System Validation

Lecture 1: Introduction

Joost-Pieter Katoen

Formal Methods and Tools Group

E-mail: `katoen@cs.utwente.nl`

URL: `fmt.cs.utwente.nl/courses/systemvalidation/`

December 6, 2002

Overview

$\Rightarrow$ *On the role of system verification*

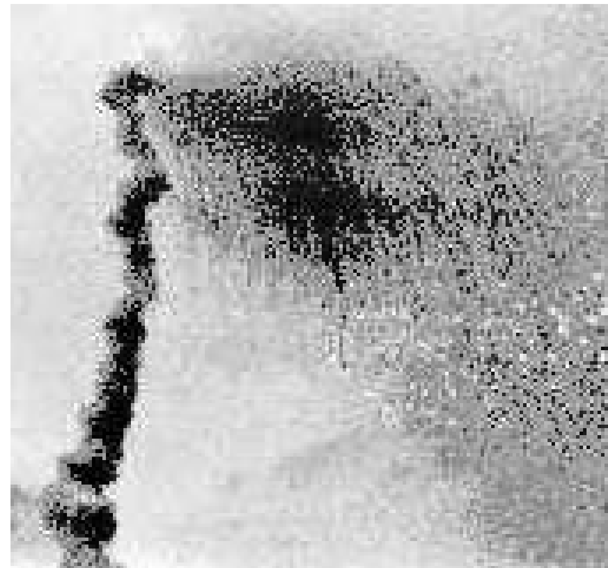
- *Formal verification techniques*
  - model-based testing
  - simulation
  - deductive approaches
- *Model checking in a nutshell*
- *Practical usage of model checking*
- *Course objectives and planning*

The quest for correctness

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

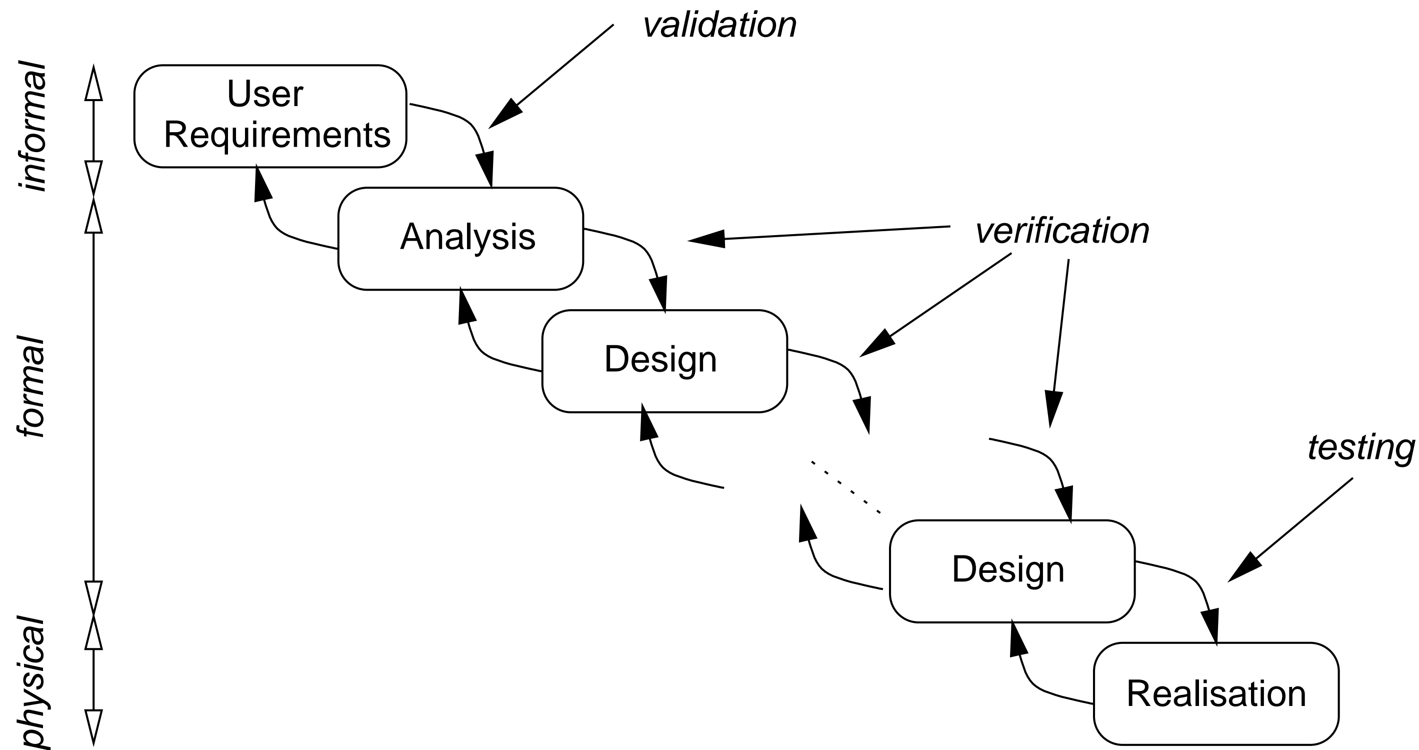
- Rapidly increasing *integration of IT* in different applications:
  - embedded systems
  - e-banking and e-shopping
  - transportation systems
- Reliability increasingly depends on hard- and software *integrity*
- Defects can be *fatal* and extremely *costly*
  - products subject to mass-production
  - 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

