Specification and Verification

Lecture 1: Introduction - To verify or not to verify?

Guillermo A. Pérez

September 27, 2024

Outline

TL;DR: This course in short

What is Verification?

Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.

Why verify?

Essentially: To avoid money and life loss.

References

Main references

- Christel Baier, Joost-Pieter Katoen: **Principles of Model Checking.** MIT Press 2018.
- Mickael Randour: Verification course @ UMONS.
- **Lectures in Game Theory for Computer Scientists.** Eds: K. Apt, E. Grädel. Cambridge University Press 2011.
- **Handbook of Model Checking.** Eds: E.M. Clarke, T.A. Henzinger, H. Veith, R. Bloem. Springer 2018.

Required and target competences

What tools do we need?

Discrete maths, Modelling, Algorithms & complexity, Formal language theory

What skills will we obtain?

We will get an overview of the verification field and

- Prepare a symbolic solver, split into two deliveries:
  1. Library with graph/automata algorithms using BDDs
  2. Parity-game solver

How will these skills be useful?

To assert strong correctness guarantees about your code/hardware.

Evaluation method

Project: a verification tool

A project that accounts for 40% of your grade:

- You prepare a model checker that can deal with provided benchmarks

Exercises: optional, but...

The exercises can be counted towards your exam grade!

Written examination

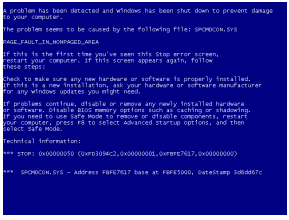
The remaining 60%:

- You can bring **one cheat sheet**
- Answers (report in LaTeX) for exercise sessions count here! To be turned in before exam date

Outline

Let's talk about bugs...

Insects

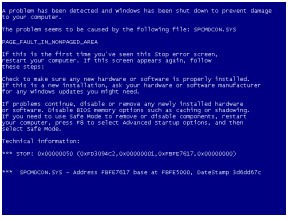


Computer bugs

Let's talk about bugs...

Insects

- Plenty of them



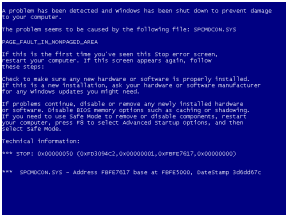
Computer bugs

- Plenty of them

Let's talk about bugs...

Insects

- Plenty of them
- Mostly annoying

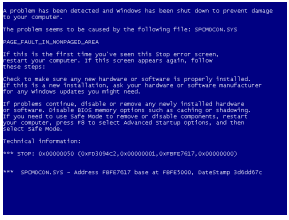


Computer bugs

- Plenty of them
- Mostly annoying

Let's talk about bugs...

Insects

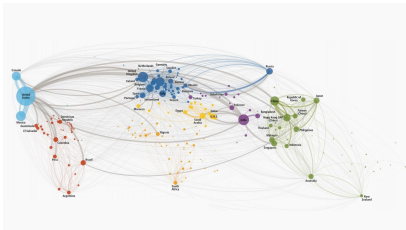


Computer bugs

- Plenty of them
- Mostly annoying
- Serve a purpose

- Plenty of them
- Mostly annoying
- At best useless, can be harmful!

It's all about money (1/4)



AT&T long-distance service (1990)

- Bug caused continuous cascade reboots of all long-distance switches.
- Impact: 9-hour outage.
- Costs: 60-100 million US$.
- Source: wrong interpretation of C `break` statement.

It's all about money (2/4)

Pentium FDIV (1994)

- Bug in the **f**loating point **div**ision unit (FDIV).
- Impact: inaccurate results for 1 in $9 \cdot 10^9$ random floating point divisions.
- Costs: ∼500 million US$ (replacement of all processors).
  + PR nightmare for Intel!
- Source: 5 missing entries in a 1066-entry look-up table.

It's all about money (3/4)

Ariane 5 (1996)

- Loss of guidance after 37s followed by self-destruction.
- Costs: $>$ 500 million US$.
- Source: data conversion from 64-bit floating point to 16-bit signed integer causing overflow in the hardware.
  - Appropriate software handler was disabled to improve efficiency.

It's all about money (4/4)

Mars Climate Orbiter (1998)

- Atmosphere entry at wrong angle resulting in disintegration.
- Costs: 327 million US$ (mission failure).
- Source: ground software sending instructions calculated in the wrong units (imperial vs. metric units as the NASA-Lockheed contract specified).

