T–79.4301 Parallel and Distributed Systems (4 ECTS)

T–79.4301 Rinnakkaiset ja hajautetut järjestelmät (4 op)

Lecture 2

2006.02.03

Keijo Heljanko

Keijo.Heljanko@tkk.fi

Common Flaws

Some common flaws in concurrent systems include:

- Deadly Embrace
- Circular Blocking
- Deadlock
- Starvation (livelock)
- Underspecification
- Overspecification

Deadly Embrace

A common problem in resource allocation

- Consider two callers A and B making a telephone call to each other simultaneously
- To connect a call two shared resources must be exclusively allocated: the caller's telephone line and the receiver's line
- It would be natural to use a protocol where we first allocate the caller's line and only then the receiver's line

Deadly Embrace (cnt.)

- However, if A and B call simultaneously each other, it can be the case that both A and B allocated their own lines but fail to allocate the receiver's line
- If there is no recovery mechanism in place, the system might deadlock

Deadly Embrace - Process A

In pseudocode process A might look like:

```
process A:
// Code removed
lock(line_A);
// Code removed
lock(line_B);
// Code removed
release(line_B);
// Code removed
release(line_A);
// Code removed
endprocess;
```


Deadly Embrace - Process B

In pseudocode process B might look like:

```
process B:
// Code removed
lock(line_B);
// Code removed
lock(line_A);
// Code removed
release(line_A);
// Code removed
release(line_B);
// Code removed
endprocess;
```


Deadly Embrace - Deadlock

An execution leading to a potential deadlock can be:

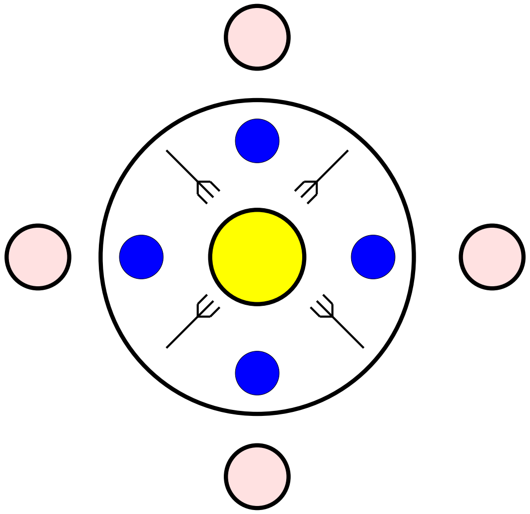
```
   Process A:        Process B:

   lock(line_A)
                     lock(line_B)

// Deadlock:
// Process A is waiting for line B
// Process B is waiting for line A
```


Circular Blocking

The deadly embrace extended over any number of processes. Classic academic example:
Dining philosophers

- There are $n \geq 2$ philosophers sitting around a round table thinking
- Because all thinking is a tough job also the philosophers need to eat
- The dish prepared for them is particularly slippery spaghetti which requires two forks to be eaten
- Unfortunately there are only n forks available, distributed one between each pair of philosophers

Philosophers



Dining Philosophers

- The philosophers have agreed on a protocol to allocate the forks:
  - Think until hungry
  - Grab left fork
  - Grab right fork
  - Eat
  - Return right fork
  - Return left fork
  - Repeat from the beginning

Dining Philosophers (cnt.)

- Assume we have four philosophers: p(0),p(1),p(2),p(3)
- The forks are: f[0],f[1],f[2],f[3]
- The fork f[0] is to left of p(0) and to the right of p(3)
- It is now easy to see that the philosophers can all starve: Can you see how?

Dining Philosophers Pseudocode

```
#define left(i) (f[(i)])
#define right(i) (f[((i)+1)%n])
process p(i):
     while true do
          think();
          lock(left(i));
          lock(right(i));
          eat();
          release(right(i));
          release(left(i));
     enddo;
endprocess;
```

Dining Philosophers - Deadlock

A deadlock execution is:

```
p(0):       p(1):        p(2):        p(3):
lock(f[0])
            lock(f[1])
                         lock(f[2])
                                      lock(f[3])
```

After this:
p(0) is waiting for p(1) to release fork f[1]
p(1) is waiting for p(2) to release fork f[2]
p(2) is waiting for p(3) to release fork f[3]
p(3) is waiting for p(0) to release fork f[0]

