Abstract. Traditionally, program analysis has been divided into two camps: Static techniques analyze code and safely determine what cannot happen; while dynamic techniques analyze executions to determine what actually has happened. While static analysis suffers from overapproximation, erring on whatever could happen, dynamic analysis suffers from underapproximation, ignoring what else could happen. In this talk, I suggest to systematically generate executions to enhance dynamic analysis, exploring and searching the space of software behavior. First results in fault localization and specification mining demonstrate the benefits of search.

Keywords: program analysis, test case generation, specifications
Fun in C

double fun(double x) {
    double n = x / 2;
    const double TOLERANCE = 1.0e-7;
    do {
        n = (n + x / n) / 2;
    } while (ABS(n * n - x) > TOLERANCE);
    return n;
}

Fun Demo

Square Roots in C

double csqrt(double x, double eps) {
    double n = x / 2;
    do {
        n = (n + x / n) / 2;
    } while (ABS(n * n - x) > eps);
    return n;
}

how do we validate this?

This is an easy exercise for Andrey Rybalchenko's terminator work, for instance.

Here's a little fun function. What does it do?

Here's a few examples. Can you guess now?

Here it is again, named. It is actually called the **Byzantine method** for computing square roots.
Square Roots in Eiffel

\[ \text{sqrt (x: REAL, eps: REAL): REAL is} \]

\[ \text{-- Square root of x with precision eps} \]

\[ \text{require} \]
\[ x \geq 0 \land \text{eps > 0} \]

\[ \text{-- precondition} \]

\[ \text{external} \]
\[ \text{csqrt (x: REAL, eps: REAL): REAL} \]

\[ \text{do} \]
\[ \text{Result := csqrt (x, eps)} \]

\[ \text{ensure} \]
\[ \text{abs (Result}^2 - x) \leq \text{eps} \]

\[ \text{-- postcondition} \]

end \[ \text{-- sqrt} \]

Here’s an Eiffel implementation, coming with pre- and postconditions we can actually use for validation.

Static C Analysis

This is hard – but we can still map all languages to one and, for instance, analyze C programs.

Real Square Roots

\[ \text{double asqrt(double x, double eps) \{} \]
\[ \quad \_\text{asm} \{ \]
\[ \quad \_\text{fld} x \]
\[ \quad \_\text{fsqrt} \]
\[ \quad \} \]
\[ \} \]
Static Binary Analysis

Eiffel in C
in ASM → C
in ASM

E.g. with the S2E platform
(by Candea and others)

Roots in the Cloud

double rsqrt(double x, double eps) {
  char url[1024];
  char *query =
    "http://www.compute.org/?sqrt(%f,%f)"
  sprintf(url, query, x, eps);
  return atof(query_url(url));
}

how do we validate this?

S2E does this nice job of going from
concrete to symbolic and back again
This is where static analysis finally comes to an end.

But does this actually happen in real life? I mean, who has multiple languages, obscure code, remote calls?

Well, everyone has. You start a browser, you have it all. None of this is what program analysis can handle these days. We’re talking scripts, we’re talking distributed, we’re talking amateurs, we’re talking security.
When you’re doing static analysis these days, you’re in some kind of dreamland. Everything is beautiful, everything is well-defined, and everything is under your control. (This is also called the academic bubble).

In real life, though, you’re stuck – and we do not have an answer to these new challenges.

---

**Dynamic Analysis**

- Originates from *execution monitoring*
- Considers (only) *actual executions*
- Covers all abstraction layers
- Tied to *run-time verification* techniques
<table>
<thead>
<tr>
<th>Static Analysis</th>
<th>Dynamic Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires perfect knowledge</td>
<td>Limited to observed runs</td>
</tr>
<tr>
<td>Originates from compiler optimization</td>
<td>Originates from execution monitoring</td>
</tr>
<tr>
<td>Considers all possible executions</td>
<td>Considers (only) actual executions</td>
</tr>
<tr>
<td>Can prove universal properties</td>
<td>Covers all abstraction layers</td>
</tr>
<tr>
<td>Tied to symbolic verification techniques</td>
<td>Tied to run-time verification techniques</td>
</tr>
</tbody>
</table>

So, is there some sort of middle ground – something that combines the coverage of static analysis with the applicability of dynamic analysis? 

