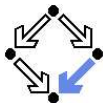
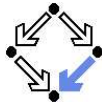


# Model Checking (Part 3)

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>





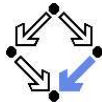
---

## 1. Basic PLTL Model Checking

## 2. Translating PLTL Formulas to Automata

## 3. Optimized PLTL Model Checking

# The Basic Approach

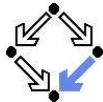


Translation of the original problem to a problem in automata theory.

- **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

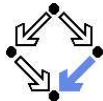


A (variant of a) labeled transition system in a finite state space.

- 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  $a$ .

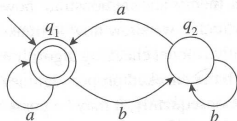
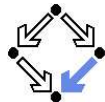


Figure 9.1  
A finite automaton.

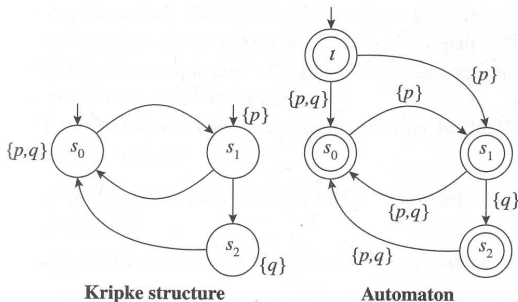
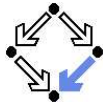
# A Finite State System as an Automaton



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.
  - **Thus all atomic propositions are evaluated on the successor state.**
- $F := State$ .
  - Every state is accepting.

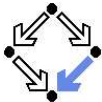
# A Finite State System as an Automaton



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

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

If  $r = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$  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  $r$ .

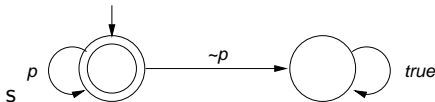


# A System Property as an Automaton

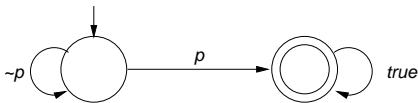
Also an PLTL formula can be translated to a finite state automaton.

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

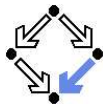
- **Example:**  $\Box p$ .



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

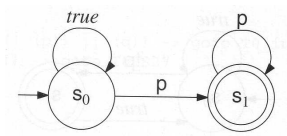






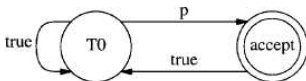
# Further Examples

- Example:  $\diamond \square p$ .



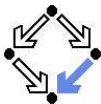
Gerard Holzmann: "The Spin Model Checker", 2004.

- Example:  $\square \diamond p$ .



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

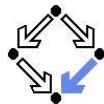
We will give later an algorithm to convert arbitrary PLTL formulas to automata.



# System Properties

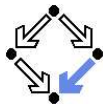
- **State equivalence:**  $L(s) = L(t)$ .
  - Both states have the same labels.
  - Both states satisfy the same atomic propositions in  $AP$ .
- **Run equivalence:**  $w(r_l) = w(r'_l)$ .
  - Both runs have the same sequences of labels.
  - Both runs satisfy the same PLTL formulas built over  $AP$ .
- **Indistinguishability:**  $w(r_l) = w(r'_l) \Rightarrow (r \models P \Leftrightarrow r' \models R)$ 
  - PLTL formula  $P$  cannot distinguish between runs  $r$  and  $r'$  whose labeled versions read the same words.
- **Consequence:**  $S \models P \Leftrightarrow \mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$ .
  - Proof that, 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.
  - “Vice versa” direction relies on indistinguishability property.

# 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  $w = w(r_i)$  exists, then  $r$  is a **counterexample**, i.e. a run of  $S$  such that  $r \not\models P$ .

# Synchronized Product of Two Automata

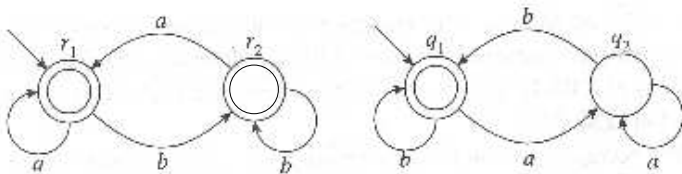
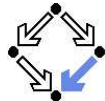


Given two finite automata  $A = \langle I_A, R_A, State_A \rangle$  and  $B = \langle I_B, R_B, F_B \rangle$ .

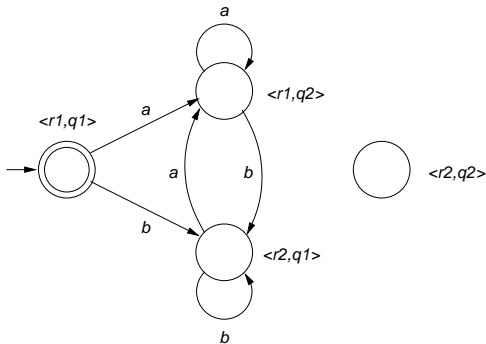
- **Synchronized product**  $A \otimes B = \langle I, R, F \rangle$ .
  - $State := State_A \times State_B$ .
  - $Label := Label_A = Label_B$ .
  - $I := I_A \times I_B$ .
  - $R(I, \langle s_A, s_B \rangle, \langle s'_A, s'_B \rangle) := R_A(I, s_A, s'_A) \wedge R_B(I, s_B, s'_B)$ .
  - $F := State_A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

