C01 – Intro

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

Why formal verification?

Formal verification - overview

Program analysis

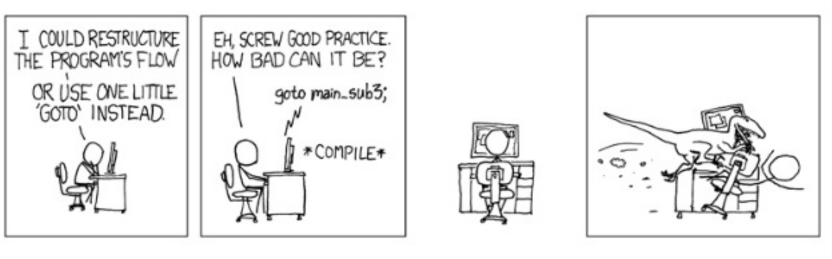
Formal semantics of programs

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).



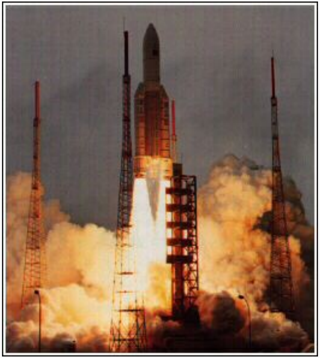
Software bugs

- 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

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...

Ariane 5

Cause: software error

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

```
P_M_DERIVE(T_ALG.E_BH) :=
  UC_16S_EN_16NS (TDB.T_ENTIER_16S
    ((1.0/C_M_LSB_BH) * G_M_INFO_DERIVE(T_ALG.E_BH)));
```

- software exception not caught
  $\Rightarrow$ computer switched off

- all backup computers run the same software
  $\Rightarrow$ all computers switched off, no guidance
  $\Rightarrow$ rocket self-destructs

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

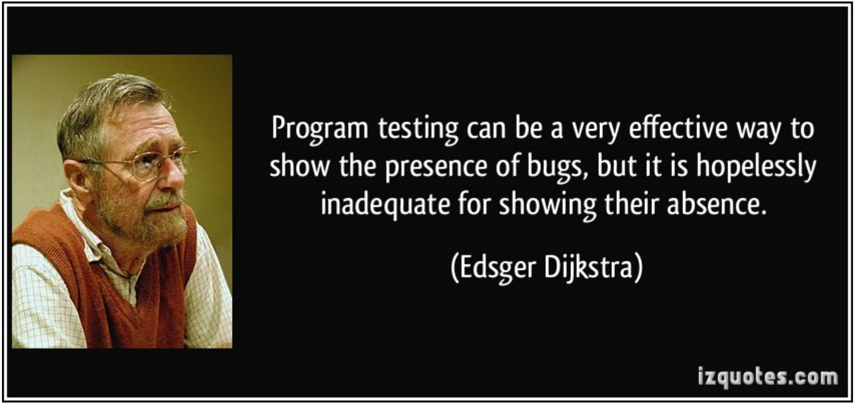
Not sufficient!

We should use **Formal Methods** and **Formal Verification!**

(provide rigorous, mathematical insurance ♡)

Formal verification - overview

Testing

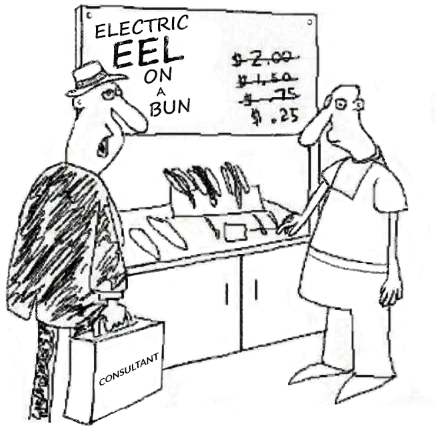

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
(Edsger Dijkstra)
izquotes.com

Formal methods

Formal methods are a particular kind of mathematically based techniques for

- specification
- development
- verification

of software and hardware systems.