Typical system design trajectory



known as the waterfall model

What is system verification?

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

Verification $\neq$ validation:

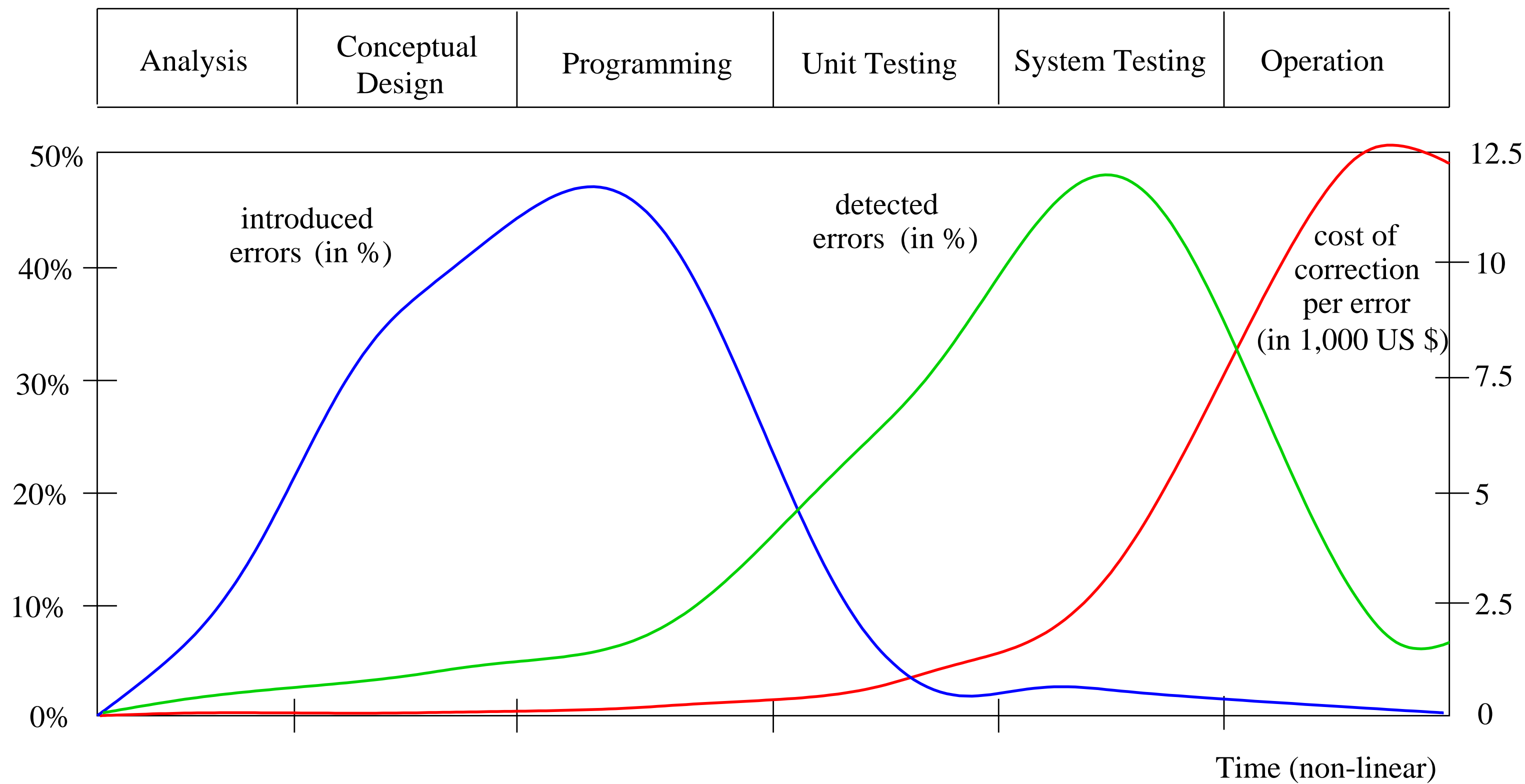
Verification = "check that we are building the thing *right*"