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

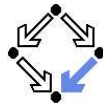
# Synchronized Product of Two Automata



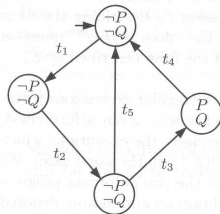
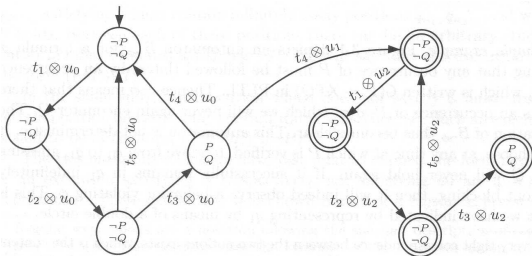
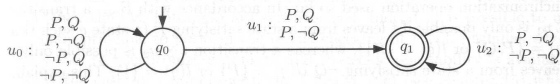
Edmund Clarke: "Model Checking", 1999.



# Example



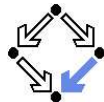
Check whether  $S \models \square(P \Rightarrow \bigcirc \diamond Q)$ .



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

The product automaton accepts a run, thus the property does not hold.

# Checking Emptiness

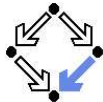


How to check whether  $\mathcal{L}(A)$  is non-empty?

- Suppose  $A = \langle I, R, F \rangle$  accepts a run  $r$ .
  - Then  $r$  contains infinitely many occurrences of some state in  $F$ .
  - Since  $State$  is finite, in some suffix  $r'$  every state occurs infinit. often.
  - Thus every state in  $r'$  is reachable from every other state in  $r'$ .
- $C$  is a **strongly connected component (SCC)** of graph  $G$  if
  - $C$  is a subgraph of  $G$ ,
  - every node in  $C$  is reachable from every other node in  $C$  along a path entirely contained in  $C$ , and
  - $C$  is maximal (not a subgraph of any other SCC of  $G$ ).
- Thus the states in  $r'$  are contained in an SCC  $C$ .
  - $C$  is reachable from an initial state.
  - $C$  contains an accepting state.
  - Conversely, *any* such SCC generates an accepting run.

$\mathcal{L}(A)$  is non-empty if and only if the reachability graph of  $A$  has an SCC that contains an accepting state.

# Checking Emptiness



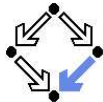
Find in the reachability graph an SCC that contains an accepting state.

- We have to find **an accepting state with a cycle back to itself**.
  - Any such state belongs to some SCC.
  - Any SCC with an accepting state has such a cycle.
  - Thus this is a sufficient and necessary condition.
- Any such a state  $s$  defines a **counterexample run**  $r$ .
  - $r = \iota \rightarrow \dots \rightarrow s \rightarrow \dots \rightarrow s \rightarrow \dots \rightarrow s \rightarrow \dots$
  - Finite prefix  $\iota \rightarrow \dots \rightarrow s$  from initial state  $\iota$  to  $s$ .
  - Infinite repetition of cycle  $s \rightarrow \dots \rightarrow s$  from  $s$  to itself.

This is the core problem of PLTL model checking; it can be solved by a *depth-first search* algorithm.



# 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*( $D$ )

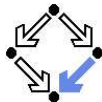
**end**

**end**

**end**

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

# Checking State Properties



Apply depth-first search to checking a state property (assertion).

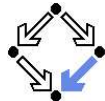
```
global
  StateSpace  $V := \{\}$ 
  Stack  $D := \langle \rangle$ 

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

function search( $s$ )
   $V := V \cup \{s\}$ 
  if  $\neg check(s)$  then
    print  $D$ 
    return true
  end
  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
  return false
end
```

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

# Depth-First Search for Acceptance Cycle



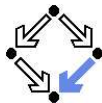
```
global
  ...
  Stack C := {}

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

