T–79.4301 Parallel and Distributed Systems (4 ECTS)

Lecture 4

9th of October 2008

Keijo Heljanko

Keijo.Heljanko@tkk.fi

Spin and XSpin Installed

- The `spin` and `xspin` binaries are installed to the computing centre Linux workstations:
  `http://www.tkk.fi/atk/luokat/computernames.html`
- Basically you need to add the directory:
  `/p/edu/t-79.4301/bin`
  to your executable search path

Installing Yourself

- Optionally, installing Spin to your own machine is also pretty straightforward, just follow the instructions for (Unix(Linux)/Windows/Mac) at:
  `http://spinroot.com/spin/Man/README.html`
- Hint for Linux users:
  The first three line of the `xspin` script need for Linux to be replaced with:

```
#!/usr/bin/wish -f
# the next line restarts using wish \
#exec wish c:/cygwin/bin/xspin -- $*
```


Advanced Promela - `d_step`

Similar, more advanced version of atomic, example:

```
d_step { /* Swap values of a and b */
   tmp = b;
   b = a;
   a = tmp
}
```

Advanced Promela - d_step (cnt.)

- Differences to atomic
  - May not contain non-determinism (deterministic step)
  - It is a runtime error if some statement inside d_step blocks
  - The states reached inside a d_step sequence do not exists in the statespace of the system, only the last state reached by the execution does
  - No goto's in or out of a d_step
  - d_step can exists inside an atomic sequence but not vice versa

Example: atomic vs. d_step

```
byte a[12];
init {
  int i = 0;
  d_step { /* d_step is a slight winner here. */
    do
      :: (i < 12) -> a[i] = (i*5)+2; i++;
      :: else -> break;
    od;
    i = 0; /* zero i to avoid introducing new states */
  };
  atomic { /* Run might block, better use atomic.*/
     run foo(); run bar();};  /* atomic startup. */
}
```


No atomic vs. atomic vs. d_step

```
byte x,y;
/* Compare the state-spaces of: */
/* Non-atomic */
active proctype P1() { x++; x++; x++}
active proctype P2() { y++; y++; y++}
/* P1 atomic */
active proctype P1() { atomic {x++; x++; x++} }
active proctype P2() { y++; y++; y++}
/* P1 d_step */
active proctype P1() { d_step {x++; x++; x++} }
active proctype P2() { y++; y++; y++}
```


atomic vs. d_step

- The use of atomic sequences might sometimes be necessary to model a feature of the system (e.g, atomic swap of two variables implemented in HW)
- Their use often allows for more efficient analysis of models
- Rule of thumb: When in doubt, use atomic, it is harder to shoot to your own foot with it
- d_step is handy for internal computation, e.g., to initialize some arrays
- Misuse of atomic and d_step (overuse) might hide the concurrency bugs you are looking for, be careful!

Example: Check for Blocking

You can check that in your models statements inside atomic are never blocked by:

```
/* Add a new variable */
bit aflag;
/* Change each atomic block:                       */
/* atomic { foo; bar; baz;}                        */
/* to:                                             */
/* atomic { foo; aflag=1; bar; baz; aflag=0;} */
/* Add an atomicity observer:                      */
active proctype aflag_monitor {
  assert(!aflag);
}
```


A Word of Warning

- The exact semantics of `atomic` and `d_step` are very involved, see:
  - The Spin Model Checker - Primer and Reference Manual
- Features which interact with `atomic` and `d_step` in "interesting" ways are (try to avoid unless you really really know what you are doing):
  - `goto`'s in and out of atomic sequences
  - Combining rendezvous and `atomic` or `d_step` in various ways
  - Complex loops inside `atomic` or `d_step` (the model checker might get stuck there!)

The Promela `timeout`

- The Promela `timeout` statement becomes executable if there is no process in the system which would be otherwise executable
- Models a global timeout mechanism
- Can be dangerous to use in modelling, as it provides an escape from deadlock states - it is easy to hide real concurrency problems (unwanted deadlocks) by using it
- Timeouts can often be alternatively modelled by just using the `skip` keyword in place of the `timeout`

Macros

Promela uses the C-language preprocessor to preprocess Promela models. Things you can do with it are e.g.,:

```
/* Constants */
#define CHANNEL_CAPACITY 3

/* Macros */
#define RESET_VARS(x)   \
    d_step { x[0] = 0; \
             x[1] = 0; \
             x[2] = 0; }
```

Macros (cnt.)

```
/* Make models conditional */
#define FOO 1

#ifdef FOO
/* Case FOO */
#endif

#ifndef FOO
/* Case not FOO */
#endif

/* Use skip to model timeouts */
#define timeout skip
```


inline - Poor Man's Procedures

Promela also has its own macro-expansion feature called `inline`. It basically works by exactly the same textual replacement mechanism as C macro expansion.

```
inline example(x, y) {
        y = a;
        x = b;
        assert(x)
}
init {
        int a, b;

        example(a,b)
}
```


inline (cnt.)

