrences of some accepting state $s$.
    - $r = \iota \to \ldots \to s \to \ldots \to s \to \ldots \to s \to \ldots$
  - Since the state space is finite, $r$ must contain a cycle including $s$.
    - Finite prefix $\iota \to \ldots \to s$ from initial state $\iota$ to $s$.
    - Infinite repetition of "acceptance cycle" $s \to \ldots \to s$ from $s$ to itself.
- Search for a reachable acceptance cycle $s \to \ldots \to s$.
  - Depth first search algorithm that starts with initial state and looks for a reachable accepting state $s$ with a cycle back to itself.
- If found, the acceptance cycle determines a counterexample run.
  - A system run that violates the stated system property.
  - If none can be found, the system satisfies the specified property.

The core of Spin is the search for reachable acceptance cycles.

# Complexity of the Search

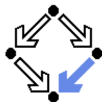


The complexity of checking $S \models P$ is as follows.

- 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 the state space of the system automaton.
  - Actually, in the number of reachable states (typically much smaller).

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

# On the Fly Model Checking

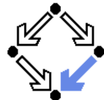


For checking $\mathcal{L}(S_A \otimes (\neg P)_A) = \emptyset$, it is not necessary to construct the states of $S_A$ in advance.

- 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.

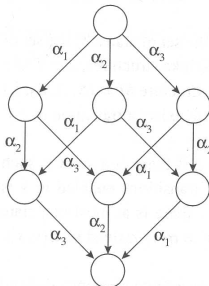
# Partial Order Reduction



Core problem of model checking: state space explosion.

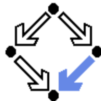
- Take asynchronous composition $S_0||S_1||\dots||S_{k-1}$.
  - Take state $s$ where one transition of each component is enabled.
    - Assume that the transition of one component does not disable the transitions of the other components and that no other transition becomes enabled before all the transitions 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.




Edmund Clarke: "Model Checking", 1999.

# Summary

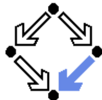


Application of Spin for deciding $S \models P$.

- 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 $(\neg P)_A$.
  - There exists an algorithm for performing the conversion.
- Construct synchronized product automaton $S_A \otimes (\neg P)_A$.
  - After that, formula labels are not needed any more.
- Find accepting run 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.
- Apply on-the-fly model checking and partial order reduction to reduce the problem of state space explosion.

Fully automatic, but only applicable to finite state systems.

# Outlook



Other approaches/tools to system verification exist.

- Model checking:
  - Symbolic Model Checking: SMV, NuSMV, ...
    - Systems and properties are modelled as binary decision diagrams (BDDs).
  - Bounded Model Checking: NuSMV2, ...
    - Model checking is reduced to checking the satisfiability of propositional formulas.
  - Counter-Example Guided Abstraction Refinement: BLAST, SLAM.
    - Abstract model is iteratively checked and refined; in principle also applicable to infinite state systems.
- Theorem proving:
  - Theorem proving assistants: PVS, Isabelle, Coq, the RISC ProofNavigator, ...
  - Not fully automatic, system invariants have to be established.
  - Complexity of verification independent of size of state space, also applicable to infinite state systems.