Formal verification:
The story about system correctness

[Automated Reasoning, 2015/2016 1b — Lecture 1]

Doina Bucur

d.bucur@rug.nl

Nov 2015

What is **automated reasoning**?

Reasoning = solving problems by mathematically proving the answer:

Take the problem assumptions.

Systematically apply deductive rules from logic

…until a conclusion is reached.

Automated reasoning simply automates this.

Tools: *software verifiers, model checkers, static analysers, theorem provers, runtime memory profilers*.

Model checking (or **formal verification**) is an example; it **verifies** the correctness of the model of a system (design or implementation) against a specification. Like any mathematical proof, it provides guarantees.

(Check the **Stanford Encyclopedia of Philosophy** for an overview; skim through topics you may be interested in. http://plato.stanford.edu/entries/reasoning-automated/)

Scientific aim:
a **"Verifying Compiler"**

(**T. Hoare**, Turing Award 1980 for fundamental contributions to the definition and design of programming languages)

"At present, the most widely accepted means of raising trust levels of software is by massive and expensive testing.

A verifying compiler uses mathematical and logical reasoning to check the correctness of the programs that it compiles.

The criterion of correctness is specified by types, assertions, and other redundant annotations associated with the code of the program."

(**The verifying compiler: A grand challenge for computing research.** Tony Hoare. J. ACM 50, 1, 63-69. 2003.)

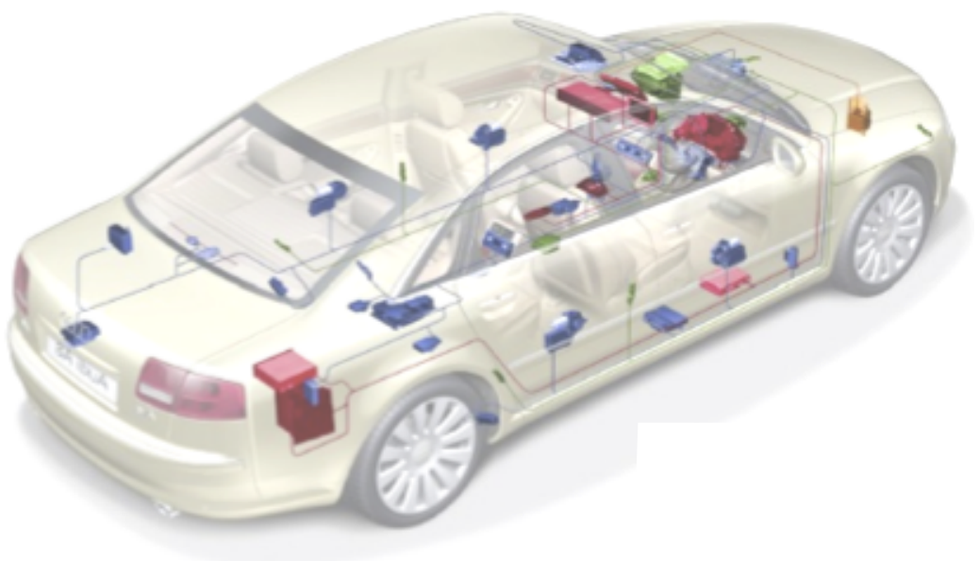
Verification vs. testing

Testing (or **Validation**): executing with a set of inputs to see that the system will **sometimes** do as intended.

Verification: formally proving that the system will **always** (i.e., for any inputs) do as intended (the system is said to be “correct”).

Critical systems

Hardware and software systems are used in applications where failure is unacceptable.



ESA European Space Agency, Ariane 5 first launch, 1996. Failure: velocity reading overflowed 16 bits, triggering an autodestruct.

Atomic Energy of Canada, Therac-25 radiation therapy machine ‘85-87. Radiation overdoses (x100). Failure: race condition

Bugs in critical systems

http://www.cs.tau.ac.il/~nachumd/horror.html

- The **Mars Climate Orbiter** crashed Sep 1999 due to wrong program units (non-metric).
- A China Airlines **Airbus Industrie A300** crashes Apr 26, 1994 killing 264. Recommendations include software modifications.
- On Oct 24, 2013, a court ruled against **Toyota** in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module's (ECM) firmware.
- 486-DX4s, **Pentiums** and Pentium clones had a bug in their floating-point division algorithm discovered in 1994. Intel set aside US $475 million to cover the costs.

http://www.edn.com/design/automotive/4423428/1/Toyota-s-killer-firmware--Bad-design-and-its-consequences

Toyota's killer firmware: Bad design and its consequences

Michael Dunn - October 28, 2013

92 Comments

Share 152 +1 749 Tweet 0 Like 3.1k

On Thursday October 24, 2013, an Oklahoma court **ruled against Toyota** in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module's (ECM) firmware.