It's all about safety (1/2)

Therac-25 radiation therapy (1985-1987)

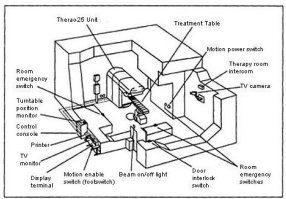
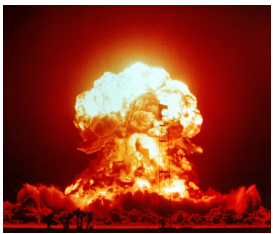


Figure 1. Typical Therac-25 facility

- Two modes: one "safe" direct mode, one powerful mode requiring shielding; bug caused mismatch.
- Impact: several deaths by radiation poisoning.
- Source: *race condition* in the software managing the choice of mode.

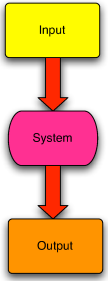
It's all about safety (2/2)

The doomsday bug (1983)



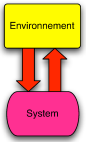
- Soviet nuclear early-warning system (Oko) falsely reports five incoming US missiles.
- Possible impact: WW3?
- Avoided by Stanislav Petrov who judged the report to be a false alarm.
- Source: bug in the Soviet satellite detection system.

Batch processing vs. reactivity



Batch processing system

- Computes results.
- Correctness *easier* to assess.

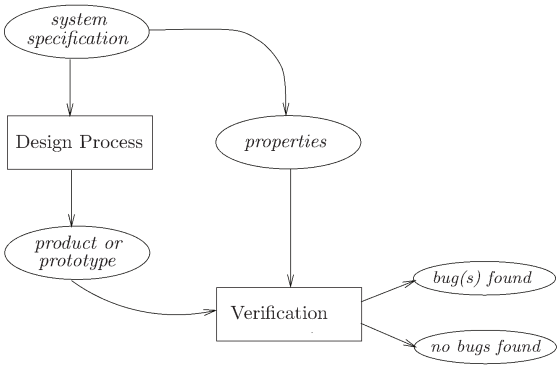


Reactive system

- Continuous interaction with the environment:
  - requests information,
  - reacts to events.
- Correctness very difficult to assess.

Outline

Hardware and software verification (1/4)



A posteriori verification

Properties to check obtained from the system's **specification**

Hardware and software verification (2/4)

Software verification

- *Peer-reviewing*: static analysis of uncompiled code
  - Useful (catches from 31% to 93%, median 60%, of defects)
  - Used in $\sim$80% of software projects
  - Difficult to catch dynamic issues: concurrency, algorithmic defects...

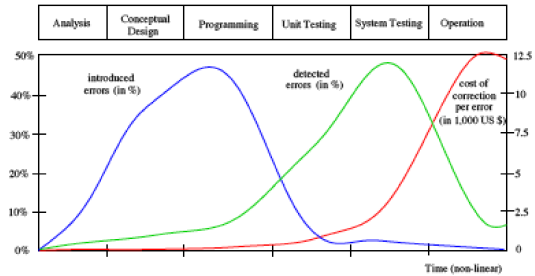
Hardware and software verification (2/4)

Software verification

- *Peer-reviewing*: static analysis of uncompiled code
  - Useful (catches from 31% to 93%, median 60%, of defects)
  - Used in ∼80% of software projects
  - Difficult to catch dynamic issues: concurrency, algorithmic defects...
- *Testing*: dynamic, confronts the software to test suites
  - Can catch dynamic defects
  - 30% to 50% of software cost devoted to testing
    - More time spent on validation than on construction!
  - Exhaustive testing infeasible
    - **Testing can only show the presence of errors, not their absence!**

Hardware and software verification (3/4)

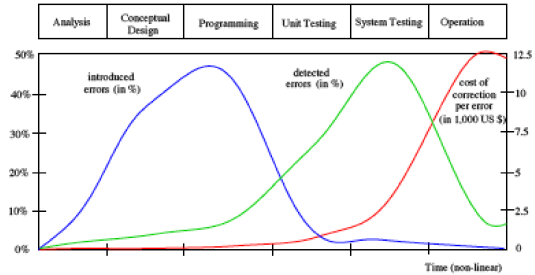
Catching bugs: the sooner, the better



Software lifecycle: error introduction, detection and repair costs

Hardware and software verification (3/4)

Catching bugs: the sooner, the better



Software lifecycle: error introduction, detection and repair costs

⇒ **We need methods that can detect bugs early in a software's life.**

Hardware and software verification (4/4)

Hardware verification

- Preventing errors is vital:
  - high fabrication costs,
  - fixing defects after delivery is difficult (no patch),
  - high quality expectations.

