Introduction

Lecture #1 of Model Checking

Joost-Pieter Katoen

Lehrstuhl 2: Softwaremodeling and Verification

E-mail: `katoen@cs.rwth-aachen.de`

April 3, 2007

Overview

⇒ *On the role of system verification*

- *Formal verification techniques*
  - model-based testing
  - simulation
  - deductive approaches
- *Model checking*
- *Course objectives and planning*

The quest for software verification

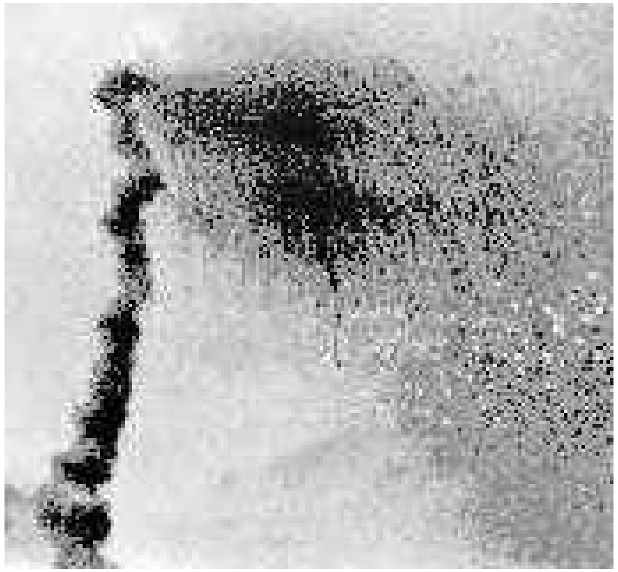
It is fair to state, that in this digital era correct systems for information processing are more valuable than gold.

Henk Barendregt (1996)

The importance of software correctness

- Rapidly increasing *integration of ICT* in different applications:
  - embedded systems
  - communication protocols
  - transportation systems
- Reliability increasingly depends on hard- and software *integrity*
- Defects can be *fatal* and extremely *costly*
  - products subject to mass-production
  - safety-critical systems

A famous example



The Ariane-5 launch on June 4, 1996; it crashed 36 seconds after the launch due to a conversion of a 64-bit floating point into a 16-bit integer value

What is system verification?

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

Verification $\neq$ validation:

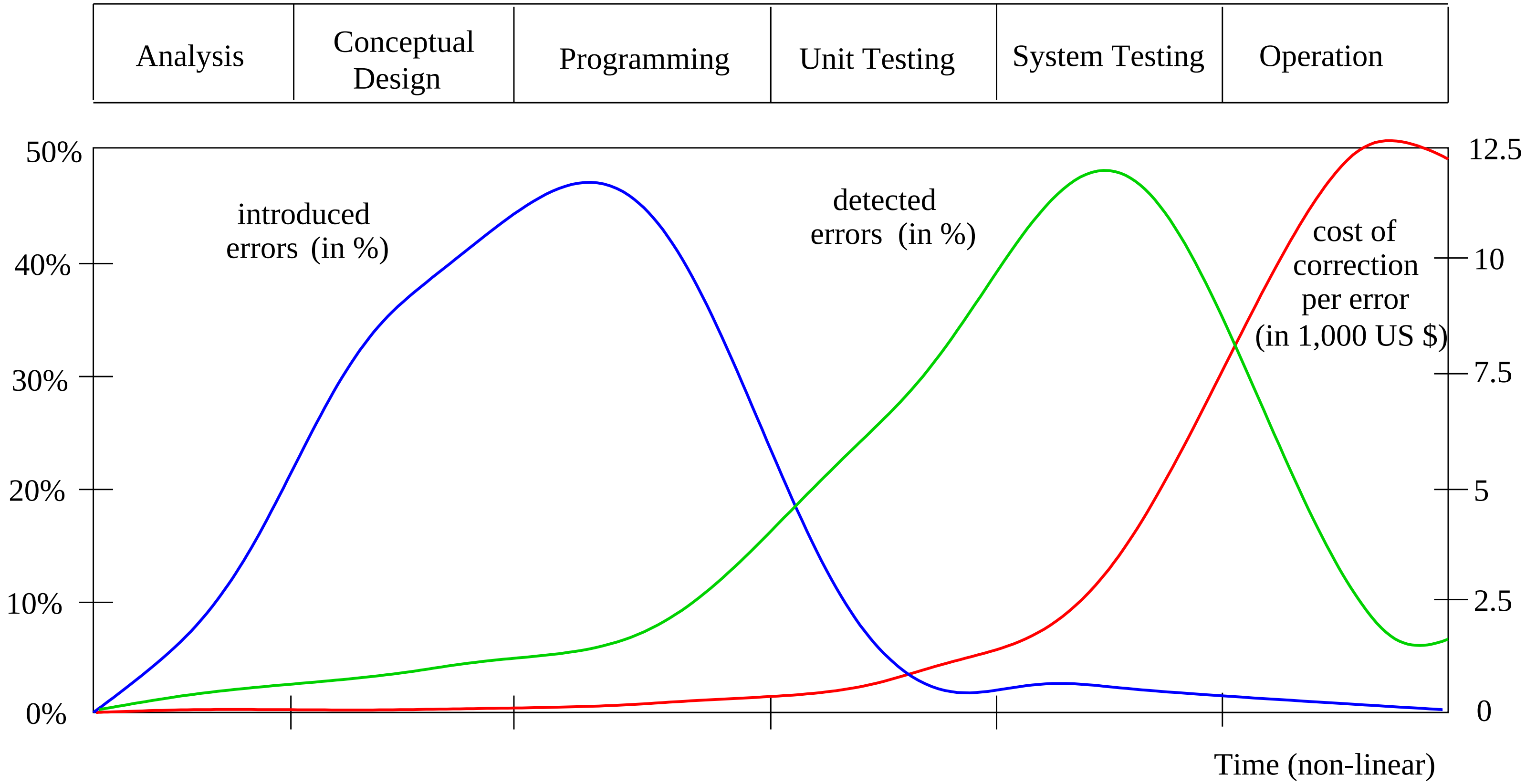
Verification = "check that we are building the thing *right*"