- Toyota's electronic throttle control system (ETCS) source code is of **unreasonable quality**.
- Toyota's source code is defective and contains **bugs**, including bugs that can cause unintended acceleration.
- A litany of faults were found in the code, including **buffer overflow**, **unsafe casting**, and **race conditions** between tasks.
- Toyota claimed only 41% of the allocated **stack** space was being used. Barr's investigation showed that 94% was closer to the truth. On top of that, stack-killing, MISRA-C rule-violating recursion was found in the code, and the CPU doesn't incorporate memory protection to guard against **stack overflow**.
- MISRA-C:1998, in effect when the code was originally written, has 93 required and 34 advisory rules. Toyota nailed 6 of them.

Formal verification methods

Errors should be eliminated in the design and implementation process (i.e., **statically**).

When this is not possible, the remaining errors should be caught **at runtime** before they occur.

Research existed early on in Computer Science around theories to prove the quality of complex concurrent systems. The **application** to real-life systems is recent.

Our model checkers:

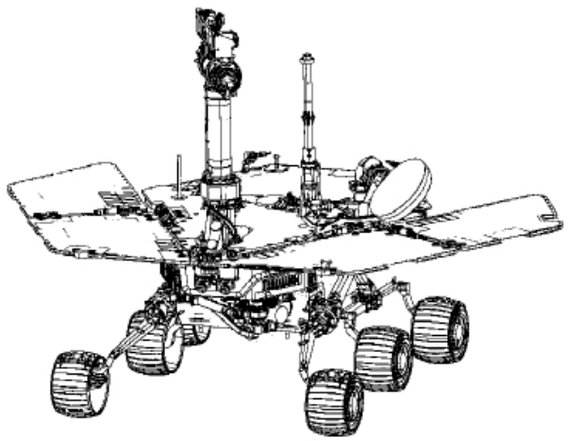
- **SPIN (Simple Promela Interpreter)**
  http://spinroot.com
- **CBMC (Bounded Model Checker for ANSI-C)**
  http://www.cprover.org/cbmc/

Model checkers for C/Java

Tool name	Tool developer	Symbolic analysis	Abstraction	Counterexample	BMC	Concurrency	Languages
ASTRÉE	École Normale Supérieure	×	×				C (subset)
CODESONAR	Grammatech Inc.	×	×				C, C++, ADA
PolySpace	PolySpace Technologies	×	×			×	C, C++, ADA, UML
PREVENT	Coverity	×	×			×	C, C++, Java
BLAST	UC Berkeley/EPF Lausanne	×	×	×		×	C
F-SOFT (abs)	NEC	×	×	×			C
Java PathFind.	NASA	×		×	×	×	Java
MAGIC	Carnegie Mellon University	×	×	×		×[1]	C
SATABS	Oxford University	×	×	×		×	C, C++, SpecC, SystemC
SLAM	Microsoft	×	×	×		×	C
SPIN	Bell Labs[2]			×	×	×	PROMELA, C[3]
ZING	Microsoft Research			×	×	×	ZING (object oriented)
CBMC	CMU/Oxford University	×		×	×		C, C++, SpecC, SystemC
F-SOFT (bmc)	NEC	×		×	×		C
EXE	Stanford University	×		×	×		C
SATURN	Stanford University	×		×	×		C

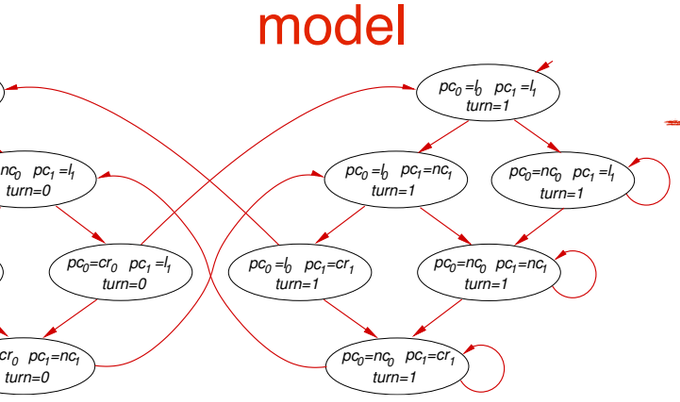
[D'Silva, Kroening, Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification", IEEE Transactions on Computer Aided Design. 2008]