Dining Philosophers vs. Real Life

The aim of the dining philosophers example is to:

- Show that circular blocking chains can be arbitrarily long
- Show that the possibility of stumbling on the deadlock by a randomly picked test run is extremely small: There is exactly one deadlocking state, and an exponential (in n) number of non-deadlocking states
- Dining philosophers was too easy: Often locking problems are much harder to spot just because the programs are larger and the locking is less structured

Deadlock

- Deadlocks are a common problem in distributed systems.
- As seen before deadlocks can occur from the use of blocking lock primitives.
- In a message passing system deadlocks might occur due to processes waiting for messages from one another in similar manner as processes are waiting for other processes to release locks
- Mixing priority based scheduling with locking is also known to easily lead to deadlocks

Starvation (livelock)

- Starvation (livelock) is a different problem. In it a part of the system is live and executing but other parts of the system are blocked indefinitely.
- Example: High priority process using a busy wait (spinlock) to wait for a low priority process to release a lock in an OS kernel. However, the low priority process is never given CPU time because the scheduler always picks the highest priority runnable task to be executed.

Deadlock and starvation will be treated more formally later.

Under- and Overspecification

Examples in a message passing (data-communications protocol) setting:

- Underspecification: A message arrives in a protocol implementation and there is no code to handle it (unexpected reception)
- Overspecification: There is code in a protocol implementation to cope with the reception of messages which are not possible in the protocol (dead code)

Non-concurrency Bugs

Of course your standard set of normal bugs not related to concurrency applies

- Incorrect control flow
- Incorrect data manipulation
- Wrong assumptions about the environment
- Null pointer exceptions
- Uninitialised data
- Array out of bounds errors
- Memory management problems (e.g., leaks, accessing freed memory)

Automata Theoretic Approach

A short theory of model checking using automata

- Assume you have a finite state automaton (FSA) of the behaviour of the system A (see Lecture 1 automaton A_M for an example)
- Assume the specified property is also specified with an FSA S
- Now the system fulfils the specification, if the language of the system is contained in the language of the specification: i.e., it holds that $L(A) \subseteq L(S)$

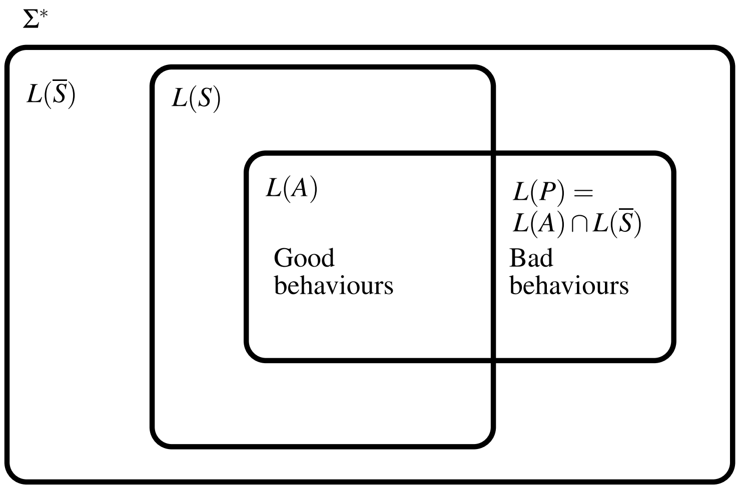
Automata Theoretic Approach (cnt.)

- If you have studied the course: "T-79.1001 Introduction to Theoretical Computer Science T" or one of its predecessors well, you know how to proceed:
- We need to generate the product automaton: $P = A \cap \overline{S}$, where $\overline{S}$ is an automaton which accepts the complement language of $L(S)$

Automata Theoretic Approach (cnt.)

- If $L(P) = \emptyset$, i.e., P does not accept any word, then the property holds and thus the system is correct
- Otherwise, there is some run of P which violates the specification, and we can generate a counterexample execution of the system from it (more on this later)

Language Inclusions



Finite State Automata

- Finite state automata (FSA) can be used to model finite state systems, as well as specifications for systems.
- In this course they form the theoretical foundations of analysis algorithms
- Next we recall and adapt automata theory from previous courses
- The classes of automata will later be extended with features such as variables and message queues to make them more suitable for protocol modelling

Finite State Automaton

Definition 1 *A (nondeterministic finite) automaton A* is a tuple $(\Sigma, S, S^0, \Delta, F)$, where

- Σ is a finite *alphabet*,
- S is a finite set of *states*,
- $S^0 \subseteq S$ is the set of *initial states*,
- $\Delta \subseteq S \times \Sigma \times S$ is the *transition relation* (no ε-transitions allowed), and
- $F \subseteq S$ is the set of *accepting states*.

