T–79.4301 Parallel and Distributed Systems (4 ECTS)

T–79.4301 Rinnakkaiset ja hajautetut järjestelmät (4 op)

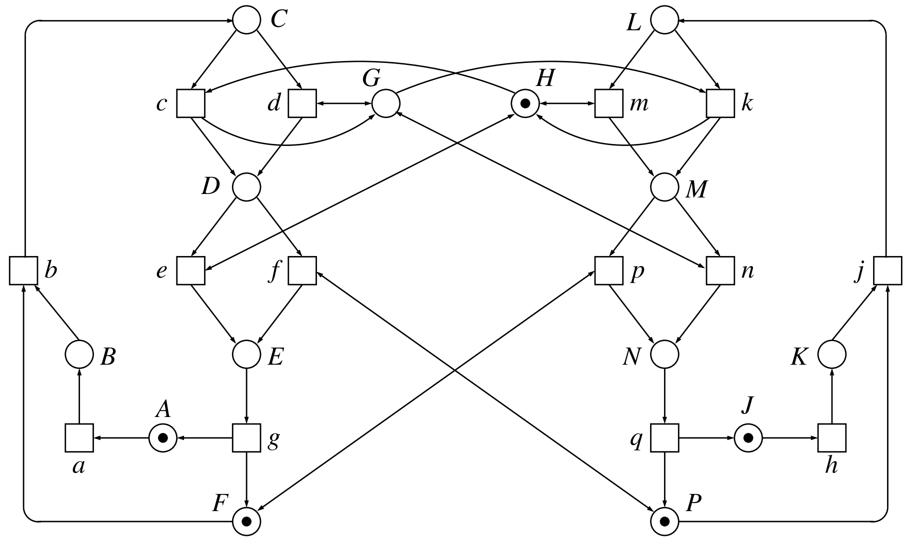
Lecture 10

14th of April 2008

Keijo Heljanko

Keijo.Heljanko@tkk.fi

Peterson's Mutex (by W. Reisig)



From 1-bounded P/T-nets to LTSs

- A 1-bounded P/T-net N with $|P|$ places can always be converted to a synchronization of LTSs $L_N = L_1||L_2||\cdots||L_n$ with $n \leq |P|$ components which have two states each. The reachability graph of L_N will be isomorphic to that of N.
- The construction is slightly too complicated to show here. The main trick is to use the set of transitions T as the alphabet Σ in L_N, and to make each L_i corresponding to a place $p \in P$ synchronize on all labels $t \in {}^\bullet p \cup p^\bullet$.

From P/T-nets to Promela

- Suppose that the net N we are looking is 255-bounded. Holzmann suggests the following scheme for translating P/T-nets (with $W(x,y) = 1$ for all $(x,y) \in F$, a restriction which can be easily removed) to Promela as shown in the next two slides.

From P/T-nets to Promela (cnt.)

```
#define Place byte /* < 256 tokens per place */

Place s0, s1, s2, r0, r1, r2;

#define inp1(x)        (x>0) -> x--
#define inp2(x,y)      (x>0&&y>0) -> x--; y--

#define out1(x)        x++
#define out2(x,y)      x++; y++
```


From P/T-nets to Promela (cnt.)

```
init
{
          atomic {s0=1;r0=1} /*initial marking*/
          do
/* t1 */  :: atomic { inp1(s0)    -> out1(s1) }
/* t2 */  :: atomic { inp1(r0)    -> out1(r2) }
/* t3 */  :: atomic { inp2(s1,r0)-> out2(s2,r1)}
/* t4 */  :: atomic { inp1(r2)    -> out1(r2) }
          od
}
```


From P/T-nets to Promela (cnt.)

- Actually, all `atomic` statements of the translation can safely be replaced with `d_step` statements.
- By using the LTS to P/T-net mapping first also LTSs can be translated to Promela.

From P/T-nets to Promela (cnt.)