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

Catching software bugs: the sooner, the better



Overview

- *On the role of system verification*

$\Rightarrow$ *Formal verification techniques*

  - model-based testing
  - simulation
  - deductive approaches

- *Model checking*

- *Course objectives and planning*

Formal methods

Formal methods are the

"applied mathematics for modelling and analysing ICT systems"

They 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

Highly recommended by IEC, ESA, FAA and NASA for safety-critical software

Model-based formal verification

- Starting-point of is a *model* of the system under consideration
- *Modelling* – a piece of art – already reveals several inconsistencies and ambiguities
- Accompanied with efficient algorithms for realistic systems
  - improvements in data structures and algorithms + better computers

Any verifi cation using model-based techniques is only as good as the model of the system.

Formal verification techniques for property ϕ

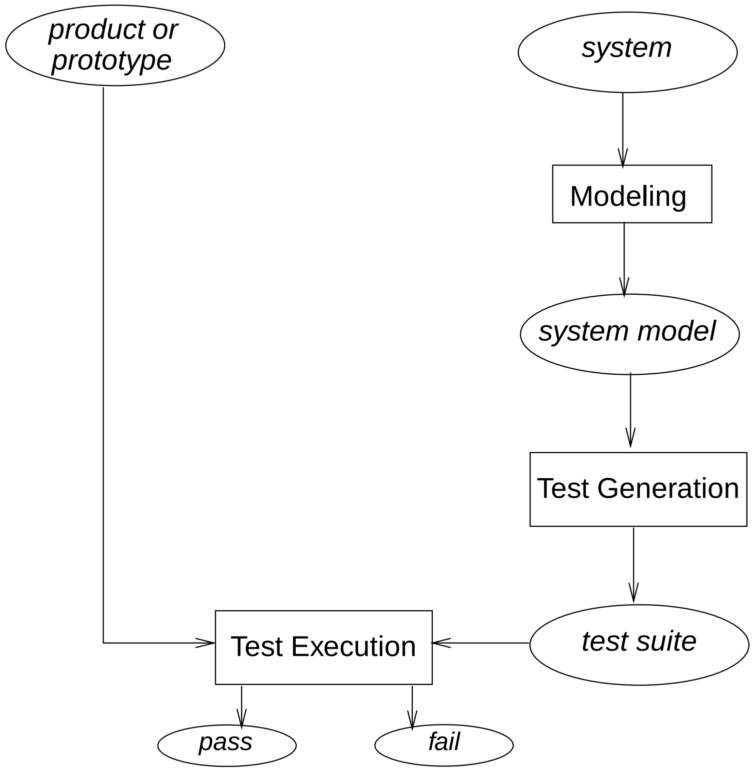
- *deductive methods*
  - method: provide a formal *proof* that ϕ holds
  - tool: theorem prover/proof assistant or proof checker
  - applicable if: system has form of a mathematical theory
- *model checking*
  - method: systematic check on ϕ in all states
  - tool: model checker (SPIN, NUSMV, UPPAAL, ...)
  - applicable if: system generates (fi nite) behavioural model
- *model-based simulation or testing*
  - method: test for ϕ by exploring possible behaviours
  - tool: simulator/tester
  - applicable if: system defi nes 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 infi nite)
  - unexplored behaviours may contain the fatal bug