Deterministic Automata (DFA)

An automaton A is *deterministic* (DFA) if $|S^0| = 1$ and for all pairs $s \in S, a \in \Sigma$ it holds that if for some $s' \in S$: $(s,a,s') \in \Delta$ then there is no $s'' \in S$ such that $s'' \neq s'$ and $(s,a,s'') \in \Delta$.

(I.e., there is only at most one state which can be reached from s with a.)

Transition Relation

- The meaning of the transition relation $\Delta \subseteq S \times \Sigma \times S$ is the following: $(s,a,s') \in \Delta$ means that there is a move from state s to state s' with symbol a.
- An alternative (equivalent) definition gives the transition relation as a function $\rho : S \times \Sigma \rightarrow 2^S$, where $\rho(s,a)$ gives the set of states to which the automaton can move with a from state s.

Synonyms for FSA

Synonyms for the word automaton are: finite state machine (FSM), finite state automaton (FSA), nondeterministic finite automaton (NFA), and finite automaton on finite strings/words.

Runs

A finite automaton A accepts a set of words $L(A) \subseteq \Sigma^*$ called the *language* accepted by A, defined as follows:

- A *run* r of A on a finite word $a_0, \ldots, a_{n-1} \in \Sigma^*$ is a sequence $s_0, \ldots, s_n$ of $(n+1)$ states in S, such that $s_0 \in S^0$, and $(s_i, a_i, s_{i+1}) \in \Delta$ for all $0 \leq i < n$.
- The run r is *accepting* iff $s_n \in F$. A word $w \in \Sigma^*$ is accepted by A iff A has an accepting run on w.

Languages

- The language of A, denoted $L(A) \subseteq \Sigma^*$ is the set of finite words accepted by A.
- A language of automaton A is said to be *empty* when $L(A) = \emptyset$.

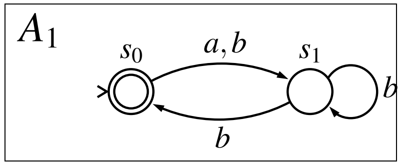
Boolean Operations with Automata

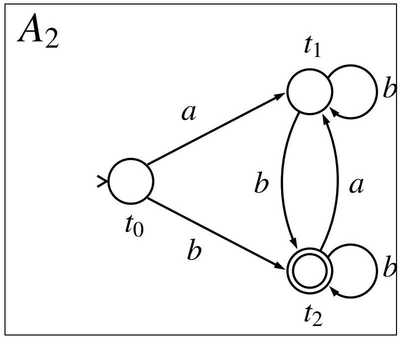
Let us now recall basic operations with finite state automata.

- We will do this by defining the Boolean operators for finite automata:
  $A = A_1 \cup A_2, A = A_1 \cap A_2$, and $A = \overline{A_1}$.
- These operations will as a result have an automaton A, such that:
  $L(A) = L(A_1) \cup L(A_2), L(A) = L(A_1) \cap L(A_2)$, and
  $L(A) = (\Sigma^* \setminus L(A_1)) = \overline{L(A_1)}$, respectively.

Example: Operations on Automata

As a running example we will use the following automata A_1 and A_2, both over the alphabet $\Sigma = \{a, b\}$. We draw boxes around automata to show which parts belong to which.





$A = A_1 \cup A_2$

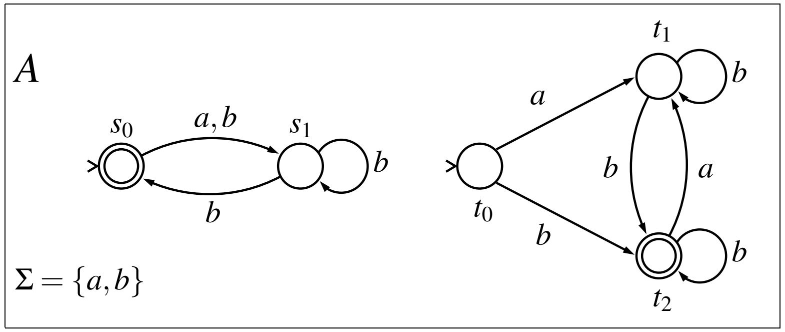
Definition 2 Let $A_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ and $A_2 = (\Sigma, S_2, S_2^0, \Delta_2, F_2)$. We define the *union* automaton to be $A = (\Sigma, S, S^0, \Delta, F)$, where:

- $S = S_1 \cup S_2$,
- $S^0 = S_1^0 \cup S_2^0$
  (Note: no ε-moves but several initial states instead),
- $\Delta = \Delta_1 \cup \Delta_2$, and
- $F = F_1 \cup F_2$.

We have $L(A) = L(A_1) \cup L(A_2)$.

Example: Union of Automata

The following automaton A is the union, $A = A_1 \cup A_2$.



$A = A_1 \cap A_2$

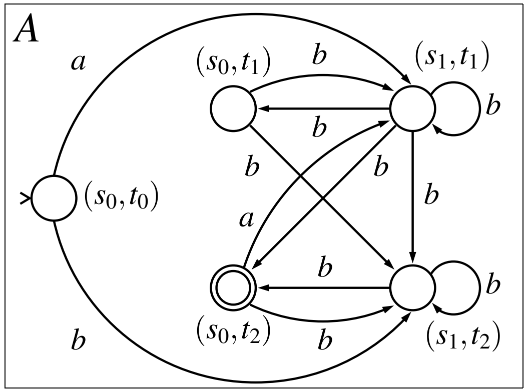
Definition 3 Let $A_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ and $A_2 = (\Sigma, S_2, S_2^0, \Delta_2, F_2)$. We define the *product* automaton to be $A = (\Sigma, S, S^0, \Delta, F)$, where:

- $S = S_1 \times S_2$,
- $S^0 = S_1^0 \times S_2^0$,
- for all $s, s' \in S_1, t, t' \in S_2, a \in \Sigma$: $((s,t), a, (s',t')) \in \Delta$ iff $(s,a,s') \in \Delta_1$ and $(t,a,t') \in \Delta_2$; and
- $F = F_1 \times F_2$.

We have $L(A) = L(A_1) \cap L(A_2)$.

Example: Intersection of Automata

The following automaton A is the intersection (product) $A = A_1 \cap A_2$.



Complementation

The definition of complementation is slightly more complicated.

- We say that an automaton has a *completely specified transition relation* if for all states $s \in S$ and symbols $a \in \Sigma$ there exist a state $s' \in S$ such that $(s, a, s') \in \Delta$.

Completely Specified Automata

Any automaton which does not have a completely specified transition relation can be turned into one by:

- adding a new *sink state* q_s,
- making q_s a non-accepting state,
- adding for all $a \in \Sigma$ an arc (q_s, a, q_s), and
- for all pairs $s \in S, a \in \Sigma$: if there is no state s' such that $(s, a, s') \in \Delta$, then add an arc (s, a, q_s).
  (Add all those arcs which are still missing to fulfil the completely specified property.)

Complementing DFAs

- Note that this construction does not change the language accepted by the automaton.
- We first give a complementation definition which *only works for completely specified deterministic automata*!

Complementing DFAs (cnt.)

Definition 4 Let $A_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ be a *deterministic* automaton with a completely specified transition relation. We define the *deterministic complement* automaton to be $A = (\Sigma, S, S^0, \Delta, F)$, where:

- $S = S_1$,
- $S^0 = S_1^0$,
- $\Delta = \Delta_1$, and
- $F = S_1 \setminus F_1$ ("flip the acceptance bit").

We have $L(A) = (\Sigma^* \setminus L(A_1))$.

Complementing NFAs

- The operations we have defined for finite state automata so far have resulted in automata whose size is polynomial in the sizes of input automata.
- The most straightforward way of implementing complementation of a non-deterministic automaton is to first determinize it, and after this to complement the corresponding deterministic automaton.

Complementing NFAs (cnt.)

- Unfortunately determinization yields an exponential blow-up. (A worst-case exponential blow-up is in fact unavoidable in complementing non-deterministic automata.)