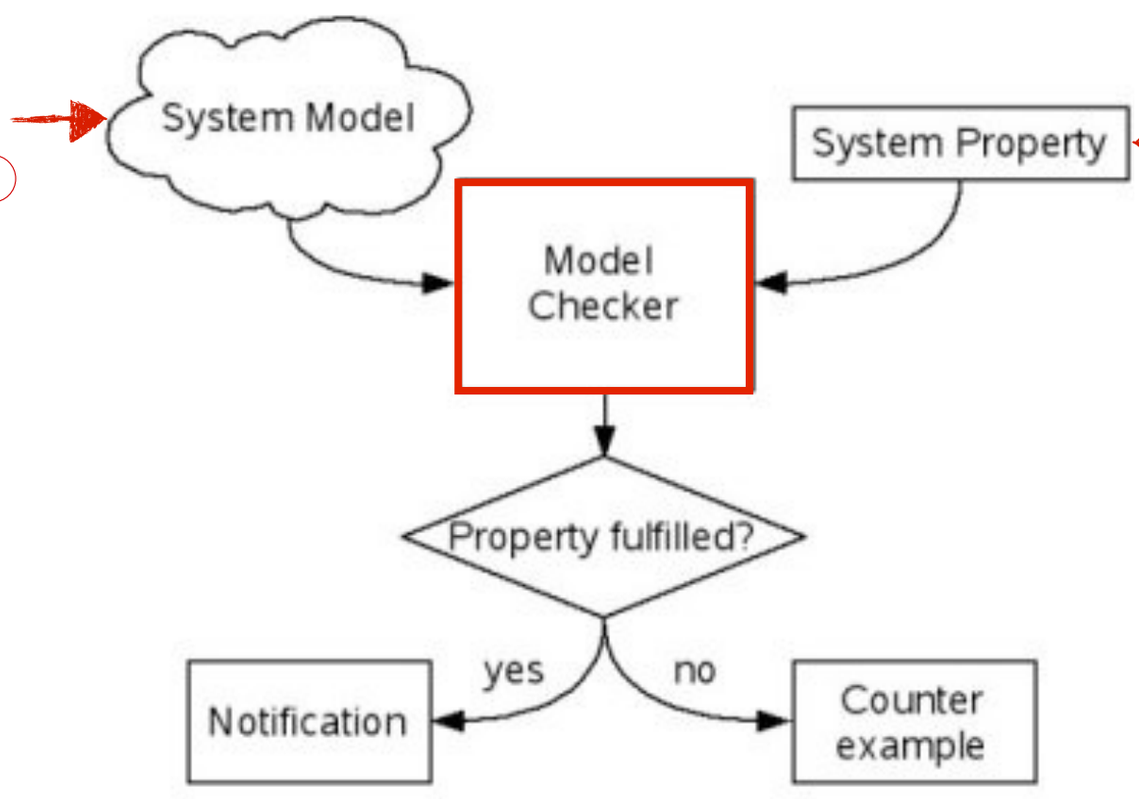
Model checking



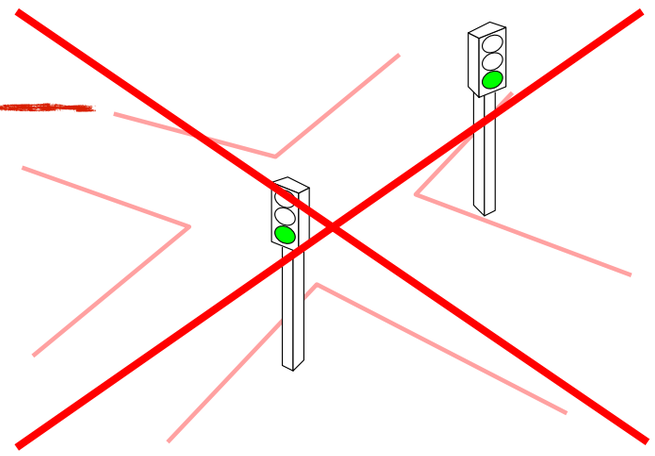
... a **static**, model-based, fully automatic, exhaustive technique for proving absence of errors in **finite-state** concurrent systems.

