Model Checking

Lecture #1: Introduction, Background, Course Organisation

Joost-Pieter Katoen

Software Modeling and Verification Group

Model Checking Course, RWTH Aachen, WiSe 2019/2020

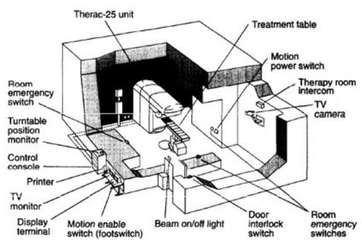
Overview

1. The Relevance of Software Reliability
2. Formal Verification
3. Model Checking in a Nutshell
4. Striking Model-Checking Examples
5. Course Organisation

Overview

1. The Relevance of Software Reliability
2. Formal Verification
3. Model Checking in a Nutshell
4. Striking Model-Checking Examples
5. Course Organisation

Software Reliability: Therac-25

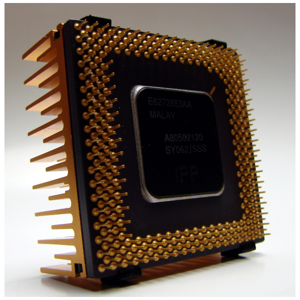


- Radiation machine for cancer patients
- At least 6 cases of overdosis (≈ factor 100) in 1985–1987
- Three cancer patients died
- Source: Design error in the control software: race condition
- Software written in assembly language

Software Reliability: Ariane 5 Flight 501

- Crash of European Ariane 5-missile in 1996
- Source: conversion from a 64-bit floating point to 16-bit signed integer
- Efficiency considerations had led to disabling of the software handler (in `Ada`)
- Overflow conditions crashed both primary and backup computers
- Costs: more than 500 million US$, 8 billion US$ development costs

Hardware Reliability: Pentium FDIV



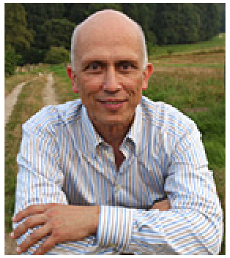
Pentium μprocessor

- FDIV = **f**loating point **div**ision unit
- Byte: 1 in 9 billion floating point divides with random parameters would produce inaccurate results
- Loss: ≈ 500 million US$[a] + serious image loss of Intel
- Source: flawless realisation of floating-point division

[a] all flawed processors were replaced

The Quest for Software Correctness

> "It is fair to state, that in this digital era correct systems for information processing are more valuable than gold."
>
> Speech@50-years CWI Amsterdam



Henk Barendregt

The Importance of Software Correctness

- Rapid increase of software in different applications
  - embedded systems
  - communication protocols
  - transportation systems

⇒ reliability increasingly depends on software!

- Defects can be fatal and extremely costly[1]
  - products subject to mass-production
  - safety-critical systems

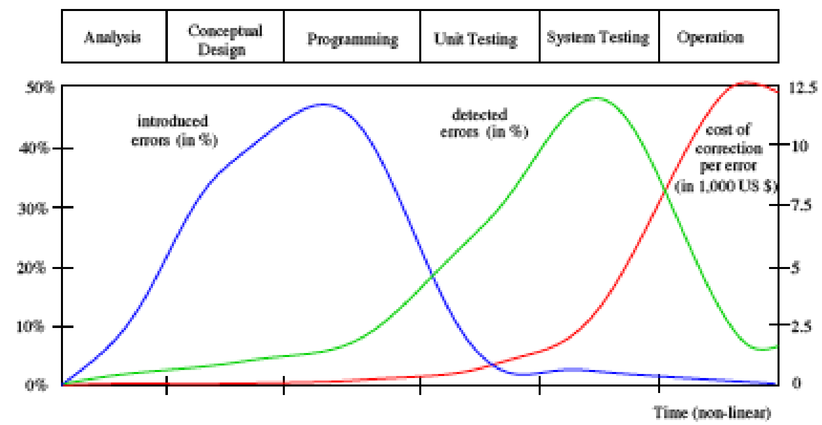
Software reliability is one of the grand challenges of the German Society of Computer Science.