Determinization

Definition 5 Let $A_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ be a non-deterministic automaton. We define a deterministic automaton $A = (\Sigma, S, S^0, \Delta, F)$, where

- $S = 2^{S_1}$, the set of all sets of states in S_1,
- $S^0 = \{S_1^0\}$, a single state containing all the initial states of A_1,
- $(Q, a, Q') \in \Delta$ iff $Q \in S, a \in \Sigma$, and $Q' = \{s' \in S_1 \mid \text{ there is } (s, a, s') \in \Delta_1 \text{ such that } s \in Q\}$; and
- $F = \{s \in S \mid s \cap F_1 \neq \emptyset\}$, those states in S which contain at least one accepting state of A_1.

Determinization (cnt.)

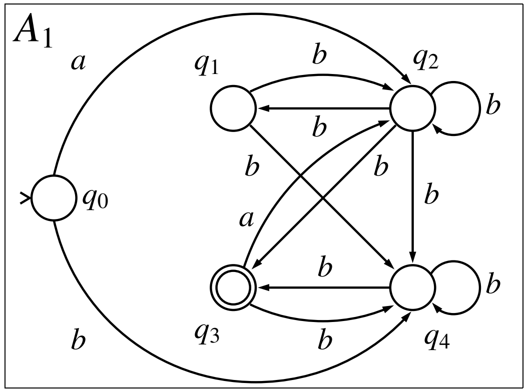
- The intuition behind the construction is that it combines all possible runs on given input word into one run, where we keep track of all the possible states we can currently be in by using the “state label”.
  (The automaton state consists of the set of states in which the automaton can be in after reading the input so far.)
- We denote the construction of the previous slide with $A = det(A_1)$ Note that $L(A) = L(A_1)$, and A is deterministic. If A_1 has n states, the automaton A will contain 2^n states.

Determinization (cnt.)

- Note also that the determinization construction gives an automaton A with a completely specified transition relation as output. Thus to complement an automaton A_1, we can use the procedure $A' = det(A_1)$, $A = \overline{A'}$, and we get that $L(A) = \Sigma^* \setminus L(A') = \Sigma^* \setminus L(A_1) = \overline{L(A_1)}$.
- To optimise the construction slightly, usually only those states of A which are reachable from the initial state are added to set of states set of A.
- One can also use the classical DFA minimisation algorithm to reduce the size of the result further

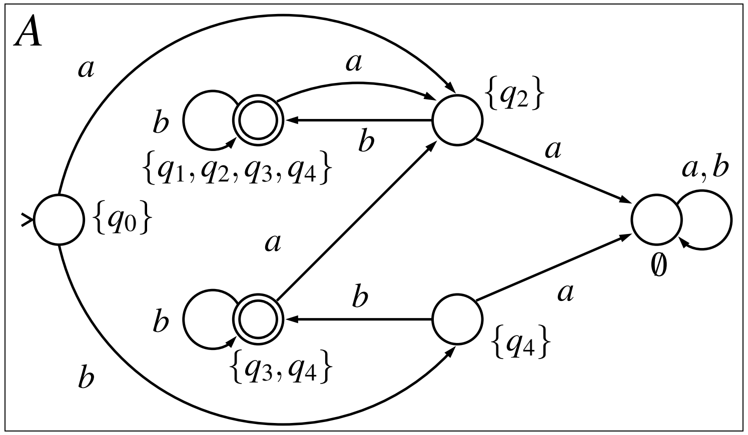
Example: Determinization

We want to determinize the following automaton A_1 over the alphabet $\Sigma = \{a, b\}$.



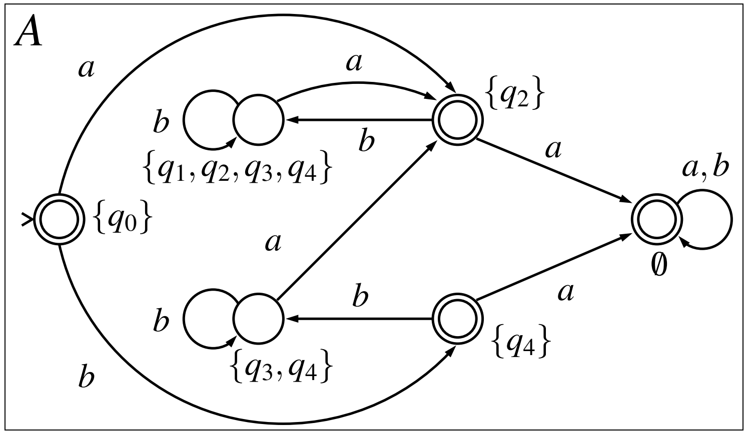
Example: Determinization Result

As a result we obtain the automaton A below. (Only the reachable part shown!)