discrete system
as state-transition model





specification

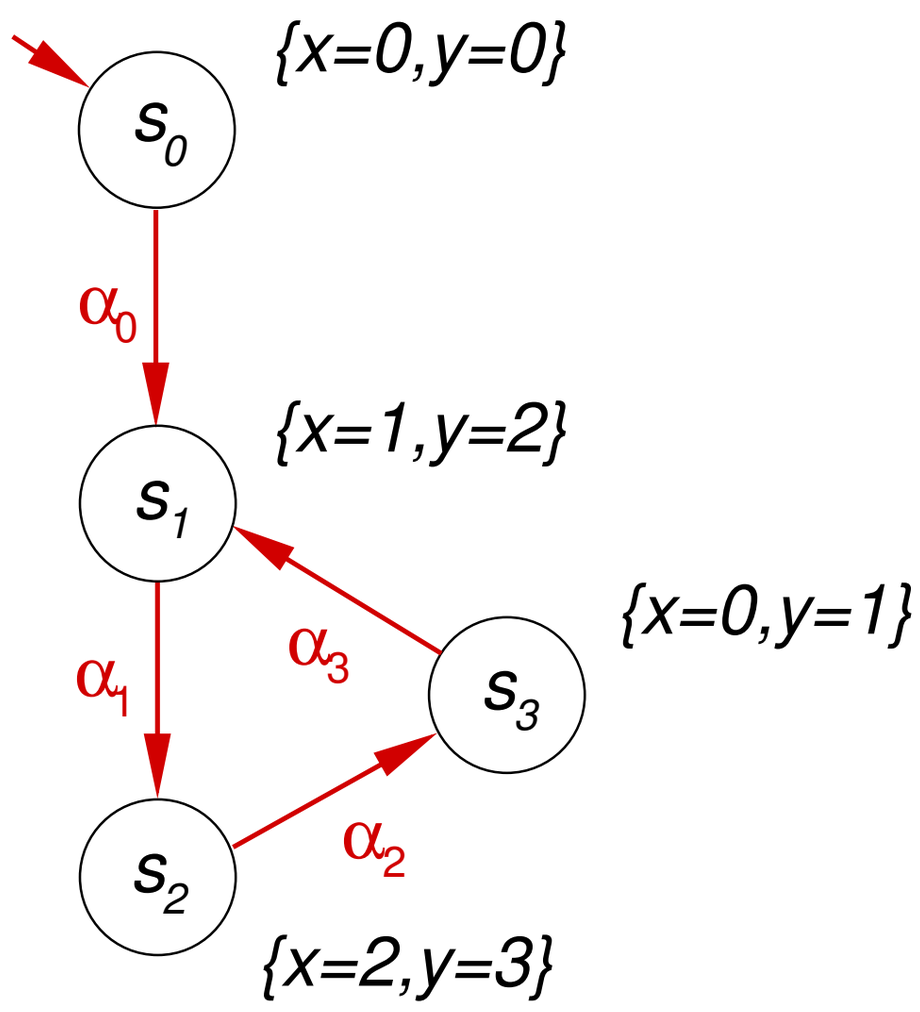


“the two traffic lights should never show a green light at the same time”

The **program** is modelled as a transition relation:

$$T \in S \times S$$

pre-state post-state



What specifications?

Take such a model, and ask whether states with the particular property, say, "ptr==0", are reachable. This is how you check for NULL-pointer dereferences.

In general, you can check for **properties** such as:

- Can the `assert(var != c)` be violated?
- Can the program **deadlock** or **livelock**?
- Can an array be accessed **out of bounds**?
- Can this variable **overflow**?
- Can a **division by zero** happen?

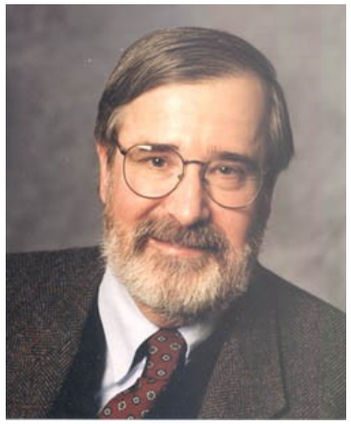
- **Temporal properties:**
  - Does `var == c` happen eventually?
  - Will `var2 > c` only happen after `var1 > c`?
  - Does `var == c` happen within 10 seconds from system boot?
  - Does `var == c` happen within 10 seconds from system boot with probability 99%?

In a nutshell: Model checking

Amir Pnueli

Introduced temporal logic into computing science for **program and systems verification**.

Turing Award 1996.

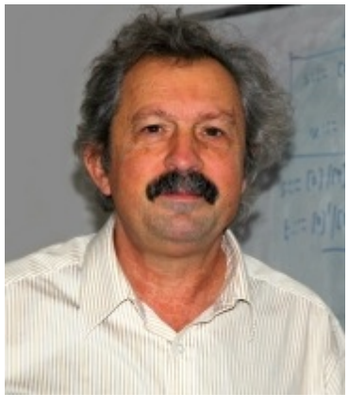


Ed Clarke

Allen Emerson

Determined validity of a LTL formula on a finite-state model.

Turing Award 2007.

