T–79.4301 Parallel and Distributed Systems (4 ECTS)

T–79.4301 Rinnakkaiset ja hajautetut järjestelmät (4 op)

Lecture 4

2006.02.17

Keijo Heljanko

Keijo.Heljanko@tkk.fi

HELSINKI UNIVERSITY OF TECHNOLOGY
Laboratory for Theoretical Computer Science

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

- See the course homepage for more detailed instructions for different shells:

  `http://www.tcs.tkk.fi/Studies/T-79.4301/`

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

Like the previous Lecture, this part is based on a nice Spin Beginners' Tutorial by Theo C. Ruys:
`http://spinroot.com/spin/Doc/SpinTutorial.pdf`
and The Spin Model Checker - Primer and Reference Manual

- Promela is somewhat like the C language - very powerful but at the same time hard to fully master
- In the following we discuss more advanced modelling features of Promela

Alternative Send/Receive Syntax

- Alternative syntax for the send-statement:
  `ch ! <expr_1> (<expr_2>,..., <expr_n>);`
- Alternative syntax for the receive-statement:
  `ch ! <var_1> (<var_2>,..., <var_n>);`

More Promela Message Passing

- Peeking at the message channel can be implemented in Promela with:
  `ch ? [<var_1>, <var_2>,..., <var_n>];`
  It is executable iff the message receive would be but does not actually remove the message from the channel. Moreover, the contents of the variables `<var_i>` remain unchanged.
- To do the same except that this time the variables `<var_i>` are changed, use:
  `ch ? < <var_1>, <var_2>,..., <var_n> >;`
  For example, `ch ? <x,y>` puts the contents of the first message in the channel `ch` to vars `x` and `y` without removing the message from the channel

Other Channel Operations

- `len(ch)` - returns the number of messages in channel `ch`
- `empty(ch)` - returns true if `ch` is empty, otherwise returns false
- `nempty(ch)` - returns true if `ch` is not empty, otherwise returns false
- `full(ch)` - returns true if `ch` is full, otherwise returns false
- `nfull(ch)` - returns true if `ch` is not full, otherwise returns false

Rendezvous Communication

- In Promela the synchronization between two processes (rendezvous) is syntactically implemented as message passing over a channel of capacity 0.
- In this case the channel cannot store messages, only pass immediately from the sender to the receiver.

Rendezvous Example

```
mtype = { msgtype };

chan name = [0] of { mtype, byte };

active proctype A()
{       name!msgtype(124);    /* Alternative syntax */
        name!msgtype(121)     /* used here */
}

active proctype B()
{       byte state;
        name?msgtype(state)   /* And here */
}
```

Rendezvous Example (cnt.)

- The processes A and B in the example synchronize: The execution of both the send and the receive is blocked until a matching send/receive pair becomes enabled.
- When a matching send/receive pair is enabled, they can execute and communicate in an atomic step the sent message from the sender to the receiver
- Note that if the channel had a capacity of 2 in the example, the process A could already terminate before the process B starts executing

Executability of Statements (recap)

- `skip` - always executable
- `assert(<expr>)` - always executable
- `<expression>` - executable if not zero
- `<assignment>` - always executable
- `if` - executable if at least one guard is
- `do` - executable if at least one guard is
- `break` - always executable
- send `ch ! msg` - executable if channel `ch` is not full
- receive `ch ? var` - executable if channel `ch` is not empty

Advanced Promela - `atomic`

In Promela a sequence of statements can be grouped together to execute atomically by using the `atomic` compound statement:

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

Atomic Sequences (cnt.)

- The sequence of statements inside an `atomic` sequence execute together in an uninterrupted manner
- In other words no other process can be scheduled until the atomic sequence has been completed
- In the example that means that no other process can be run to see the state where both `a` and `b` contains the old value of `a`

Atomic - Examples

- The following Promela statement sequences are not atomic:

```
nfull(ch) -> ch!msg0;  /* Not atomic! */
ch?[msg0] -> ch?msg0;  /* Not atomic! */
```

- They can be replaced by:

```
atomic { nfull(ch) -> ch!msg0 };/* Atomic! */
ch?msg0;               /* Trivially atomic! */
```

Atomic Sequences - Details

- The atomic sequences are also allowed to contain branching and non-determinism
- If any statement inside an atomic sequence is found to be unexecutable (i.e., it blocks the execution), other processes are allowed to run
- The states reached inside an `atomic` sequence still exists in the statespace of the system, not only the last state reached by the execution

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`

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 2:

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.)

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

Basic LTS Notation (cnt.)

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

Basic LTS Notation (cnt.)

- $L \longrightarrow$ iff for some $s \in S^0$ it holds that $s \longrightarrow$
- $L \Longrightarrow$ iff for some $s \in S^0$ it holds that $s \Longrightarrow$

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 $s = (s_1, s_2, \ldots, s_n)$),
- $S^0 = S_1^0 \times S_2^0 \times \cdots \times S_n^0$
  (all combinations of initial states of the component LTSs L_i), 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 $x \notin \Sigma_i$.