[1] See `https://raygun.com/blog/costly-software-errors-history/`

Overview

1. The Relevance of Software Reliability
2. Formal Verification
3. Model Checking in a Nutshell
4. Striking Model-Checking Examples
5. Course Organisation

Bug Hunting: the Sooner, the Better



Formal Methods

Formal methods are:

- "applied mathematics for modelling and analysing ICT systems"

Formal methods offer a large potential for:

- obtaining an early integration of verification in the design process
- providing more effective verification techniques (higher coverage)
- reducing the verification time

Usage of formal methods:

- Highly recommended for safety-critical software by FAA, NASA, ...
- Required by ISO for autonomous vehicles at ASIL[2] Level D

[2] Automotive Safety Integrity Level

Formal Methods for Verifying Property φ

Deductive methods: provide a formal proof that φ holds

- tool: theorem prover (Isabelle/HOL, Coq, ...)
- applicable if: system has form of a mathematical theory
- pros: general applicable, high user involvement, hard guarantees

Model checking: systematic check on φ in all states

- tool: model checker (Spin, NuSMV, UppAal, ...)
- applicable if: system generates (finite) behavioural model
- pros: highly (fully) automatable, hard guarantees

Model-based testing: test for φ by program execution

- applicable if: system defines an executable model
- tool: text generation and execution (JTorX, RT-Tester, ...)
- pros: useful for finding bugs, not their absence

Milestones in Formal Verification

- Mathematical program correctness (Turing, 1949)
- Syntax-based technique for sequential programs (Hoare, 1968)
  - for a given input, does a program generate the correct output?
  - based on compositional proof rules expressed in predicate logic
- Syntax-based technique for concurrent programs (Pnueli, 1977)
  - handles properties referring to states during the computation
  - based on proof rules expressed in temporal logic
- Automated verification of concurrent programs (Clarke & Emerson 1981 Queille & Sifakis 1982)
  - model-based instead of proof-rule based approach
  - does the concurrent program satisfy a given (logical) property?

Overview

1 The Relevance of Software Reliability

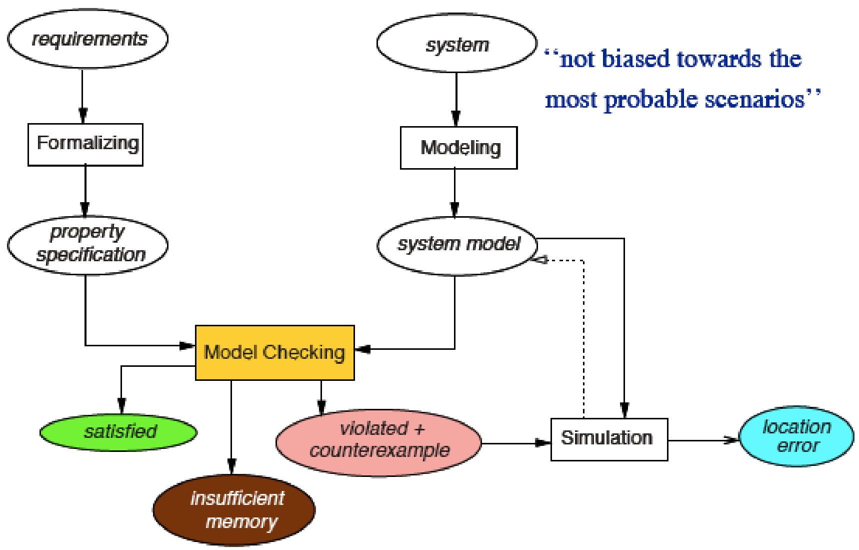
2 Formal Verification

3 Model Checking in a Nutshell

4 Striking Model-Checking Examples

5 Course Organisation

Model Checking Overview



Paris Kanellakis Theory and Practice Award 1998

Randal Bryant

Edmund Clarke

E. Allen Emerson

Ken McMillan

For their invention of "symbolic model checking," a method of formally checking system designs, which is widely used in the computer hardware industry and starts to show significant promise also in software verification and other areas.

Some other winners: Rivest et al., Paige and Tarjan, Buchberger, ...

Gödel Prize 2000

Moshe Vardi

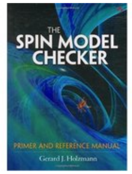
Pierre Wolper

"For work on model checking with finite automata."

Some other winners: Shor, Sénizergues, Agrawal et al., . . .

ACM System Software Award 2001

Gerard J. Holzmann



SPIN book

SPIN is a popular open-source software tool, used by thousands of people worldwide, that can be used for the formal verification of distributed software systems.

