T–79.4301 Parallel and Distributed Systems (4 ECTS)

Lecture 5

16th of October 2008

Keijo Heljanko

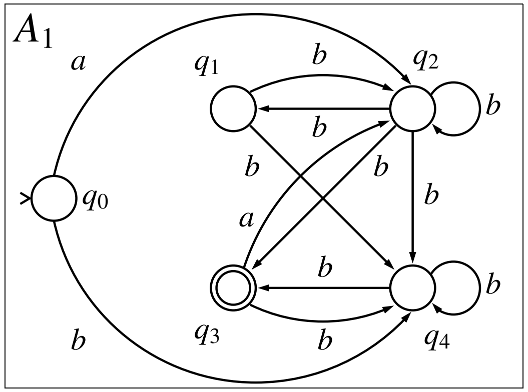
`Keijo.Heljanko@tkk.fi`

Home Exercise 1

- The home exercise 1 is now available through the course Noppa page:
  `https://noppa.tkk.fi/noppa/kurssi/t-79.4301/etusivu`
- The exercise is to be done individually, and the topic is modelling an elevator controller in Promela and verifying some safety properties of it with Spin
- The deadline is on Thursday 6th of November at 12:15
- The deadline is strict!

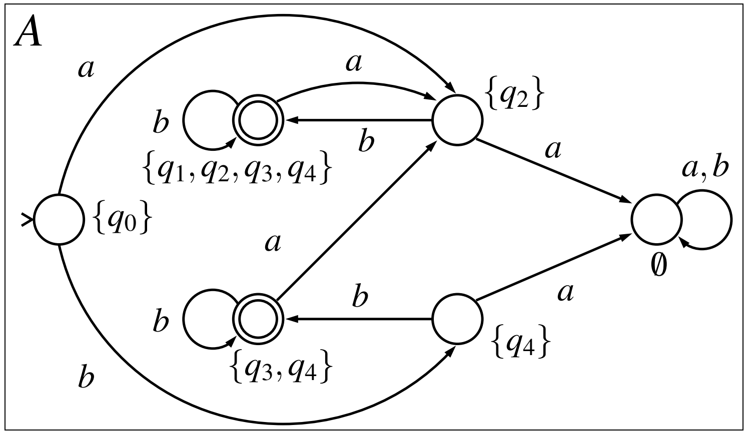
Example: Determinization

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



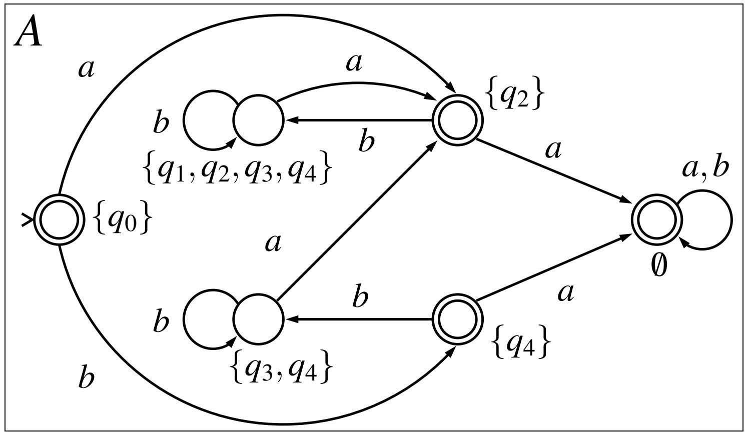
Example: Determinization Result

As a result we obtain the automaton A below. (In this course it always suffices to only consider the part reachable from the initial state!)



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 behavior 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 behavior of a system can be described by a finite state automaton, call it A.
- Also in the allowed behaviors 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 behaviors of the system. In other words a system is incorrect if it has some behavior (accepts a word) that is not accepted by the specification. In other words a correct implementation has less behavior than the specification, or more formally $L(A) \subseteq L(S)$.