Need more runs
Test Case Generation

- generates as many executions as needed
- random / search-based / constraint-based
- typically *directed* towards specific goals
- achieves high coverage on real programs
Experimental Program Analysis

- *generate* executions as needed
- *analyze* resulting executions and results
- analysis results *drive* test case generation
- *explore* as much behavior as possible

Enriching specifications

Dallmeier et al.: “Generating Test Cases for Specification Mining”, ISSTA 2010

Dallmeier et al.: “Generating Test Cases for Specification Mining”, ISSTA 2010
Generate test cases to systematically explore behavior
Assess executions to learn about software behavior

Do enriched specs contain more information?

Enriched specs have more regular and exceptional transitions

init vs enrich consistent for 3 other subjects
Enrich more trans. ALSO BETTER FOR VERIF?
Evaluation

How effective are enriched specifications?

- Enriched specs can be almost as effective as manually crafted specs

Enriching specifications

- Execute and extract initial spec
- Generate test mutants and enrich specs

Dallmeier et al.: "Generating Test Cases for Specification Mining", ISSTA 2010

two types: report at correct call, at least report a violation
for comp, manually created model again consistent with other 3 test cases
A new kind of Analysis

- Static analysis
- Dynamic analysis
- Experimental analysis

- 0 runs
- n given runs
- n generated runs

Generate test cases to systematically explore behavior
Assess executions to learn about software behavior

Are these real executions?

Here’s a simple addressbook.
Here's a test case generated by Randoop. It’s >200 lines long...

... and in the end, it fails. What do you do now?

A simplified version of the above. If you use two address book objects and make one's category depend on one the other, it'll crash.
Catch: There’s only one addressbook! So the Randoop test makes little sense, because it violates an implicit precondition.

The catch is: There’s never more than one addressbook! So the Randoop test makes little sense, because it violates an implicit precondition. When testing the Addressbook classes, Randoop detects *112 failures. However, all of them are false, pointing to an error in the generated test case rather than the application itself, which has *0 problems.

We examined a suite of five applications; overall, Randoop reported 181 failures, but all of them were false. For a little test suite of applications, we find real bugs:

Addressbook crashes when editing empty list
Calculator crashes when computing 500*10+5 with “,” as separator
Spreadsheet crashes when pasting empty clipboard
Search-based System Testing

- Generate tests at the user interface level
- Aim for code coverage and GUI coverage
- Synthesize artificial input events
- Any test generated is a valid input

Joint work with Florian Gross and Gordon Fraser

What I'm going to demo you now is our prototype called EXSYST, for Explorative SYstem Testing. EXSYST takes a Java program with a graphical user interface, such as our Addressbook example. It then generates user inputs such as mouse clicks or keystrokes and feeds them into the program. What you see here is EXSYST clicking and typing into the address book program; at the top, you see the statement coverage achieved so far. (Normally, all of this takes place in the background, so you don't see it, and it is also much much faster).

At first, these inputs are completely random, as you can see in these initial steps. But then, the search-based

The results are clear. Although it’s going through the GUI, EXSYST achieves a far higher coverage than Randoop.

Here are the results for *

Addressbook and
generating system tests:
higher coverage,
no false alarms,
realistic specs

generating unit tests:
lower coverage,
false alarms,
fuzzy specs

pea
Generate test cases to systematically explore behavior

Assess executions to learn about software behavior

real executions

real specifications

real executions

real specifications

Do we get real specifications?

Carving Invariants

We map the pre- and postconditions, as implemented in the system interface, down to the code — and thus down to the extracted specs.

