T–79.4301 Parallel and Distributed Systems (4 ECTS)

T–79.4301 Rinnakkaiset ja hajautetut järjestelmät (4 op)

Lecture 11

2006.04.21

Keijo Heljanko

Keijo.Heljanko@tkk.fi

Exam Info

- The exam is on Wed 10th of May 2006, 13:00-16:00 in lecture hall T1 in the CS building.
- Remember to register for the exam in WWWTopi on 8th of May at the latest.
- The exam will cover the material of Lectures 1-11 (Lecture 12 is not part of the exam requirements), Tutorials 1-8, as well as the home exercises 1-3.
- If you have not received the $\geq$50% score from the home exercises but still want to take the exam, please contact the Lecturer first.

Exam Info (cnt.)

- The questions will be available both in Finnish and in English.
- The preliminary plan is that the next exam is in August/September, and the one after that in December.

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 (cont.)

- 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 (cont.)

- 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 (cont.)

- 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.

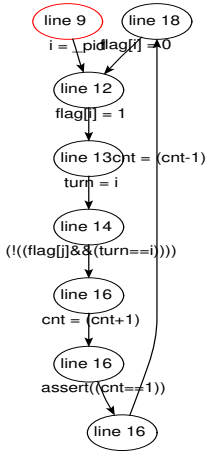
Promela and EFSMs

- 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, go-tos, 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.

Spin `neverclaim`

- Spin has a feature called `neverclaim` which for safety properties allows one to add an observer automaton to the system that observes each transition of the Promela program.
- Thus essentially, the reachability graph of the Promela program is synchronized with an observer automaton essentially using the finite state machine (not LTS!) synchronization construction.

Example

The following neverclaim detects all safety violations of the past safety formula $\mathbf{G}(p)$:

```
never {
        do
        :: true
        :: !p -> break
        od
}
```

Neverclaims

- Each transition of the Promela program is followed by one transition of a neverclaim.
- The neverclaim can not change the state of the system but can evaluate expressions based on the current value of atomic propositions.
- Thus a neverclaim can be seen as an EFSM which does not contain any operations, just expressions.
- Control flow is usually accomplished by using gotos.
- If the end of a neverclaim is ever reached, Spin reports an error.

Checking Safety with Neverclaims

- Intuitively neverclaims accept behaviors of the system that are counterexamples to the safety property being model checked.
- Thus any violation of a safety property expressible as an NFA can easily be mapped to a neverclaim.
- Neverclaims can also express liveness properties, but handling those is outside the scope of this course.
- When using partial order reductions the neverclaims used should be stuttering invariant, otherwise counterexamples can be erroneously missed!

Fairness

Fairness is a property of a system model often required to prove liveness properties of systems. They place additional constraints on what kind of looping (infinite) behaviors of the system are allowed. The two main types of fairness are:

- Weak fairness: Each weakly fair transition of the system is either disabled in infinitely many times or it is taken infinitely many times.
- Strong fairness: Each strongly fair transition of the system that is enabled infinitely many times is also fired infinitely many times.

Fairness (cnt.)

The rest of this lecture will be a demonstration, not part of the exam requirements.

Fair P/T-nets

- A fair P/T-net is a P/T-net with a fairness mapping $f : T \mapsto \{n, w, s\}$, where n stands for no fairness, w stands for weak fairness, and s stands for strong fairness.
- By definition, all finite runs of a fair P/T-net are fair.

Fair P/T-nets (cnt.)

- An infinite run of a P/T-net $\sigma = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \ldots$ is fair iff for each transition $t \in T$:
  - $f(t) = n$: $\top$ - no requirements for σ,
  - $f(t) = w$: Either $t_i = t$ for infinitely many $i \geq 0$, or $t \notin enabled(M_i)$ for infinitely many $i \geq 0$.
  - $f(t) = s$: If $t \in enabled(M_i)$ for infinitely many $i \geq 0$, then $t_i = t$ for infinitely many $i \geq 0$.

Fair P/T-nets (cnt.)

- It is easy to prove that every fair Petri net has a fair run. (It is easy to define alternative notions of fairness where this is not the case.)
- A fair P/T-net satisfies a temporal logic formula ψ iff $\pi \models \psi$ holds for every fair run of the P/T-net.
- The Maria model checker contains a direct support for both weak and strong fairness constraints of fair Petri nets.

Fair P/T-nets (cnt.)

- If ψ is a safety formula, the satisfaction of formulas is not affected by fairness.
- In the case ψ is a liveness formula, fairness constraints say that all runs of the system should satisfy the liveness property, while we don't care what happens in the non-fair runs.

Fairness Example

- Consider a system consisting of two processes, where the first process wants to execute a single local action in order to terminate.
- If we do not assume anything about the scheduling speeds of the two processes, we cannot prove that the first process will eventually terminate, as the second process can run in a loop without the first process ever being scheduled.
- If we make the single transition of the first process weakly fair, then in all fair runs of the system the first process will in fact terminate.

Uses of Fairness of Modelling

Often the different kinds of fairness are used in:

- No fairness: Events controlled by the environment, subroutines which might not terminate, etc.
- Weak fairness: Transitions of the system fully controlled by the running process, subroutines that will terminate, exits from critical sections.
- Strong fairness: Allocation of shared resources, entries to the critical section, different scheduling decisions by the scheduler, packet loss in a channel.

Implementing Fairness

- Suppose that you have managed to prove that some progress properties of the system hold under fairness in the model, and the model needs to be implemented in a programming language.
- It want be very hard to implement fairness in practice!
- For example, if some shared resources are allocated in a strongly fair fashion, you basically have to implement a scheduler (round-robin, etc.) to allocate the resources in a way that is fair towards all participants.
- Weak fairness is often simpler as it is usually a side-product of the operating system scheduler.

mplementing Fairness (cnt.)

- Sometimes it is infeasible/impossible to implement a scheduler.
- There are several ways to overcome such problems, which include:
  - Using timers/counters to detect when no progress is being made and resorting to a backup scheme when the timer fires / the counter indicates no progress has been made in a long time.
  - Using randomization to make the probability of not making progress small. (See for example Ethernet CSMA/CD.)

Fairness Teaser

- How would you implement a shared memory multiprocessor memory system with $n = 1024$ processors using 2^{30} cache lines worth of memory in a fashion that guarantees progress for all processors but is still of high performance? (Hint: There is no easy answer...)