```
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
```

```
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
```

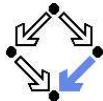
# Depth-First Search for Acceptance Cycle



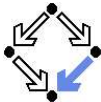
- At each call of  $search(s)$ ,
  - $s$  is a reachable state,
  - $D$  describes a path from  $\iota$  to  $s$ .
- $search$  calls  $searchCycle(s)$ 
  - on a reachable accepting state  $s$
  - in order to find a cycle from  $s$  to itself.
- At each call of  $searchCycle(s)$ ,
  - $s$  is a state 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.**



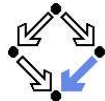
# Complexity of the Search

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.**

# Adding Weak Scheduling Fairness



Assume that system is an asynchronous composition of  $k$  components.

- **Weak scheduling fairness:**

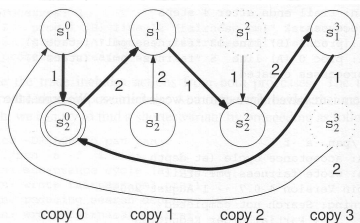
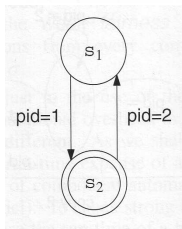
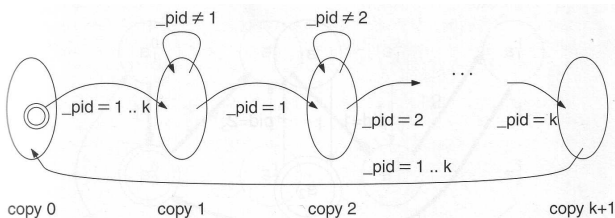
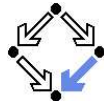
- A run satisfies w.s.f., if it has infinitely many transitions from every component which is permanently ready to perform some transition.

- **Implementation in a model checker.**

- Construct  $k + 2$  copies  $0, \dots, k + 1$  of the state space.
  - Actually just  $k + 2$  bits need to be added to every state.
  - Only copy 0 has the **original acceptance states**.
- In copy 0, the destinations of all transitions **from acceptance states** are replaced by the corresponding states in copy 1.
- In copy  $i, 1 \leq i \leq k$ , the destinations of all transitions **of component  $i$**  are replaced by the corresponding states in copy  $i + 1$ ; a null transition is added from every state to the corresponding state in copy  $i + 1$  which is enabled if no other transition of component  $i$  is enabled.
- In copy  $k + 1$ , the transitions of **all components** are replaced by the corresponding states in copy 0.

**Every accepted run has (possibly null) transitions from all components.**

# Example

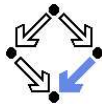


Gerard Holzmann: "The Spin Model Checker", 2004.

Runtime complexity of model checking is increased by a factor  $k + 2$ .



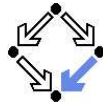
# Summary



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, remains to be described.
- Construct synchronized product automaton  $S_A \otimes (\neg P)_A$ .
  - After that, formula labels are not needed any more.
- Find SCC 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.



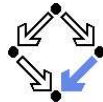
---

1. Basic PLTL Model Checking

2. Translating PLTL Formulas to Automata

3. Optimized PLTL Model Checking

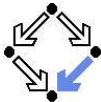
# From a PLTL Formula to an Automaton



How to translate an PLTL formula  $P$  to an automaton  $P_A$ ?

- **Automaton states:** “atoms”.
  - An **atom** is a particular set of subformulas of  $P$ .
    - May influence the value of the PLTL formula.
    - May be simultaneously satisfied by some state.
- **Construction:**  $P_A$  is the synchronous product of two automata.
  - A **local automaton:** checking the safety aspect of  $P$ .
    - Atoms describe system states.
    - Transitions describe the system transitions allowed by  $P$ .
  - An **eventuality automaton:** checking the liveness aspect of  $P$ .
    - Atoms describe goals that have to be satisfied by a system run.
    - Transitions describe the changes of these goals.

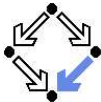
Today there are better approaches generating smaller automata, but the presented one is easier to understand.



# The Closure of a Formula

- We will only consider the operators  $\neg, \wedge, \circ, \mathbf{U}$ .
  - $f_0 \vee f_1 \Leftrightarrow \neg(\neg f_0 \wedge \neg f_1), \dots$