$\Longrightarrow$ testing/simulation can show the presence of errors, *not their absence*

Model-based testing



As model checking verifi es models and not realisations, testing is an essential complementary technique

Overview

- *On the role of system verification*

- *Formal verification techniques*

  - model-based testing
  - simulation
  - deductive approaches

$\Rightarrow$ *Model checking*

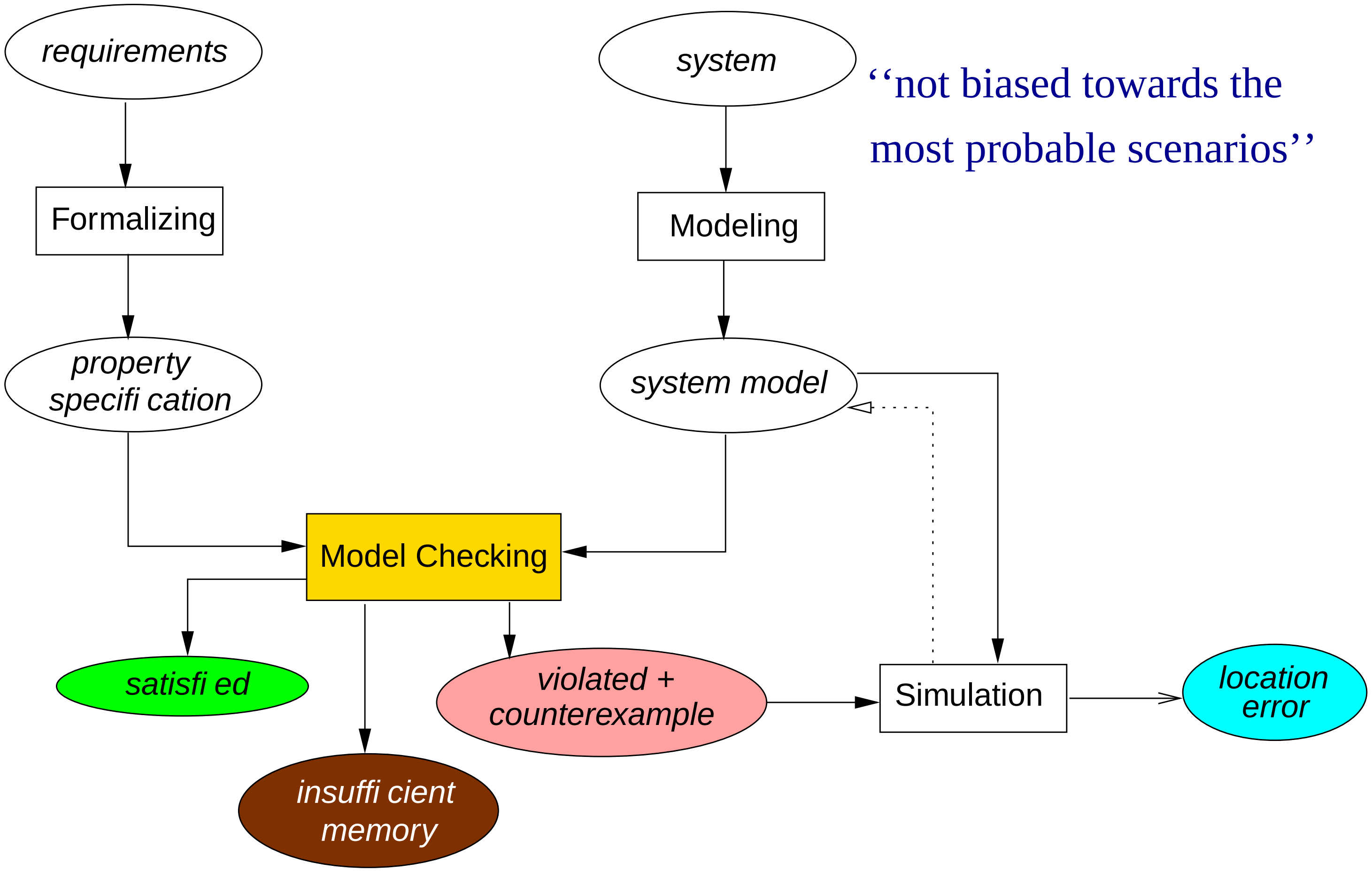
- *Course objectives and planning*

Milestones in formal verification

- Mathematical approach towards 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)
  - can handle properties referring to situations during the computation
  - based on proof rules expressed in temporal logic
- Automated verification of concurrent programs (Emerson & Clarke, 1981)
  - model-based instead of proof-rule based approach
  - does the concurrent program satisfy a given (logical) property?

these formal techniques are not biased towards the most probable scenarios

Model checking overview



‘‘not biased towards the most probable scenarios’’

What is model checking?