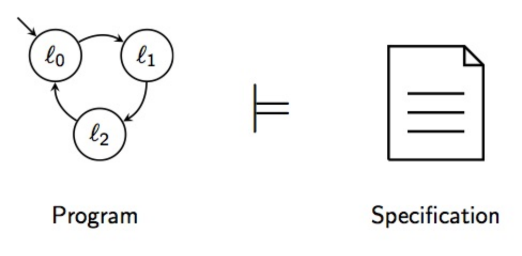


THE ONLY THING HARDER TO SELL THAN FORMAL METHODS

Formal program verification

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

In particular, it is about proving they meet their specifications.



Formal program verification

Rice's Theorem (1951):

The question Program $\models$ Specification

is undecidable!

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

Formal program verification

Rice's Theorem (1951):

The question 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

Powerful 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

Formal program verification

Why don't more people use formal verification?

- Time consuming
- Expensive

Formal program verification

Why don't more people use formal verification?

- Time consuming
- Expensive

Formal or Informal?

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

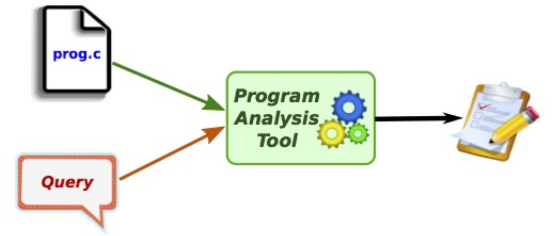
Formal verification methods

- Axiomatic semantics
- Deductive verification
  - SAT solvers
  - SMT solvers
  - Interactive theorem provers
  - Automatic theorem provers
- Model checking
- Abstract interpretation
- Type systems
- Lightweight formal methods
- Proof Assistants
- ...

Program analysis

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.



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

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?

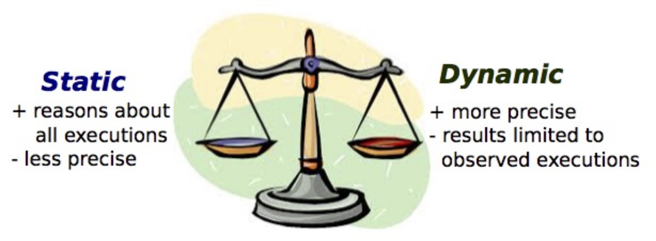
- Automatic parallelization

  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



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

Static Analysis

- Program analysis without running the program.
- It is nearly as old as programming
- It is a big field, with different approaches and applications
- List of tools for static code analysis
- Analysis paradigms:
    - Type systems
    - Dataflow analysis
    - Model checking

Type Systems

- Types are most widely used static analysis.
- A type is an example of an abstract value
  - Represents a set of concrete values
  - Every static analysis has abstract values
- A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute. - Pierce.
- Unfortunatelly, we will not cover this topic in this course.

Formal semantics of programs

Formal semantics

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

Formal semantics

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

Formal semantics - three approaches:

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

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

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

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

Operational semantics

- Gordon Plotkin in the 1980s
- 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.

Operational semantics

- Describes how programs compute
- Relatively easy to define
- Close connection to implementation
- This is 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 $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$

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 $\langle e, \sigma \rangle \to \langle e', \sigma' \rangle$

$\langle \texttt{int}\ x = 0; x = x + 1;\ ,\ \emptyset \rangle$

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 $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$

$$\langle \texttt{int}\ x = 0; x = x + 1;\ ,\ \emptyset \rangle \quad \rightarrow \quad \langle x = x + 1;\ ,\ x \mapsto 0 \rangle$$

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 $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$

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

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 $\langle e, \sigma\rangle \to \langle e', \sigma'\rangle$

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

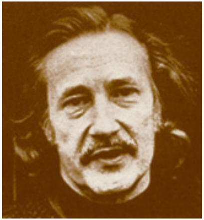
Example (Small-step semantics)

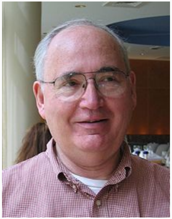
- Assume an abstract machine whose configurations have two components:
  - the expression e being evaluated
  - a store σ that records the values of variables
- Then, a small-step semantics describes the execution as a succession of one-step transitions of the form $\langle e, \sigma \rangle \rightarrow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

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

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://www.questionpro.com/t/AT4NiZrHIa

See you next time!