Some other winners: TeX, Postscript, UNIX, TCP/IP, Java, Smalltalk

ACM Turing Award 2007

Edmund Clarke

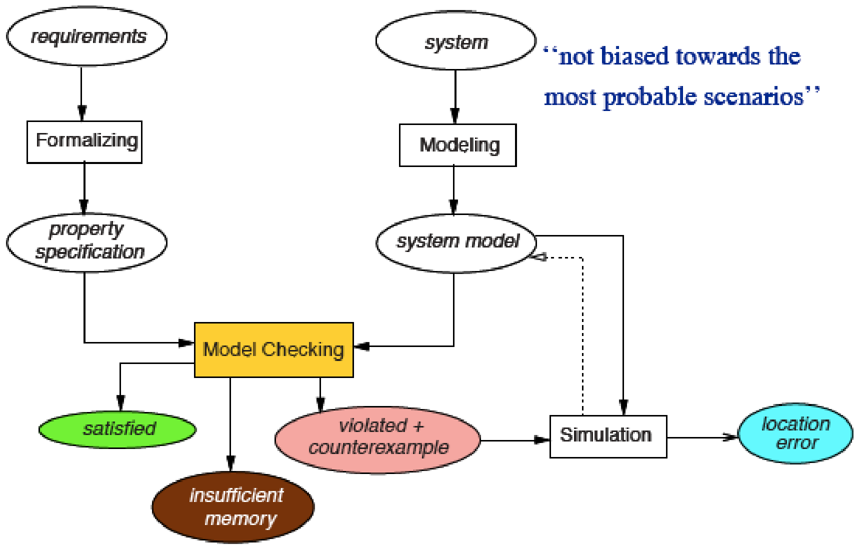
E. Allen Emerson

Joseph Sifakis

"For their role in developing Model-Checking into a highly effective verification technology, widely adopted in the hardware and software industries."

Some other winners: Dijkstra, Cook, Hoare, Rabin and Scott

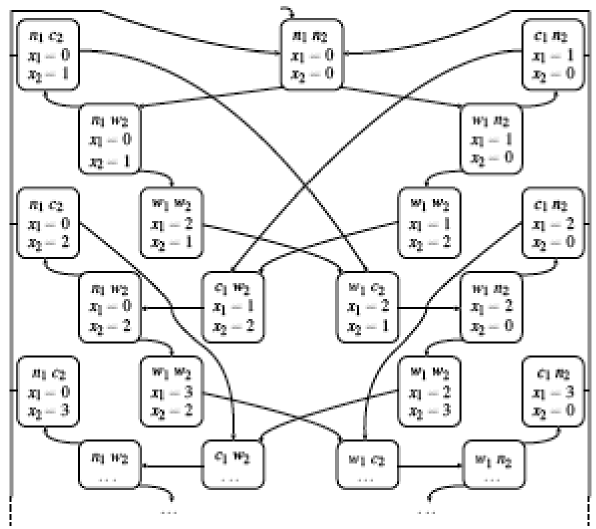
Model Checking Overview



What is Model Checking?

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

What are Models?



What are Models?

Transition systems

- States labelled with basic propositions
- Transition relation between states
- Action-labelled transitions to facilitate composition

Expressivity

- Programs are transition systems
- Multi-threaded programs are transition systems
- Hardware circuits are transition systems
- Petri nets are transition systems
- ......

What are Properties?

Example properties:

- Can the system reach a deadlock situation?
- Can two processes ever be simultaneously in a critical section?
- On termination, does a program provide the correct output?
- Can the system be reset in every possible system state?

Temporal logic:

- Propositional logic
- Modal operators such as $\Box$ “always” and $\Diamond$ “eventually”
- Interpreted over infinite state sequences (linear)
- Or over infinite trees of states (branching)

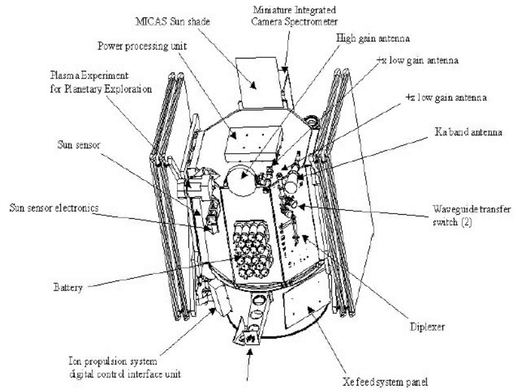
The Model Checking Problem

Let M be a model, i.e., a finite transition system.
Let φ be a formula in temporal logic, i.e., the specification.

