Introduction to Model Checking

Lecture # 1: Motivation, Background, and Course Organization

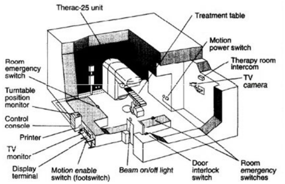
Prof. Dr. Ir. Joost-Pieter Katoen

Lehrstuhl Software Modellierung and Verifikation

RWTH AACHEN UNIVERSITY

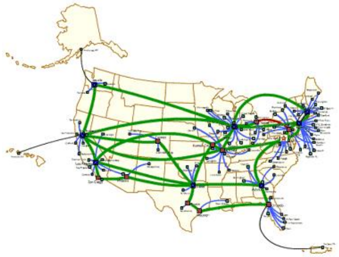
October 21, 2008

Therac-25 Radiation Overdosing (1985-87)



- Radiation machine for treatment of cancer patients
- At least 6 cases of overdosis in period 1985–1987 ($\approx$ 100-times dosis)
- Three cancer patients died
- Source: Design error in the control software (*race condition*)

AT&T Telephone Network Outage (1990)

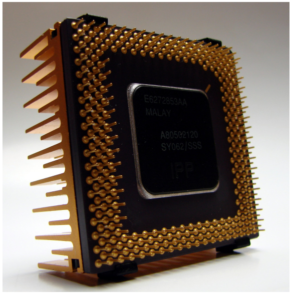


- January 1990: problem in New York City leads to 9 h-outage of large parts of U.S. telephone network
- Costs: several 100 million US$
- Source: software flaw (wrong interpretation of `break` statement in `C`)

Ariane 5 Crash (1996)

- Crash of the european Ariane 5-missile in June 1996
- Costs: more than 500 million US$
- Source: software flaw in the control software
- A data conversion from a 64-bit floating point to 16-bit signed integer
- Efficiency considerations had led to the disabling of the software handler (in `Ada`)

Pentium FDIV Bug (1994)

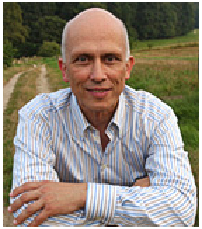


- FDIV = **f**loating point **div**ision unit
- Certain floating point division operations performed produced incorrect results
- Byte: 1 in 9 billion floating point divides with random parameters would produce inaccurate results
- Loss: $\approx$ 500 million US$ (all flawed processors were replaced) + enormous image loss of Intel Corp.
- Source: flawless realization of floating-point division

The Quest for Software Correctness

Speech@50-years Celebration CWI Amsterdam

"It is fair to state, that in this digital era correct systems for information processing are more valuable than gold."


Henk Barendregt

The Importance of Software Correctness

Rapidly increasing integration of ICT in different applications

- embedded systems
- communication protocols
- transportation systems

$\Rightarrow$ reliability incrasingly depends on software!

Defects can be fatal and extremely costly

- products subject to mass-production
- safety-critical systems

What is System Verification?

Folklore "definition"

System verification amounts to check whether a system fulfills the qualitative requirements that have been identified

Verification $\neq$ validation

- Verification = "check that we are building the thing right"
- Validation = "check that we are building the right thing"

Software Verification Techniques

Peer reviewing

- static technique: manual code inspection, no software execution
- detects between 31 and 93% of defects with median of about 60%
- subtle errors (concurrency and algorithm defects) hard to catch

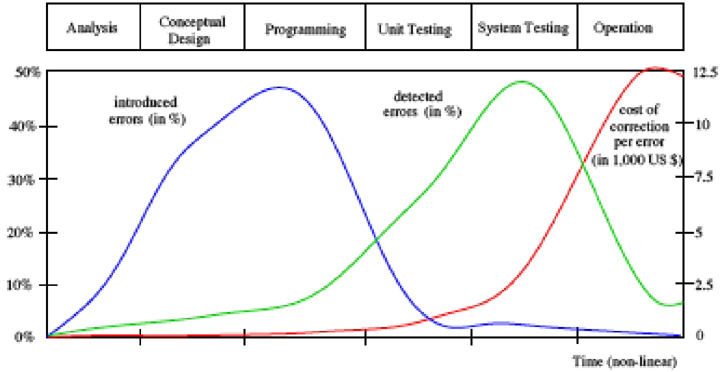
Testing

- dynamic technique in which software is executed

Some figures

- 30% to 50% of software project costs devoted to testing
- more time and effort is spent on validation than on construction
- accepted defect density: about 1 defects per 1,000 code lines

