Formal Methods for System Design

Chapter 1: Formal verification

Mickael Randour

Mathematics Department, UMONS

September 2023

1 Motivations

2 Formal verification in a nutshell

3 Going further: synthesis

4 Course organization

1 Motivations

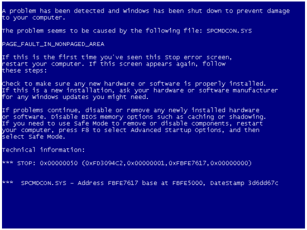
2 Formal verification in a nutshell

3 Going further: synthesis

4 Course organization

Let's talk about bugs. . .

Insects



```
A problem has been detected and Windows has been shut down to prevent damage
to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBFE7617,0x00000000)


***  SPCMDCON.SYS - Address FBFE7617 base at FBFE5000, DateStamp 3d6dd67c
```

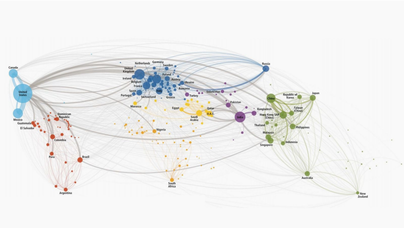
Computer bugs

- Plenty of them.
- Pesky as hell.
- Serve a purpose.

- Plenty of them.
- Pesky as hell.
- At best useless. . . In the worst-case, threaten the purpose of the software and 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 `break` statement (c code).

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 (pound-seconds instead of newton-seconds as the NASA-Lockheed contract specified).

It’s all about safety (1/2)

Therac-25 radiation therapy (1985-1987)

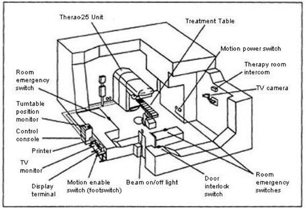
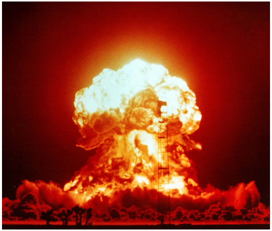


Figure 1. Typical Therac-25 facility

- Two modes: one “safe” direct mode and one very powerful mode requiring appropriate shielding. Bug caused mismatch of the chosen mode.
- Impact: several deaths by radiation poisoning.
- Source: design error causing *race condition* in the software managing the choice of mode.
  - ▷ Bug already present in previous machines but with no consequence due to hardware limitations.
  - ↪ Never reuse code without proper testing in the new environment!

It’s all about safety (2/2)

The doomsday bug (1983)



- Soviet nuclear early-warning system (Oko) falsely reports five incoming US missiles.
  - ▷ Three weeks after the Soviet military had shot down Korean Air Lines Flight 007.
- Possible impact: WW3?
- Avoided by Stanislav Petrov who judged the report to be a false alarm.
- Source: bug in the Soviet satellite detection system.

Ubiquity of software integration

Software is *everywhere*:

- embedded systems,
- communication protocols,
- transportation systems. . .

Reliability increasingly depends on software!

▷ E.g., cars: fuel injection, central locking unit, ABS, ESP. . .

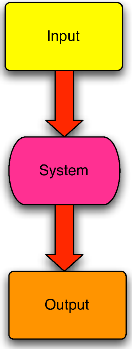
. . . and less legal systems (VW's pollution *defeat device*).

Defects can be **fatal** and **extremely costly**.

▷ Safety-critical systems.

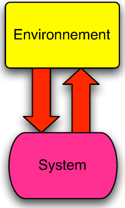
▷ Mass-produced systems (correcting a bug is a huge task).

Batch processing systems vs. reactive systems



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.

