C01 – Organization & Intro

Program Verification

FMI · Denisa Diaconescu · Spring 2025

Overview

Course organisation

Course details

- Denisa Diaconescu
    - `https://cs.unibuc.ro/~ddiaconescu/`
    - **denisa.diaconescu [at] gmail.com**
    - denisa.diaconescu [at] g.unibuc.ro
- Access to Materials
  `https://tinyurl.com/FMI-PV2025`
- Microsoft Teams
  `https://tinyurl.com/FMI-PV2025-Teams`

Grading

Default	:	10 points
Exam	:	60 points
Project	:	30 points
Maximum Grade	:	100 points
Minimum Passing Grade	:	50 points

*A bonus of 30 points can be awarded for certain projects (stay tuned)

Exam: 60 points

- 1 hour exam
- Multiple choice questions (aka *grile*)
- Questions similar to the quiz questions and the examples from the lecture notes
- All materials at hand

Project: 30 points (+ potentially 30 points bonus)

You will hunt bugs
applying existing verification tools
on open-source projects.

Phase 1: Get familiar with some formal verification tools

DEADLINE: 11 March 2025

TODO: nothing for now.

Some formal verification tools:

- CppCheck
- SonarQube
- SpotBugs
- Semgrep
- Infer
- LuaCheck
- Sanitizers from Google
- Valgrind
- any other formal verification tool you like

We are not searching for code smells and stylistic errors.

Phase 2: Choose an open-source project

DEADLINE: 7 April 2025

TODO: fill-in the form to sign up (see below)

- Choose an open-source project from GitHub.
    - It should have at least 1000 stars, but special cases can be discussed.
    - The programming language of the project is not important, as long as you can verify it!
    - Run a verification tool on the project and make sure you find bugs.
    - If you didn't find any bug, look for a different project or tool.
- Fill-in the form to sign up.
    - `https://tinyurl.com/FMI-PV2025-ProjectForm`
    - No signing up, no project
    - Please respect the deadline
- We will work on the principle "first come, first served"
    - There will be no projects using the same open-source repository.

Phase 3: Write a report

DEADLINE: 4 May 2025

TODO: submit the report (see below)

Write a report describing your experience with the verification of the project and present it.

Report of the project (20p)

- an overview of the open-source repository
- an overview of the verification tool used and how to use it
- presentation of the bugs found and suggestions to fix them
- max 10 pag
- submit the report as a pdf
  `https://tinyurl.com/FMI-PV2025-Project`

Presentation of the project (10p)

- 15 mins presentation in which you will give on overview of the project, the verification tool used, and the bugs found.

Project: bonus

Bonus paths (only one applies)

30p If you submit a pull request that fixes one of the bugs presented in the project and it gets merged.

10p If you open an issue on the source-repository describing one of the bugs presented in the project, and it gets acknowledged.

These situations should be handled before you present your report and details should be included in the report.

Project: 30 points (+ potentially 30 points bonus)

DEADLINE 1: 11 March 2025
TODO: nothing for now.

DEADLINE 2: 7 April 2025
TODO: fill-in the form to sign up
`https://tinyurl.com/FMI-PV2025-ProjectForm`

DEADLINE3: 4 May 2025
TODO: submit the report `https://tinyurl.com/FMI-PV2025-Project`

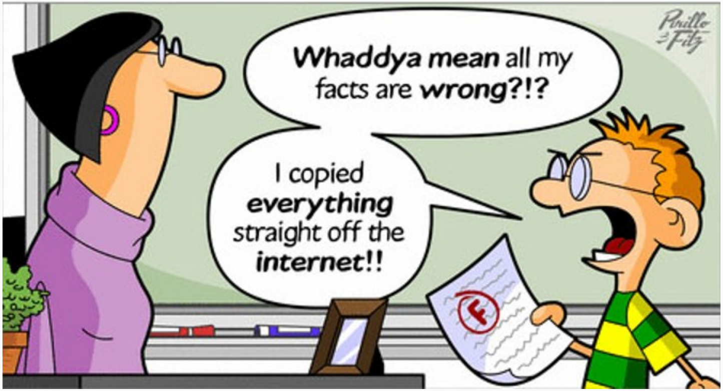
After 5 May
Project presentations; the schedule will be announced

Bibliography

- *Logic in Computer Science: Modeling and Reasoning about Systems*, 2nd edition, Michael Huth, Mark Ryan, Cambridge University Press, 2004.
- *Model Checking*, Edmund M. Clarke, O. Grumberg, Doron A. Peled, MIT Press, 2000.
- *Systems and Software Verification: Model-Checking Techniques and Tools*, B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, Springer, 2001.
- *Practical Foundations for Programming Languages*, 2nd edition, Robert Harper, Cambridge University Press, 2016.
- *Verification of Sequential and Concurrent Programs*, 3rd edition, Krzysztof R. Apt, Frank S. de Boer, Ernst-Rüdiger Olderog, Springer, 2009.

