# T–79.4301 Parallel and Distributed Systems (4 ECTS)

## *Lecture 1*

*11th of September 2008*

Keijo Heljanko

Keijo.Heljanko@tkk.fi



## T–79.4301 Parallel and Distributed Systems (4 ECTS)

- Design methods for parallel and distributed systems: modelling and verification.
- The language of the course is English.
- Lectures: Thu 12:15–14:00 in B353, first Lecture on 11th of Sep.
- Tutorials: Thu 14:15–15:00 in B353, first Tutorial on 18th of Sep.
- Course homepage available in NOPPA portal: `http://noppa.tkk.fi/`



## Course Personnel

- Lectures: Prof. Keijo Heljanko
  - Email: Keijo.Heljanko@tkk.fi
  - Homepage: `http://www.tcs.hut.fi/~kepa/`
- Tutorials: Tuomas Launiainen
  - Email: t794301-autumn08@tcs.hut.fi
  - News: `nntp://news.tky.fi/opinnot.tik.rhj`



## Course requirements

To pass the course you have to:

- Pass the exam. Preliminary exam date: Thursday 17th of Dec 2008 at 9:00-12:00 in T1
- Get enough points from home exercises:
  - $\geq$ 50% of points from home exercises to pass,
  - $\geq$ 80% of points gives +1 to exam grade.

  The exercises should be done individually, no exercise groups/sharing of solutions allowed.

## Home exercise schedule

The deadlines are tight!

- Thursday 16.10 at 12:15 - Exercise 1 distributed
- Thursday 6.11 at 12:15 - Deadline of Exercise 1, Exercise 2 distributed
- Thursday 20.11 at 12:15 - Deadline of Exercise 2, Exercise 3 distributed
- Thursday 4.12 at 12:15 - Deadline of Exercise 3

The deadlines are tight!



## Course material

- All material needed for the exam will be distributed through the NOPPA portal.
- Material will mostly consist of the lecture slides.
- Also some tutorial material will be added.
- There is no single book the course will be based on but we can recommend a few related ones.



## Course replacement

The course T–79.4301 Parallel and Distributed Systems (4 ECTS) can be used to replace either:

- T–79.179 Parallel and Distributed Digital Systems (3 cr), or
- T–79.231 Parallel and Distributed Digital Systems (3 cr).



## Software failures

Software is used widely in many applications where a bug in the system can cause large damage:

- Safety critical systems: airplane control systems, medical care, train signalling systems, air traffic control, power plants, etc.
- Economically critical systems: e-commerce systems, Internet, microprocessors, etc.

## Price of Software Defects

Two very expensive software bugs:

- Intel Pentium FDIV bug (1994, approximately $500 million).
- Ariane 5 floating point overflow (1996, approximately $500 million).



## Pentium FDIV - Software bug in HW


Image © CPU-World.com

```
4195835 - ((4195835 / 3145727) * 3145727) = 256
```

The floating point division algorithm uses an array of constants with 1066 elements. However, only 1061 elements of the array were correctly initialised.



## Ariane 5

Exploded 37 seconds after takeoff - the reason was an overflow in a conversion of a 64 bit floating point number into a 16 bit integer.



## More Software Bugs

Prof. Thomas Huckle, TU München: Collection of Software Bugs
`http://www5.in.tum.de/~huckle/bugse.html`

# The Cost of Software Defects

The national economic impacts of software defects are significant. In the USA the cost of software defects has been estimated to be $59 billion, that is 0.6% of the gross domestic product.

Source: National Institute of Standards & Technology (NIST): The Economic Impacts of Inadequate Infrastructure for Software Testing
`www.nist.gov/director/prog-ofc/report02-3.pdf`



# Reducing the Cost

According to the report by NIST 1/3 of the software defects could be avoided by using better software development methodology.

In this course the major focus is on development methods for parallel and distributed systems. The main focus is on modelling and computer aided verification methods.



# Finding Bugs in Parallel Systems

The principal methods for the validation of complex parallel and distributed systems are:

- Testing (using the system itself)
- Simulation (using a model of the system)
- Deductive verification (mathematical (manual) proof of correctness, in practice done with computer aided proof assistants/proof checkers)
- Model Checking ($\approx$ exhaustive testing of a model of the system)

Use also good design methodology!



