Automatic Testing & Verification

Introduction

Juan Pablo Galeotti, Alessandra Gorla, Software Engineering Chair – Computer Science, Saarland University, Germany
About Us...

- Juan Pablo Galeotti
- Alessandra Gorla

- http://www.st.cs.uni-saarland.de
  - Computer Science – Saarland University
  - Research on:
    - Mining software archives
    - Automated Debugging
    - Mutation Testing
    - Mining Models
  - Bachelor and Master students
    - Always welcome!
Automated analysis is rapidly gaining ground on all software development activities.

**Industry**: Tools are more often used to enhance more traditional QA methodologies.  
- More results are obtained at a lower cost.
Impact on Industry

- "Static Analysis can reduce defects by up to a factor of six!"
  - (Capers Jones, Software Productivity Group)

- Microsoft
  - Code is automatically analyzed using automatic verifiers (PREfix, PREfast)
  - Visual Studio + Code Contracts: Static Analysis + Test Case Generation
  - Kit for verifying drivers is delivered to third-parties vendors.
Impact on Industry

- **Java FindBugs**: > 1.5 million downloads
  - Actively used in Google, Sun, Ebay, etc.
  - It finds “mechanical” errors, error patterns
- Experience in **Google**:
  - >4000 problems in deployed code!
  - More than 80 infinite loops!
Automatic program analysis is increasingly used in software engineering activities

**Industry**: Tools are more often used to enhance traditional QA methodologies
- Concrete results are obtained at lower cost.

**Academia**: A lot of activity
- Program analysis topics gain presence in more important conferences in software engineering, programming and systems
  - Hybrid approaches (static analysis + testing).
  - Bug detection, Multicore (data races, automatic parallelization).
    - More precision, usability, scalability (no longer a bottleneck)
  - Security
Some disasters

- **Therac 25:**
  - Due to a bug, a radio therapy machine applied 100 times the radiation dose on patients
    - A race condition did not allow the machine to detect a change in the operation mode

- **Ariane V Flight 501:**
  - An arithmetic overflow in inertial computation
  - Exception badly handled
  - Reused code from Ariane 4

Any mini-disaster at home?
Why is so hard to build quality programs?

- In other fields,
  - Many failures, but unfrequent
    - A building, a bridge, or a car do not need weekly patches...
  - Well-defined techniques for quality assurance
  - Quality can be measured and predicted
  - Relatively simple parts, composition can be done.

- In software...
  - Systems fail everyday
  - It is very complex to determine quality and predict problems.
  - Building a failure-tolerant system is more complex than “concrete” engineering
    - Replicate components can lead to new problems...
Software Complexity

- A lot of functionality!

- Size:
  - Code millions lines length
    - Each line might be important!
  - State-explosion:
    - A single 32bits variable has $2^{32}$ potential values!!!

- Complexities:
  - Arithmetic, concurrency, dynamic heap memory
  - Complex hardware, heterogenous
  - etc.
“Program analysis is the systematic examination of a software «artifact» to discover its properties”

- **Examination:**
  - Automatic vs. manual

- **Systematic:**
  - Coverage in testing, inspection checklist, exhaustive model checking, etc...

- **Artifact:**
  - Program, execution trace, test case, design, requirements document.

- **Properties:**
  - Functional: correction
  - Non-functional: memory consumption, performance, availability, security
Program Analysis

Analyze (infer/prove properties over) programs

- **Dynamic Analysis:**
  - Program analysis using the execution of the program
  - Characterizes some executions
  - Precise: it knows all concrete states

- **Static Analysis:**
  - The program is analysed without executing code.
  - Characterizes all possible executions
  - Conservative: it approximates concrete states
Verification, Validation, Synthesis, Inference

- Verification
  - Against a specification
    - It might be an implicit specification

- Validation
  - Does the system do what the user wants?
  - Failures in specifications

- Inference
  - Discover some interesting properties about the program

- Synthesis
  - Create a new program: optimize (compiler), control (scheduler)

We will focus on verification and inference
A problem has been detected and Windows has been shut down to prevent damage to your computer.