  - $\diamond f \Leftrightarrow \text{true } \mathbf{U} f$ .
  - $\square f \Leftrightarrow \neg \diamond \neg f$ .
- The closure  $\mathcal{C}(P)$  of a PLTL formula  $P$ .
  - The smallest set satisfying the following conditions:
    - $P \in \mathcal{C}(P)$ .
    - If  $\neg f_0 \in \mathcal{C}(P)$ , then  $f_0 \in \mathcal{C}(P)$ .
    - If  $f_0 (\neq \neg f_1) \in \mathcal{C}(P)$ , then  $\neg f_0 \in \mathcal{C}(P)$ .
    - If  $f_0 \wedge f_1 \in \mathcal{C}(P)$ , then  $f_0, f_1 \in \mathcal{C}(P)$ .
    - If  $\circ f_0 \in \mathcal{C}(P)$ , then  $f_0 \in \mathcal{C}(P)$ .
    - If  $f_0 \mathbf{U} f_1 \in \mathcal{C}(P)$ , then  $f_0, f_1 \in \mathcal{C}(P)$ .
  - Example:  $P :\Leftrightarrow (\neg p) \mathbf{U} q$ .  
 $\mathcal{C}(P) = \{P, \neg P, p, \neg p, q, \neg q\}$ .

The closure of  $P$  is the set of its subformulas and their negation.

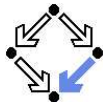


# The Atoms of a Formula

The closure  $\mathcal{C}(P)$  gives the material to construct sets of formulas.

- An atom  $K \subseteq \mathcal{C}(P)$  of  $P$ :
  - An atom  $K$  of  $P$  is a subset of the subformulas of  $P$  that do not have a propositional contradiction.
    - If  $f_0 \in \mathcal{C}(P)$ , then  $f_0 \in K$  iff  $\neg f_0 \notin K$ .
    - If  $f_0 \wedge f_1 \in \mathcal{C}(P)$ , then  $f_0 \wedge f_1 \in K$  iff  $f_0 \in K$  and  $f_1 \in K$ .
  - $P_K := \{K : K \text{ is an atom of } P\}$ .
    - The set of atoms of  $P$ .
- Example:  $P := (\neg p) \mathbf{U} q$ .  
$$P_K = \{\{P, p, q\}, \{P, p, \neg q\}, \{P, \neg p, q\}, \{P, \neg p, \neg q\},$$
$$\{\neg P, p, q\}, \{\neg P, p, \neg q\}, \{\neg P, \neg p, q\}, \{\neg P, \neg p, \neg q\}\}.$$

The atoms represents the states of the automaton to be constructed.

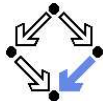


# The Local Automaton of a Formula

- The **local automaton**  $P_L = \langle I_L, R_L, F_L \rangle$  of  $P$ :
  - $State := P_K, Label := \mathbb{P}(\mathcal{C}(P))$ .
    - States are atoms, labels are sets of formulas from the closure of  $P$ .
  - $I_L := \{a \in P_K : P \in a\}$ .
    - The initial states are all atoms that contain  $P$ .
  - $R_L(l, a, a') : \Leftrightarrow l = a \wedge$ 
    - $\forall \circ f \in \mathcal{C}(P) :$ 
      - $\circ f \in a \Leftrightarrow f \in a' \wedge$
    - $\forall f_0 \mathbf{U} f_1 \in \mathcal{C}(P) :$ 
      - $f_0 \mathbf{U} f_1 \in a \Leftrightarrow f_1 \in a \vee (f_0 \in a \wedge f_0 \mathbf{U} f_1 \in a')$ .
  - $F_L := P_K$ .
    - All states are accepting; i.e. every accepted run passes some state infinitely often.

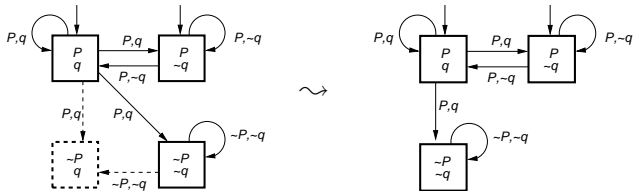
The transition label describes the obligations of the predecessor state; the local automaton checks all obligations except for the **U** formulas.

# Example



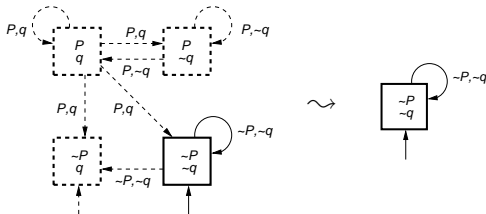
- Local automaton  $P_L$  where  $P \Leftrightarrow \text{true } U q$  ( $\Leftrightarrow \diamond q$ ).

- Accepts every run (no safety aspect).

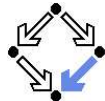


- Local automaton  $(\neg P)_L$  where  $\neg P \Leftrightarrow \neg(\text{true } U q)$  ( $\Leftrightarrow \square \neg q$ ).

- Accepts only runs where  $q$  does not occur.



# The Eventuality Automaton of a Formula

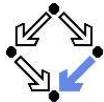


- The **eventuality automaton**  $P_E = \langle I_E, R_E, F_E \rangle$  of  $P$ :
  - $State := \mathbb{P}(\{f_0 \mathbf{U} f_1 : f_0 \mathbf{U} f_1 \in P_K\})$ ;  $Label := \mathbb{P}(C(P))$ .
    - States are sets of  $\mathbf{U}$  formulas; labels are formula sets.
  - $I_L := \{\emptyset\}$ .
    - The initial state is the empty formula set.
  - $R_L(l, a, a') : \Leftrightarrow$ 
    - $(a = \emptyset \wedge \forall f_0 \mathbf{U} f_1 \in l : f_1 \in l \vee f_0 \mathbf{U} f_1 \in a') \vee$