Bug Hunting: the Sooner, the Better



Formal Methods

Intuitive description

Formal methods are the

"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 by IEC, FAA, and NASA for safety-critical software

Formal Verification Techniques for Property P

Deductive methods

- method: provide a formal proof that P holds
- tool: theorem prover/proof assistant or proof checker
- applicable if: system has form of a mathematical theory

Model checking

- method: systematic check on P in all states
- tool: model checker (Spin, NuSMV, UppAal, ...)
- applicable if: system generates (finite) behavioural model

Model-based simulation or testing

- method: test for P by exploring possible behaviours
- applicable if: system defines an executable model

Simulation and Testing

Basic procedure:

- take a model (simulation) or a realisation (testing)
- stimulate it with certain inputs, i.e., the tests
- observe reaction and check whether this is "desired"

Important drawbacks:

- number of possible behaviours is very large (or even infinite)
- unexplored behaviours may contain the fatal bug

About testing . . .

testing/simulation can show the presence of errors, not their absence

Milestones in Formal Verification

- Mathematical program correctness (Turing, 1949)
- Syntax-based technique for sequential programs (Hoare, 1969)
  - for a given input, does a computer 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
  - model-based instead of proof-rule based approach
  - does the concurrent program satisfy a given (logical) property?

Example Proof Rules

Backward axiom

$$\frac{}{\{\mathcal{A}[e/x]\}\ x := e\ \{\mathcal{A}\}}$$

Invariant rule

$$\frac{\{\mathcal{I} \wedge b\}\ P\ \{\mathcal{I}\}}{\{\mathcal{I}\}\ \textbf{while}\ b\ \textbf{do}\ P\ \{\mathcal{I} \wedge \neg b\}}$$

Cut rule

$$\frac{\{\mathcal{A}\}\ P\ \{\mathcal{B}\} \qquad \{\mathcal{B}\}\ Q\ \{\mathcal{C}\}}{\{\mathcal{A}\}\ P; Q\ \{\mathcal{C}\}}$$

Logical rule

$$\frac{\mathcal{A} \Rightarrow \mathcal{A}' \quad \{\mathcal{A}'\}\ P\ \{\mathcal{B}'\} \quad \mathcal{B}' \Rightarrow \mathcal{B}}{\{\mathcal{A}\}\ P\ \{\mathcal{B}\}}$$

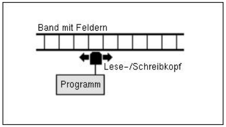
The ACM Turing Award

Alan M. Turing (1912 - † 1954)

- Mathematician, logician, crypto-specialist
- Computational model: Turing Machine

Some Turing Award Winners

- Edsger Dijkstra (1972)
- Donald Knuth (1974)
- Michael Rabin and Dana Scott (1976)
- Stephen Cook (1982)
- Rivest, Shamir and Adleman (2002)



ACM Turing Award 2007

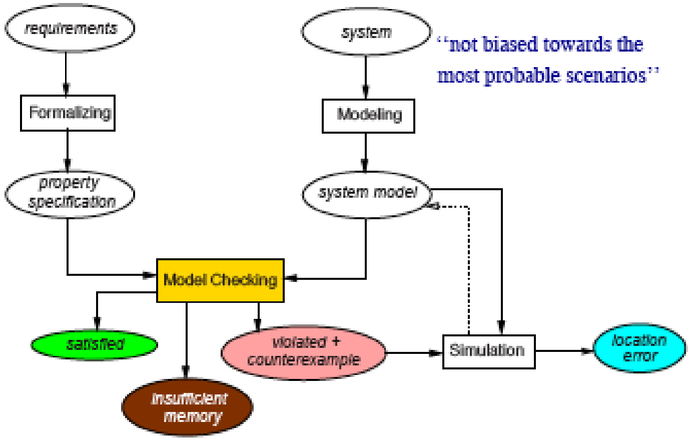
Recipients in February 2008

- Edmund M. Clarke jr. (CMU, USA)
- Allen E. Emerson (Texas at Austin, USA)
- Joseph Sifakis (IMAG Grenoble, F)