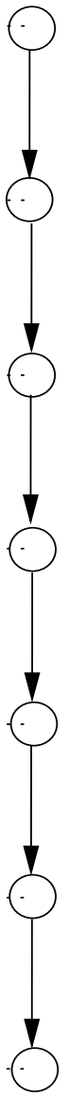


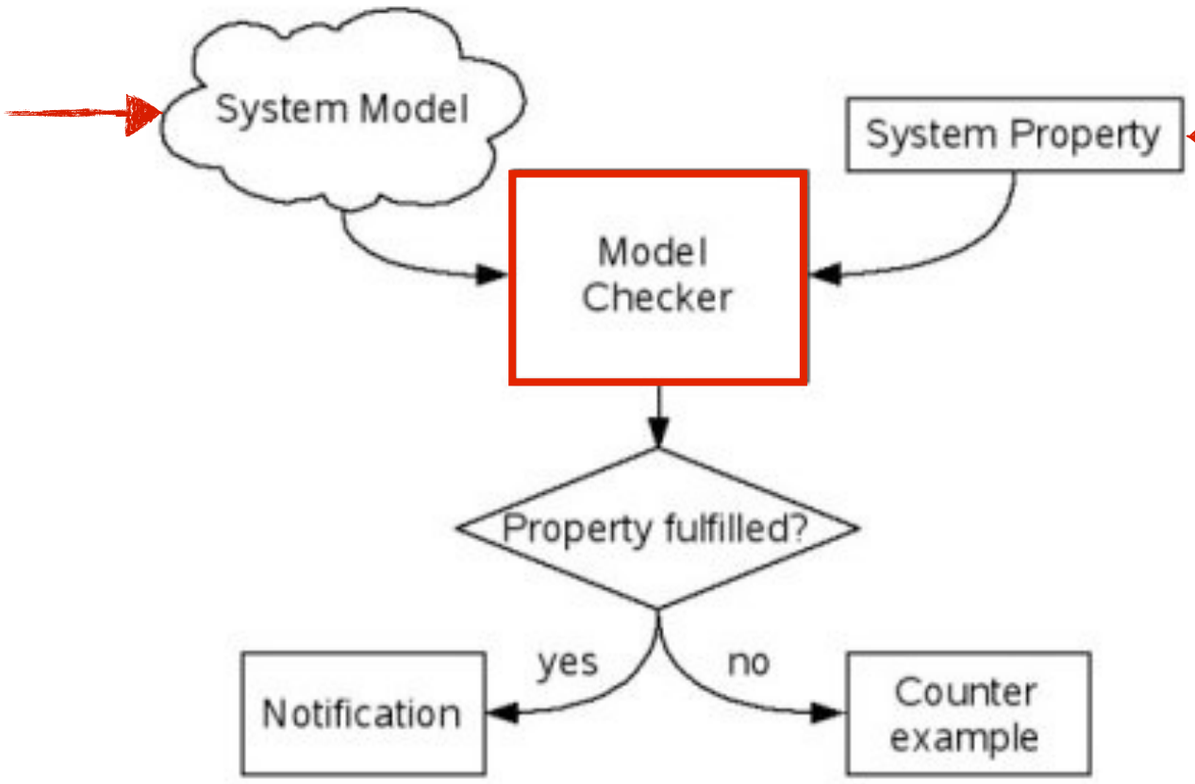
Joseph Sifakis

Runtime checking

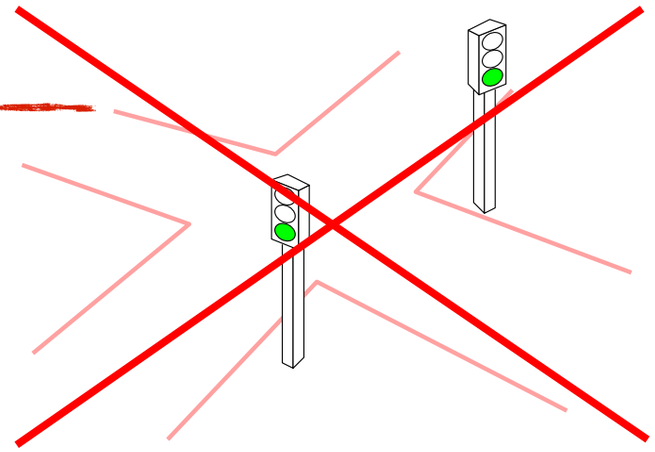
... is a **dynamic**, model-based, automatic, exhaustive technique for proving absence of errors in executions of systems.

system execution
as state-machine model





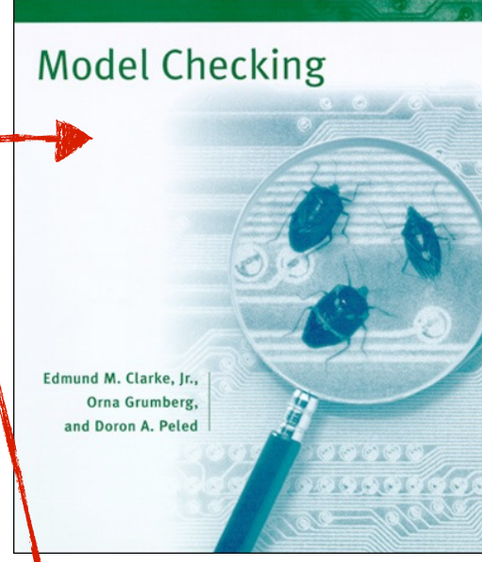
specification



“the two traffic lights should never show a green light at the same time”