- It may be more efficient to use a Petri net model checker such as PROD (`http://www.tcs.hut.fi/Software/prod/`) to do the model checking as for example the partial order reductions in Spin are not really effective for the model obtained from the translation. (The concurrency of the model is hidden inside the data manipulation of a single process.)
- Another Petri net model checker is Maria (`http://www.tcs.hut.fi/Software/maria/index.en.html`).
- Both of the tools actually use high-level Petri nets, which contain extensions to deal with structured data

Structural Analysis via Example

We want to prove mutual exclusion of Peterson's mutex algorithm. The critical sections correspond to places E and N, and thus our proof objective is:

$$M(E)+M(N) \leq 1 \qquad (1)$$

We can easily check that the net satisfies the following place invariants as they hold in the initial state and are preserved by every transition:

$$M(C)+M(D)+M(E)+M(F)=1 \qquad (2)$$
$$M(G)+M(H)=1 \qquad (3)$$
$$M(L)+M(M)+M(N)+M(P)=1 \qquad (4)$$

Example (cnt.)

By linear algebra, we can sum up the invariants (2), (3), and (4) to obtain a new invariant:

$$M(C)+M(D)+M(E)+M(F)+M(G)+$$
$$M(H)+M(L)+M(M)+M(N)+M(P)=3 \quad (5)$$

We need expressions on the markings which do not use equality to a constant on the right hand side to proceed further.

Example (cnt.)

It is easy to check that the following equation holds in the initial state and is preserved by every transition:

$$M(C)+M(F)+M(G)+M(M) \geq 1 \qquad (6)$$

Next subtract (6) from (5), to get the result:

$$M(D)+M(E)+M(H)+M(L)+M(N)+M(P) \leq 2 \quad (7)$$

Example (cnt.)

We also have:

$$M(D)+M(H)+M(L)+M(P) \geq 1 \qquad (8)$$

When we subtract (7) from (6), we get the result:

$$M(E)+M(N) \leq 1 \qquad (9)$$

Now, (8) is our proof objective (1), and thus we are done. Therefore the mutual exclusion property holds for the Peterson's mutex algorithm.

Structural Analysis Summary

With structural analysis, it is possible to prove properties of a model without exploring its dynamic behavior.

Structural analysis is not a complete method and always requires some human assistance. Thus it cannot replace model checking methods. However, it is can be a very powerful technique in the hands of a person performing the verification.

Extending LTSs with Data

- Sometimes it is convenient to extend the LTS model with data in order to more conveniently model Promela like languages.
- We will sketch the idea below in an informal manner.
- The idea is the following: Assume we have a system with n LTS components L_i, which manipulate m global variables x_j with a value range $0 \ldots r$.

ELTSs

- The state vector of the extended LTS system (ELTS) will consist of a tuple $(s_1, s_2, \ldots, s_n, v_1, v_2, \ldots, v_m)$, where s_i is the current local state of the component L_i and v_j is the current value of the global variable x_j.
- The initial state will be extended to give initial values for all the global variables.
- For each global variable we can define some operations, for example `inc(x)` to increment the value of global variable x, `dec(x)` to decrement it, and expressions like `iszero(x)` to check whether the variable is zero. (Expressions must be side-effect free.)

ELTSs (cnt.)

- Now we can for each local transition of the LTS add a guard: a list of expressions evaluated using the current values of the global variables. The guard will evaluate to true iff all the expressions evaluate to true.
- A global transition will be enabled iff all guards of all its component transitions evaluate to true.

ELTSs (cnt.)

- To update the global variables, each local transition is also associated with a list of operations.
- When a global transition is fired, each of the local transitions participating in it will in their turn execute its list of operations on the global variables.
- The state of global variables obtained after all operations have been executed is recorded as the state reached after firing the global transition.

ELTSs (cnt.)

- It is fairly straightforward to include other data manipulation features of Promela such as FIFOs and all their expression and operations in an ELTS model.
- The part that is hard to faithfully handle using ELTSs are the `atomic` and `d_step` features of Promela.
- There are many variants of the ELTS model, also state machines variants (extended finite state machines, EFSMs) are pretty common in the literature.

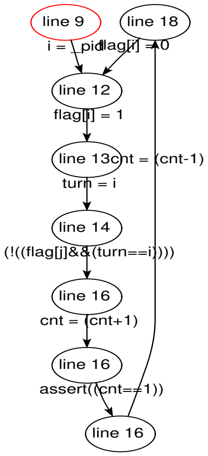
Promela and EFSMs

- Internally inside Spin all Promela programs are first translated into (a Spin variant of) extended finite state machines EFSMs.
- Consider for example the Peterson's Mutex algorithm shown in the next slide.
- Its EFSM can be produced in xspin using the feature "View Spin automaton for each Proctype" available in the run menu. The automaton is shown in the next slide following the Promela code.