Jury justification

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

Model Checking Overview



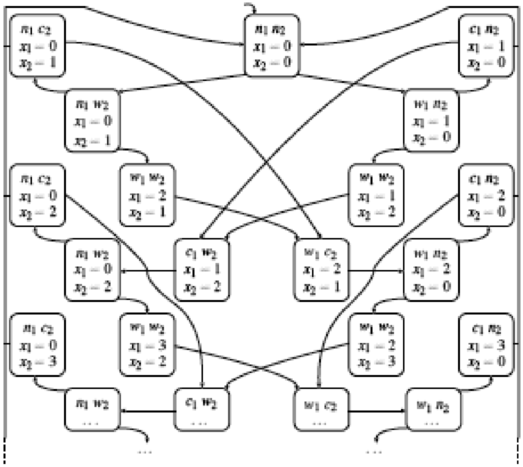
“not biased towards the most probable scenarios”

What is Model Checking?

Informal description

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?


$n_1\ c_2$ $x_1 = 0$ $x_2 = 1$
$n_1\ n_2$ $x_1 = 0$ $x_2 = 0$
$c_1\ n_2$ $x_1 = 1$ $x_2 = 0$
$n_1\ w_2$ $x_1 = 0$ $x_2 = 1$
$w_1\ n_2$ $x_1 = 1$ $x_2 = 0$
$n_1\ c_2$ $x_1 = 0$ $x_2 = 2$
$w_1\ w_2$ $x_1 = 2$ $x_2 = 1$
$w_1\ w_2$ $x_1 = 1$ $x_2 = 2$
$c_1\ n_2$ $x_1 = 2$ $x_2 = 0$
$n_1\ w_2$ $x_1 = 0$ $x_2 = 2$
$c_1\ w_2$ $x_1 = 1$ $x_2 = 2$
$w_1\ c_2$ $x_1 = 2$ $x_2 = 1$
$w_1\ n_2$ $x_1 = 2$ $x_2 = 0$
$n_1\ c_2$ $x_1 = 0$ $x_2 = 3$
$w_1\ w_2$ $x_1 = 3$ $x_2 = 2$
$w_1\ w_2$ $x_1 = 2$ $x_2 = 3$
$c_1\ n_2$ $x_1 = 3$ $x_2 = 0$
$n_1\ w_2$ ...
$c_1\ w_2$ ...
$w_1\ c_2$ ...
$w_1\ n_2$ ...
...
...

What are Models?

Transition systems

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

Expressivity

- Programs are transition systems
- Multi-threading programs are transition systems
- Communicating processes are transition systems
- Hardware circuits are transition systems
- What else?

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?

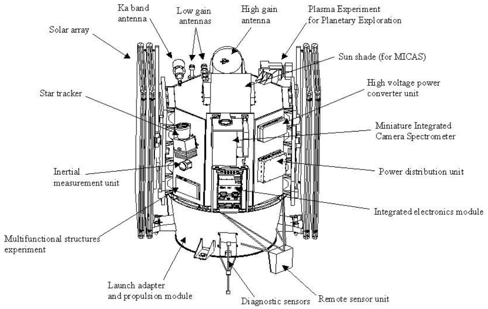
Temporal logic

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

NASA's Deep Space-1 Spacecraft

Model checking

has been applied to several modules of this spacecraft


Ka band antenna
Low gain antennas
High gain antenna
Plasma Experiment for Planetary Exploration
Solar array
Sun shade (for MICAS)
High voltage power converter unit
Star tracker
Miniature Integrated Camera Spectrometer
Inertial measurement unit
Power distribution unit
Integrated electronics module
Multifunctional structures experiment
Launch adapter and propulsion module
Diagnostic sensors
Remote sensor unit

launched in October 1998

A Small Program Fragment

$$\begin{array}{c}
\textbf{process } \mathrm{Inc} = \textbf{while } \mathit{true} \textbf{ do if } x < 200 \textbf{ then } x := x + 1 \textbf{ od} \\
\textbf{process } \mathrm{Dec} = \textbf{while } \mathit{true} \textbf{ do if } x > 0 \textbf{ then } x := x - 1 \textbf{ od} \\
\textbf{process } \mathrm{Reset} = \textbf{while } \mathit{true} \textbf{ do if } x = 200 \textbf{ then } x := 0 \textbf{ od}
\end{array}$$