* The more we can cover behavior, the more we learn about the system

* In presence of obscure code, search-based techniques are first choice
**Calculator Panel**
Object Invariants

- **EvoSuite** + Daikon
- **EXSYST** + Daikon

The image contains a calculator interface with the current state showing `1245 C`. The following invariants are highlighted:

1. `this.calculator.operator == null`
2. `(no such invariant: explores multiple operators)`
3. `this.calculator.state.getClass() != this.calculator.operator.getClass()`

**Calculator Panel**
Object Invariants

The image contains a calculator interface with the current state showing `1245 C`. The following invariants are highlighted:

1. `this.calculator.operator == null`
2. `(no such invariant: explores multiple operators)`

**Calculator Operand**
EnteringFirstOperandState(Calculator, char c)

- **EvoSuite** + Daikon
- **EXSYST** + Daikon

The image contains a calculator interface with the current state showing `1245 C`. The following invariants are highlighted:

1. `(no such invariant: c takes random values)`
2. `c in {"0"..."9"}`
Generate test cases to systematically explore behavior.

Do we get proven specifications?

Automated program proving requires loop and recursion invariants.

Proving a Multiplier

\[ \text{(requires } 0 \leq x < 65535) \]
\[ \text{(requires } 0 \leq y < 65535) \]
\[ \text{(ensures } \text{result} = x \times y) \]

```
mult = i = 0;
while (i < y) {
    mult += x; i++;
}
return mult;
```

Mining Loop Invariants

Juan Pablo Galeotti, Andreas Zeller

```
mult = i = 0;
while (i < y) {
    mult += x; i++;
}
return mult;
```

\[ #1 \ x \ \text{one of } (1, 1316) \]
\[ #2 \ y \ \text{one of } (1, 131) \]
\[ #3 \ i \geq 0 \]
\[ #9 \ i \leq y \]
\[ #10 \ i = (\text{mult} / x) \]
\[ #11 \ \text{mult} = (x \times i) \]
Proven Specifications

\[(\text{requires } 0 \leq x < 65535)\]
\[(\text{requires } 0 \leq y < 65535)\]
\[(\text{ensures } \text{result} == x \times y)\]

```
mult = i = 0;
while (i < y) {
    mult += x; i++;
}
return mult;
```

Challenges

Mine specifications that are

- complete
- real
- proven

But then, remember: all of this build on a finite number of executions. Will we ever be able to reach the completeness of static and symbolic techniques?

Picture © Myla Fox Productions
http://mylafox.deviantart.com/art/My-Little-Pony-Rainbow-Dash-199094976
The maybe best part of experimental analysis, however, is that it **smoothly** blends with all sorts of static analysis and verification. That's because we can use the mined specifications as **surrogates** for individual components, allowing for **local verification and analysis.**

At the end, we thus get the best of both worlds – we get dynamic analysis, static analysis, verification and validation all into one. We have a long way before us, but I think that this is a nice way to make verification scalable…
Static Analysis
requires perfect knowledge
- Originates from compiler optimization
- Considers all possible executions
- Can prove universal properties
- Tied to symbolic verification techniques

Dynamic Analysis
limited to observed runs
- Originates from execution monitoring
- Considers (only) actual executions
- Covers all abstraction layers
- Tied to run-time verification techniques

SAMBAMBA
- Compiler and runtime system for online adaptive parallelization
- Based on LLVM
- Target: Common C/C++ programs

...scalable to the challenges that await for us, every day, everywhere in the wide world of software.
```c
long performTask(int size1, int size2) {
    list *x = makeList(size1);
    list *y = makeList(size2);

    long hashX = hashList(x);
    long hashY = hashList(y);

    freeList(x);
    freeList(y);

    return hashX * hashY;
}

long hashList(list *x) {
    if (x == 0) return 0;
    return hashElem(x) + 31 * hashList(x->next);
}
```

Demo 1:
- gcc
- execute gcc version
- run sambamba (parallelized both functions)
- execute sambamba version
Challenges

- Finding relevant specifications
  Ranking wrt usage, bug-finding capabilities
- Expressing specifications
  Choosing a generic, domain-specific vocabulary
- Continuous specification
  Abstract feedback while you program

* The more we can cover behavior, the more we learn about the system
* In presence of obscure code, search-based techniques are first choice

And this is not only what we should do – this is something we must do. Thank you!