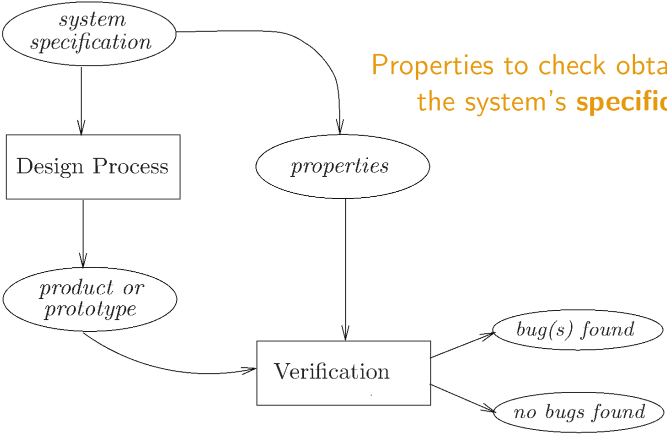
Characteristics of reactive systems

- Not necessarily terminating: in general, termination (deadlock) is to avoid.
- Should always be ready for interaction.
  - ▷ Interaction = basic unit of computation: event - condition - action.
- Sequence of interactions = computation.
- Allowed ordering of interactions determine correctness.
- Specific constraints of embedded systems: energy consumption, real-time, sparse resources, etc.

⇒ **Specific methods are needed to analyze those systems.**

1. Motivations
2. Formal verification in a nutshell
3. Going further: synthesis
4. Course organization

Hardware and software verification (1/4)



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

A posteriori verification [BK08].

Hardware and software verification (2/4)

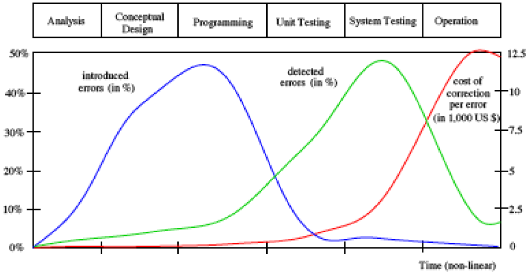
Some classical techniques

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 [BK08].

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

Hardware and software verification (4/4)

Some classical techniques

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**),

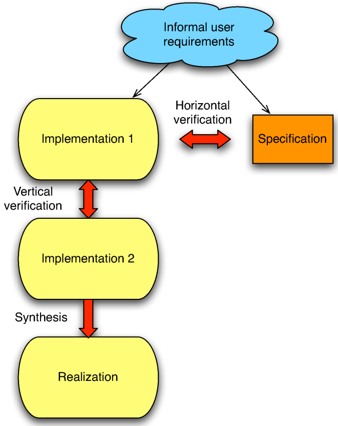
check that the system satisfies the specification by (semi-)automatically generating some sort of *mathematical proof* .

Usefulness

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

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

Horizontal vs. vertical verification



- Horizontal: system vs. spec.
- Vertical: system vs. refinement.
- Synthesis: correctness-preserving refinement.

  ↪ More on that later!

Checkable properties

Non-exhaustive selection (informal definitions)

- **Safety**: unwanted system *states* are never reached.
  - ▷ E.g., avoid deadlock.
- **Liveness**: desired behavior *eventually* happen.
  - ▷ E.g., coffee machine eventually provides coffee.
- **Persistence**: after some time, desired state set is never left.
  - ▷ E.g., after initial warm-up, the system always stays online.
- **Fairness**: infinitely done requests are infinitely satisfied.
  - ▷ E.g., access to critical section (mutex).
- **Quantitative properties**: energy consumption, response time, etc.
  - ↪ Much more complex.

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**.

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

Limits of formal verification

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 formulae.
- ▷ Difficult to validate: does it really represent the expected behavior of the system?

Is the specification complete?

- ▷ Are all important properties specified?

Three approaches to formal 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 behaviors of the model.
- Applicable if the system defines an *executable model*.

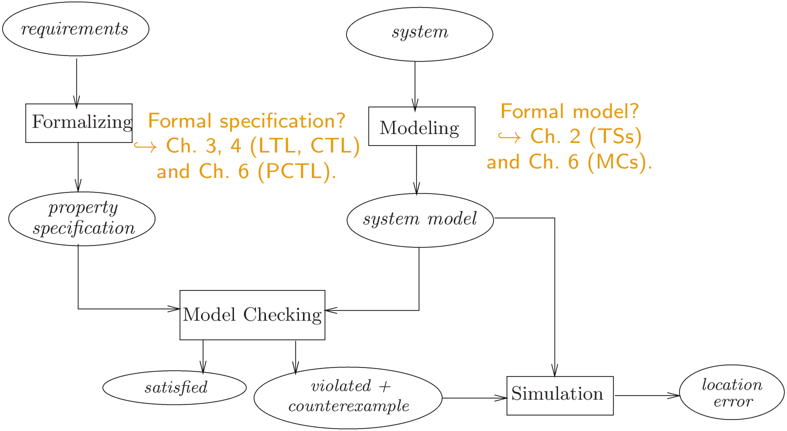
Three approaches to formal verification (2/2)

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 behavioral model*.
- Efficient techniques and tools.
- If the property is not satisfied, can provide *counter-examples* (thus guiding repairs).