    - $(a \neq \emptyset \wedge \forall f_0 \mathbf{U} f_1 \in a : f_1 \in l \vee f_0 \mathbf{U} f_1 \in a')$ .
    - From the initial state, the successor state has to fulfill all obligations of the label not satisfied by the label itself.
    - From each other state, the successor state has to fulfill all obligations of the current state not satisfied by the label.
  - $F_L := \{\emptyset\}$ .
    - Only the initial state is accepting; i.e. every accepted run passes the initial state infinitely often.

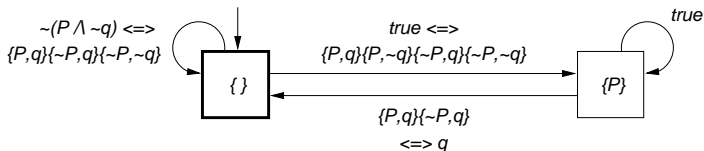
The transition label describes the obligations of the predecessor state involving  $\mathbf{U}$  formulas; the eventuality automaton checks these obligations.



# Example



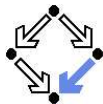
- Eventuality automaton  $P_E$  where  $P \Leftrightarrow \text{true } \mathbf{U} \ q$  ( $\Leftrightarrow \diamond q$ )  
(also for  $(\neg P)_E$  where  $\neg P \Leftrightarrow \neg(\text{true } \mathbf{U} \ q) \Leftrightarrow \square \neg q$ ).



- If the initial obligation is  $\{P, \neg q\}$ , the only possible transition is to the non-accepting state.
  - In all other cases, there is an accepting run through the initial state.
- If the non-accepting state eventually encounters  $q$ , the obligation is fulfilled and a transition back to the accepting state is possible.

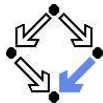
Only runs are accepted where all obligations are infinitely often fulfilled.

# Synchronous Product of Both Automata

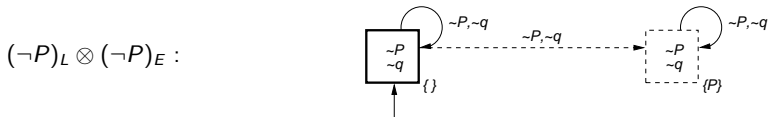
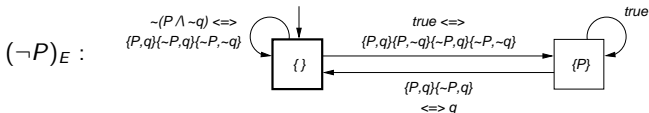


- Synchronous product  $P_L \otimes P_E$ .
  - As already defined; finally all non-atomic propositions are removed from the labels.
- **Intuitive understanding:**
  - $P_L \otimes P_E$  accepts only runs that both  $P_L$  and  $P_E$  accept.
  - $P_L$  accepts a run with transition  $a \xrightarrow{a} a'$ , if the transition does not violate the safety aspect of  $P$ , even if  $a$  does not satisfy the **U** formulas with which it was labeled.
  - $P_E$  accepts a run with transition  $b \xrightarrow{a} b'$ , if either  $b$  satisfies the **U** formulas in label  $a$ , or if this obligation is passed on to  $a'$  and eventually a state is reached that fulfills the obligation.
  - $P_L \otimes P_E$  thus accepts a run with transition  $\langle a, b \rangle \xrightarrow{a} \langle a', b' \rangle$ , if the transition does not violate the safety aspect of  $P$  and if either  $\langle a, b \rangle$  satisfies the **U** formulas in  $a$ , or if this obligation is passed on to  $\langle a', b' \rangle$  and eventually a state is reached that fulfills the obligation.

# Example

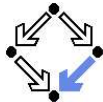


Product  $(\neg P)_L \otimes (\neg P)_E$  where  $\neg P \Leftrightarrow \neg(\text{true } \mathbf{U} \ q) \Leftrightarrow \Box \neg q$ .



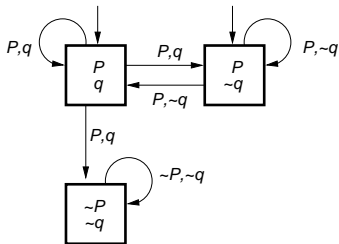
$(\neg P)_L \otimes (\neg P)_E$  is equivalent to  $(\neg P)_L$  (no liveness aspect in  $\neg P$ ).

# Example

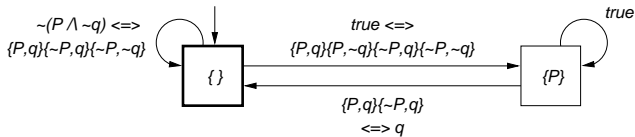


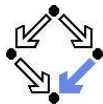
Product  $P_L \otimes P_E$  where  $P \Leftrightarrow \text{true}$  **U**  $q \Leftrightarrow \diamond q$ .

$P_L$  :



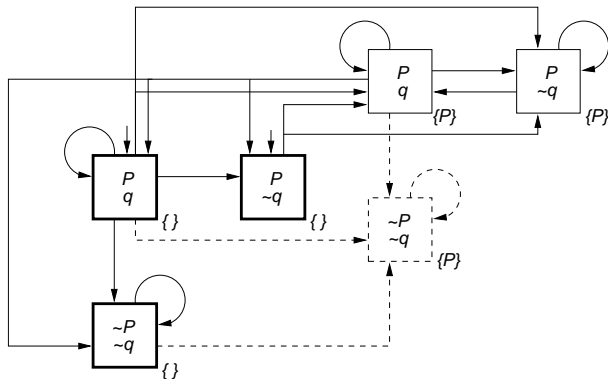
$P_E$  :





# Example

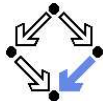
Product  $P_L \otimes P_E$  where  $P \Leftrightarrow \text{true}$   $\mathbf{U}$   $q \Leftrightarrow \diamond q$ .



Can remove a state from which no acceptance state can be reached.

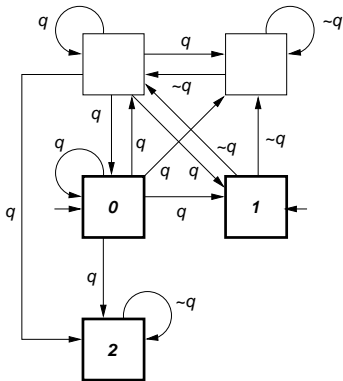


# Example

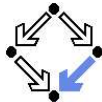


Product  $P_L \otimes P_E$  where  $P \Leftrightarrow \text{true}$   $\mathbf{U} q \Leftrightarrow \diamond q$ .

- In every accepting run,  $q$  must occur.
