Program Testing and Analysis:
Introduction and Foundations

Dr. Michael Pradel
Software Lab, TU Darmstadt
About Me

- Michael Pradel
- At TU Darmstadt since 2014

Before joining TUD

- Master-level studies in Dresden and Paris
- Master thesis at EPFL, Switzerland
- PhD at ETH Zurich, Switzerland
- Postdoctoral researcher at UC Berkeley, USA
About the Software Lab

- My research group; since 2014
- Research on tools and techniques for building reliable, efficient, and secure software
  - Program analysis
  - Test generation
  - Focus: JavaScript-based applications & Concurrency
- Thesis and job opportunities
Plan for Today

- Introduction
  - What the course is about
  - Why it is interesting
  - How it can help you

- Organization
  - Course projects
  - Term paper
  - Mid-term and final exam

- Foundations
  - Grammars, ASTs, CFGs, CGs, PDGs, etc.
What you probably know:

- Manual testing or semi-automated testing:
  JUnit, Selenium, etc.

- Manual ”analysis” of programs:
  Code inspection, debugging, etc.

Focus of this course:

**Automated** testing and program analysis
Why Do We Need It?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm
Why Do We Need It?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm

0.5-25/KLoC in delivered software
Why Do We Need It?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm

1.5 years to find a bug

[Palix2011]
Why Do We Need It?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm

Ariane 5
Northeast blackout
Therac-25
What is Program Analysis?

- Automated analysis of program behavior, e.g., to
  - find programming errors
  - optimize performance
  - find security vulnerabilities
What is Program Analysis?

- Automated analysis of program behavior, e.g., to
  - find programming errors
  - optimize performance
  - find security vulnerabilities

Input → Program → Output

Additional information
What is Program Analysis?

- Automated analysis of program behavior, e.g., to
  - find programming errors
  - optimize performance
  - find security vulnerabilities
Static vs. Dynamic Analysis

Static

- Analyse source code, byte code, or binary
- Typically:
  - Consider all inputs
  - Overapproximate possible behavior

Dynamic

- Analyze program execution
- Typically:
  - Consider current input
  - Underapproximate possible behavior
## Static vs. Dynamic Analysis

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Analyse source code, byte code, or binary</td>
<td>- Analyze program execution</td>
</tr>
<tr>
<td>- Typically:</td>
<td>- Typically:</td>
</tr>
<tr>
<td>- Consider all inputs</td>
<td>- Consider current input</td>
</tr>
<tr>
<td>- Overapproximate possible behavior</td>
<td>- Underapproximate possible behavior</td>
</tr>
</tbody>
</table>

**E.g., compilers, lint-like tools**

**E.g., automated testing, profilers**
Example

// JavaScript
var r = Math.random(); // value in [0, 1)
var out = "yes";
if (r < 0.5)
    out = "no";
if (r === 1)
    out = "maybe";
console.log(out);

Quiz: What are the possible outputs?
Example

// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
    out = "no";
if (r === 1)
    out = "maybe"; // infeasible path
console.log(out);

Overapproximation: "yes", "no", "maybe"

- Consider all paths (that are feasible based on limited knowledge)
Example

// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
    out = "no";
if (r === 1)
    out = "maybe"; // infeasible path
console.log(out);

Underapproximation: "yes"

- Execute the program once
Example

// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
    out = "no";
if (r === 1)
    out = "maybe"; // infeasible path
console.log(out);

Sound and complete: "yes", "no"

■ For this example: Can explore both feasible paths
Another Example

// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);
Another Example

// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);

Overapproximation: Any value

- Consider all paths (that are feasible based on limited knowledge about Math.random())
Another Example

// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);

Underapproximation:
Some number in [0,2), e.g., 1.234

- Execute the program once
Another Example

// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);

Sound and complete?

- Exploring all possible outputs:
  Practically impossible
- This is the case for most real-world programs
Over- vs. Underapproximation

Handwritten notes...
Program $P$, Input $i$, Behavior $P(i)$

All possible behavior (what we want, ideally)

Underapproximation (e.g., testing, dyn. a.) $\Rightarrow$ False negative

Overapproximation (e.g., most static a.) $\Rightarrow$ False pos.