# Why is Testing Hard?

Testing should always be done! However, testing parallel and distributed systems is not always cost effective:

- Testing concurrency related problems is often done only when rest of the system is in place
  $\Rightarrow$ fixing bugs late can be very costly.
- It is labour intensive to write good tests.
- It is hard if not impossible to reproduce bugs due to concurrency encountered in testing.
  - Did the bug-fix work?
- Testing can only prove the existence of bugs, not their in-existence.

# Simulation

The main method for the validation of hardware designs:

- When designing new microprocessors, no physical silicon implementation exists until very late in the project.
- Example: Intel Pentium 4 simulation capacity (Roope Kaivola, talk at CAV05):
  - 8000 CPUs
  - Full chip simulation speed 8 Hz (final silicon > 2 GHz).
  - Amount of real time simulated before tape-out: well under 5 minutes.
- Consider using simulation/prototyping for software.



# Deductive Verification

- Proving things correct by mathematical means (mostly invariants + induction).
- Computer aided proof assistants used to keep you honest (it will nag you if you've missed a case in you proof) and to prove small sub-cases.
- Very high cost, requires highly skilled personnel:
  - Only for truly critical systems.
  - HW examples: Pentium 4 FPU, Pentium 4 register rename logic (Roope Kaivola: 2 man years, 2 'time bomb' silicon bugs found - thankfully masked by surrounding logic)



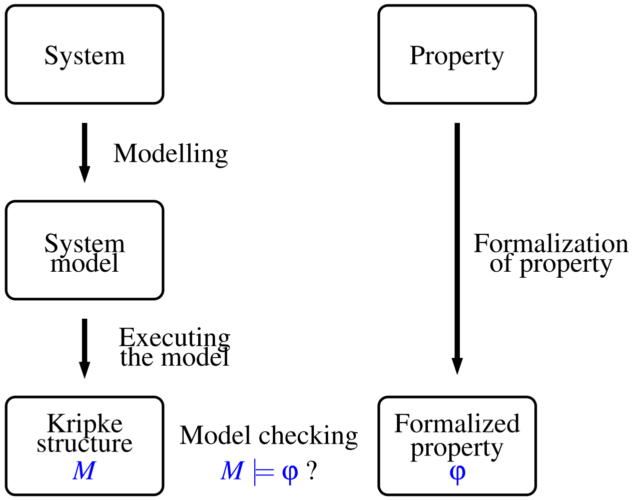
# Model Checking

In model checking every execution of the model of the system is simulated obtaining a Kripke structure $M$ describing all its behaviours. $M$ is then checked against a system property $\varphi$:

- Yes: The system functions according to the specified property (denoted $M \models \varphi$). The symbol $\models$ is pronounced "models", hence the term model checking.
- No: The system is incorrect (denoted $M \not\models \varphi$), a counterexample is returned: an execution of the system which does not satisfy the property.



# Models and Properties



# Benefits of Model Checking

- In principle automated: Given a system model and a property, the model checking algorithm is fully automatic
- Counterexamples are valuable for debugging
- Already the process of modelling catches a large percentage of the bugs: rapid prototyping of concurrency related features



# Drawbacks of Model Checking

- State explosion problem: Capacity limits of model checkers often exceeded
- Manual modelling often needed:
  - Model checker used might not support all features of the implementation language
  - Abstraction needed to overcome capacity problems
- Reverse engineering of existing already implemented systems to obtain models is time consuming and often futile



# Model Checking in the Industry

- Microprocessor design: All major microprocessor manufacturers use model checking methods as a part of their design process
- Design of Data-Communications Protocol Software: Model checkers have been used as rapid prototyping systems for validating new data-communications protocols under standardization.
- Critical Software: NASA space program is model checking code used by the space program.
- Operating Systems: Microsoft is using model checking to verify the correct functioning of new Windows device drivers.



# Modelling Languages

As a language describing system models we can for example use:

- Java programs,
- UML (unified modelling language) state machines,
- SDL (specification and description language),
- Promela language (input language of the Spin model checker),
- Petri nets (model checkers from HUT: Maria, PROD),
- process algebras, and
- VHDL,Verilog, or SMV languages (mostly for HW design).

# Choosing the Modelling Language

If the modelling language of choice has a well defined semantics (i.e., it is possible to write an algorithm to generate all the possible executions of the system), it is in principle applicable as a modelling language.

The most important thing in choosing your modelling language is the ease of modelling and the existence of a sufficiently good model checker for it



# Modelling Language of this Course

In this course we will mainly be using the Promela language because of the relative ease of modelling and existence of a good model checker Spin for it.