Aim: Find all initial states s in M such that $M, s \models \varphi$.

Example: NASA's Deep Space-1 Spacecraft

Model checking has been applied to several modules of this spacecraft



launched in October 1998

A Program Snippet

process Inc = **while** *true* **do if** $x < 200$ **then** $x := x + 1$ **od**

process Dec = **while** *true* **do if** $x > 0$ **then** $x := x - 1$ **od**

process Reset = **while** *true* **do if** $x = 200$ **then** $x := 0$ **od**

is x always between (and including) 0 and 200?

Modelling in NanoPromela

```
int x = 0;

proctype Inc() {
  do :: true -> if :: (x < 200) -> x = x + 1 fi od
}

proctype Dec() {
  do :: true -> if :: (x > 0) -> x = x - 1 fi od
}

proctype Reset() {
  do :: true ->  if :: (x == 200) -> x = 0 fi od
}

init {
  atomic{ run Inc() ; run Dec() ; run Reset() }
}
```

How to Check?

Extend the model with a “monitor” process that checks $0 \leqslant x \leqslant 200$:

```
proctype Check() {
  assert (x >= 0 && x <= 200)
}

init {
  atomic{ run Inc() ; run Dec() ; run Reset() ; run Check() }
}
```


A Counterexample

```
..............
605: proc  1 (Inc)   line   9 "pan_in" (state 2) [((x<200))]
606: proc  1 (Inc)   line   9 "pan_in" (state 3) [x = (x+1)]
607: proc  3 (Dec) line 5 "pan_in" (state 2)    [((x > 0))]
608: proc  1 (Inc)   line   9 "pan_in" (state 1) [(1)]
609: proc  3 (Reset) line  13 "pan_in" (state 2) [((x==200))]
610: proc  3 (Reset) line  13 "pan_in" (state 3) [x = 0]
611: proc  3 (Reset) line  13 "pan_in" (state 1) [(1)]
612: proc  2 (Dec)   line   5 "pan_in" (state 3) [x = (x-1)]
613: proc  2 (Dec)   line   5 "pan_in" (state 1) [(1)]

spin: line  17 "pan_in", Error: assertion violated
spin: text of failed assertion: assert(((x>=0)&&(x<=200)))
```


Breaking the Error

```
int x = 0;

proctype Inc() {
  do :: true -> atomic{ if :: x < 200 -> x = x + 1 fi } od
}

proctype Dec() {
  do :: true -> atomic{ if :: x > 0 -> x = x - 1 fi } od
}

proctype Reset() {
  do :: true -> atomic{ if :: x == 200 -> x = 0 fi } od
}

init {
  atomic{ run Inc() ; run Dec() ; run Reset() }
}
```


The Model Checking Process

- Modeling phase
  - model the system under consideration
  - as a first sanity check, perform some simulations
  - formalise the property to be checked
- Execution phase
  - run the model checker to check the validity of the property in the model
- Analysis phase
  - property satisfied? → check next property (if any)
  - property violated? →
    1. analyse generated counterexample by simulation
    2. refine the model, design, or property . . . and repeat the entire procedure
  - out of memory? → try to reduce the model and try again

The Pros of Model Checking

- widely applicable (hardware, software, communication protocols, ...)
- potential "push-button" technology (software-tools)
  Uppaal, SPIN, NuSMV, CBMC, Java Pathfinder, Storm, ...
- increased usage in hardware and software industry
  Siemens, Amazon, FaceBook, Intel, Cadence, Ford, ESA, ...
- provides a counterexample if property is refuted
  model checking is an extremely effective bug-hunting technique
- sound mathematical foundations
  logic, automata, data structures and algorithms, complexity
- unlike testing, not biased to the most probable scenarios