Promela: Peterson's Mutex

```
bool turn, flag[2]; /* Code reformatted, old line numbers below */
byte cnt;
active [2] proctype P1()
{       pid i, j;
        i = _pid;                                    /* line 9 */
        j = 1 - _pid;
again:  flag[i] = true;                              /* line 12 */
        turn = i;
        !(flag[j] && turn == i) ->
                cnt++; assert(cnt == 1); cnt--; /* line 16 */
        flag[i] = false;                             /* line 18 */
        goto again
}
```

EFSM for Peterson’s Mutex



Notes on the Spin EFSM

- Notice how all the control flow statements have been removed, and all that remains is a state machine with expressions added to the edges. For example, the `goto` on line 19 has been removed. No `goto` statements will exists in any Spin EFSMs. (The picture is incomplete wrt. expressions.)
- Spin has done some internal optimizations. For example, there is no state of the automaton corresponding to line 10 of the program. This optimization is safe because j is a local variable.
- At runtime there are two instances of the same EFSM running.

Spin EFSMs

The statements appearing on edges of Spin EFSMs are:

- Assignments
- Assertions
- Print statements
- Send or Receive Statements
- Promela expressions (expression statements)

All other features of Promela (if-statements,do-loops, gotos, etc.) are mapped to the structure of the state machine part of the Spin EFSM.

Stuttering

- Recall from Lecture 8 how past formulas were defined over finite paths $\pi = x_0 x_1 x_2 \ldots x_n \in (2^{AP})^*$.
- By stuttering we mean a situation where π contains two consecutive indexes such that $x_i = x_{i+1}$, i.e., two consecutive states where the valuation of the atomic propositions did not change.

Cause of Stuttering

- In a parallel system quite a few things cause stuttering. For example, firing an invisible transition τ in some component not linked to the property under model checking causes the τ to be observable by the stuttering of current valuation of atomic propositions.
- It has been argued, that a temporal logic should not be able to observe the firing of such invisible transitions, and temporal logics insensitive to stuttering should be used instead.
- In other words: If the logic is not insensitive to stuttering, the verification results can differ due to a single firing of an “invisible transition”, which conflicts with our intuitive notion of what “invisible” means.

Stuttering Equivalence

- Two sequences π and π' are said to be stuttering equivalent, if π can be obtained from π' by executing a finite sequence of stuttering removals and insertions, where:
  - A stuttering removal takes two letters $x_i x_{i+1}$ at consecutive indexes of π' such that $x_i = x_{i+1}$, and replaces them in π' with a single letter x_i.
  - A stuttering insertion takes a single letter x_i of π' and replaces it in π' with two copies: $x_i x_i$.

Stuttering Invariance

- A logic is said to be *invariant under stuttering* (also called *stuttering insensitive*) iff for every formula ψ of the logic and every pair of stuttering equivalent words π, π' it holds that $\pi \models \psi$ iff $\pi' \models \psi$.
- In other words, a stuttering invariant logic cannot distinguish two sequences which only differ by the amount of stuttering in the sequences.

Stuttering and Past Safety Formulas

Recall the definition of past safety formulas from Lectures 8 and 9.

- The set of past safety formulas in not stuttering invariant because for example the formula $\mathbf{G}\,(p \Rightarrow \mathbf{Y}\,q)$ can distinguish two stuttering equivalent words.
- By disallowing the use of the “yesterday” operator $\mathbf{Y}$ (and its variant $\mathbf{Z}$) the logic becomes stuttering invariant.
- For future time logics, similarly, the “next” operator $\mathbf{X}$ needs to be disallowed to obtain a stuttering invariant logic.

Benefits of Stuttering Invariance

- The partial order reductions algorithms such as the ample sets employed by Spin require the specification logic to be stuttering invariant.
- For safety properties that are stuttering invariant, one can synchronize the specification automaton with only transitions that change the valuation of atomic propositions. (You need to synchronize on all of them in order not to introduce spurious counterexamples, see Tutorial 8.)
- Especially for run-time verification it can be hard to synchronize with all actions of the system in an efficient manner but limiting to observing changes to the atomic propositions may be much more feasible.