Promela is a natural choice for data-communications protocol modelling.



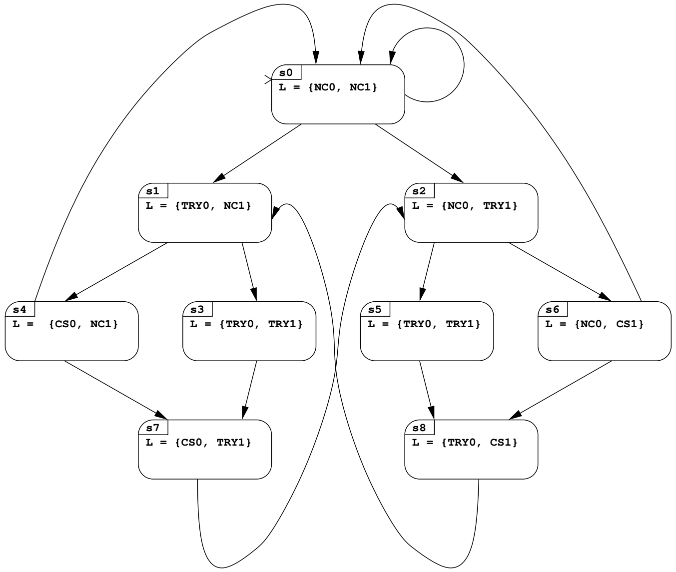
# Kripke Structures

Kripke structure is a fully modelling language independent way of representing the behaviour of parallel and distributed system.

Kripke structures are graphs which describe all the possible executions of the system, where all internal state information has been hidden, except for some interesting atomic propositions.



# Example: Mutex - Kripke structure



# Kripke structure

Kripke structure is a directed graph, where:

- The states of the graph are all possible reachable states of the system.
- There is an arc from state $s$ to state $s'$ if and only if (iff from now on) it is possible to move with an atomic action from state $s$ to the state $s'$.
- The valuation $L$ of each state contains exactly those atomic propositions which hold in that state.



# Formal Definition

**Definition 1** Let $AP$ be a finite set of atomic propositions. Kripke structure is a four-tuple $M = (S, s^0, R, L)$, where

- $S$ is a finite set of states,
- $s^0 \in S$ is the initial state (marked with a wedge),
- $R \subseteq S \times S$ is the transition relation, $((s, s') \in R$ is drawn as an arc from $s$ to $s'$), and
- $L : S \to 2^{AP}$ is a valuation, i.e. a function which maps each state to those atomic propositions which hold in that state.



# Kripke Structures and Automata

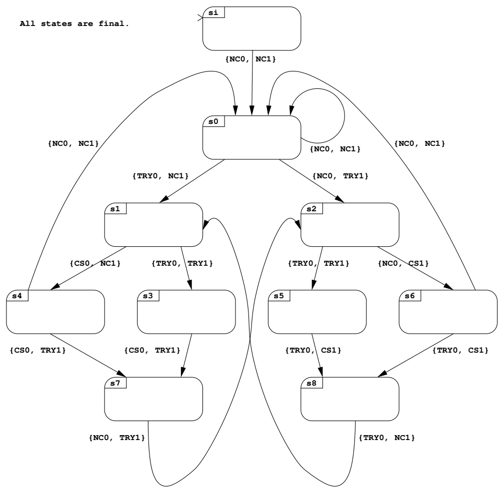
Kripke structures have a close relationship with finite state automata (FSA) (recall from: T–79.1001/T-79.158 Introduction to Theoretical Computer Science):
The changes are the following:

- labelling is on states instead of having labels on arcs,
- alphabet $\Sigma$ consists of the subsets of $AP$,
- there is at most one arc between any two states, and
- there is no definition of final states. (All the states are final.)

It is easy to derive a FSA out of a Kripke structure.



# Example: The Mutex Automaton $\mathcal{A}_M$



## Model Checking - Ingredients

- A way of modelling the system conveniently - modelling language
- A way of describing all the behaviours of the system model in a modelling language independent way - Kripke structure
- A way of specifying properties - assertions, automata, regular expressions, temporal logics
- An algorithm to check whether the property holds for the system - model checker



## Main Topics for the Course

The rest of the course will concentrate especially on:

- modelling of parallel and distributed systems,
- specifying properties,
- using model checkers to verify them, and
- basic theoretical background of verification methods.