Program analysis: from proving correctness to proving incorrectness

Roberto Bruni, Roberta Gori
(University of Pisa)
Lecture #01

BISS 2024
March 11-15, 2024
A software bug is an error, flaw or fault in the design, development, or operation of computer software that causes it to produce an incorrect or unexpected result.
Software Verification

Correctness
the aim is to prove the absence of bugs

Incorrectness
the aim is to prove the presence of bugs
The need for verification

Friday, 24th June [1949]

Checking a large routine by Dr A. Turing.

How can one check a routine in the sense of making sure that it is right?
Ariane 5 Rocket Explosion (1996)

Caused due to numeric overflow error

Attempt to fit 64-bit format data into 16-bit space

Cost: $100M for loss of mission

Multi-year set back to the Ariane program

Read more at:
https://www.bugsnag.com/blog/bug-day-ariane-5-disaster/
Unfortunately

It was one of the most serious but not the only one….

SOFTWARE HORROR STORIES

https://www.cs.tau.ac.il/~nachumd/horror.html

Toyota unintended acceleration
4 people died

Boeing 747 Max Crashes
350 people died
Costs of SW bugs

Knight Capital Trading Glitch (2012)
$ 440 M

Nissan Airbag Malfunction (2014)
1 Million Vehicles Recalled


Software Fails Watch (Tricentis, 2017): SW bugs lead to $ 1.7 Trillion revenue lost.


Complexity of programs

Size of Linux Kernel

Avg. Size of Android Apps

always increasing!
The main question

Will our program behave as we intended?

We need to analyse all executions of the program

The semantics of a program is a description of its run-time behaviors

Checking if a software will run as intended is equivalent to checking if the code satisfies a (semantic) property of interest
Success stories

A long time before success

Computer-assisted verification is an old idea
- Turing, 1948
- Floyd-Hoare logic, 1969

Success in practice: only from the mid-1990s
- Importance of the increase of performance of computers

A first success story:
- Paris metro line 14, using Atelier B (1998, refinement approach)

Other Famous Success Stories

  http://www.astree.ens.fr/
- Microsoft’s hypervisor: using Microsoft’s VCC and the Z3 automated prover (2008, deductive verification)
  More recently: verification of PikeOS
- Certified C compiler, developed using the Coq proof assistant (2009, correct-by-construction code generated by a proof assistant)
  http://compcert.inria.fr/
- L4.verified micro-kernel, using tools on top of Isabelle/HOL proof assistant (2010, Haskell prototype, C code, proof assistant)
Forward semantics for deterministic programs

We start from input state $\sigma$ and we want to characterise the reachable output states.

Input store $\sigma$ \rightarrow A program $c$ \rightarrow Output store $[[c]]\sigma$

$\sigma \rightarrow[[c]]\sigma = \bot$ Non terminating execution

Denotational semantics $[[c]] : \Sigma \rightarrow \Sigma_{\bot}$
Collecting semantics for deterministic programs

A program $c$

Input stores $\sigma$

Output store $[[c]]\sigma$

$[[c]]P = \bigcup_{\sigma \in P} [[c]]\sigma$

Denotational semantics $[[c]] : \Sigma \rightarrow \Sigma_{\perp}$

Collecting semantics $[[c]] : \wp(\Sigma) \rightarrow \wp(\Sigma)$
Ideal exact analysis

\[[c]\] : \wp(\Sigma) \rightarrow \wp(\Sigma)

\( \forall P . \forall \sigma \in [c]P . \sigma(x) \neq 0 \)

semantic property of a program: a property about \( [c] \)
Undecidability in the way

non trivial property:
- there exists a program $c$ such that $\mathcal{P}(c)$ holds true
- and there exists also some program $c$ such that $\mathcal{P}(c)$ is false

Rice theorem.
Let $\mathcal{P}(c)$ be a non trivial semantic property of programs $c$.
There exists no algorithm such that, for every program $c$, it returns true if and only if $\mathcal{P}(c)$ holds true

no analysis method that is automatic, universal, exact!
For some program...