In this course:

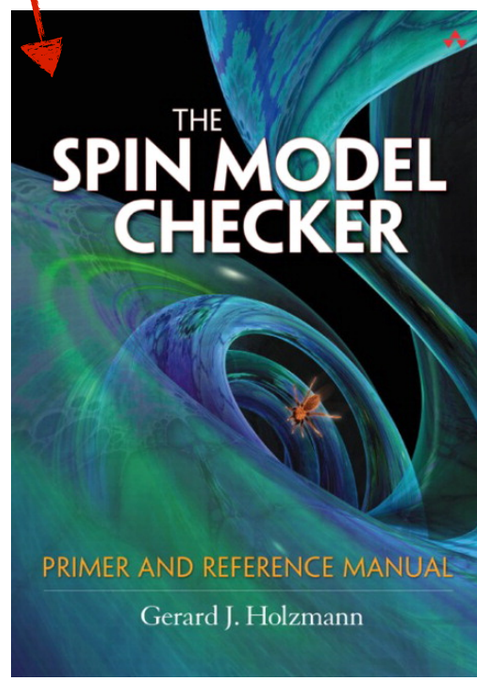
recommended

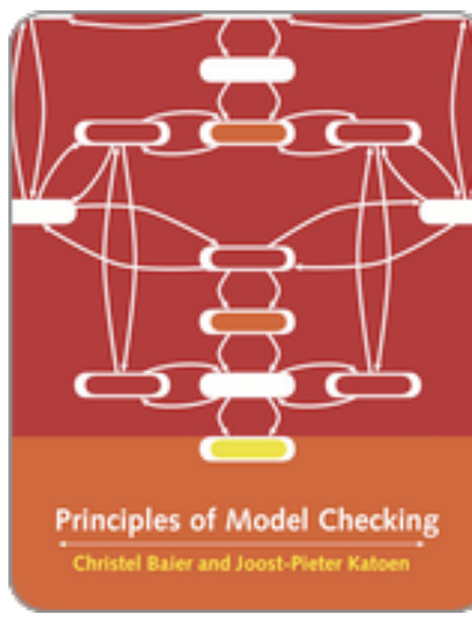


1. You get an intro to the principle and practice of model checking. You check a model against its specification both (i) **statically** (i.e., at system design-time or compile-time) and (ii) **dynamically** (i.e., at system runtime). Both build on the same fundamentals.

2. We cover **transition systems**, a **temporal logic** (LTL) to write specifications, and some **checking algorithms** and their complexity. We see how to automatically extract formal models out of sequential/ concurrent software.

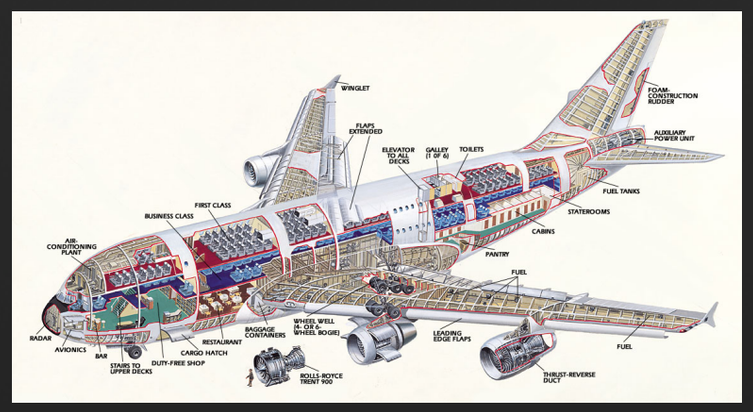
3. We take examples from the software domain, and use the model checkers **SPIN** (now developed at NASA JPL) and **CBMC** (Carnegie Mellon, University of Oxford).



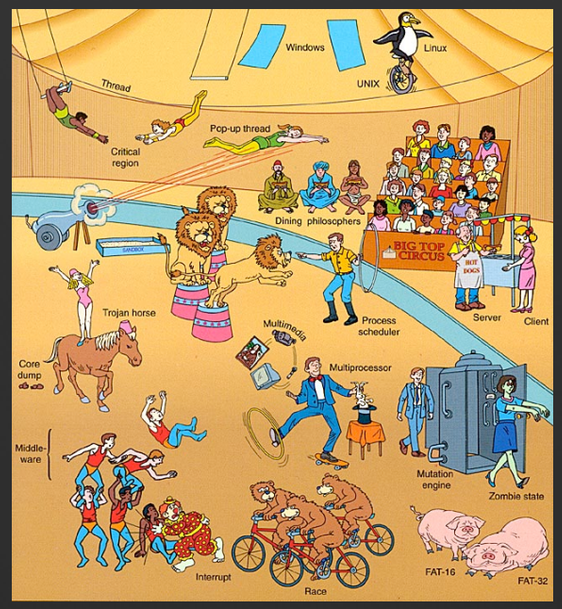


Some success stories for model checking

NASA runs in-house **formal methods** group**. SPIN** is developed there. Deep Space 1 controls were debugged with Spin.