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

Modeling 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() }
}
```

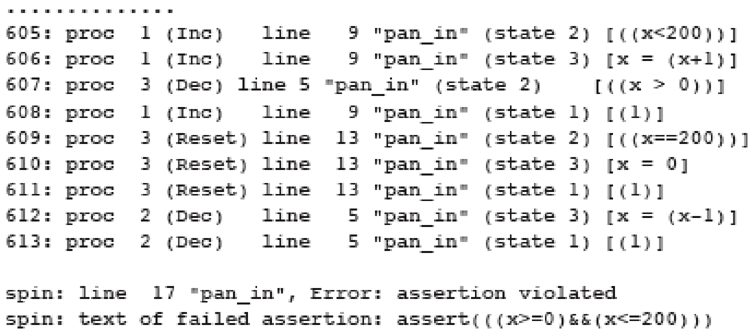
How to Check?

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
- Running phase
  - run the model checker to check the validity of the property in the model
- Analysis phase
  - property satisfied? $\longrightarrow$ check next property (if any)
  - property violated? $\longrightarrow$
    1. analyse generated counterexample by simulation
    2. refine the model, design, or property . . . and repeat the entire procedure
  - out of memory? $\longrightarrow$ try to reduce the model and try again

The Pros of Model Checking

- widely applicable (hardware, software, protocol systems, ...)
- allows for partial verification (only most relevant properties)
- potential “push-button” technology (software-tools)
- rapidly increasing industrial interest
- in case of property violation, a counterexample is provided
- sound and interesting mathematical foundations
- not biased to the most possible scenarios (such as testing)

The Cons of Model Checking

- main focus on control-intensive applications (less data-oriented)
- model checking is only as “good” as the system model
- no guarantee about completeness of results
- impossible to check generalisations (in general)

Nevertheless:

Model checking is a effective technique to expose potential design errors

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 current/next generation of space missiles
  - NASA's Mars Pathfinder, Deep Space-1, JPL LARS group

Course Topics

What are appropriate models?

- transition systems
- from programs to transition systems
- from circuits to transition systems
- multi-threading, communication, . . .
- `nanoPromela`: an example modeling language

What are properties?

- safety: "something bad never happen"
- liveness: "something good eventually happens"
- fairness: "if something may happen frequently, it will happen"

Course Topics

How to check regular properties?

- finite-state automata and regular safety properties
- Büchi automata and ω-regular properties
- model checking: nested depth-first search

How to express properties succinctly?

- Linear-time Temporal Logic (LTL): syntax and semantics
- What can be expressed in LTL?
- LTL model checking: algorithms, complexity
- How to treat fairness in LTL

Course Topics

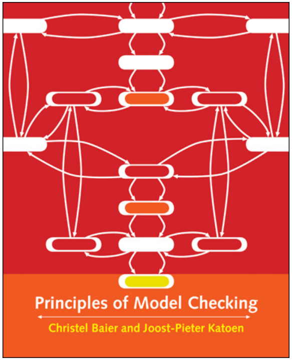
How to express properties succinctly?

- Computation Tree Logic (CTL): syntax and semantics
- What can be expressed in CTL?
- CTL model checking: algorithms, complexity
- How to treat fairness in CTL

How to make models smaller?

- Equivalences and pre-orders on transition systems
- Which properties are preserved?
- Minimization algorithms

Course Material


Principles of Model Checking
Christel Baier and Joost-Pieter Katoen

Principles of Model Checking

Christel Baier

TU Dresden, Germany

Joost-Pieter Katoen

RWTH Aachen University, Germany, and
University of Twente, the Netherlands

Gerard J. Holzmann, NASA JPL, Pasadena:

"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

- Tue 14:00 - 15:30, Wed 10:00-11:30 in AH2
- Check regularly course webpage for possible "no shows"

Material

- Lecture slides (with gaps) are made available on webpage
- Copies of the book are available in the CS library

Website

`moves.rwth-aachen.de/i2/249`

Exercises and Exam

Exercise Classes

- Fri 10:00 - 11:30 in AH2 (start: October 31)
- Instructor: Martin Neuhäusser

Weekly exercise series

- Intended for groups of 2 students
- New series: every Fri on course webpage (start: October 24)
- Solutions: Fri (before 10:00) one week later
- Student assistants: Denise Nimmerrichter and Stefan Herting

Exam:

- February 13, 2009 (written exam)
- participation if at least 50% of all points in the weekly exercises are gathered (BSc and MSc-students)

Course Embedding

Aim of the course

It's about the foundations of model checking, not its usage!

Prerequisites

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

Some follow-up courses

- Advanced model checking (SS 2009)
- Practical exercises model checking (SS 2009)
- Automata and reactive systems (Thomas)
- Satisfiability checking (Abrahám)