Academic Dishonesty

Don't do it!

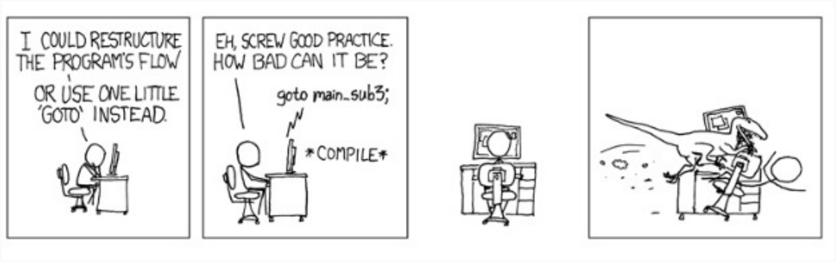

Whaddya mean all my facts are wrong?!?
I copied everything straight off the internet!!
F
Pirillo & Fitz

Why formal verification?

Computer scientists?

What is (or should be) the essential preoccupation of computer scientists?

The production of reliable software, its maintenance, and safe evolution year after year (up to 20 even 30 years).


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE

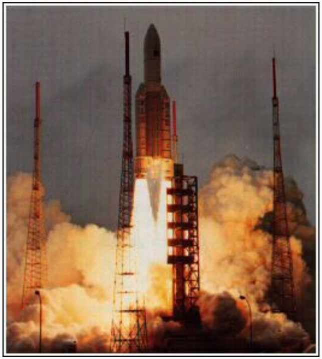
Software bugs

- Bugs are everywhere!
- Bugs can be very difficult to discover in huge software
- Bugs can have catastrophic consequences either very costly or inadmissible
- Here you can find a collection of "famous" bugs
- Here you can find 10 biggest software bugs and tech fails of 2023

The cost of software failure

- Patriot MIM-104 failure, 25 February 1991
  - death of 28 soldiers
  - An Iraqi Scud hit the Army barracks in Dhahran, Saudi Arabia. The Patriot defense system had failed to track and intercept the Scud.
  - R. Skeel. *Roundoff Error and the Patriot Missile*.
- Ariane 5 failure, 4 June 1996
  - cost estimated at more than 370 000 000$
  - M. Dowson. *The Ariane 5 Software Failure*
- Toyota electronic throttle control system failure, 2005
  - at least 89 deaths
  - CBSNews. *Toyota "Unintended Acceleration" Has Killed 89*.
- Heartbleed bug in OpenSSL, April 2014
- The DAO attack on the Ethereum Blockchain in June 2016
- . . .

Ariane 5



Maiden flight of the Ariane 5 Launcher, 4 June 1996

Ariane 5

40s after launch...

Cause: software error

- arithmetic overflow in unprotected data conversion
  from 64-bit float to 16-bit integer types

The DAO Attack on the Ethereum Blockchain — 2016

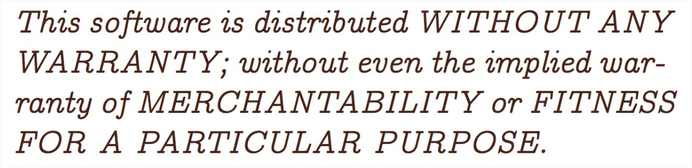
Timeline

30th April	The DAO is launched with a 28 day crowd-founding window
15th May	More than 100 million US dollars were raised
12th June	Stephan Tual, one of The DAO's creators: • a "recursive call bug" has been found in the software • ... but "no DAO funds [are] at risk".
by 18th June	More than 3.6m ether ($\approx$ 50m US dollars) were drained from the DAO account using that bug
15th July	Ethereum splits in two

ETH	Rewrite the history to reverse effects of the attack
ETC	Accept the aftermath of the attack

Who cares?

- No one is legally responsible for bugs:


This software is distributed WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

- Even more, one can even make money out of bugs!
  (customers buy the next version to get around bugs in software)

How can we avoid such failures?