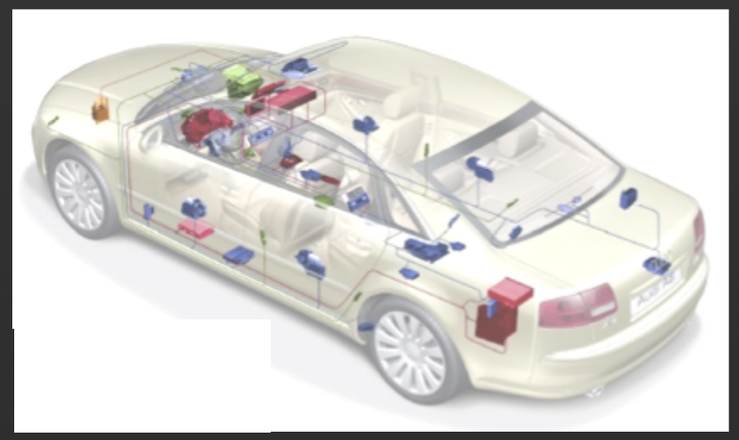


Prover Technologies does verification for **Airbus** control software, railway/metro interlocking systems.



Modern OSes and **drivers: Linux, MS Windows.** Microsoft Research develops and uses the SLAM model checker.

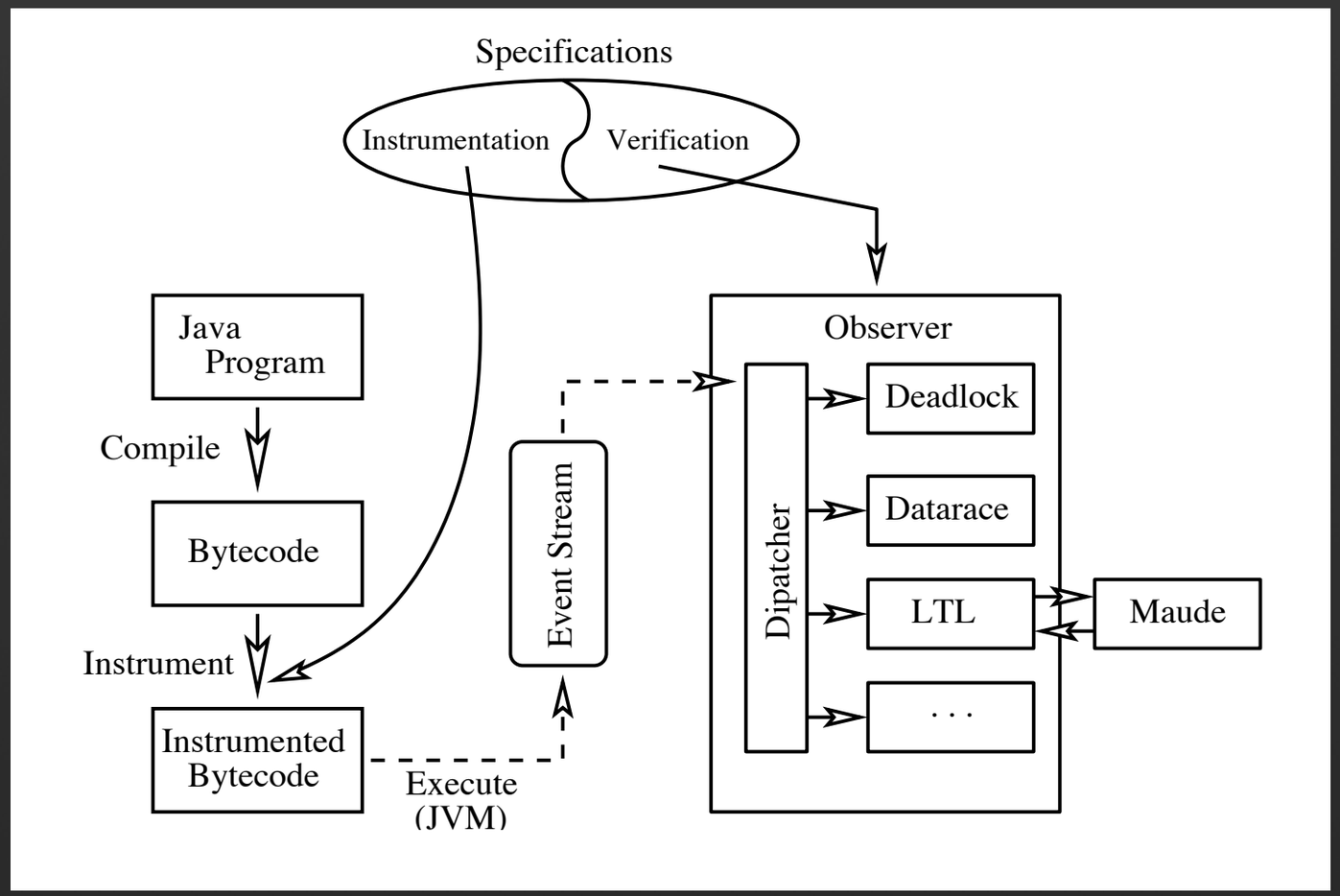
Part of the **Maeslantkering decision system** was debugged with SPIN.



