T-79.5305 Formal Methods (4 ECTS)

T-79.5305 Formaalit menetelmät (4 op)

2006-09-13

Tommi Junttila, Keijo Heljanko, Ilkka Niemelä, and Heikki Tauriainen

HELSINKI UNIVERSITY OF TECHNOLOGY
Laboratory for Theoretical Computer Science

T-79.5305 Formal Methods (4 ECTS)

- Model checking techniques (for software systems): abstraction methods, advanced state space analysis, model reduction by slicing
- Seminars: Wed 16:15– in TB353, first meeting on 13th of Sep.
- Course homepage: `http://www.tcs.hut.fi/Studies/T-79.5305/`

Course Personnel

- Coordinator: D.Sc.(Tech.) Tommi Junttila
- Tutors:
  - Prof. Ilkka Niemelä
  - Doc., Academy Research Fellow Keijo Heljanko
  - Lic.Sc.(Tech) Heikki Tauriainen
- Contact information: see the course web page

Course requirements

To pass the course you have to:

- Attend the seminars (recommended, not required)
- Give a seminar presentation. The length of the presentation should be 45–90 minutes.
- Write a short report (5–8 pages) on your presentation topic. The report shoud review the problem, solution, and techniques in the presentation material and include a small self-made example on applying the solution/techniques.
- Act as an opponent for one other presentation. This includes reading the presentation material, preparing and presenting two questions, and generally being active during the presentation.

Grading

The grading will be based on

- the seminar presentation (weight 60 percent),
- the short report (weight 20 percent), and
- acting as an opponent (weight 20 percent).

The deadline for delivering the short report is the wednesday *two weeks* after your presentation.

Course material

- Seminar presentations will be given on research articles on the topics.
- The course web page will list some background material that can be consulted if needed.
- Each presentation will be assigned a tutor who will
  - consult during the preparation of your presentation,
  - grade your presentation, short report, and acting as an opponent.

Hints for Preparing your Presentation

- It will take some time!
- Consult your tutor!
- Introduce the context, goal, and general structure of your presentation material
- For each subproblem, proceed top-down:
  - introduce the problem,
  - outline the solution, and
  - then go deeper in the details.
- Prepare examples in advance.

Course replacement

The course T–79.5305 Formal Methods (4 ECTS) replaces the course

- T-79.157 Formal Description and Verification of Computing Systems

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, etc.
- Economically critical systems: ecommerce 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


```
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 recent developments on advanced computer aided verification methods for software systems, including parallel and distributed ones. methods.

Finding Bugs in Software 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 a 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

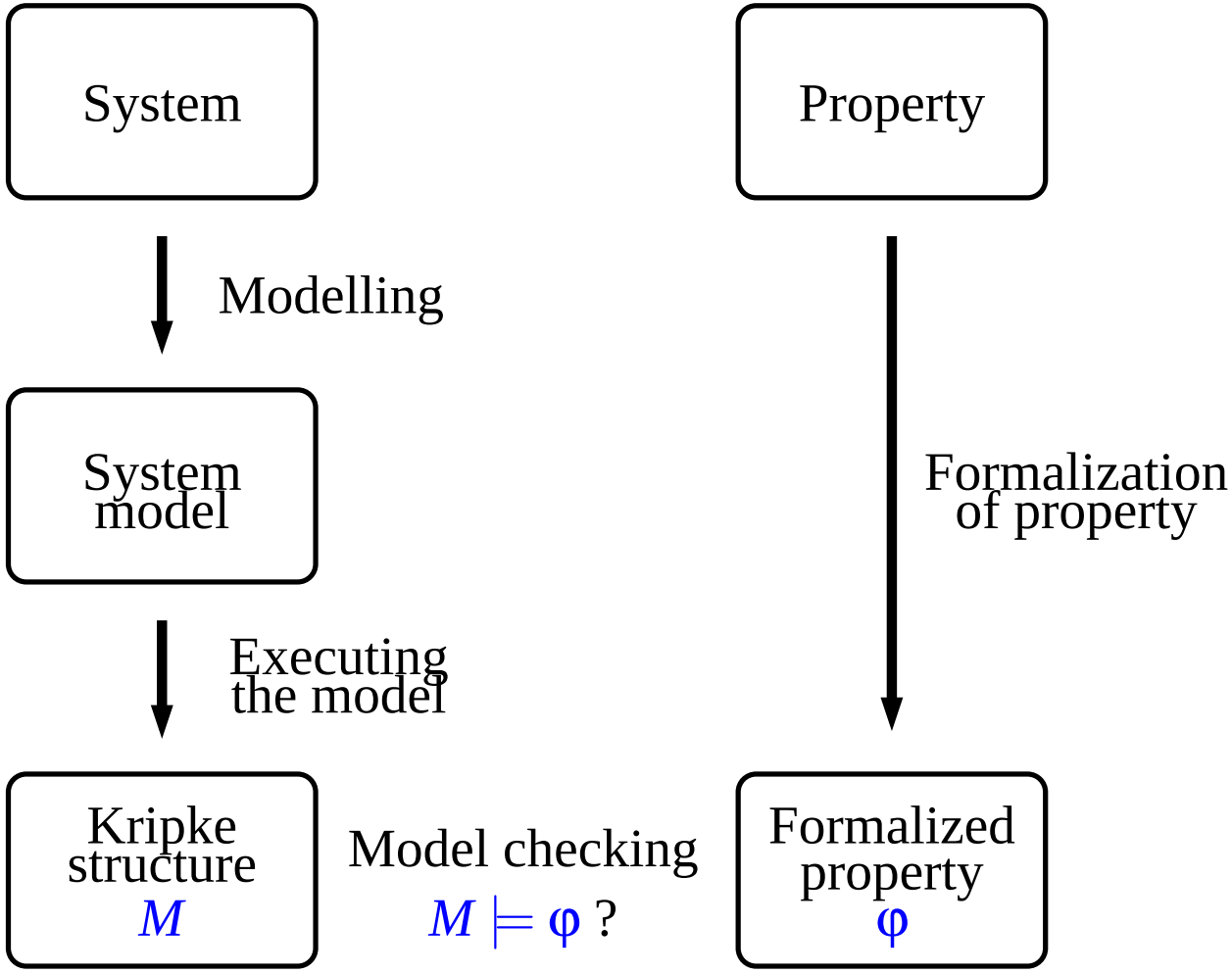
- Proving things correct by mathematical means (mostly invariants + induction).
- Computer aided proof assistants used to keep you honest (it will nag you if you've missed a case in you proof) and to prove small sub-cases.
- Very high cost, requires highly skilled personnel:
  - Only for truly critical 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 φ:

- 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 standardisation. They've also been used as verification tool of protocol implementations (Bell Labs, Nokia)

Model Checking in the Industry

- Critical Software: NASA space program is currently developing and using model checking technology for verifying code used by the space program.
- Operating System Kernel Software: Microsoft has applied model checking to analyze the locking discipline of 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).

Main Topics for the Course

The seminar presentations in the course will concentrate on

- using abstraction techniques when model checking software systems,
- approaches to model checking concurrent Java programs,
- model/program reduction by using slicing,
- ...