  - Accepting states have only incoming edges labeled with  $q$ .
  - Exception:  $2 \rightarrow 2$ , but 2 is not initial.
- All runs where  $q$  occurs are accepted.
  - $q$  in first state:  $0 \rightarrow \dots$
  - $q$  not in first state:  $1 \rightarrow \dots$
  - Occurrences of  $q$ :  
 $\dots \rightarrow 0 \rightarrow \dots \rightarrow 0 \rightarrow \dots$
  - $q$  finitely often:  
 $\dots \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow \dots$
  - $q$  eventually forever:  
 $\dots \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow \dots$



This automaton is much bigger than necessary (typical for the approach).



---

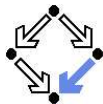
## 1. Basic PLTL Model Checking

## 2. Translating PLTL Formulas to Automata

## 3. Optimized PLTL Model Checking



# On the Fly Model Checking

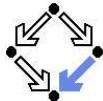


For checking  $\mathcal{L}(S_A \otimes (\neg P)_A) = \emptyset$ , it is not necessary to construct the states of  $S_A$  in advance.

- 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.**

# On the Fly Model Checking

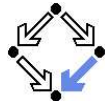


Expansion of state  $s = \langle s_0, s_1 \rangle$  of product automaton  $S_A \otimes (\neg P)_A$  into the set  $R(s)$  of transitions from  $s$  (**for**  $\langle l, s, s' \rangle \in R(s)$  **do** ...).

- Let  $S'_1$  be the set of all successors of state  $s_1$  of  $(\neg P)_A$ .
  - Property automaton  $(\neg P)_A$  has been precomputed.
- Let  $S'_0$  be the set of all successors of state  $s_0$  of  $S_A$ .
  - Computed on the fly by applying system transition relation to  $s_0$ .
- $R(s) := \{ \langle l, \langle s_0, s_1 \rangle, \langle s'_0, s'_1 \rangle \rangle : s'_0 \in S'_0 \wedge s'_1 \in S'_1 \wedge s_1 \xrightarrow{l} s'_1 \wedge L(s'_0) \in I \}$ .
  - Choose candidate  $s'_0 \in S'_0$ .
  - Determine set of atomic propositions  $L(s'_0)$  true in  $s'_0$ .
  - If  $L(s'_0)$  is not consistent with the label of any transition  $\langle s_0, s_1 \rangle \xrightarrow{l} \langle s'_0, s'_1 \rangle$  of the proposition automaton,  $s'_0$  it is ignored.
  - Otherwise,  $R$  is extended by every transition  $\langle s_0, s_1 \rangle \xrightarrow{l} \langle s'_0, s'_1 \rangle$  where  $L(s'_0)$  is consistent with label  $l$  of transition  $s_1 \xrightarrow{l} s'_1$ .

Actually, depth-first search proceeds with first suitable successor  $\langle s'_0, s'_1 \rangle$  before expanding the other candidates.

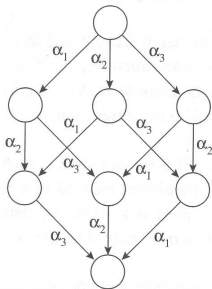
# Partial Order Reduction



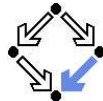
Core problem of model checking: state space explosion.

- 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 three transitions have been performed.
  - Take state  $s'$  after execution of all three transitions.
    - There are  $k!$  paths leading from  $s$  to  $s'$ .
    - There are  $2^k$  states involved in the transitions.

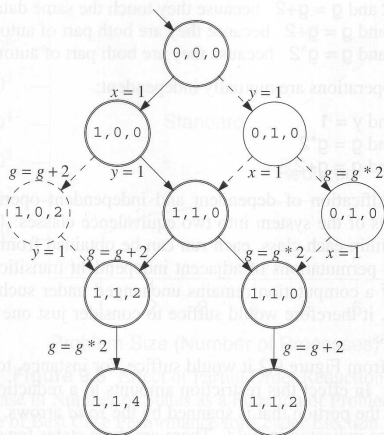
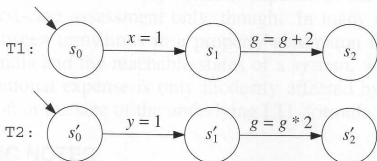
Sometimes it suffices to consider  
a *single path* with  $k + 1$  states.



# Example



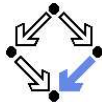
Check  $(T1 || T2) \models \diamond g \geq 2$ .



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  $\square x \geq y$ ).

# Partial Order Reduction



Check  $S \models P$ .

**boolean** *search*( $s$ )

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

**boolean** *search*( $s$ )

...  
**for**  $\langle l, s, s' \rangle \in ample_P(s)$  **do**

$\rightsquigarrow$

■  $ample_P(s) \subseteq R(s)$ .

■ The ample set  $ample_P(s)$ .

■ The set of transitions from  $s$  to be considered for checking  $P$ .

■  $R(s) := \{ \langle l, s, s' \rangle : l \in Label \wedge s' \in State \}$ .

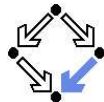
■ The set of all transitions from  $s$ .