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: 1,5 defects per 1,000 code lines

Catching software bugs: the sooner, the better



[Liggesmeyer et al. 1998]

The importance of hardware verification

- high fabrication costs
- hardware bug fixes after delivery to customers very difficult
- high-quality expectations (software bugs are anticipated...)
- time-to-market affects potential revenue
  - 1 week delay for high-end μ-processor = revenue loss of $\geqslant$ 20 million US dollar
- techniques: emulation, simulation, testing and structural analysis

$\Longrightarrow$ considerable effort in fault detection and prevention

$\Longrightarrow$ design takes just 27% of development time!

Overview

- *On the role of system verification*

$\Rightarrow$ *Formal verification techniques*

  - model-based testing
  - simulation
  - deductive approaches

- *Model checking in a nutshell*
- *Practical usage of 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 ESA, FAA and NASA for safety-critical software

Formal verification

- Aim: establish system correctness with mathematical rigour
- Promising techniques accompanied with powerful software tools
- Two brands: *deductive* methods and *model-based* techniques
- Starting-point of model-based techniques is a *model* of the system under consideration
- *Modelling* – a piece of art – already reveals several inconsistencies and ambiguities

Any verification 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 (finite) behavioural model
- *model-based simulation or testing*
  - method: test for ϕ by exploring possible behaviours
  - tool: simulator/tester
  - applicable if: system defines an executable model

Simulation and testing

Basic procedure:

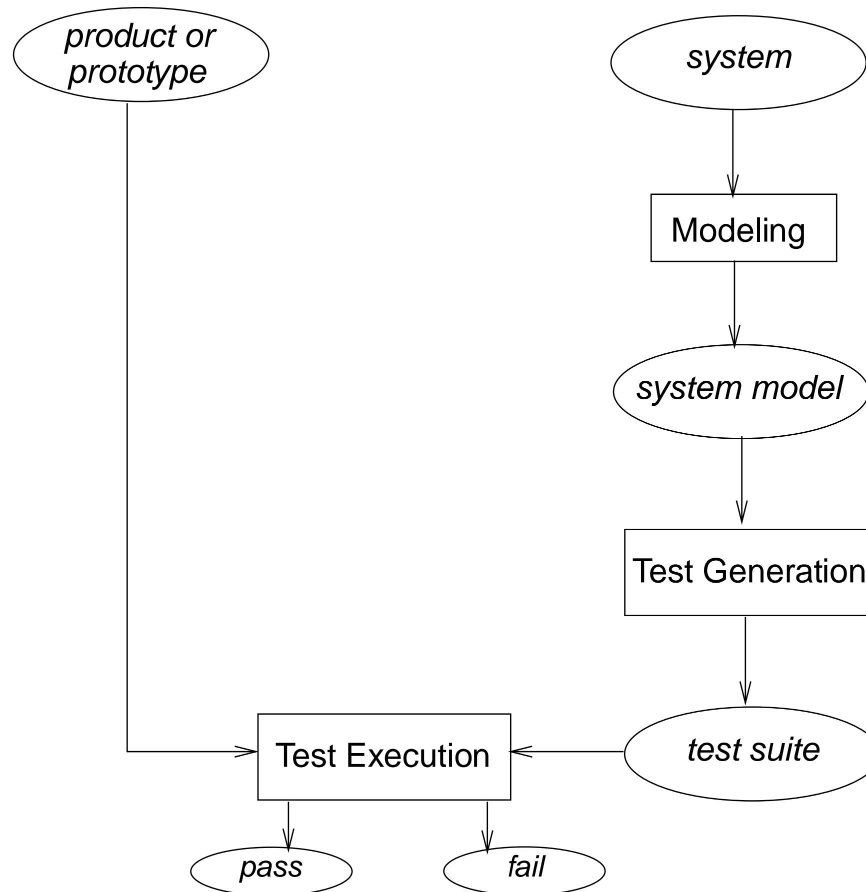
- take a model (simulation) or a realisation (testing)
- stimulate it with certain inputs, i.e., the tests
- observe reaction and check whether this is “desired”

Important drawbacks:

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

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

Model-based testing



Testing

Testing is a very useful technique when, for instance:

- it is difficult to construct a system model
- system parts (physical devices) cannot be formally modeled
- when model is proprietary (e.g., third-party testing)

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

Course on “Testing Techniques” in 3rd trimester!