↪ **main focus of this course.**

Model checking process



Formal specification?
↪ Ch. 3, 4 (LTL, CTL)
and Ch. 6 (PCTL).

Formal model?
↪ Ch. 2 (TSs)
and Ch. 6 (MCs).

Schematic view of the model checking approach [BK08].

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).

All in all:

a quite effective technique to expose design errors.

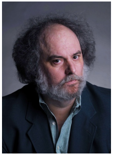
$\hookrightarrow$ **interesting addition to most design processes.**

Industry usage

Model checking techniques are increasingly present in industrial design processes.

- *Security*. A flaw in the Needham-Schroeder public-key protocol remained undiscovered for 17 years before being revealed by model checking [Low96].
- *Model checkers for C, C++ and `Java`*. Developed and used by Microsoft, Digital, NASA. Successfully applied to the design of *device drivers*.
- In 2013, Facebook invested in a startup specialized in software verification: Monoidics.
- Even medium-size businesses may benefit from formal methods.
  - ↪ E.g., CASSTING FP7 European project with industrial partners EnergiNord (energy provider) and Seluxit (smart homes and smart grids).

Some awards for model checking advances



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

Turing Award 2007 Gödel Prize 2000

- Clarke, Emerson and Sifakis "for their role in developing model checking into a highly effective verification technology, widely adopted in the hardware and software industries."
- Vardi and Wolper "for work on model checking with finite automata."
- Many important people: Büchi, Petri, Rabin, Scott, Floyd, Hoare, Dijkstra, Pnueli, Milner, Queille, Kozen, Harel, Bryant, McMillan, Holzmann, Alur, Dill, Thomas, Henzinger. . .

1 Motivations

2 Formal verification in a nutshell

3 Going further: synthesis

4 Course organization

Synthesis vs. verification (1/2)

Verification operates *a posteriori*: it checks that an existing model satisfies a specification.

What if we tried to work the other way around?

Verification

- ▷ Input: model $\mathcal{M}$, spec. $\mathcal{S}$
- ▷ Output: $\mathcal{M} \overset{?}{\models} \mathcal{S}$.

Synthesis

- ▷ Input: spec. $\mathcal{S}$
- ▷ Output: model $\mathcal{M}$ such that $\mathcal{M} \models \mathcal{S}$, or $\textsc{No}$ if none exists.

Goal

Automatic design of a suitable system from the specification.

Synthesis vs. verification (2/2)

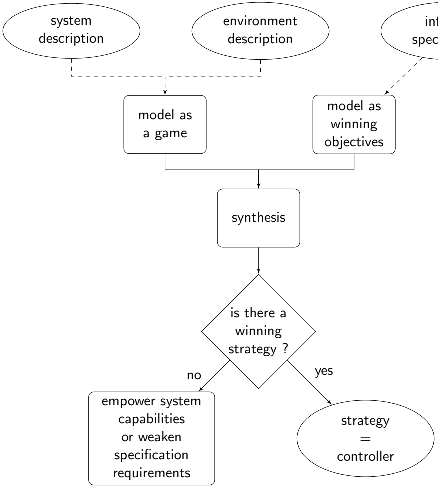
Synthesis is much more difficult!

↪ Consider the role of the **uncontrollable environment** for reactive systems.

In practice, instead of checking a temporal formula (spec.) against an automaton-based model, one may consider a two-player game between the system and its environment (assumed antagonistic).

- ▷ Basic model, much richer ones exist.
- ▷ Maybe Ch. 7?
- ▷ We look for **winning strategies** for the system.

Synthesis process


system description
environment description
informal specification
model as a game
model as winning objectives
synthesis
is there a winning strategy ?
no
yes
empower system capabilities or weaken specification requirements
strategy = controller

Synthesis process [Ran13].

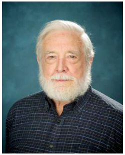
1. Can one player guarantee victory?
2. Can we decide which one?
3. How complex his strategy needs to be?

Important research area (incl. in UMONS).