Hardware and software verification (4/4)

Hardware verification

- Preventing errors is vital:
  - high fabrication costs,
  - fixing defects after delivery is difficult (no patch),
  - high quality expectations.
- >50% of Application-Specific Integrated Circuits do not work properly after initial design and fabrication
- >70% of the total development time is devoted to error detection and prevention

Hardware and software verification (4/4)

Hardware verification

- Preventing errors is vital:
  - high fabrication costs,
  - fixing defects after delivery is difficult (no patch),
  - high quality expectations.
- >50% of Application-Specific Integrated Circuits do not work properly after initial design and fabrication
- >70% of the total development time is devoted to error detection and prevention
- Some techniques: *emulation* ($\sim$ testing), *simulation* ($\sim$ testing executed on models), *hardware testing* (to find fabrication faults)

Formal verification

Goal

Given

- a *formal model* of the system (= how it behaves)
- and a *formal specification* (= what it should do **and not do**),

check that the system satisfies the specification by (semi-)automatically generating some sort of *mathematical proof*.

Formal verification

Goal

Given

- a *formal model* of the system (= how it behaves)
- and a *formal specification* (= what it should do **and not do**),

check that the system satisfies the specification by (semi-)automatically generating some sort of *mathematical proof*.

Usefulness

- Early integration of verification in the design process.
- More effective verification (higher coverage).
- Reduced verification time.

Formal verification

Goal

Given

- a *formal model* of the system (= how it behaves)
- and a *formal specification* (= what it should do **and not do**),

check that the system satisfies the specification by (semi-)automatically generating some sort of *mathematical proof*.

Usefulness

- Early integration of verification in the design process.
- More effective verification (higher coverage).
- Reduced verification time.

$\Rightarrow$ **safety** $\nearrow$ **and costs** $\searrow$

Specification formalisms

Formal encoding of such properties requires appropriate **specification formalisms**

Specification formalisms

Formal encoding of such properties requires appropriate **specification formalisms**

- Most are **temporal logics** (LTL, CTL, etc).
- *Not all logics can express all properties!*

Specification formalisms

Formal encoding of such properties requires appropriate **specification formalisms**

- Most are **temporal logics** (LTL, CTL, etc).
- *Not all logics can express all properties!*

Trade-off between **expressiveness** and **tractability**.

$\hookrightarrow$ think about *decidability* and *complexity*: e.g., no hope of checking termination for Turing-powerful models.

Limits of formal verification (cf. validation)

Is the model right?

- Is it a faithful representation of the implementation?

Limits of formal verification (cf. validation)

Is the model right?

- Is it a faithful representation of the implementation?

Is the specification right?

- Often difficult to formalize, from oral language to logical formulas
- Difficult to check: does it really represent the expected behaviour of the system?

Limits of formal verification (cf. validation)

Is the model right?

- Is it a faithful representation of the implementation?

Is the specification right?

- Often difficult to formalize, from oral language to logical formulas
- Difficult to check: does it really represent the expected behaviour of the system?

Is the specification complete?

- Are all important properties specified?

Three approaches to verification (1/2)

Deductive methods (logical inference)

- Method: provide a formal proof that the property holds
- Tools: theorem provers and proof assistants/checkers (e.g., HOL, Isabelle)
- Applicable if the system has the form of a *mathematical theory*

Three approaches to verification (1/2)

Deductive methods (logical inference)

- Method: provide a formal proof that the property holds
- Tools: theorem provers and proof assistants/checkers (e.g., HOL, Isabelle)
- Applicable if the system has the form of a *mathematical theory*

Model-based simulation/testing

- Method: test the property by exploring possible behaviours of the model
- Applicable if the system defines an *executable model*

Three approaches to verification (2/2)

Model checking

- Method: systematic check of the property in all states of the model
- Tools: model checkers (e.g., Spin, NuSMV, Uppaal)
- Applicable if the system generates a *finitely representable behavioural model*
- Efficient techniques and tools
- If the property is not satisfied, can provide *counter-examples* (thus guiding repairs)

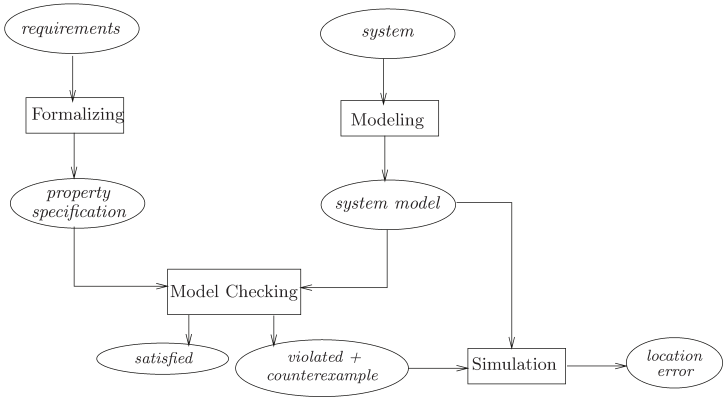
Three approaches to verification (2/2)