Overview

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

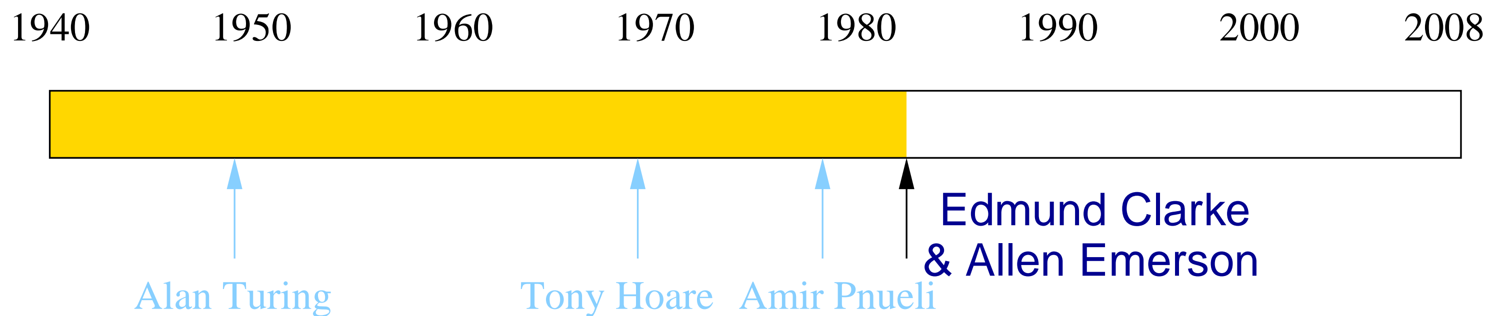
$\Rightarrow$ *Model checking in a nutshell*