UNEXPECTEDKERNEL_MODE_TRAP

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 that 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: 0x00000007F (0x00000000, 0xB9FC8E84, 0x00000008, 0xC00000000)

*** Beep.SYS - Address B9FC8E84 base at B9FC7000, DateStamp 36B04C16
Find the bug!

```plaintext
x := 8;
y := x;
z := 0;

while y > -1 do
  x := x / y;
  y := y-2;

z := 5;
```

Division by zero!
Find the bug!

\[ x := 8; \]
\[ y := x; \]
\[ z := 0; \]
\[ \text{while } y > f(x, y, z) \text{ do} \]
\[ \quad x := x / y; \]
\[ \quad y := y - 2; \]
\[ \quad z := 5; \]

Division by zero!
The BIG program

- Remove bugs from our programs.
  - As soon as possible (before deployment).

- Testing:
  - Ejecución directa de código en un ambiente controlado.
  - Identifica y localiza fallas: pero no garantiza ausencia.
  - A veces es demasiado costoso.

- Inspecciones:
  - Evaluación humana de artefactos.
  - Especificaciones, Diseños, Código.
  - Esfuerzo humano importante: propenso a errores, no puede ser exhaustivo por un problema de escala.
  - A veces los errores son "sutiles", se dan luego de una secuencia no trivial de eventos.
Classic techniques

- Remove bugs from our programs.
  - As soon as possible (mainly before deployment).

- **Testing**: Direct execution of a program in a controlled environment
  - Identify and localize faults: it does not guarantee absence
  - It may be too expensive

"Program testing can be used to show the presence of bugs, but never to show their absence!" -- Edsger Dijkstra
Classic Techniques

- Remove bugs from our programs.
  - As soon as possible (mainly before deployment).

- **Dynamic Analysis**: Tools for mining information from program executions
  - Find failures (memory)
  - Invariants
  - Precise but under approximates.

- **Inspections**: Human evaluation of artifacts.
  - Specifications, Designs, Code.
  - Important human effort: error prone, it can not be exhaustive due to scale of artifacts
  - Sometimes failures are “subtle”, they happen after a non-trivial sequence of events.
Static Analysis

- The program is analyzed without being executed.
- It is the systematic examination of an abstraction of the program states.

**Systematic**
- We examine all program paths within one procedure
  - What about loops?
  - The exploration is exhaustive

**Abstraction**
- Keep only relevant information with regard to the property to infer
  - Variable sign (-, 0, +)
  - Set of variables to be read in the future
  - Etc...
More known techniques

- **Deductive verification methods**
  - Formal proofs of correction against specifications.
  - Tool support:
    - Compilers+ theorem provers
    - Type checkers
    - Semi automatic or incomplete, often too costly.
  - They require **annotations** *(types or contracts)*.

- **Model checking**
  - A formal model of a system is analyzed
  - Good for analyzing event interaction (protocols, concurrency, etc)
  - Expensive

- **Dataflow / Abs. Interpretation**
  - “Mechanical” Errors (difficult to exhibit through Testing or inspections):
    - Memory usage (null dereference, non initialized date).
    - Resource “Leaks” (memory, locks, files).
    - Vulnerabilities (buffers overruns, non validated data).
    - Non handled exceptions, concurrency (race conditions), etc.

- **Bug finding**
  - Search for common patterns
  - Good practice enforcement
Static Analysis: typical usage

- **To optimize code**
  - Detect unused variables;
  - Remove dead code
  - Detect more frequently used expressions.
  - Purity analysis.
  - Null dereference.

- **Verification**
  - Implicit or explicit contracts
  - Functional properties
  - “Mechanical” Errors

- **Program Understanding**
  - Type inference
  - Pre/post computation
  - Invariants
  - Memory requirements
  - Reverse engineering
    - Call graph of a OO program
    - Behavioral Models
    - Architectural view
    - ...