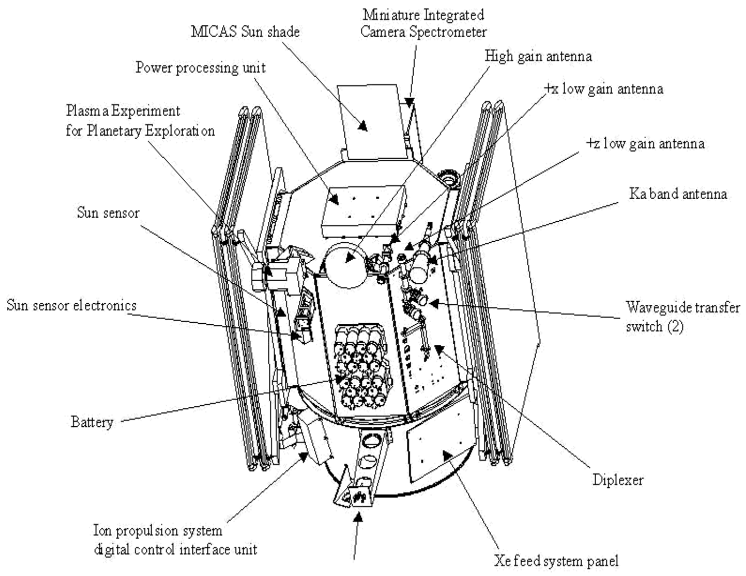
Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

Typical model-check properties

- Is the generated result ok?
- Can the system reach a deadlock situation, e.g., when two concurrent programs are mutually waiting for each other and thus halt the entire system?
- Can a deadlock occur within 1 hour after a system reset?
- Is a response always received within 8 minutes?

Model checking requires a precise and unambiguous statement of the properties to be examined; this is typically done in *temporal logic*

Deep Space-1 Spacecraft



modules of NASA's Deep Space 1 space-craft (launched in October 1998) have been thoroughly examined using model checking

A simple concurrent program

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?

A small example

```
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 for the values of x?

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() }
}
```

And let the model checker verify whether the assertion holds in every state of the concurrent system....

```
pan: assertion violated ((x >= 0) && (x <= 200)) (at depth 1802)
pan: wrote pan_in.trail
.....................
State-vector 32 byte, depth reached 3598, errors: 1
      12609 states, stored
```

The counter-example

```
...............
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? → 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, 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 counter-example is provided
- sound and interesting mathematical foundations
- not biased to the most possible scenarios (such as testing)

The cons of model checking

- mainly focused on control-intensive applications (less data-oriented)
- any validation 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 Pathfi nder, Deep Space-1, JPL LARS group

Overview

- *On the role of system verification*
- *Formal verification techniques*
  - model-based testing
  - simulation
  - deductive approaches
- *Model checking*

$\Rightarrow$ *Course objectives and planning*

Course topics

- Modeling hard- and software systems
  - transition systems, parallelism, nanoPromela, state-space explosion problem
- Linear-time properties
  - deadlock, reachability, safety, invariants, liveness and fairness
- Regular properties
  - fi nite-state automata and safety, Büchi automata and persistence
- Spin and Promela
  - hands-on experience with some small examples

Course topics

- Linear-time temporal logic
  - syntax, semantics, model-checking algorithms
- Computation tree logic
  - . . . as above . . .
  - counterexample generation, expressiveness LTL vs CTL,
  - symbolic model checking, CTL^*, fairness
- Equivalences and abstraction
  - trace and (bi)simulation, logical characterization
  - minimization algorithms

Course organization (1)

- Prerequisites
  - automata theory, complexity theory (a bit), algorithms and data structures
- Lectures: twice per week (AH1+6, Tue+Wed)
  - check regularly course web-page for possible “no shows”
  - slides (with gaps) are made available on web page
- Exercises: once per week (AH3, Fri)
  - marked exercises
  - master students: 50% of points needed
  - assistent: Martin Neuhäusser
  - student assistants: Denise Nimmerrichter und Ulrich Schmidt-Görtz

Course organization (2)

- Course material:
  - draft book “Principles of Model Checking” (Baier & Katoen)
  - hard copy available at secretary Lehrstuhl i2
  - fi nd flaws? please report them (katoen@cs.rwth-aachen.de)
  - one set of exercises waived if you fi nd serious flaw
- Exam:
  - written exam Friday July 13, 2007
  - copy of lecture notes allowed at exam
- Outlook
  - Model Checking Lab (WS 07/08), Advanced Model Checking