Language Containment

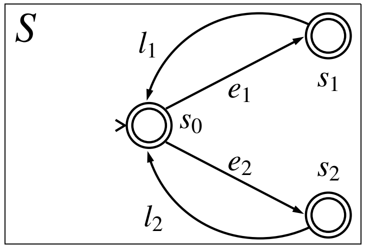
- Checking whether $L(A) \subseteq L(S)$ holds is referred to as performing a language containment check.
- Recall: By using simple automata theoretic constructions given above, we can now check whether the system meets its specification. Namely, we can create a product automaton $\mathcal{P} = A \cap \overline{S}$ 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.

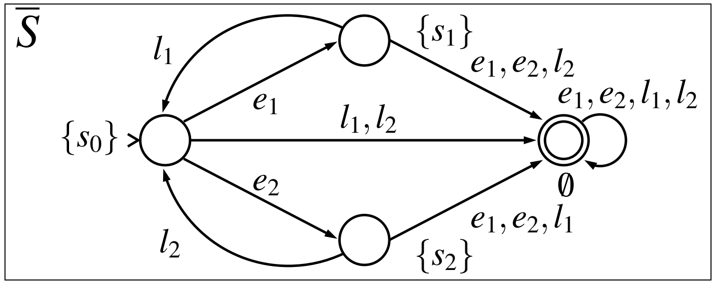
Example: Safety Property

- 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 behavior 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.

Labeled Transition System (LTS)

- Labeled transition system (LTS) is a variant of the finite state automaton (FSA) model better suited for modelling asynchronous systems (software)
- They are a very simple model of concurrency and as such they are simple to understand and there are very few variants
- We will use them in the course to demonstrate concurrency related phenomena
- The simplicity of model is intentional in order not to focus too much on the modelling language but on the concurrency related phenomenon at hand

LTSs (cnt.)

- Because LTSs are so simple, modelling with them can be cumbersome. We will later show how the LTS model can be extended with features to make modeling with them closer to Promela
- Promela models also have all the same concurrency phenomena as LTS based models
- We will start introducing LTSs by recalling the definition of finite state automata

Finite State Automaton (recap)

Recall the definition of FSA from Lecture 4:

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*.

Labeled Transition System (LTS)

Definition 2 *A labeled transition system L* is a tuple (Σ, S, s^0, Δ), where

- Σ is a finite *alphabet* not containing the symbol τ,
- S is a finite set of *states*,
- $S^0 = \{s^0\}$ where $s^0 \in S$ is the *initial state*, and
- $\Delta \subseteq S \times \Sigma \cup \{\tau\} \times S$ is the *transition relation* (containing also τ-transitions).

LTS vs. FSA

Changes:

- A new special symbol τ (“tau”), denoting an internal action (also called the invisible action)
- The alphabet Σ now specifies those visible actions on which the LTS can synchronize with other LTSs
- A single initial state s^0
- The transition relation also includes τ-transitions internal to the component (these are almost but not quite the same as ε-moves in some FSA models)
- No final states (think of all the states being final)

LTS vs. FSA (cnt.)

Why LTSs instead of FSAs?

- FSA based models are more natural for synchronous systems such as hardware, while LTS based models are more natural for asynchronous systems such as concurrent software
- The main difference is the parallel composition operator $||$ (also called the asynchronous product) is used to compose a system out of its components: $L = L_1 || L_2 || \cdots || L_n$ instead of using the synchronous product (also called the intersection $\cap$): $A = A_1 \times A_2 \times \cdots \times A_n$.

Basic LTS Notation

Let $L = (\Sigma, S, S^0, \Delta)$ be an LTS, $s, s' \in S$, $s_0, s_1, \ldots s_n \in S$, $x_1, x_2, \ldots x_n \in \Sigma \cup \{\tau\}$. We define:

- $s \xrightarrow{x} s'$ iff $(s, x, s') \in \Delta$
- $s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \cdots \xrightarrow{x_n} s_n$ iff for all $1 \leq i \leq n$: $s_{i-1} \xrightarrow{x_i} s_i$
- $s \xrightarrow{x_1 x_2, \ldots, x_n} s'$ iff there are some $s_0, s_1, \ldots, s_n$ such that $s_0 = s$, $s_n = s'$, and $s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \cdots \xrightarrow{x_n} s_n$

Basic LTS Notation (cnt.)

- $s \longrightarrow s'$ iff for some $\sigma \in (\Sigma \cup \{\tau\})^*$ it holds that $s \xrightarrow{\sigma} s'$
- $s \longrightarrow$ iff for some s' it holds that $s \longrightarrow s'$

Basic LTS Notation (cnt.)