- **Improve code quality/readability**
  - It might check if a program satisfies established “good practices” patterns
# Static Analyzer Glossary

## Concepts that appear recurrently:

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Inference vs. Checking</th>
<th>Sensibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approximation</td>
<td>Verification vs. Bug Finding</td>
<td>Flow-sensitive</td>
</tr>
<tr>
<td>Correction vs Completeness</td>
<td></td>
<td>Path-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Context-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intra vs. Interprocedural</td>
</tr>
</tbody>
</table>

- May vs Must
- False positives
- False negatives
Abstraction

- Focus on property to analyze

- Which aspects of the problem are interesting?
  - Control aspects
    - e.g: event sequences, instruction sequences, concurrency, etc.
  - Data
    - Division by zero
    - Memory consumption
  - Security

- Functionality

- To what degree of precision?
Abstraction: example

- I want to know about potential null dereferences
  - $\sigma$: Var $\rightarrow \{\text{null, notNull, maybeNull}\}$

- Division by Zero
  - $\sigma$: : Var $\rightarrow \{\text{indef, Z, NZ, QZ}\}$

- Can I free memory used by the iterator on method exit?
  - $\sigma$: : points-to graph
  - Check reachability

```java
float avg_list(List L) {
    int c = 0, s = 0;
    Iterator itL = L.iterator();
    for (; itL.hasNext();)
        { 
            val = (Value) itL.next();
            s += val.value();
            c++;
        }
    return s/c;
}
```
Abstraction requires approximation

- Abstraction => handle an incomplete picture
  - We do not handle “concrete”/“real” information

- Examples: Positive integers
  - $3 - 3 = 0$
  - $\text{Abs}(3) = \text{NZ}$
  - What is NZ – NZ?
    - Answer $Z \circ \text{NZ} \Rightarrow \text{MZ}$
MAY vs MUST Analysis

- We need to approximate and be conservative
  - We require to approximate from “above” (MAY) or from “below” (MUST)

Points-to analysis (e.g.: call graph)

```java
class B extends A {}
...
Object a;
[a -> null]
if(y<10) {
    a = new A();
    [a -> A]
} else {
    a = new B()
    [a -> B]
}
[a -> A ó B]
a.foo()
```

Available expressions (e.g.: remove redundant computation)

```java
...
int c = b*b;
{b*b }
int d = c + 1;
{b*b, c+1}
int a = b*b;
{b*b, c+1}
c = 4;
{ b*b }
if(b < c) b = 2;
else a = b*4;
{}
return d;
```
Soundness vs Completeness

- False positives vs. False negatives.

- Bugs detected by a "correct" analysis
- Bugs detected by a "complete" analysis (eg: testing)
- Real Bugs in the system
Given an analysis A that checks certain property Q on a program Pr:

- **Soundness**
  - If program Pr satisfies Q, then analysis A can prove/find it
  - If analysis reports **no null dereference** => program **has no** null dereferences

- **Completeness**
  - If analysis A reports property Q on program Pr, then property Q exists in program Pr.
  - If analysis reports **null dereference** => program **has** null dereference

- **Ideal** = Soundness + Completeness
Inference vs Checking

- **Type-Checking:**
  - Given a program and a set of annotations (property) check if the annotations are correct

- **Inference:**
  - Given a program infer annotations that exhibit a property

- Checking is easier than infering!

```java
int max(int[] a) {
  1: int i=0;
  2: int m = a[0];
  3: while (i < a.length) {
  4:    if(a[i]> m)
  5:       m = a[i];
  6:    i++;
  7:  }
  8: return m;
}
```

```plaintext
requires |a|>0;
ensures \forall j: [0, |a|)| res>=a[j];
```

```plaintext
1: assert i == 0;
2: assert m == a[0];
3: assert 0<=i<|a| && \forall j: [0, i)| m>=a[j];
4: assert 0<=i<|a| && \forall j: [0, i)| m>=a[j] && a[i]>m;
5: assert 0<=i<|a| && \forall j: [0, i)| m>=a[j] && m==a[i];
6: assert 0<i<=|a| && \forall j: [0, i)| m>=a[j];
7: assert i==|a| && \forall j: [0, i)| m>=a[j];
8: assert i==|a| && \forall j: [0, i)| res>=a[j];
```
Inference
Verification vs Bug Finding

- Verification: prove that a program obeys a specification
  - Specification could be implicit
  - Analysis is usually conservative

- Bug finding
  - Focus in *some* classes of errors with “real” impact.
    - Misusage of equals, hash
    - Null pointers
    - Synchronization
  - Apply several techniques on these errors
    - Dataflow, syntactic, statistical, etc.
  - Practical: it may miss bugs
Sensibility: which code aspects will be considered?