Model checking

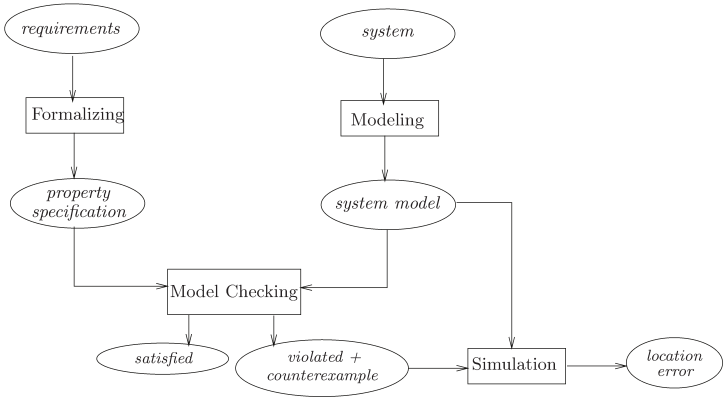
- Method: systematic check of the property in all states of the model
- Tools: model checkers (e.g., Spin, NuSMV, Uppaal)
- Applicable if the system generates a *finitely representable behavioural model*
- Efficient techniques and tools
- If the property is not satisfied, can provide *counter-examples* (thus guiding repairs)

$\hookrightarrow$ **main focus of this course**

Model checking process



Model checking process



Formal model
transition systems/Markov chains

Formal specification
LTL/CTL/PCTL

Pros of model checking

Pros:

- widely applicable (hardware, software, protocols),
- allows partial verification (most relevant properties),
- heavily automated,
- growing industrial interest,
- counter-example generation,
- sound mathematical foundations,
- not biased to the most probable scenarios (in contrast to *testing*)

Cons of model checking

Cons:

- focus on *control-intensive* applications (reactive systems)—less on *data-oriented* applications (batch processing systems),
- **model checking is only as good as the model**,
- decidability and complexity issues (state explosion problem),
- completeness is not guaranteed (if the specification omits important properties)

Cons of model checking

Cons:

- focus on *control-intensive* applications (reactive systems)—less on *data-oriented* applications (batch processing systems),
- **model checking is only as good as the model**,
- decidability and complexity issues (state explosion problem),
- completeness is not guaranteed (if the specification omits important properties)

All in all:

a quite effective technique to expose design errors.

$\hookrightarrow$ **interesting addition to most design processes**

Industry usage

Model checking techniques are increasingly present in industrial design processes

Industry usage

Model checking techniques are increasingly present in industrial design processes

- *Security*. A flaw in the Needham-Schroeder public-key protocol remained undiscovered for 17 years before being revealed by model checking

Industry usage

Model checking techniques are increasingly present in industrial design processes

- *Security*. A flaw in the Needham-Schroeder public-key protocol remained undiscovered for 17 years before being revealed by model checking
- *Model checkers for C, C++ and Java*; Developed and used by Microsoft, Digital, NASA; Successfully applied to the design of *device drivers*

Industry usage

Model checking techniques are increasingly present in industrial design processes

- *Security*. A flaw in the Needham-Schroeder public-key protocol remained undiscovered for 17 years before being revealed by model checking
- *Model checkers for C, C++ and Java*; Developed and used by Microsoft, Digital, NASA; Successfully applied to the design of *device drivers*
- Facebook has the verification tool Infer and its own verification team
- Amazon does verification within their automated-reasoning team

Awards for model checking advances





E. Clarke A. Emerson J. Sifakis M. Vardi P. Wolper

— Turing Award 2007 — Gödel Prize 2000

Awards for model checking advances





E. Clarke A. Emerson J. Sifakis M. Vardi P. Wolper

— Turing Award 2007 — Gödel Prize 2000

- Turing award: "for their role in developing model checking into a highly effective verification technology, widely adopted in the hardware and software industries."

Outline

Outline of the course

1. Introduction
2. Transition systems
3. Linear temporal logic
4. Omega automata
5. Binary decision diagrams - Proj. 1
6. Model checking LTL
7. Reactive synthesis
8. Parity games - Proj. 2