T–79.4301 Parallel and Distributed Systems (4 ECTS)

Lecture 11

4th of December 2008

Keijo Heljanko

Keijo.Heljanko@tkk.fi

Exam Info

- The exam is on Wed 17th of December 2008, 9:00-12:00 in lecture hall T1 in the CS building.
- Please register for the exam in WebOodi.
- The exam will cover the material of Lectures 1-11 (Lecture 12 will not be part of exam requirements), Tutorials 1-11, 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 currently known date for the next exam is on 6th of May 2009, and the one one after that in August/September 2009.

Stuttering

- Recall from Lecture 9 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 Lecture 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 9.)
- 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!

Model Checking Approaches

- Explicit state model checking - reachability analysis combined with advanced reduction techniques such as ample sets, often used for data communications protocol SW, example tool: Spin
- Symbolic model checking with BDDs - reachability graph is stored compactly with binary decision diagrams, and computing the set of reachable states symbolically, often used for small hardware designs, example tool: NuSMV
- Bounded model checking with SAT - states reachable within k transitions are represented as a propositional satisfiability formula, incomplete, often used for debugging HW, example tool: NuSMV

Model Checking Approaches (cnt.)

- Counterexample guided abstraction refinement - an abstract model is generated from the input language (e.g., C-language) program and it is checked against the specification. If a counterexample is found that is spurious due to the fact that the abstraction is too coarse, the model is refined to remove this counterexample, and model checking is done again in the refined model. Example tools: SLAM, Blast.

Testing guided by Model Checking

- Concolic testing - this is a testing method utilizing model checking to improve coverage of test cases generated. First a random test case against the (Java) program is run. From this test run a set of path constraints of the execution are collected (with code instrumentation) reflecting paths that were not covered by the test run. These constraint are fed to a constraint solver, that finds inputs to the program that drive the next test run of the program through uncovered path in the control flow graph. Example tool: jCute.

Property Specification Languages

- In the hardware design community there has been a push towards standardized property specification languages. The most common ones are:
  - PSL - Property Specification Language, IEEE 1850 (`http://www.eda-stds.org/ieee-1850/`, `http://www.pslsugar.org/`). It includes the LTL temporal logic and other features such as regular expressions.
  - SVA - SystemVerilog Assertions (`http://www.systemverilog.org/`). A assertions language with similar goals as PSL built into SystemVerilog HW design language.

Property Specification Languages

- For software model checking usually LTL or one of its subsets is used as the specification language.
- Also regular expressions and finite state machines are used to express safety properties of software.
- The challenge in the software side in standardizing the property specification language lies in the vastly more complex data handling compared to the HW case.

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 fair 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.

Implementing 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...)

Model Checking Tools

- In the following slides model checking tools other than Spin are described
- All the tools are freely available (under various licences) unless otherwise stated
- The comments on the strengths of the tools are highly subjective
- See the table of model checkers at: `http://anna.fi.muni.cz/yahoda/`

NuSMV 2

- Homepage: `nusmv.irst.itc.it/`
- A model checker (mainly) for hardware, a remake of the SMV model checker
- BDD based symbolic model checker
- Bounded model checker
- Licence: LGPL

IBM Rulebase

- Homepage: `http://www.haifa.ibm.com/projects/verification/RB_Homepage/index.html`
- A commercial hardware model checker by IBM
- BDD based symbolic model checker
- Bounded model checker
- Parallelized model checkers
- Licence: Commercial, University program available

Java Pathfinder 2

- Homepage: `http://javapathfinder.sourceforge.net/`
- A model checker for Java programs
- Implementation technique: A full custom Java virtual machine
- See also other Java model checkers such as Bandera (`http://bandera.projects.cis.ksu.edu/`) and Bogor (`http://bogor.projects.cis.ksu.edu/`).

Uppaal

- Homepage: `http://www.uppaal.com/`
- A model checker for timed systems
- Free for academic use, commercial licences available
- See also other model checkers for timed systems such as: IF (`http://www-verimag.imag.fr/~async/IF/`) which also handles untimed systems

SLAM

- Homepage: `http://research.microsoft.com/slam/`
- A model checker for sequential C programs (correct use of locking primitives in Windows device drivers) heavily employing abstraction
- Licence: Not available outside Microsoft
- See also: Zing (`http://research.microsoft.com/zing/`)

Blast

- BLAST – Berkeley Lazy Abstraction Software Verification Tool
- Homepage: `http://mtc.epfl.ch/software-tools/blast/`
- Model checker for C programs
- Employs lazy abstraction refinement
- Licence: Modified BSD licence

DiVinE

- DiVinE – Distributed Verification Environment
- Homepage: `http://anna.fi.muni.cz/divine/`
- A distributed model checker for computing clusters
- Accepts Promela programs and is thus also available for verifying Spin models
- Uses NIPS: `http://wwwhome.cs.utwente.nl/~michaelw/nips/` as a virtual machine to interpret Promela models. See NIPS homepage for further details about Promela semantics.

Maria

- Homepage: `http://www.tcs.hut.fi/Software/maria/`
- A model checker for high-level Petri nets
- Good support for LTL model checking under fairness
- Very extensive data manipulation support, quite flexible as a model checker back-end
- Licence: GPL

PROD

- Homepage: `http://www.tcs.hut.fi/Software/prod/`
- A model checker for high-level Petri nets (Pr/T-nets)
- Very good partial order reduction algorithms available (even better than Spin in many cases)

Murφ

- A model checker for asynchronous systems in a formalism closely related to Petri nets
- Good symmetry reduction algorithms available

The Model Checking Kit

- Homepage:

  `http://www.fmi.uni-stuttgart.de/szs/tools/mckit/overview.shtml`

- A collection of different model checking tools behind a single interface
- Provides an easy way to try different methods on small model checking problems