- $s \overset{a}{\Rightarrow} s'$ iff there is $a \in \Sigma$ and $s_0, s_1, s_2, s_3 \in S$ such that $s_0 = s$, $s_3 = s'$, and $s_0 \xrightarrow{\tau^*} s_1 \xrightarrow{a} s_2 \xrightarrow{\tau^*} s_3$
- $s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} s_2 \overset{a_3}{\Rightarrow} \cdots \overset{a_n}{\Rightarrow} s_n$ iff for all $1 \leq i \leq n$: $a_i \in \Sigma$ and $s_{i-1} \overset{a_i}{\Rightarrow} s_i$
- $s \xRightarrow{a_1 a_2, \ldots, a_n} s'$ iff there are some $s_0, s_1, \ldots, s_n$ such that $s_0 = s$, $s_n = s'$, $a_i \in \Sigma$, and $s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} s_2 \overset{a_3}{\Rightarrow} \cdots \overset{a_n}{\Rightarrow} s_n$
- $s \Rightarrow s'$ iff for some $\sigma \in \Sigma^*$ it holds that $s \overset{\sigma}{\Rightarrow} s'$

Basic LTS Notation (cnt.)

- $L \longrightarrow$ iff for s^0 it holds that $s^0 \longrightarrow$

Parallel Composition ||

Let's now create an LTS $L = (\Sigma, S, S^0, \Delta)$ by composing n LTSs:
$L_1 = (\Sigma_1, S_1, S_1^0, \Delta_1)$,
$L_2 = (\Sigma_2, S_2, S_2^0, \Delta_2)$,
$\ldots$,
$L_n = (\Sigma_n, S_n, S_n^0, \Delta_n)$
in parallel:

$L = L_1 || L_2 || \cdots || L_n$

Parallel Composition || (cnt.)

The intuition:

- Pick an initial state from each LTS
- Any process can do a τ-transition on its own, and others remain in their current states during its execution
- If a is in the alphabet for several LTSs, all of them must be able to perform it before it can be executed
  - When executing a, all LTSs with a in their alphabet move, while all other LTSs remain in their current states

Definition of ||

Definition 3 *Parallel composition* $L = L_1 || L_2 || \cdots || L_n$ is an LTS (Σ, S, S^0, Δ), where

- $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \cdots \cup \Sigma_n$,
- $S = S_1 \times S_2 \times \cdots \times S_n$
  (states of the parallel composition are tuples $s = (s_1, s_2, \ldots, s_n)$),
- $S^0 = \{(s_1^0, s_2^0, \cdots, s_n^0)\}$
  (a single initial state where each component LTSs L_i is in its initial state), and
- $\Delta \subseteq S \times \Sigma \cup \{\tau\} \times S$ is the *transition relation*, where:

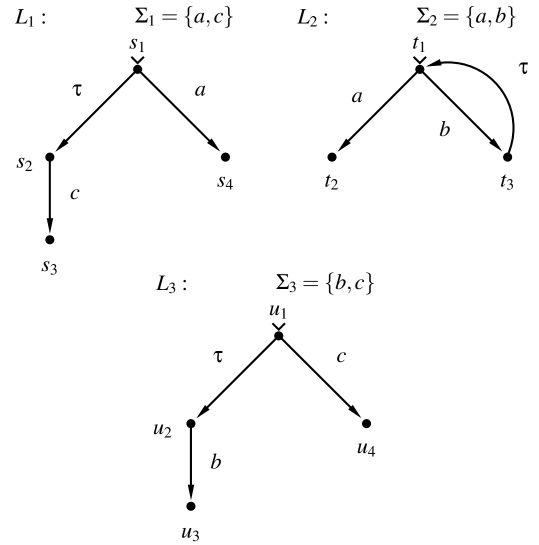
Definition of || (cnt.)