- Statement order? Flow sensitive
- Call Stack? Context sensitive
- Conditional branches? Path sensitive

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Sensibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-checking</td>
<td>Insensible</td>
</tr>
<tr>
<td>Dataflow</td>
<td>Flow-sensitive</td>
</tr>
<tr>
<td>Model Checking</td>
<td>Flow-sensitive and Path-sensitive</td>
</tr>
<tr>
<td>Points-to</td>
<td>C: Flow-sensitive</td>
</tr>
<tr>
<td></td>
<td>Java: Context-sensitive</td>
</tr>
</tbody>
</table>
Static Analysis limitations

- Why we need to approximate the static analysis?
  - Because of performance issues?
  - Because of shortage of resources?

- Because of indecibility of the analysis!
  - In general there is no program capable of computing precisely the properties of arbitrary.
  - This leads us to the Halting problem
    - “Has program P property X?” ≡ “Program P can reach a state where X holds?”
About this course

- **Objectives**
  - Present techniques for analyzing programs
    - Static and dynamic techniques
  - Learn their fundamentals:
    - In order to evaluate them, compare them
    - Pros & Cons.
    - Challenges
  - Promote using automatic tools for testing and program analysis
Outline

Verification and Analysis

- Design by contract
- Verification using automatic theorem provers
- Static verification, dynamic verification, bounded verification
- Typestates
- nullness, inmutability analysis
- Intraprocedural / Interprocedural Dataflow
- Dynamic Symbolic Execution

Testing

- Test case generation
- Structural and functional testing
- Search based testing
- Regression testing
- Mutation testing
- Model based testing
- Web applications testing
- GUI testing

Tools, tools, tools, demos, demos, tools, demos!
Course Organization

- 3 lectures every 2 weeks
- 1 lab session every 2 weeks
- 3 student projects
- Final written exam
Inferring properties

- Dataflow analysis

- It is the most heavily used static analysis technique.

- **Purpose**: Infer **automatically** interesting properties of a given program

- **Principle**: Model the execution of a program as the solution of a set of equations. The equations describe the flow of values through the program instructions.
An example

\[ \begin{align*}
\sigma &= [] \\
[& x \rightarrow \text{NZ}] \\
[& x \rightarrow \text{NZ}, y \rightarrow \text{NZ}] \\
[& x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{MZ}] \\
[& x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{MZ}] \\
[& x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{MZ}] \\
[& x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{MZ}] \\
[& x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{NZ}] \\
[& x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{MZ}] \\
[& x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{MZ}] \\
\end{align*} \]
An example

\[ \sigma = [] \]

1: \( x := 8 \) 

2: \( y := x \) 

3: \( z := 0 \) 

\( y > -1 \) 

4: \( x := x/y \) 

5: \( y := y - 2 \) 

6: \( z := 5 \)
An Example: Using the results

\[
\sigma = [] \\
x := 8; \\
\sigma = [x \rightarrow \text{NZ}] \\
y := x; \\
\sigma = [x \rightarrow \text{NZ}, y \rightarrow \text{NZ}] \\
z := 0; \\
\sigma = [x \rightarrow \text{NZ}, y \rightarrow \text{NZ}, z \rightarrow \text{Z}] \\
\text{while } y > -1 \text{ do} \\
\quad \sigma = [x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{MZ}] \\
\quad x := x / y; \\
\quad \sigma = [x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{MZ}] \\
\quad y := y - 2; \\
\quad \sigma = [x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{MZ}] \\
\quad z := 5; \\
\quad \sigma = [x \rightarrow \text{NZ}, y \rightarrow \text{MZ}, z \rightarrow \text{NZ}] \\
\]

Warning: This program \textit{may} try to divide a number by zero
Contract Verification