Some great minds behind synthesis

A. Church

P. Ramadge



W. Wonham

A. Pnueli

R. Rosner

Turing Award 1996

Seminal papers [Chu57, RW87, PR89].

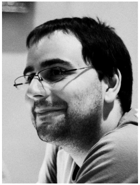
1 Motivations

2 Formal verification in a nutshell

3 Going further: synthesis

4 Course organization

Teaching staff



Mickael Randour
Professor
(lectures)

James C. A. Main
Teaching assistant
(exercise sessions, project supervision)

We are available for discussion and help if needed. Please do not hesitate to contact us!

Feedback on the course is welcome!

Learning outcomes

At the end of the course, students should be able to

(i) model reactive systems using mathematical formalisms,

(ii) analyze these models using classical verification techniques,

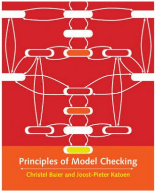
(iii) use the results of this analysis to debug their models/systems,

(iv) master the core mathematical concepts and algorithms for prominent techniques,

(v) use verification tools supporting these techniques.

Course material

- Slides available on *Moodle*.
  - Inspired by the reference book and slides by Thierry Massart (ULB), Jean-François Raskin (ULB), Joost-Pieter Katoen (RWTH Aachen), etc.
- Notes in class: *pay attention to the blackboard!*
- Optional reference book: *Principles of Model Checking* by C. Baier and J.-P. Katoen, MIT Press, 2008 [BK08].


Principles of Model Checking
Christel Baier and Joost-Pieter Katoen

Course schedule

- Check *Hyperplanning* and announcements via *Moodle*.

Exercise sessions

- Exercise sheets will be available on *Moodle*.
- *Crucial for the oral exam.*
  - ▷ Many exercises will be available for each session.
    - ↪ Feedback available from the TA.

Question/discussion sessions will also be scheduled within the theory lectures.

Course outline

Detailed outline available on *Moodle*.

1. Formal verification
2. Modeling systems
3. Linear temporal logic
4. Computation tree logic
5. Symbolic model checking
6. Model checking probabilistic systems
7. TBD: book chapter / article

Exercise sessions

General instructions and grading

Please read the detailed instructions on Moodle.

- Individual **oral exam** at the end of Q1 (AAs 1 and 2).
- **Group tool presentations** (AAs 1 and 2).
- Presentations of **advanced work** (AA3).
- **Group project** (AA3).

Oral exam

You should prove that

- you *understand* the theory,
- you *master* the essential techniques.

Format

One large question with 20 minutes to prepare (course material allowed, around 35% of the grade) followed by smaller questions exploring all the course, without preparation.

Mastering the exercise sessions is crucial!

Tool presentation (groups)

Goal

Choose a verification tool in the list (see Moodle) and prepare a 30-minute presentation of the tool for the class.

Evaluation: quality of the presentation.

All students of the group are expected to participate in the presentation and to know the tool.

Date for the presentations: TBA.
Choice of the tool before 01/11/2023 (first come, first served).

Presentations of advanced work

Goal

Study an article or book chapter on an advanced topic and present it to the other students: 4-hour lectures.

Evaluation: quality of the presentation, understanding of the subject. Students will also be questioned on all presentations.

Project

Details to be refined (#students, etc).

In a nutshell:

- combine learning and formal methods,
- continuous and common project,
- teachers as coaches.

$\rightsquigarrow$ **Check the instructions for the last project.**

References I

C. Baier and J.-P. Katoen.
Principles of model checking.
MIT Press, 2008.

A. Church.
Applications of recursive arithmetic to the problem of circuit synthesis.
Summaries of the Summer Institute of Symbolic Logic, 1:3–50, 1957.

G. Lowe.
Breaking and fixing the Needham-Schroeder public-key protocol using FDR.
In Tools and Algorithms for the Construction and Analysis of Systems, pages 147–166. Springer, 1996.

A. Pnueli and R. Rosner.
On the synthesis of a reactive module.
In Proc. of POPL, pages 179–190. ACM Press, 1989.

M. Randour.
Automated synthesis of reliable and efficient systems through game theory: A case study.
In Proceedings of the European Conference on Complex Systems 2012, Springer Proceedings in Complexity XVII, pages 731–738. Springer, 2013.

P.J. Ramadge and W.M. Wonham.
Supervisory control of a class of discrete event processes.
SIAM journal on control and optimization, 25(1):206–230, 1987.