Model Checking based Software Verification

18.5-2006

Keijo Heljanko

Keijo.Heljanko@tkk.fi

Laboratory for Theoretical Computer Science
Department of Computer Science and Engineering
Helsinki University of Technology
http://www.tcs.tkk.fi/~kepa/

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, microprocessors, Internet, etc.

Cost of Software Bugs

Three software bugs:

- Intel Pentium FDIV bug (1994, approx $500 million).
- Ariane 5 floating point overflow (1996, approx $500 million).
- Mars Pathfinder priority inversion problem (fixable)

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`

According to NIST, 1/3 of the costs could be avoided by using better software development methods.

Pentium FDIV bug


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

The floating point division algorithm uses a table of constants with 1066 rows. A bug in the initialization of the table caused only 1061 rows to be correctly initialized.

Cost: $500 million

Ariane 5

Self destructed 37 seconds after takeoff - the cause was an overflow in the conversion from a 64 bit floating point number to a 16 bit integer.

Cost: $500 million

Mars Pathfinder

- A priority inversion problem in the Pathfinder space craft made the system lock up and reboot every once in a while.
- The bug consisted of a low priority thread holding a mutex lock and not being able to obtain any CPU time to release the lock needed for higher priority threads to make progress. A watchdog timer then rebooted the system.

More Software Bugs

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

Finding Bugs in Software

The principal methods for the validation of complex software/hardware systems are:

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

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 the rest of the system implementation is available
  $\Rightarrow$ 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.

Deductive Verification

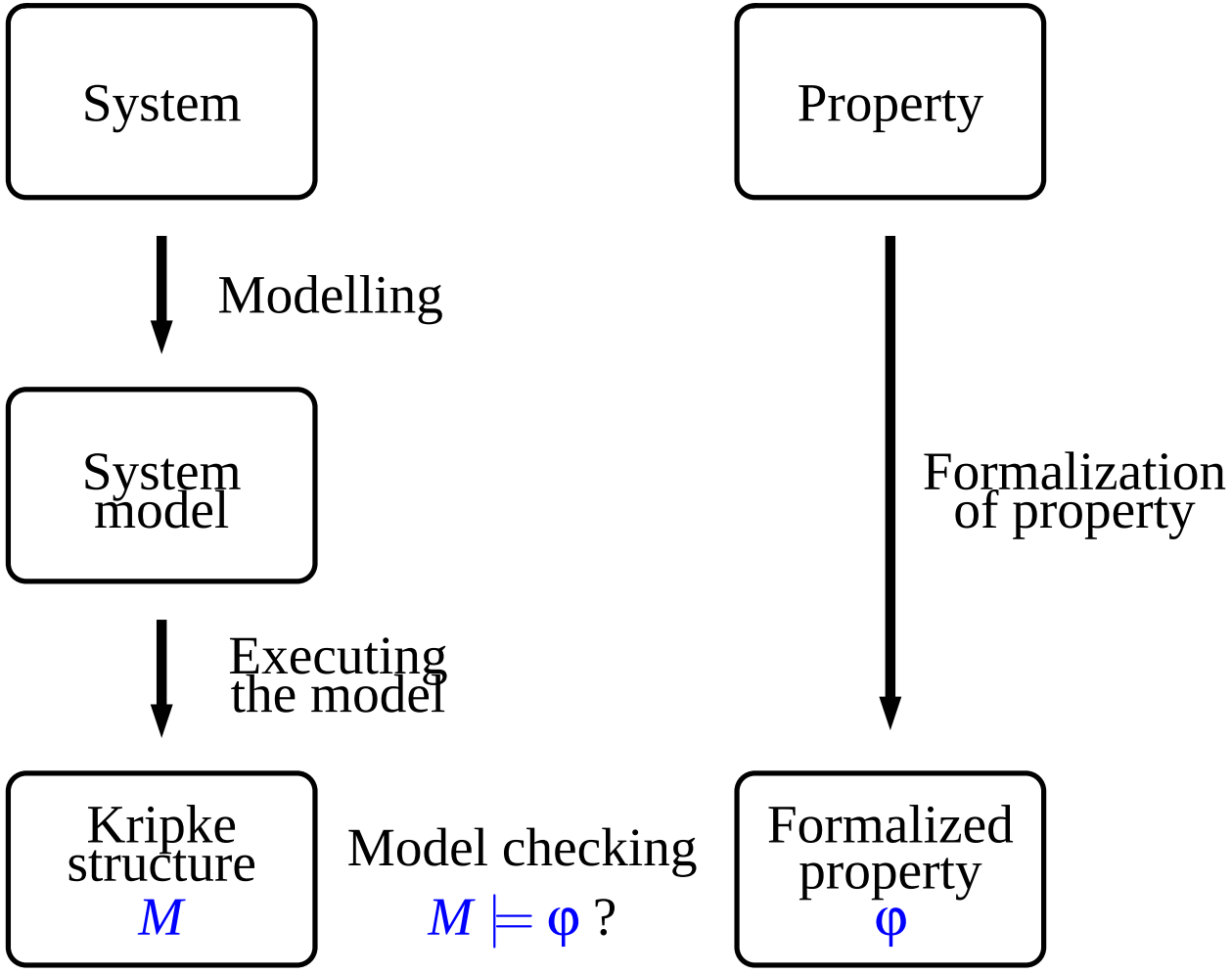
- Proving things correct by mathematical means (mostly invariants + induction).
- Computer aided proof assistants used to keep you honest (it will remind you if you've missed a case in you proof) and to prove small sub-cases.
- Very high cost, requires highly skilled personnel:
  - Only for truly critical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