■ Optimization:  $ample_P(s) \subsetneq R(s)$ .

■ Search space is reduced.

We will now investigate the calculation of the ample set.

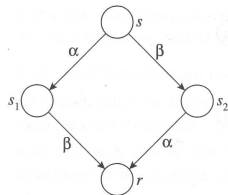
# Independence



When can two transitions of two components be executed in any order?

## ■ Independence relation $I \subseteq Trans \times Trans$ .

- $I$  is a relation between transitions.
  - $Trans := Label \times State \times State$ .
  - $I(\alpha, \beta) \dots \alpha$  and  $\beta$  are independent.
- $I$  satisfies the following conditions for every  $s \in State$  and  $\alpha, \beta \in Trans$ :



### ■ $I$ is antireflexive and symmetric:

$$\neg I(\alpha, \alpha) \wedge (I(\alpha, \beta) \Rightarrow I(\beta, \alpha)).$$

Edmund Clarke: "Model Checking", 1999.

### ■ Enabledness: $\alpha, \beta \in R(s) \Rightarrow \alpha \in R(\beta(s))$ .

If  $\alpha$  and  $\beta$  are enabled, then  $\alpha$  is still enabled after executing  $\beta$ .

### ■ Commutativity: $\alpha, \beta \in R(s) \Rightarrow \alpha(\beta(s)) = \beta(\alpha(s))$ .

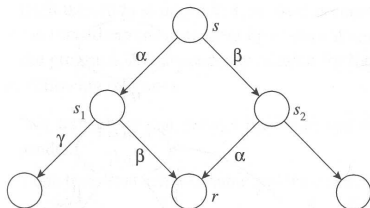
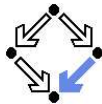
If  $\alpha$  and  $\beta$  are enabled, then they may be executed in any order.

## ■ Dependence relation $D(\alpha, \beta) := \Leftrightarrow \neg I(\alpha, \beta)$ .

- Two transitions are dependent, if they are not independent.

$I(\alpha, \beta)$  is necessary for considering only one transition  $\alpha$  or  $\beta$  from  $s$ .

# Open Problems



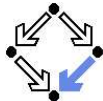
Edmund Clarke: "Model Checking", 1999.

$I(\alpha, \beta)$  is not sufficient for considering only one transition  $\alpha$  or  $\beta$ .

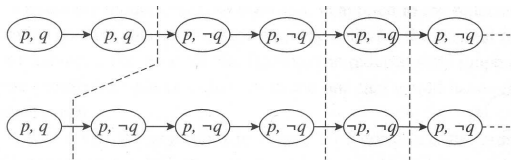
- **Problem 1:** property  $P$  may distinguish between  $s_1$  and  $s_2$ .
  - $s \rightarrow s_1 \rightarrow r$  may be recognized as different from  $s \rightarrow s_2 \rightarrow r$ .
- **Problem 2:** states  $s_1$  and  $s_2$  may have successors in addition to  $r$ .
  - If  $\alpha$  is not considered, not every successor of  $s_1$  may be explored.

Need some more checks to make sure that these problems do not occur.

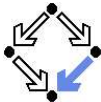
# Invisibility and Stuttering Equivalence



- A transition  $\alpha$  is invisible:
  - $\forall s, s' \in State : s' = \alpha(s) \Rightarrow L(s) = L(s')$ .
  - Predecessor and successor state fulfill the same atomic propositions.
- Two runs  $r$  and  $s$  are stuttering equivalent:
  - There are two infinite sequences  $i_0, i_1, \dots$  and  $j_0, j_1$  of positions in  $r$  respectively  $s$  such that for all  $k \in \mathbb{N}$ 
$$L(r(i_k)) = L(r(i_k + 1)) = \dots = L(r(i_{k+1} - 1)) =$$
$$L(s(j_k)) = L(s(j_k + 1)) = \dots = L(s(j_{k+1} - 1)).$$
  - $r$  and  $s$  have identical sequences of labels except for the number of occurrences of each label in subsequent positions.





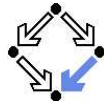


# Stuttering Invariance

- A PLTL formula  $P$  is invariant under stuttering:
  - For each pair of stuttering equivalent runs  $r$  and  $s$ :  $r \models P$  iff  $s \models P$ .
  - $P$  cannot distinguish among  $r$  and  $s$ .
- Every PLTL formula without  $\bigcirc$  is invariant under stuttering.
  - Only temporal operators  $\square$ ,  $\diamond$ ,  $\mathbf{U}$ .
  - Only these operators are allowed in Spin as well.
  - Also converse of statement is true: every path property that does not distinguish between stuttering equivalent runs can be expressed by a PLTL formula without  $\bigcirc$ .

Partial order reduction only applies to checking formulas that are invariant under stuttering.

# Computation of the Ample Set



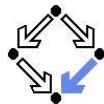
Check  $(S_0 \parallel \dots \parallel S_{k-1}) \models P$  where  $P$  is invariant under stuttering.