- *Practical usage of model checking*
- *Course objectives and planning*

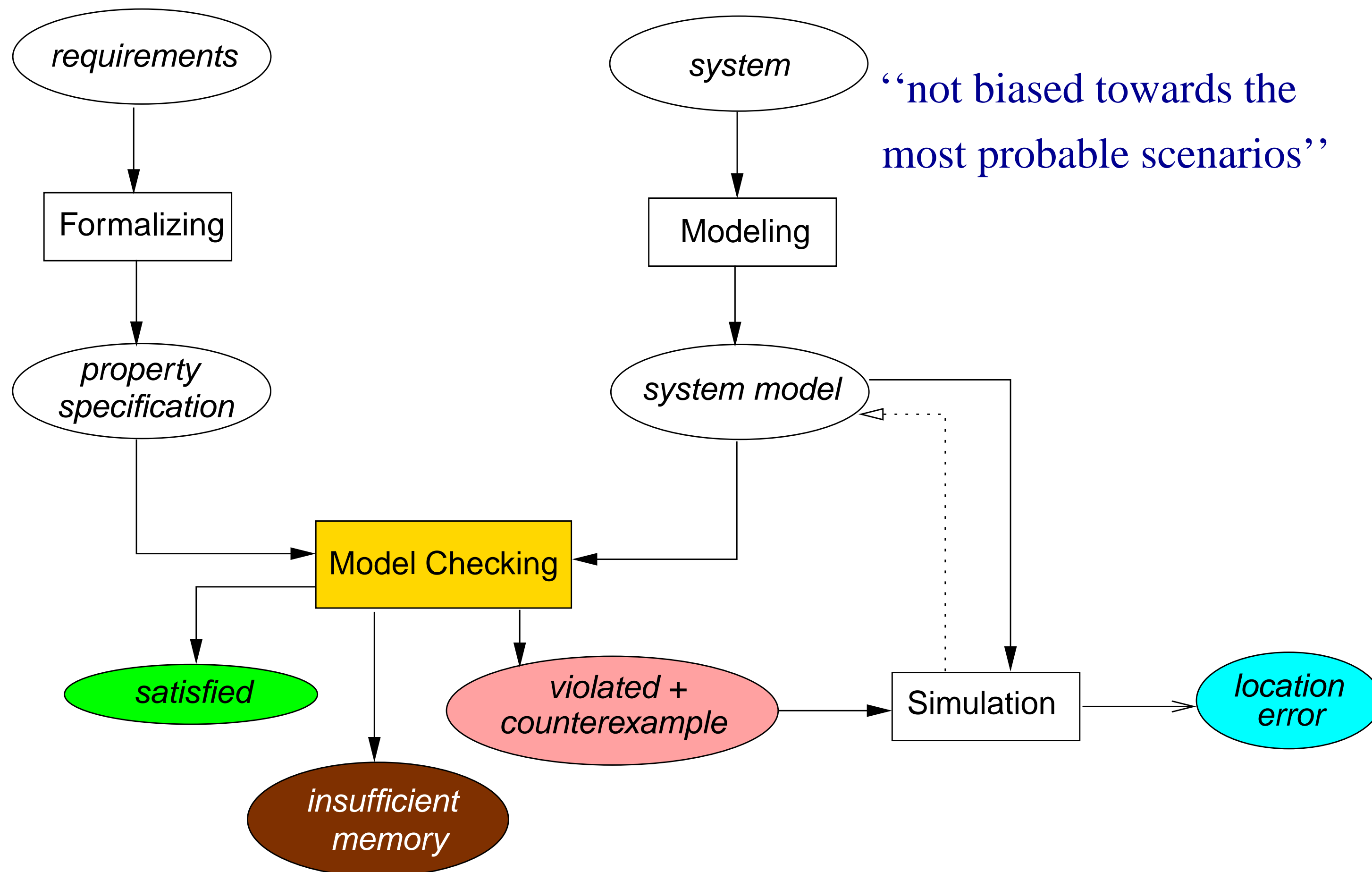
Model checking

breakthrough towards automated verification of concurrent software

- alternative to proof-based approaches
- checks validity of modal logic formula
- based on systematic state-space search