Modelling Languages

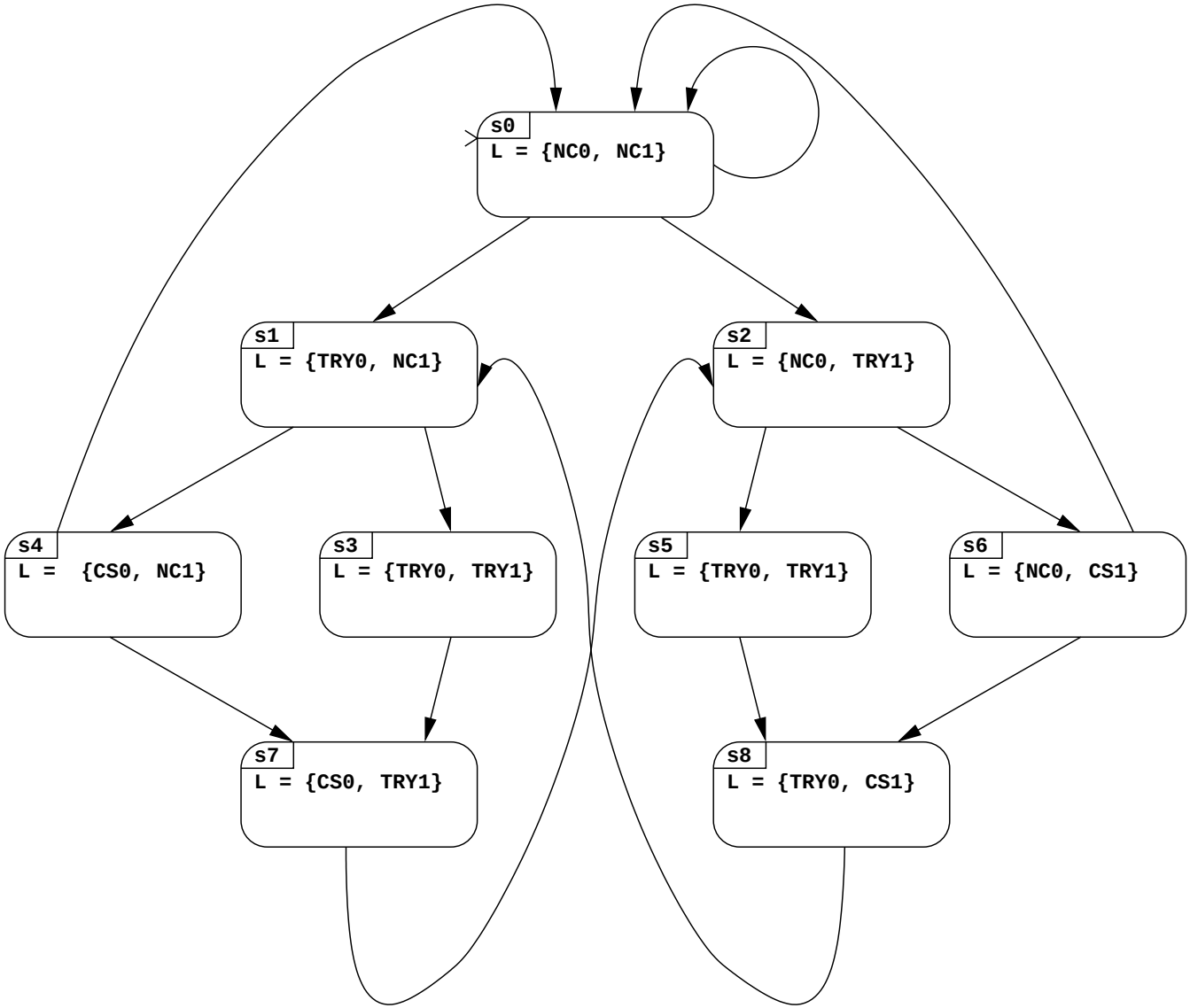
As a language describing system models we can for example use:

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

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.
- Kripke structures analyzed by model checkers can contain tens of millions of states of the system. It is not always possible to store all of them explicitly.

Example: Mutex - Kripke structure



Specification Formalisms

- Temporal logics such as the linear temporal logic LTL are usually used as the specification formalism.
- Also finite state automata or regular expressions can be used.
- The (hardware) industry standard property specification language IEEE 1850 - PSL combines temporal logics with regular expressions.

Properties of the Mutex system

Below are two properties the Mutex system fulfills and their formalization in LTL.

- It is never the case that both processes are at the same time in the critical section:
  $\Box\neg(CR0 \wedge CR1)$
- Whenever process 0 wants to get into the critical section, it always eventually gets there:
  $\Box(TRY0 \Rightarrow \Diamond CR0)$

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 can be often exceeded. $\Rightarrow$ Nice research problems for researchers creating new model checking algorithms.
- 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 often futile. Early stage specifications are the most fruitful target.

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.

Model Checking Research at HUT

Main research topics:

- Bounded model checking (BMC): A variant of model checking where all the executions of the system are investigated only upto a fixed number of time steps (the bound).
- Use of highly efficient propositional satisfiability (SAT) solvers to implement bounded model checking.
- BMC encodings of property specification languages such as LTL and PSL.
- Optimizing BMC for analyzing systems that contain concurrency.

Conclusions

- Model checking is a systematic way of checking whether all behaviours of a model of a system fullfil its specification.
- The approach is at its best in early stages of protocol development: E.g., as a rapid prototyping system to ease protocol standardization work.
- Model checking is in use in the industry in several different areas.
- Highly optimized model checking algorithms and tools exist, however, there is still much work to be done in the area.