\[ \mathcal{P}(c) \equiv \forall P \neq \emptyset . \exists \sigma \in [[c]]P . \sigma(x) \neq 0 \]

c \triangleq

x := 1;
and for some other program...

\[ \mathcal{P}(c) \equiv \forall P \neq \emptyset. \exists \sigma \in [c]P. \sigma(x) \neq 0 \]

c \triangleq 
while (n>1) {
    n := n+1;
    x := 0;
}

x := 1;
but for Collatz’s conjecture?

\[ \mathcal{P}(c) \equiv \forall P \neq \emptyset . \exists \sigma \in [c]P . \sigma(x) \neq 0 \]

\[ c \triangleq \]

while (n>1) {
    if (even(n)) { n := n/2; }
    else { n := 3n+1; }
}% does it terminate for any value of n?

\[ x := 1; \]

As of 2020, the conjecture has been checked by computer for all starting values up to \( 2^{68} \approx 10^{20} \).
Limitations of the analysis

no analysis method that is automatic, universal, exact!

We need to give something up:

automation: machine-assisted techniques

the universality “for all programs”:
targeting only a restricted class of programs

claim to find exact answers: introduce approximations
Over approximations

Good for proving correctness

Bad for bug-findings!

true positive

false positive

true negative
Under approximations

Good for bug-findings!

- True positive

Bad for proving correctness

- False negative
- True negative
<table>
<thead>
<tr>
<th></th>
<th>Automatic</th>
<th>Over-approximation</th>
<th>Under-approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Machine-assisted Verification</td>
<td>Yes/No</td>
<td>Yes/No</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Bounded model checking</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Abstract Interpretation</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Correctness: forward approach

A program $P$ such that $\exists c \in P. [[c]]P \subseteq Q$.

For all $\sigma \in P$, $[[c]]\sigma$ either does not terminate or terminates in $Q$. 

\[
\forall \sigma \in P. [[c]]\sigma \text{ either does not terminate or terminates in } Q
\]
Correctness: backward approach

A program

$c$

$P$ ⊆ $wlp(c, Q)$

$[c]P ⊆ Q$

Dijkstra's weakest liberal precondition

$wlp(c, Q) = \{ \sigma \mid [c] \{ \sigma \} ⊆ Q \}$
Nondeterministic programs

Some programs may exhibit nondeterministic behaviour (lack of information, approximation, programming constructs $c_1 + c_2$)

A program $c$

$[[c]] : \Sigma \rightarrow \wp(\Sigma)$

$[[c]]P \subseteq Q$

all the outputs starting from $\sigma \in P$ either do non terminate or terminate in $Q$

$P \subseteq \text{wlp}(c, Q)$
An example: non-termination analysis

Given a program $c$ and an input store $\sigma$ does $[[c]]_\sigma = \emptyset$?

Using over-approximation: we try to prove $[[c]]_\sigma \subseteq \emptyset$

Using under-approximation: we try to prove $[[c]]_\sigma \supseteq Q$ for some $Q \neq \emptyset$
## What we will see

<table>
<thead>
<tr>
<th></th>
<th>Forward</th>
<th>Backward</th>
<th>Over-approximation</th>
<th>Under-approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hoare Logic (HL)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Incorrectness Logic (IL)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locally Complete Logic (LCL)</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Necessary Condition (NC)</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Sufficient Incorrectness Logic (SIL)</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Separation Logic (SL)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Incorrectness SL</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Separation SIL</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>UNTer</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Questions
Let $c \triangleq (z := x) + (z := y)$
and let $P \triangleq (x = y = 0)$
What is $[[c]]P$?

$(x = y = z = 0)$
Question 2

Let \( c \triangleq \text{if } x < y \text{ then } x := y \text{ else (while true do skip)} \)
and let \( Q \triangleq (x = y = 0) \)
What is \( wlp(c, Q) \)?

\[ (x \geq y \lor y = 0) \]