- Choose a common programming language.
  C (low level) / Ada, Java (high level)
- Carefully design the software.
  There exists many software development methods
- Test the software extensively.

How can we avoid such failures?

- Choose a common programming language.

  C (low level) / Ada, Java (high level)

  yet, Ariane 5 software was written in Ada

- Carefully design the software.

  There exists many software development methods

  yet, critical embedded software follow strict development processes

- Test the software extensively.

  yet, the erroneous code was well tested. . . on Ariane 4

Not sufficient!

How can we avoid such failures?

- Choose a common programming language.

  C (low level) / Ada, Java (high level)

  yet, Ariane 5 software was written in Ada

- Carefully design the software.

  There exists many software development methods

  yet, critical embedded software follow strict development processes

- Test the software extensively.

  yet, the erroneous code was well tested. . . on Ariane 4

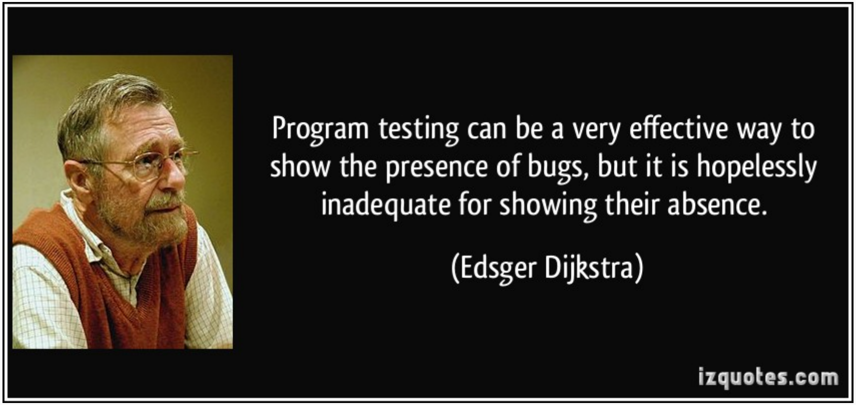
Not sufficient!

We can also use **Formal Methods** and **Formal Verification!**

(provide rigorous, mathematical insurance ♡)

Formal Verification - overview

Testing is necessary, but insufficient



Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

(Edsger Dijkstra)

izquotes.com

For correctness, we need Formal Methods

"The job of formal methods is to elucidate the assumptions upon which formal correctness depends."

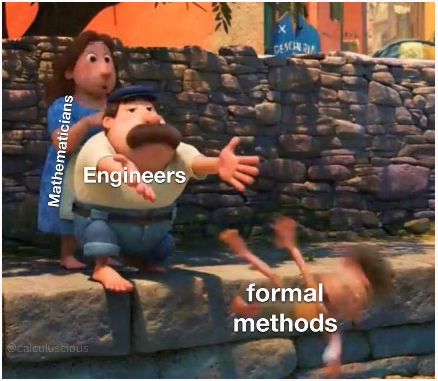
Tony Hoare

Formal Methods

Formal methods are a particular kind of mathematically based techniques for

- specification
- development
- verification

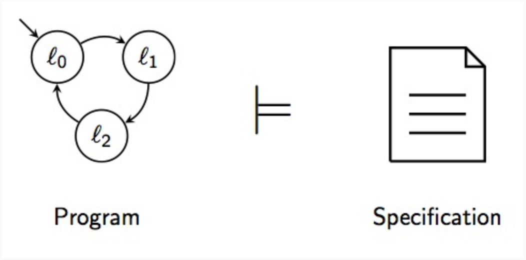
of software and hardware systems.



Formal Program Verification

Formal program verification is about proving properties of programs using logic and mathematics.

In particular, it is about proving that programs meet their specifications.



Formal program verification

Rice's Theorem (1951):

The problem Program $\models$ Specification

is undecidable!

Automated software verification by formal methods is undecidable whence thought to be impossible.

There are powerful workarounds!

Current state-of-the-art

We can check for the absence of large categories of bugs
(maybe not all of them but a significant portion of them).

Some bugs can be found completely automatically,
without any human intervention.

What gets verified?