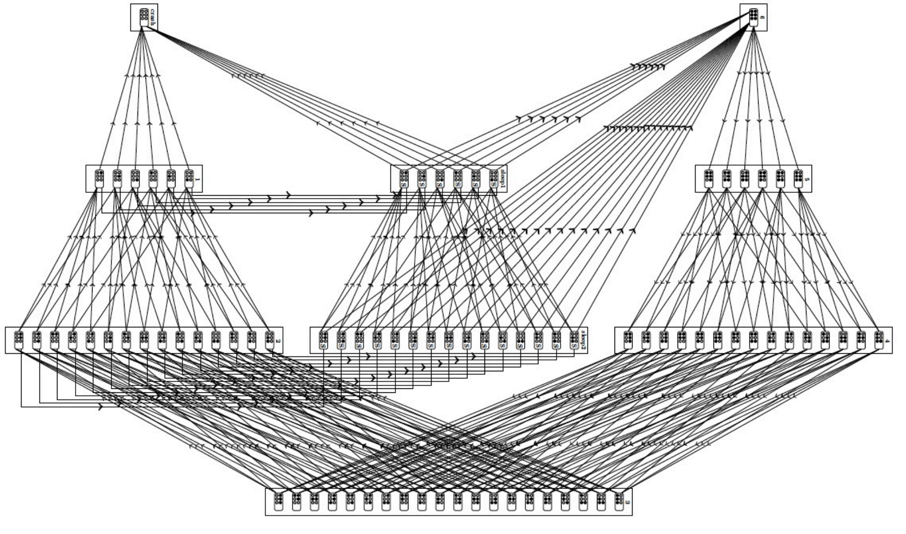
The Cons of Model Checking

- main focus on control-intensive applications (less data-oriented)
- model checking is only as "good" as the system model
- the state-space explosion problem
- mostly not possible to check generalisations

Nevertheless:

> Model checking is a very effective technique
> to expose potential design errors

State Spaces Can Be Gigantic

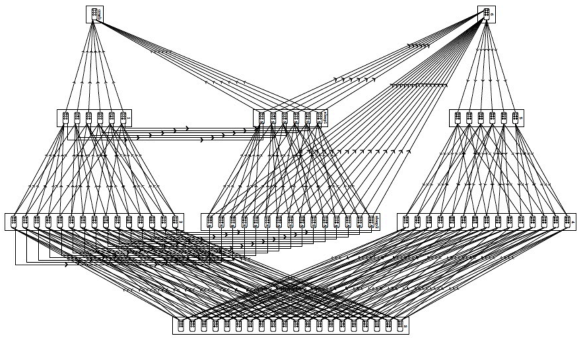


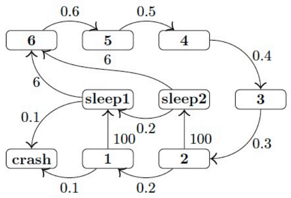
A model of the Hubble telescope

Treating Gigantic Models?

- Use compact data structures
- Make models smaller prior to (or: during) model checking
- Try to make them even smaller
- If possible, try to obtain the smallest possible model
- While preserving the properties of interest
- Do this all algorithmically and possibly fast

Abstraction





Gigantic versus smallest

Is a crash state reachable?	✔	✔
Is a failure repaired on time?	✖	✖

Overview

1 The Relevance of Software Reliability

2 Formal Verification

3 Model Checking in a Nutshell

4 Striking Model-Checking Examples

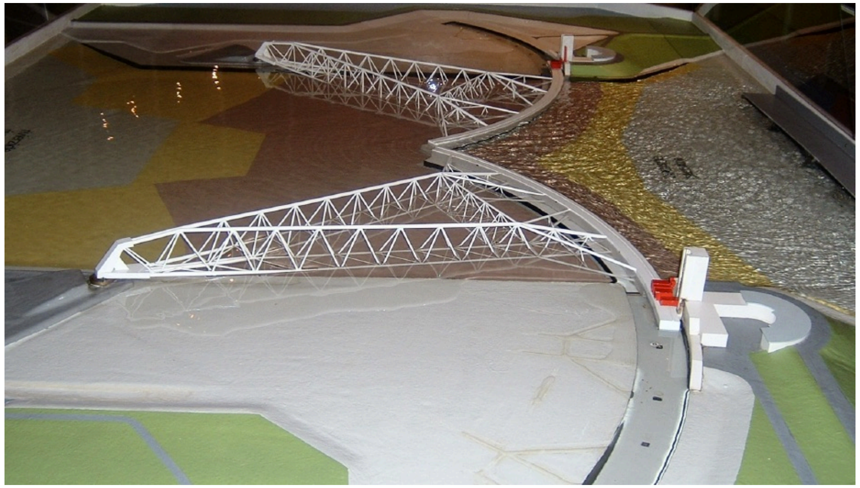
5 Course Organisation

Striking Model-Checking Examples

- Security: Needham-Schroeder encryption protocol
  - error that remained undiscovered for 17 years unrevealed
- Transportation systems
  - train model containing 10^{476} states
- Model checkers for C, Java and C++
  - used (and developed) by Microsoft, Digital, NASA
  - successful application area: device drivers
- Dutch storm surge barrier in Nieuwe Waterweg
- Software in the space missiles
  - NASA's Mars Curiosity Rover, Deep Space-1, Galileo
  - LARS group@Jet Propulsion Lab