False negative (e.g., missed bug)

False positive (e.g., benign warning)
Test Generation

- Dynamic analysis:
  Requires input to run the program

- Test generation:
  Creates inputs automatically

- Examples
  - Generate JUnit tests:
    Input = sequence of method calls
  - UI-level test generation:
    Input = sequence UI events
  - Fuzz-test a compiler: Input = program
How Does All This Help Me?

Improve the quality of your code

- Fewer bugs
- Better performance
- More secure software

Save time during manual testing

Become a better developer

- Get better understanding of program’s behavior
- Avoid common pitfalls
- Learn to use and write tools
Plan for Today

■ Introduction
  □ What the course is about
  □ Why it is interesting
  □ How it can help you

■ Organization
  □ Course projects
  □ Term paper
  □ Mid-term and final exam

■ Foundations
  □ Grammars, ASTs, CFGs, CGs, PDGs, etc.
Organization

■ Weekly lectures
■ Weekly reading material
■ Throughout the semester:
  □ Course project
  □ Term paper
■ End of November: Mid-term exam
■ End of semester: Final exam
Grading

- Weekly lectures
- Weekly reading material
- Throughout the semester:
  - Course project 33%
  - Term paper 33%
- End of November: Mid-term exam +10%
- End of semester: Final exam 33%
Grading

- Weekly lectures
- Weekly reading material
- Throughout the semester:
  - Course project 33%
  - Term paper 33%
- End of November: Mid-term exam +10%
- End of semester: Final exam 33%
Grading

- Weekly lectures
- Weekly reading material
- Throughout the semester:
  - Course project: 33%
  - Term paper: 33%
- End of November: Mid-term exam: +10%
- End of semester: Final exam: 33%

Must pass all three to pass the course
A Friendly Warning

This is not going to be an easy course!

- Read regularly (otherwise, you won’t be able to catch up)
- Work regularly on the course project
- Schedule enough time to work on the term paper
Programming Language

Most concepts taught in this course: Language-independent

Course projects and most examples: JavaScript (specifically: ECMAScript 5)

- Very popular
- Client-side web applications, but also for server, mobile, and desktop applications
- Various interesting research challenges
Piazza

Platform for discussions, in-class quizzes, and sharing additional material

- Please register and enroll for the class
- Use for all questions related to the course
- Starting from next week, messages sent to all students go via Piazza (not TUCaN!)
Learning Material

There is no script or single book that covers everything

- Slides and hand-written nodes:
  Available after lecture
- Pointers to papers, book chapters, and web resources
Course Project

- Independent research project
- Design, implement, and evaluate a program analysis and/or test generator
- Teams of 2 or 3 students
  - One team, one grade
Course Project: Tools

Based on existing frameworks and tools

- Jalangi: Dynamic analysis framework
- WebAppWalker: UI-level test generation framework
- Esprima & Escodegen: ASTs, parsing, code generation
Course Project: Organization

Timeline

- Oct 19: Register teams and preferred projects
- Week of Oct 26–30: Meeting with mentor
- Week of Dec 1–4: Meeting with mentor
- Week of Jan 18–22: Meeting with mentor
- Feb 8: Presentation of results
- Feb 19: Final submission

(There are office hours for additional meetings.)

Project proposals will be available via Piazza
1) Implementation and results
   - Source code and everything needed to reproduce the results
   - Packaged as VM image

2) Report
   - 10 pages maximum, English
   - Written like a scientific paper

Due on Feb 19
Write a scientific article that summarizes and compares three existing papers

- Topic & papers: Based on lecture content
- Individual work
- 6 pages maximum, English
- Peer reviewing
Term Paper

Write a scientific article that summarizes and compares three existing papers

- Topic & papers: Based on lecture content
- Individual work
- 6 pages maximum, English
- Peer reviewing

Grading: $\frac{2}{3} \cdot$ final paper + $\frac{1}{3} \cdot$ reviews
Term Paper: Some Advice

- Don’t waste space on basics
- Examples are your secret weapon
- Most important part: Comparison of the three papers
- Bad English distracts from good content
- Revise, revise, revise
Term Paper: Rules

- No verbatim copying of text (exception: quotes)
- You may copy figures (e.g., result graphs)
- You must use your own example(s)
Term Paper: Reviews

- Imitates peer reviewing process
- Each student reviews three term papers
- Revise your term paper after getting reviews
  - Grade will be for final term paper
- Plain text format
- About 1 page, English
Reviews: Some Advice

- Be constructive
- Be polite
- Your reviews contribute to your grade, not to the reviewee’s grade
Term Paper: Organization