- Hardware
- Compilers
- Programs
- Specifications

Current state-of-the-art

Tools/programming languages for formal verification:

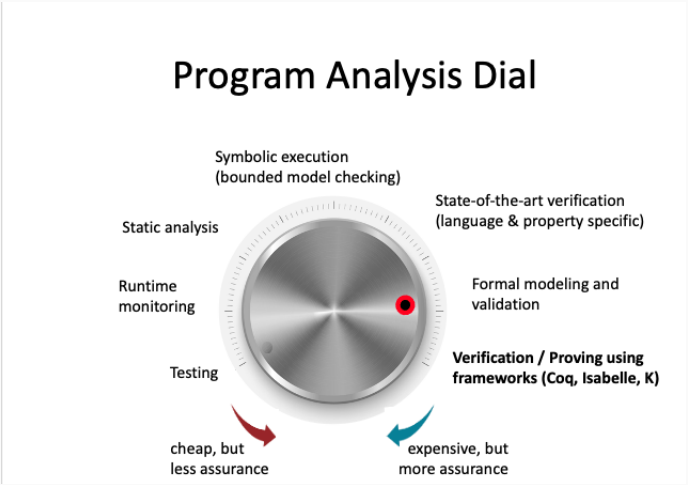
- Infer
- Spark Pro
- Dafny
- KeY
- Alloy
- ASTRÉE
- Terminator
- Frama-C
- Model checkers
- SAT solvers
- SMT solvers
- VeriFast
- SAGE
- KLEE
- Spec #
- . . .

Tools for static code analysis

Model checking tools

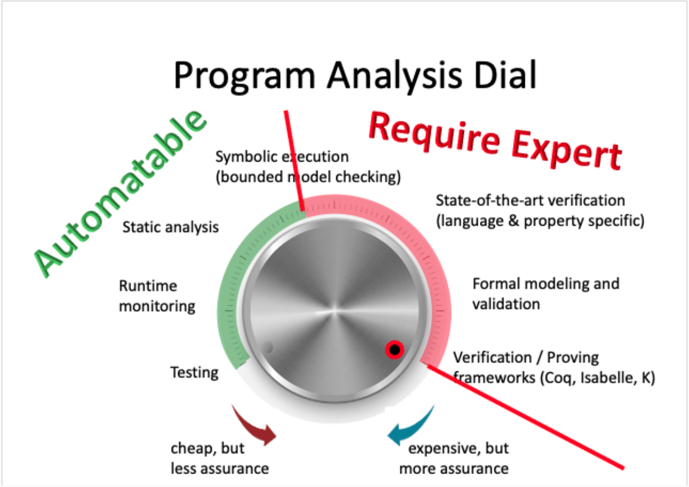
More tools

Why don't more people use formal verification?


Program Analysis Dial
Symbolic execution (bounded model checking)
State-of-the-art verification (language & property specific)
Static analysis
Runtime monitoring
Formal modeling and validation
Testing
Verification / Proving using frameworks (Coq, Isabelle, K)
cheap, but less assurance
expensive, but more assurance

The question of whether to verify formally or not ultimately comes down to how disastrous occasional failure would be.

Why don't more people use formal verification?


Program Analysis Dial
Automatable
Require Expert
Symbolic execution (bounded model checking)
Static analysis
Runtime monitoring
Testing
State-of-the-art verification (language & property specific)
Formal modeling and validation
Verification / Proving frameworks (Coq, Isabelle, K)
cheap, but less assurance
expensive, but more assurance

The question of whether to verify formally or not ultimately comes down to how disastrous occasional failure would be.

What is Program Analysis?

- Very broad topic, but generally speaking, (automated) analysis of program behaviour.
- Program analysis is about developing algorithms and tools that can analyze other programs.

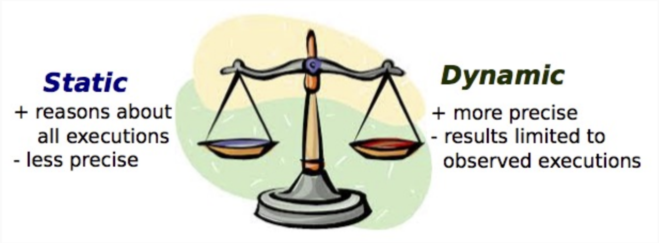
Applications of program analysis

- Bug finding (e.g., expose as many assertion failures as possible)
- Security (e.g., does an app leak private user data?)
- Verification (e.g., does the program analysis behave according to its specifications?)
- Compiler optimizations (e.g., which variables should be kept in registers for fastest memory access?)
- Automatic parallelization (e.g., is it safe to execute different loop iterations on parallel?)