```
function ampleP(s) :  
  for Si with Ri(s) ≠ ∅ do  
    if C1(Ri(s)) ∧ C2(s, Ri(s)) ∧ C3(s, Si) then  
      return Ri(s)  
    end  
  end  
  return R(s)  
end
```

- Transition relation  $R_i$  of component  $i$ :
  - Total system transition relation  $R := \bigcup_{0 \leq i < k} R_i$ .
- Result of  $\text{ample}_P(s)$ :
  - Some component  $i$ 's non-empty set of transitions from  $s$  that satisfies conditions  $C1$ ,  $C2$ ,  $C3$  (if such a set exists).
  - All transitions from  $s$ , if no such set exists.

If successful, only transitions from component  $i$  are considered.

# Condition 1: Invisibility

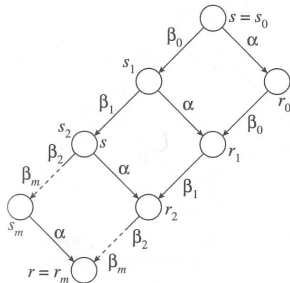


$C1(T) : \Leftrightarrow$

$\forall \alpha \in T : \alpha \text{ is invisible.}$

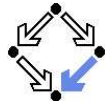
- If  $ample(s) \neq R(s)$ , then every  $\alpha \in ample(s)$  is invisible.

A reduced ample set has only invisible transitions; otherwise the discarded runs would not be stuttering equivalent to the considered run.



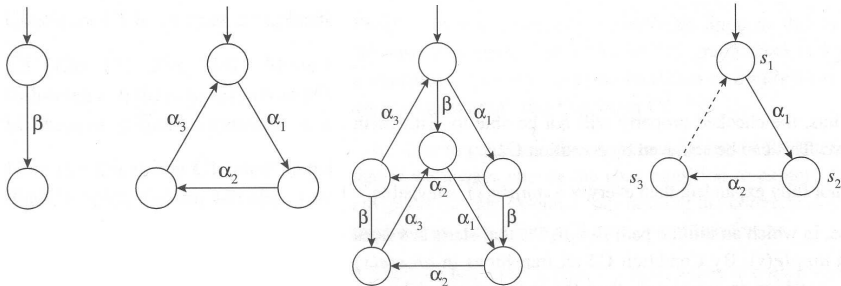
Edmund Clarke: "Model Checking", 1999.

## Condition 2: Cycle Condition



$$C2(s, T) :\Leftrightarrow \forall \alpha \in T : \neg has(D, \alpha(s)).$$

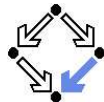
- If  $ample(s) \neq R(s)$ , then no  $\alpha \in ample(s)$  may lead to a cycle.



Edmund Clarke: "Model Checking", 1999.

$C2$  ensures that  $ample_P(s_3) = \{\beta\}$ ; otherwise  $\beta$  would not appear at all in the reduced state graph.

## Condition 3: Independence

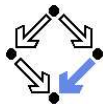


$$\begin{aligned} C3(s, S_i) : &\Leftrightarrow \\ &\forall P_j \neq P_i : \\ &\quad \forall \beta \in R_j(s) : \\ &\quad (\forall \alpha \in R_i(s) : I(\beta, \alpha) \wedge \\ &\quad \forall \alpha \in \text{current}_i(s) \setminus R_i(s) : \neg \text{enable}(\beta, \alpha)) \end{aligned}$$

- $\text{current}_i(s) := \{\alpha : \alpha \in R_i \wedge \exists s' : \alpha \in R(s') \wedge pc_i(s) = pc_i(s')\}$ .
  - The set of transitions of  $S_i$  that are enabled in any state where  $S_i$  is at the same statement as in  $s$  (including those not enabled in  $s$  itself).
- $\text{enable}(\beta, \alpha) : \Leftrightarrow \exists s : \alpha \notin R(s) \wedge \beta \in R(s) \wedge \alpha \in R(\beta(s))$ 
  - Transition  $\beta$  may enable transition  $\alpha$ .

Every transition  $\beta$  that may be executed before any transition  $\alpha \in \text{ample}(s)$  is independent of  $\alpha$  and cannot enable  $\alpha$ ; thus it is safe to execute  $\alpha$  first and  $\beta$  later.

# Interpretation

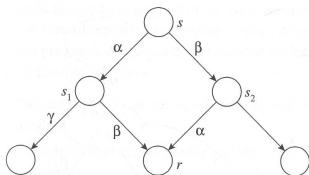
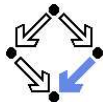


Which transitions depend on each other respectively enable each other?

- $D(\beta, \alpha)$  iff
  - $\beta$  and  $\alpha$  belong to the same process or
  - $\beta$  and  $\alpha$  share a variable that is changed by  $\beta$  or  $\alpha$ , or
  - $\beta$  and  $\alpha$  are two send statements or two receive statements on the same channel.
- $enable(\beta, \alpha)$  iff
  - $\beta$  and  $\alpha$  belong to the same process and  $\alpha$  is a possible successor statement of  $\beta$ , or
  - $\alpha$  is a statement that depends on the value of a shared variable and  $\beta$  is a statement that may change this variable, or
  - $\beta$  and  $\alpha$  are send respectively receive statements on the same channel.

For checking Condition 3, only certain pairs of transitions are of interest.

# Closing the Open Problems



Edmund Clarke: "Model Checking", 1999.

- **Problem 1:** property  $P$  may distinguish between  $s_1$  and  $s_2$ .
  - Not possible because of Condition 1 (Invisibility) which makes  $s \rightarrow s_1 \rightarrow r$  and  $s \rightarrow s_2 \rightarrow r$  stuttering equivalent.
- **Problem 2:** states  $s_1$  and  $s_2$  may have successors in addition to  $r$ .
  - Transition  $\gamma$  enabled in  $s_1$  is independent of  $\beta$  (Condition 3) and, since enabled in  $s_1$ , thus also enabled in  $r$ .
  - Since  $\beta$  is invisible (Condition 1), sequences  $s \xrightarrow{\alpha} s_1 \xrightarrow{\gamma} s'_1$  and  $s \xrightarrow{\beta} s_2 \xrightarrow{\alpha} r \xrightarrow{\gamma} r'$  are stuttering equivalent.

# Example

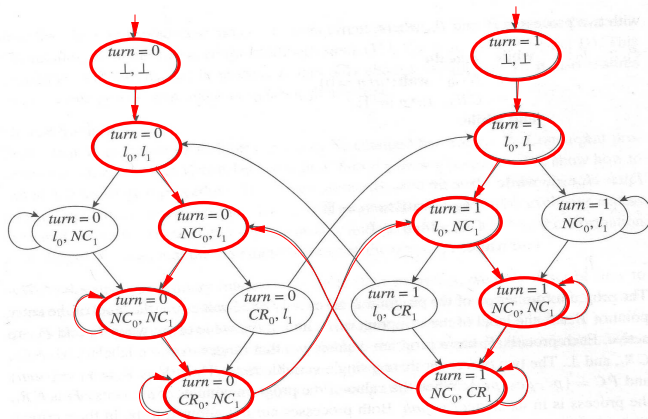
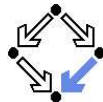


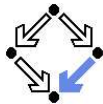
Figure 2.2  
Reachable states of Kripke structure for mutual exclusion example.

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.