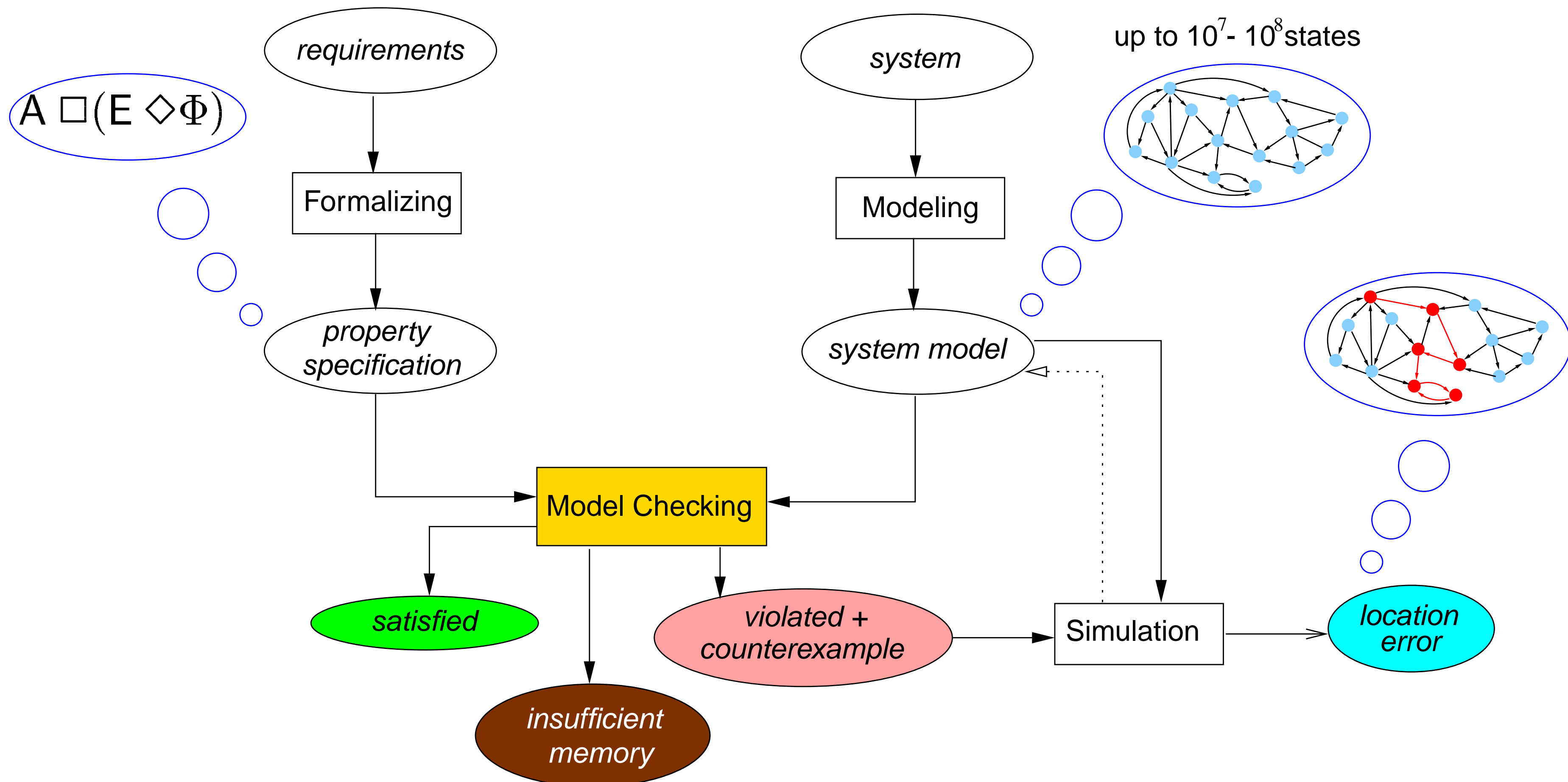


Model checking overview



"not biased towards the most probable scenarios"

The model-checking approach



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 precies and unambiguous statement of the properties to be examined; this is typically done in *temporal logic*

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 using 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 can provide a significant increase in the level of confidence of a system design

Overview

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

$\Rightarrow$ *Practical usage of model checking*

- *Course objectives and planning*

Typical interleaving / atomicity problem

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 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

```
...............
606: proc  1 (Inc)   line   9 "pan_in" (state 2) [((x<200))]
607: proc  1 (Inc)   line   9 "pan_in" (state 3) [x = (x+1)]
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() }
}
```

Case: NewCore-project (AT&T)

- Design of 5ESS Switching Center at Bell Labs (USA) in 1990-1992
- ISDN User Part protocol, two design teams
- Team 1: 40-50 traditional designers and team 2: 4-5 “verification engineers”
- 7,500 lines specification in SDL (Specification and Description Language)
  - 112 (!) design errors were found (only counting serious ones)
  - 145 formal requirements
  - 10,000 verification runs (100/week) with model checker SPIN
  - 55% of original design requirements were inconsistent

Case: IEEE Futurebus+ cache coherence protocol

- Cache coherence protocol must ensure data consistency:
  - if two caches contain a data copy, the copies must be equal
  - if the global memory has a "non-modified" data item with value v, then any copy of this item in any cache has value v
- Protocol has been modeled in 2,300 lines of SMV (after abstraction)
- Configuration: 3 bus segments, 8 processors. 10^{30} states
- Several non-trivial errors were revealed
- Result: a substantial revision of the original IEEE standard protocol

Overview

- *On the role of system verification*
- *Formal verification techniques*
  - model-based testing
  - simulation
  - deductive approaches
- *Model checking in a nutshell*
- *Practical usage of model checking*

$\Rightarrow$ *Course objectives and planning*

System validation: objectives

- Course objectives:
  - model simple systems and specify requirements
  - obtain experience with usage of model-checking tools
  - understand main ideas model-checking algorithms
- Organisation of the course:
  - **7** lectures (weeks 49, 50, 51, 3, **4**, 6 and 8)
  - self-study by means of 3 exercise series
  - tackled by groups of maximally two students
  - final mark of course is average of marks of assignments
  - *Hand in your* **own** *work! No plagiarism!*
- Exercises assistant: *Henrik Bohnenkamp* (bohnenka@cs.utwente.nl)

System validation: planning

Lecture	*Topic*	*Handed out*	*Handed in*
December 13	SPIN and PROMELA *by Theo Ruys*		
December 20	Effective modelling with PROMELA *by Theo Ruys*	Exercise 1	
January 17	Linear Temporal Logic		Exercise 1
January 31	Computational Tree Logic	Exercise 2	
February 7	Timed automata		
February 21	UPPAAL	Exercise 3	Exercise 2
March 15			Exercise 3