- $(s, x, s') \in \Delta$ where
  $s = (s_1, s_2, \ldots, s_n)$,
  $x \in \Sigma \cup \{\tau\}$, and
  $s' = (s'_1, s'_2, \ldots, s'_n)$ iff:
  - $x = \tau$: there is $1 \leq i \leq n$ such that
    $(s_i, \tau, s'_i) \in \Delta_i$ and
    $s'_j = s_j$ for all $1 \leq j \leq n$, when $j \neq i$.
  - $x \neq \tau$: for every $1 \leq i \leq n$:
    $(s_i, x, s'_i) \in \Delta_i$, when $x \in \Sigma_i$ and
    $s'_i = s_i$, when $x \notin \Sigma_i$.

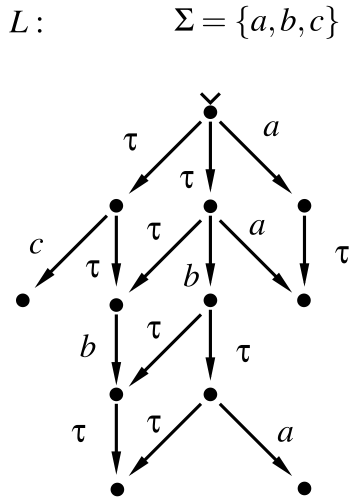
Example: Parallel Composition

- Compute the parallel composition $L = L_1 || L_2 || L_3$, where the LTSs L_1, L_2, and L_3 are given on the next slide

Example: Parallel Composition (cnt.)



Example: Result $L = L_1 || L_2 || L_3$



Reachability Analysis

- Reachability analysis is a way to implement model checking
- We have now shown how parallel composition of LTSs is done directly based on the definition
- Most model checking algorithms are based on an algorithm which implements the generation of a graph containing all the reachable global states of the system
- Let's now give this algorithm in an abstract setting, independent of the used model of concurrency: Thus the algorithm works for, e.g., the parallel composition of LTSs or a Promela

Reachability Graph

- We want to generate a graph $G = (V, T, E, v^0)$, where
- V is the set of reachable global states of the system,
- T is the set of executable global transitions of the system,
- $E \subseteq V \times T \times V$ is the set of executable global state changes of the system (arcs/edges of the reachability graph), and
- $v^0 \in V$ is the initial global state of the system.

Reachability Graph: Subroutines

- We need the following subroutines:
  - `enabled(v)`: Given a global state `v` it returns the list of all global transitions `t` which are enabled in `v`
  - `v' = fire(v,t)`: Given a global state `v`, and a global transition `t` which is enabled at `v`, it returns the global state `v'` reached from `v` by firing `t`

Reachability Graph Algorithm (part 1)

```
graph RG;  /* Global - empty reachability graph */


reachability_graph(state v_0) {

    RG.init();             /* Initialize data structures  */
    RG.add_node(v_0);      /* Add initial state to the RG */
    RG.mark_initial(v_0);  /* Mark the initial state      */
    search(v_0);           /* Process initial state       */

    /* RG now contains the reachability graph */
}
```


Reachability Graph Algorithm (part 2)

```
search(state v) {
    state v';
    transition t;
    forall t in enabled(v) {
        /* Optionally add here: code to add t to T */
        v' = fire(v,t); /* firing t at v results in v' */
        if !RG.has_node(v') { /* v' already processed? */
            RG.add_node(v');  /* Add new state v' to V */
            search(v');       /* Process v'            */
        }
        RG.add_edge(v,t,v');  /* Add arc (v,t,v') to E */
    }
}
```

Implementation Issues

- Modern model checkers such as Spin can handle reachability graphs with the number of reachable states in tens of millions
- The most time and memory critical routines are `RG.has_node(v')` and `RG.add_node(v')`
- Usually the state storage inside model checker is very carefully engineered to minimize memory usage
- In more complex system models the routine `enabled(v)` can become the bottleneck
- In many cases the line `RG.add_edge(v,t,v')` can be removed if only state properties are of interest. Also, usually `enabled(v)` can be recomputed at will

Implementation Issues (cnt.)

- The algorithm presented is depth-first search (DFS), which is the default in Spin
- Also breadth-first search (BFS) is often implemented as it guarantees shortest paths to assertion failure states
- If the set of nodes is too large to fit in the memory, database techniques (B-trees etc.) can be used to implement `RG.has_node(v')` and `RG.add_node(v')`. However, this slows down search by several orders of magnitude.