Example: Complementation

Let's call the result of the previous slide A_1, and complement the result. We get:



Boolean Operations

- We have now shown that finite state automata are closed under all Boolean operations, as with $\cup$, $\cap$, and $\overline{A}$ all other Boolean operations can be done.
- All operations except for determinization (which is also used to complement nondeterministic automata!) created a polynomial size output in the size of the inputs.

State Explosion from Intersection

- Note, however, that even if A_1, A_2, A_3, A_4 have k states each, the automaton $A'_4 = A_1 \cap A_2 \cap A_3 \cap A_4$ (sometimes alternatively called the synchronous product and denoted $A'_4 = A_1 \times A_2 \times A_3 \times A_4$) can have k^4 states, and thus in the general A'_i will have k^i states.
- Therefore even if a single use of $\cap$ is polynomial, repeated applications often will result in a state explosion problem.
- In fact, the use of $\times$ as demonstrated above could in principle be used to compose the behaviour of a parallel system from its components.

Checking Safety Properties with FSA

- A safety property can be informally described as a property stating that “nothing bad should happen”. (We will come back to the formal definition later in the course.)
- When checking safety properties, the behaviour of a system can be described by a finite state automaton, call it A.
- Also in the allowed behaviours of the system can be specified by another automaton, call it the specification automaton S.

Checking Safety (cnt.)

- Assume that the specification specifies all legal behaviours of the system. In other words a system is incorrect if it has some behaviour (accepts a word) that is not accepted by the specification. In other words a correct implementation has less behaviour than the specification, or more formally $L(A) \subseteq L(S)$.

Language Containment

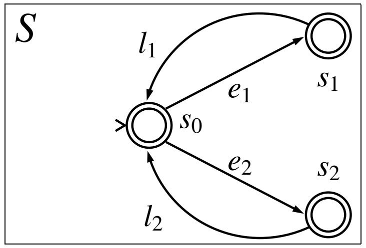
- Checking whether $L(A) \subseteq L(S)$ holds is referred to as performing a language containment check.
- Recall: By using simple automata theoretic constructions given above, we can now check whether the system meets its specification. Namely, we can create a product automaton $\mathcal{P} = A \cap \overline{S}$ and then check whether $L(P) = \emptyset$.
- In case the safety property does *not* hold, the automaton P has a counterexample run r_p which accepts a word w, such that $w \in L(A)$ but $w \notin L(S)$.

Creating the Counterexample

- By projecting r_p on the states of A one can obtain a run of r_a of the system (a sequence of states of the system) which violates the specification S.

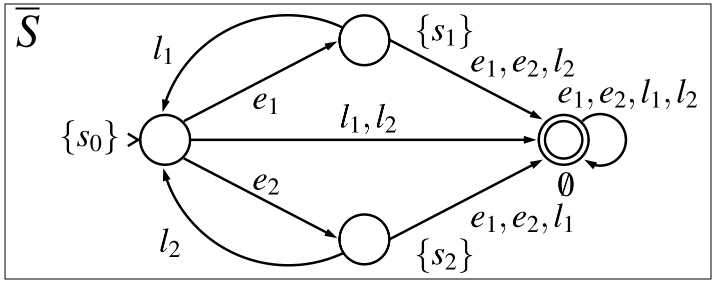
Example: Safety Property

- Consider the problem of mutual exclusion. Assume that the alphabet is $\Sigma = \{e_1, e_2, l_1, l_2\}$, where e_1 means that process 1 enters the critical section and l_1 means that process 1 leaves the critical section.
- The automaton S specifying correct mutual exclusion property is the following.



Example: Safety Property (cnt.)

If we want to check whether $L(A) \subseteq L(S)$, we need to complement S. We get the following:



Example: Safety Property (cnt.)

If we now have an automaton A modelling the behaviour of the mutex system, we can create the product automaton $P = A \cap \overline{det(S)}$. Now the mutex system is correct iff the automaton P does not accept any word.