When using `inline` keep in mind that

- Promela only has two scopes: global and process local
- Thus all variables should be declared outside the `inline`
- `inline` cannot be used as an expression
- Use `spin -I` to debug problems with inline definitions (it shows the inlines extended)

Advanced Modelling Tips

If you want to know more, the following papers contain advanced Promela modelling tips:

- Theo C. Ruys: SPIN Tutorial: how to become a SPIN Doctor, In Proceedings of the 9th SPIN Workshop, LNCS 2318, pp. 6–13, 2002. Available from: `http://spinroot.com/spin/Workshops/ws02/ruys_abs.pdf`
- Theo C. Ruys: Low-Fat Recipes for SPIN, In Proceedings of the 7th SPIN Workshop, LNCS 1885, pp. 287–321, 2000. Available from: `http://spinroot.com/spin/Workshops/ws00/18850290.pdf`

Automata Theoretic Approach

A short theory of model checking using automata

- Assume you have a finite state automaton (FSA) of the behavior 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 $L(A) \subseteq L(S)$

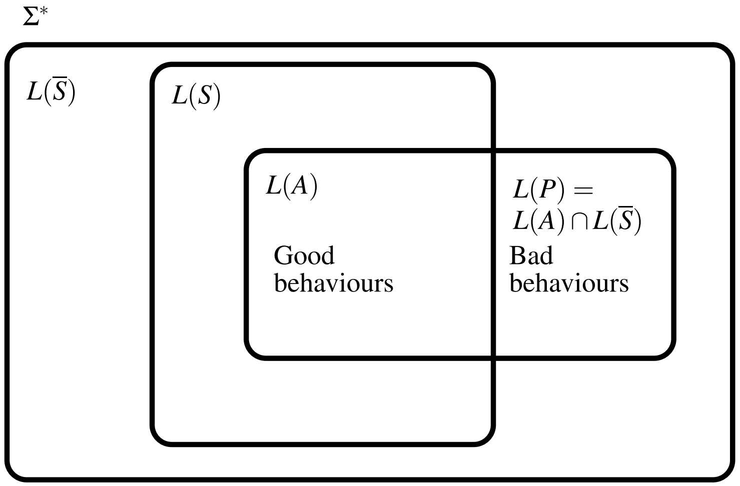
Automata Theoretic Approach (cnt.)

- 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 model checking 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 $L(A) = \emptyset$.

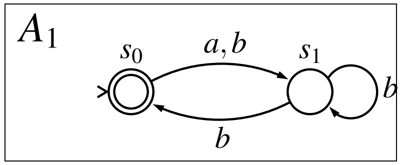
Boolean Operations with Automata

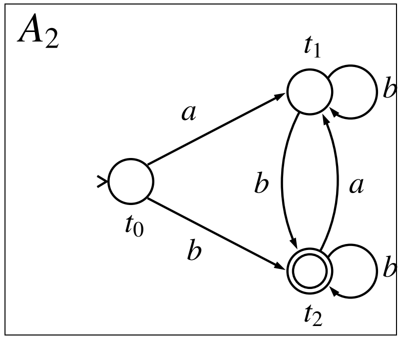
Let us now recall basic operations with finite state automata.

- 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.





$A = A_1 \cup A_2$

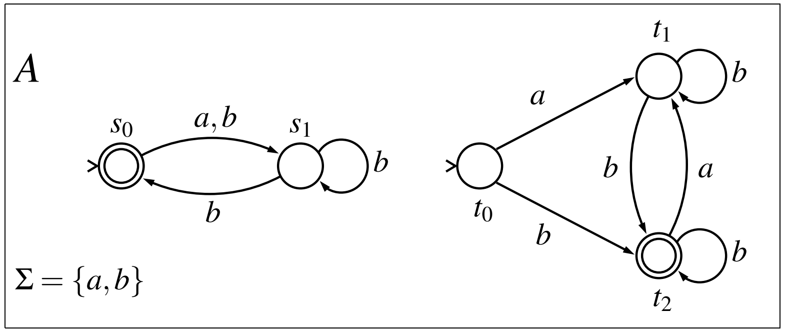
Definition 2 Let $A_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ and $A_2 = (\Sigma, S_2, S_2^0, \Delta_2, F_2)$. We define the *union* automaton to be $A = (\Sigma, S, S^0, \Delta, F)$, where:

- $S = S_1 \cup S_2$,
- $S^0 = S_1^0 \cup S_2^0$
  (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, $A = A_1 \cup A_2$.



$A = A_1 \cap A_2$

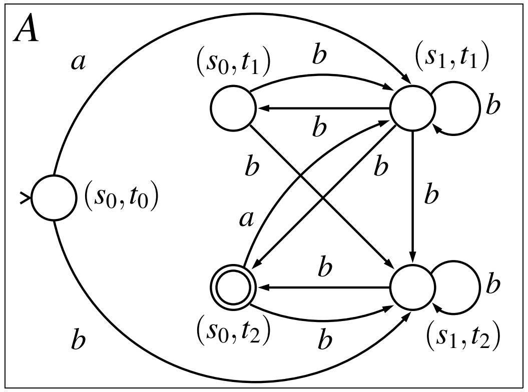
Definition 3 Let $A_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ and $A_2 = (\Sigma, S_2, S_2^0, \Delta_2, F_2)$. We define the *product* automaton to be $A = (\Sigma, S, S^0, \Delta, F)$, where:

- $S = S_1 \times S_2$,
- $S^0 = S_1^0 \times S_2^0$,
- 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 optimize 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 minimization algorithm to reduce the size of the result further.