Dynamic vs. Static Program Analysis

Two flavours of program analysis:

- Dynamic analysis: analyses programs while it is running
- Static analysis: analyses source code of the program


Static
+ reasons about all executions
- less precise
Dynamic
+ more precise
- results limited to observed executions

source: Lecture Notes "A Gentle Introduction to Program Analysis" by Işıl Dilig

Formal semantics of programs

What is a programming language?

- Syntax – symbols, keywords, well-formed expressions
- Practical – how we can use the programming language
  - manual and good practices
  - implementation (compiler/interpretor)
  - various tools (debugger etc)
- Semantics – the meaning of instructions/programs
  - most of the time ignored

Care este comportamentul corect?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

Care este comportamentul corect?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

- GCC4, MSVC: valoarea întoarsă e 4
- GCC3, ICC, Clang: valoarea întoarsă e 3

Conform standardului limbajului C (ISO/IEC 9899:2018)
Comportamentul programului este nedefinit.

Formal semantics

Why just few languages have a formal semantics?

Too Hard?

- Modeling a real-world language is hard
- Notation can get very dense
- Sometimes requires developing new mathematics
- Not yet cost-effective for everyday use

Overly General?

- Explains the behaviour of a program on every input
- Most programmers are content knowing the behavior of their program on this input (or these inputs)

Who needs semantics?

Unambiguous description

- Anyone who wants to design a new feature
- Basis for most formal arguments
- Standard tool in Programming Languages research

Exhaustive reasoning

- Sometimes have to know behaviour on all inputs
- Compilers and interpreters
- Static analysis tools
- Program transformation tools
- Critical software

Formal semantics

To analyze/reason about programs, we must know what they mean.

Formal semantics - three approaches:

- Operational semantics
  - Models program by its execution on an abstract machine
  - Useful for implementing compilers and interpreters
- Denotational semantics
  - Models program as mathematical objects
  - Useful for theoretical foundations
- Axiomatic semantics
  - Models program by the logical formulas it obeys
  - Useful for proving program correctness

Operational semantics

- Gordon Plotkin in the 1980s
- Describes how programs compute
- Relatively easy to define
- Close connection to implementation

- Describe how a valid program is interpreted as sequences of computational steps. These sequences are the meaning of the program
- Works well for sequential, object-oriented programs, parallel, distributed programs
- The most popular style of semantics

Operational semantics

- Evaluation is described as transitions in some (typically idealized) abstract machine. The state of the machine described by current expression.
- There are different styles of abstract machines.
- The meaning of a program can be
    - its fully reduced form (aka a value), for deterministic languages
    - all its possible executions and interactions, for nondeterministic/interactive languages
- A small-step semantics describes how such an execution proceeds in terms of successive reductions.

Operational semantics

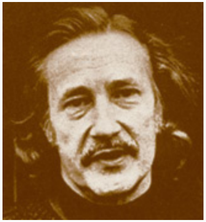
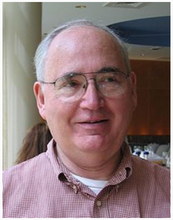
Example (Small-step semantics)

- Assume an abstract machine whose configurations have two components:
  - the expression e being evaluated
  - a store σ that records the values of variables
- Then, a small-step semantics describes the execution as a succession of one-step transitions of the form $\langle e, \sigma \rangle \to \langle e', \sigma' \rangle$

$$\begin{aligned}
\langle \texttt{int}\ x = 0; x = x + 1;\ ,\ \emptyset \rangle &\to \langle x = x + 1;\ ,\ x \mapsto 0 \rangle \\
&\to \langle x = 0 + 1;\ ,\ x \mapsto 0 \rangle \\
&\to \langle x = 1;\ ,\ x \mapsto 0 \rangle \\
&\to \langle \{\}\ ,\ x \mapsto 1 \rangle
\end{aligned}$$

Denotational semantics

- Christopher Strachey and Dana Scott published in the early 1970s

- Construct mathematical objects that describe the meaning of the blocks in the language
- Works well for sequential programs, but it gets a lot more complicated for parallel and distributed programs.

Axiomatic semantics

- Operational and denotational semantics let us reason about the meaning of a program.
- Axiomatic semantics define a program's meaning in terms of what one can prove about it.
- Useful for reasoning about correctness of programs

Quiz time!

https://tinyurl.com/FMI-PV2023-Quiz1

See you next time!