Storm Surge Barrier Maeslantkering

Storm Surge Barrier Maeslantkering



Storm Surge Barrier Maeslantkering

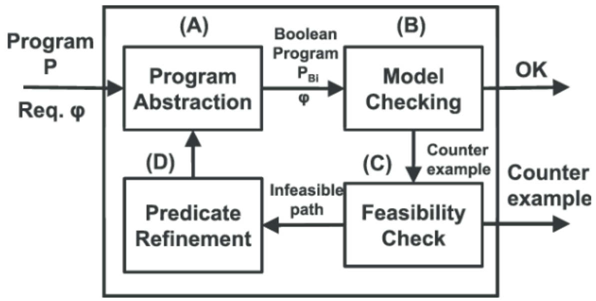
[Kars, Formal Methods in the Design of a Storm Surge Barrier Control System, 1996]

Checking Device Drivers

- 85% of system crashes of Windows XP caused by bugs in third-party kernel-level device drivers (2003)
- Main reason: complexity of the Windows drivers API
- SLAM model checker: automatically checks device drivers for certain correctness properties with respect to the Windows device drivers API
- Nowadays core of Static Driver Verifier (SDV), a tool-set for drivers developers

How to Model Check Device Drivers?

- Abstract `C` programs into Boolean programs
- Apply iterative abstraction-refinement scheme (CEGAR, see below)
- Key: recursive procedure calls (push-down automata)
- Symbolic model checking (binary decision diagrams)
- Points-to analysis + temporal safety properties (monitor)

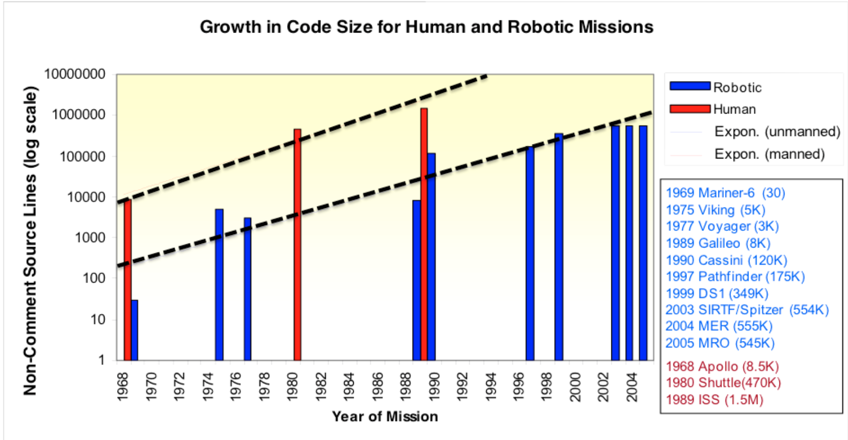


Checking Device Drivers

During development of Windows 7, 270 real bugs found
in 140 device drivers (of ≤ 30,000 lines of code) with SLAM

[Ball et al., A decade of software model checking with SLAM, 2011]

Spacecraft := Flying Software



NASA Study Flight Software Complexity (2009)

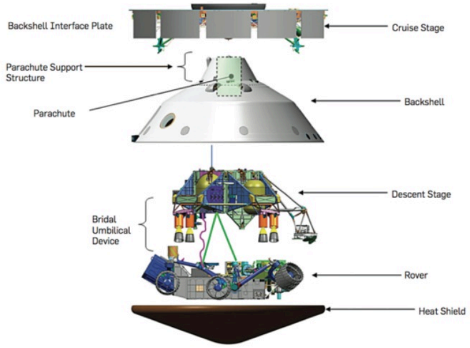
The NASA Curiosity Rover

Software in Space:

- Extremely high reliability requirements are imposed
  - Any small mistake can lead to the loss of a mission
- Extraordinary measures taken in both hardware and software design
  - system debugging and repair from millions of miles away
- Model checking verified intricate software subsystems for absence of races and deadlocks

Mars Rover Landing

The most critical part of the mission



Controlled by one of two computers
allocated within the body of the rover.