Timeline

- Nov 2: Register with preferred topics
- Jan 11: Submit paper for peer review
- Feb 1: Reviews due
- Feb 19: Final version of paper due
Exams

Mid-term exam (written)
- Advisable but not mandatory
- Can improve overall grade up to 10%
- Last week of November

Final exam (written)
- After the last lecture (exact date TBD)
Exams

Mid-term exam (written)
- Advisable but not mandatory
- Can improve overall grade up to 10%
- Last week of November

Final exam (written)
- After the last lecture (exact date TBD)

For both:
- Open book: Bring books, papers, etc.
- Corollary: Will test your understanding, not your memory!
Academic Integrity

- Work you submit must be your own/your team’s work
- Unauthorized group efforts and any form of plagiarism are considered academic dishonesty and will be punished
- Allowed to discuss the problem with your peers, but not to copy or reuse any part of an existing solution
Plan for Today

■ Introduction
  □ What the course is about
  □ Why it is interesting
  □ How it can help you

■ Organization
  □ Course projects
  □ Term paper
  □ Mid-term and final exam

■ Foundations
  □ Grammars, ASTs, CFGs, CGs, PDGs, etc.
Foundations

Hand-written notes ...
1) Programming Languages

→ Paradigms:
  - Imperative, e.g., C
  - Functional, e.g., Haskell
  - Object-oriented, e.g., Java
  - Logic, e.g., Prolog

→ Components:
  - Syntax (= form)
  - Semantics (= meaning)
  - Implementation (= execution)

{ our focus }
2) Language Implementation

```
source code -> lexical analysis -> tokens

Syntax (parsing)
```

```
Syntax -> semantic analysis

Machine code generator

machine code
```
b) Interpretation

c) Hybrid implementation

Example 1: Java
- compile to bytecode
- interpreted by VM
- (optionally), compiled to machine code → JIT

Example 2: JavaScript
- interpret source code
- JIT compilation
3) Syntax
   a) Grammar → which programs are syntactically correct

4 parts: terminals $\Sigma$, non-terminals $N$, productions $P$, 
initial symbol $S \in N$

Example: Arithmetic expressions

$\Sigma = \{0, 1, 2, \ldots, 9, +, -\}$

$N = \{\text{Exp}, \text{Num}, \text{Op}, \text{Digit}\}$

$S = \text{Exp}$

$P: \text{Exp} \rightarrow \text{Num} | \text{Exp Op Exp}$

$\text{Op} \rightarrow + | -$ 

$\text{Num} \rightarrow \text{Digit} | \text{Digit Num}$

$\text{Digit} \rightarrow 0 | 1 \ldots 9$

What is part of lang.?

A) 12-2
B) 2+ (12-4)
C) 11 * 4
D) 1234567890
5) Abstract Syntax Trees

- abstract grammar for example:
  \[ E \rightarrow n \mid \text{Op} (E, E) \]
  \[ \text{Op} \rightarrow + \mid - \]
- terminals = tokens

- e.g.: \[ 3 + 45 \]
4) Control flow graphs → models flow of control of program

\[ G = (N, E) \] where \( N \) is basic blocks = seq. of operations executed together

\( E \) is possible transfer of control

\( e = (n_1, n_2) \in E \) means that \( n_2 \) may execute after \( n_1 \)

**Ex 1:**

\[
\text{if } (c) \\
\quad x = 5 \\
\text{else} \\
\quad x = 7 \\
\text{console.log}(x)
\]
while (c) {
  x++;
  y = x;
}
console.log(x);
5) **Data Dependence Graph** → models flows of values from "definition" to "use".

\[ G = (N, E) \] where \( N \) = operations

\( E \) = possible def-use relations

\( e = (n_1, n_2) \in E \) means that \( n_2 \) may use data defined in \( n_1 \).

**Example 1**

\[ x = 5 \]

\[ y = x + 1 \]
Ex. 2:

\[ x = \ldots \]
\[ y = 5 \]
\[ \text{if} \ (x \geq 1) \]
\[ y = x \]
\[ z = x + y \]
Outlook

- Operational semantics
- Manual testing
- Random and fuzz testing
- Search-based testing
- Symbolic and concolic testing
- Testing concurrent programs
- Path profiling
- Program slicing
- Information flow analysis
- Differential testing
- Specification mining
- Performance profiling