BMW, GM, Toyota do some R&D of verification tools for **automotive software.**

Intel has a large in-house verification group.

Some **success** stories for runtime checking



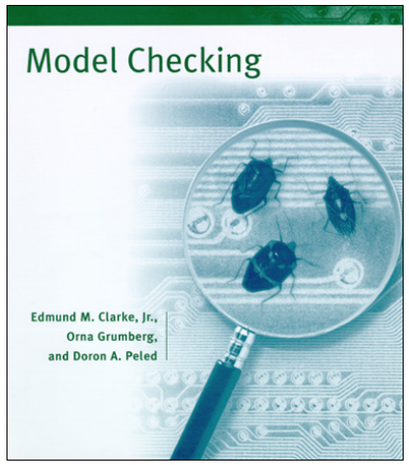
[**Java PathExplorer: A Runtime Verification Tool**, http://ti.arc.nasa.gov/m/pub-archive/archive/0262.pdf. Experimented with on the NASA Ames K9 Rover Executive.

Generally, search the Intelligent Systems Division at NASA for applications of verification: http://ti.arc.nasa.gov/publications/]

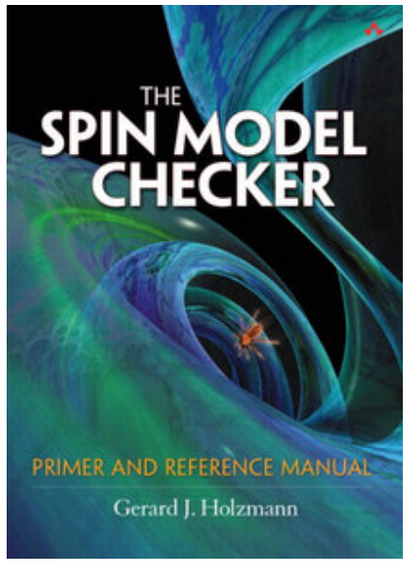
Course structure

1	Formal verification: The story about system correctness (this lecture)	
2	**Modelling** systems and model extraction	Exam topics
3	**Specifications** with Linear Temporal Logic (LTL)	
4-5	**Model checking** for LTL	
6	**Bounded model checking** with SAT	
7	**Runtime checking** for LTL	
8	The big picture: other techniques for system analysis and verification	

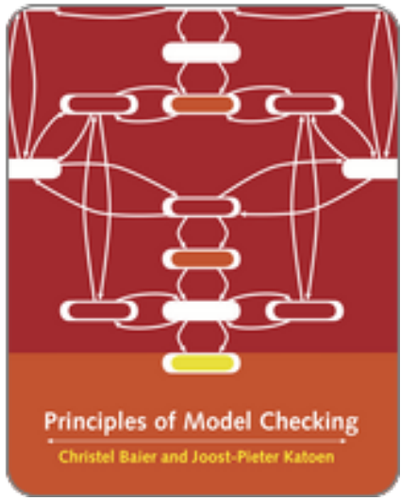
Where to read more:



Some of the content in this lecture is covered in Ch.1, *Introduction*, from *Model Checking*, E. M. Clarke, O. Grumberg, D. A. Peled.



You find a short history of SPIN and some motivation for model checking in Ch. 1, *Finding Bugs in Concurrent Systems*, from *The* SPIN *Model Checker*, G. J. Holzmann.



You may also read on the topic in Ch. 1, *System Verification*, from *Principles of Model Checking*, C. Baier, J.-P. Katoen.

Contact info and course administration

Course page: doina.net/AR.html

Contact: Doina Bucur <d.bucur@rug.nl>

Assignments: Individual. Have strict deadlines. I count on you trying to solve these problems through self-study; don't hesitate to resort to me for any sort of help.

Use the **lab sessions** as

- help desk for the assignments, and/or
- handing-in hour.