Model Checking of Mars Rover

Despite 145 code reviews, model checking of critical parts (e.g., file system) with SPIN revealed several subtle concurrency flaws.

Model checking was used in the design loop: performed routinely after every change in the code of the file system.

[Holzmann, Mars Code,2014]

Model Checking@FaceBookg

Facebook Acquires Assets of U.K. Software Startup Monoidics

JULY 18, 2013 AT 8:34 AM PT

Tweet | Share | +1 | Share | Share | Print

Facebook announced on Thursday that the company has acquired the assets of Monoidics, a software-verification company based in the United Kingdom.

As it is not a full company acquisition, only Monoidics' technical team will be joining Facebook's London engineering offices after the deal closes.

"In 2009 we started this company with the goal of making the best automatic formal verification and analysis software in the industry," Monoidics sa

Facebook

Monoidics creates programs that check other software for bugs and problems, and Facebook will use Monoidics' tools to help improve development of Facebook's mobile products. That's obviously an area Facebook is focused on, as the company has sped up product development for its Android and iOS apps over the past few years, due to the shifting of the industry toward mobile.

[Calcagno et al., Moving fast wirh software verification, 2014]

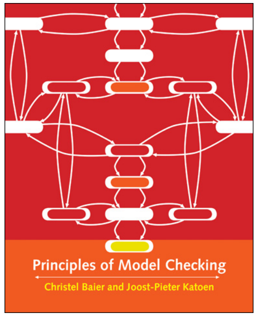
Overview

1. The Relevance of Software Reliability
2. Formal Verification
3. Model Checking in a Nutshell
4. Striking Model-Checking Examples
5. Course Organisation

Course Content

- What are transition systems and properties?
- Model checking linear temporal logic
  automata on infinite words, regular properties, complexity
- Model checking branching-time logic
  CTL, expressiveness CTL versus LTL, recursive descent
- How to make models smaller?
  - bisimulation minimisation, simulation, partial-order reduction, CEGAR
- Symbolic model checking
  - binary decision diagrams, bounded model checking, PDR

Course Material



Principles of Model Checking:

Christel Baier

TU Dresden, Germany

Joost-Pieter Katoen

RWTH Aachen University, Germany

Gerard J. Holzmann, NASA JPL:

"This book offers one of the most comprehensive introductions to logic model checking techniques available today. The authors have found a way to explain both basic concepts and foundational theory thoroughly and in crystal clear prose."

Lectures

Lecture:

- Thu 10:30 - 12:00 (AH 2), Fri 14:30-16:00 (AH 3)
- Check regularly RWTH Moodle for possible "no shows"

Material:

- Lecture slides are made available on RWTH Moodle
- Many copies of the book are available in the CS library

Website:

`moves.rwth-aachen.de/teaching/ws-19-20/introduction-to-model-checking/`

Exercises and Examination

Exercise classes:

- Fri 10:30 - 12:00 in 5056 (start: Oct 25)
- Instructors: Sebastian Junges and Lutz Klinkenberg

Weekly exercise series:

- Intended for groups of three students
- New series: every Friday on course web page (start: Oct 18)
- Solutions: Friday (before 10:00) one week later
- Participation to exercises strongly encouraged
- Starred exercises are example exam questions

Examination:

- February 20, 2020 and March 13, 2020 (written exam)
- No particular pre-requisites for exam participation

Course Prerequisites

Aim of the course:

It's about the theoretical foundations of model checking.

Not its usage.

Prerequisites:

- Automata and language theory
- Algorithms and data structures
- Computability and complexity theory
- Mathematical logic

Related Courses

- Modelling and verification of probabilistic systems (Katoen)
- Probabilistic programming (Katoen)
- Automata on infinite words (Löding)
- Satisfiability checking (Abráhám)
- Modelling and analysis of hybrid systems (Abráhám)
- Theoretical Foundations of the UML (Katoen)
- Semantics and Verification of Software (Noll)

As well as various master and bachelor theses

Questions?

Next Lecture: Friday October 11, 14:30