- Basic Idea: Translate the program (and also the contract) into a logical formula.
  - Prove that the contract holds in the formula
    - We may use an automatic theorem prover (SMT or SAT solvers)

Example: Using Dijsktra Weakest Precondition

```c
bool P(bool a, bool b) {
    if (a)
        c:=true
    else
        c:=b
}
```

```
returns c
requires true
ensures c = a || b
```

WP(P, c=a||b) = a => WP(c:=true,c=a||b) & & !a => WP(c:=b,c=a||b) = (a=> true=a||b) & & !a => b=a||b)

• Conjecture to prove: true=>(a=> true=a||b) & & !a => b=a||b)
class C {
    D f;
    C() {
        if (bluemoon()) {
            f = new D();
        }
    }
    C(D x) {
        f = x;
    }
    void m() {
        f.q();
    }
}

- Was $f$ really initialized?
- It depends on the condition
- May field $f$ be null?
Modelling using type-states

class C {
    D! f;

    C() {
        if (bluemoon()) {
            f = new D();
        }
    }

    C(D x) {
        f = x;
    }

    void m() {
        f.q();
    }
}

Field f is non null!

Errors are now explicit

1. There are executions which do not initialize f
2. x can not be assigned to f because they do not share the same type!

Now, if a program typechecks, it means there are no null dereferences
Automatic test case generation (using DSE)

Code to generate inputs for:

```csharp
void CoverMe(int[] a)
{
    if (a == null) return;
    if (a.Length > 0)
        if (a[0] == 1234567890)
            throw new Exception("bug");
}
```

<table>
<thead>
<tr>
<th>Constraints to solve</th>
<th>Data</th>
<th>Observed constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>a==null</td>
<td>null</td>
<td>a==null</td>
</tr>
<tr>
<td>a!=null &amp;&amp; !a.Length&gt;0</td>
<td>{}</td>
<td>a!=null &amp;&amp; !(a.Length&gt;0)</td>
</tr>
<tr>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]!=1234567890</td>
<td>{123..}</td>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]!=1234567890</td>
</tr>
</tbody>
</table>

Choose next path

Solve

Execute&Monitor

Negated condition

Done: There is no path left.
## Bug finders

Apache Ant 1.6.2
org.apache.tools.ant.taskdefs.optional.metamata.MAudit

```java
if (out == null) {
    try { out.close(); }
    catch (IOException e) { }
}
```

Eclipse 3.0.1,
org.eclipse.jdt.internal.debug.ui.JDIModelPresentation

```java
if (sig != null || sig.length() == 1) {
    return sig;
}
```

J2SE version 1.5 build 63 (released version),
java.lang.annotation.AnnotationTypeMismatchException

```java
public String foundType() {
    return this.foundType();
}
```

Less harmful....

```java
String dateString = getHeaderField(name);
dateString.trim();
```

- **Specialized tools in finding Bugs**
  - **Search for recurrent patterns**
    - Syntactically
    - Using dataflow analysis
    - Bug databases
Some tools

- **Contract verification**
  - Automatic:
    - Spec#, ESC-Java, HAVOC: SMT solvers
    - Code Contracts: Int Abs.
    - JForge, F-Soft, Siriam, TACO
  - Semi-automatic: Jahob, Krakatoa
  - Typestates: Plural, JSR, JavaRI

- **Bug finding**
  - FindBugs, Jlint, PMD, Astreé
  - Check’Crash, DSD

- **Inference**
  - Daikon, DySy
  - JConsume

- **ModelCheckers:**
  - JPF, Bandera,…

- **Abstract refinement**
  - Blast, SLAM

- **Test Case generation**
  - Pex, Randoop,…
  - Sage

- **Validation/ Understanding**
  - Contractor
  - CodeCity: Metrics
  - Eclipse: Navigation
- **Principles of Program Analysis**. Flemming Nielson, Hanne Riis Nielson, Chris Hankin.
- **Compilers: Principles, Techniques & Tools 2nd Edition**: Aho, Lam, Sethi, Ullman
- **Modern compiler implementation in Java**. Andrew Appel. 2nd Edition.

- **Software Testing and Analysis. Process, Principles and Techniques**. Mauro Pezzè and Michal Young.