

Search-Based Program Analysis

Andreas Zeller
Saarland University

Search-Based Program Analysis

Andreas Zeller • Saarland University, Saarbrücken, Germany, zeller@cs.uni-saarland.de, <http://www.st.cs.uni-saarland.de/zeller/>

Program Analysis

- Verification and validation
- Understanding and debugging
- Optimization and transformation

Abstract. Traditionally, program analysis has been divided into two camps: Static techniques analyze code and safely determine what can- not happen; while dynamic techniques analyze executions to determine what actually has happened. While static analysis suffers from overap- proximation, 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 anal- ysis, exploring and searching the space of software behavior. First results in fault localization and specification mining demonstrate the benefits of search-based analysis.

Static Analysis

- Originates from *compiler optimization*
- Considers *all possible* executions
- Can prove *universal properties*
- Tied to *symbolic verification* techniques

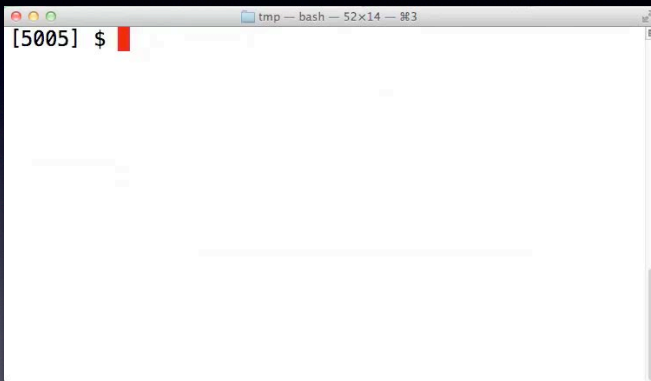
Keywords: program analysis, test case generation, specifications

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;  
}
```

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

Fun Demo



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

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;  
}
```

Here it is again, named. It is actually called the Byzantine method for computing square roots.

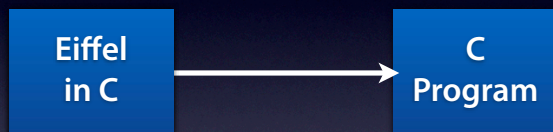
how do we validate this?

Square Roots in Eiffel

```
sqrt (x: REAL, eps: REAL): REAL is
  -- Square root of x with precision eps
  require
    x >= 0 ^ eps > 0          - precondition
  external
    csqrt (x: REAL, eps: REAL): REAL
  do
    Result := csqrt (x, eps)
  ensure
    abs (Result ^ 2 - x) <= eps - postcondition
  end -- 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

```
double asqrt(double x, double eps) {
  __asm {
    fld x
    fsqrt
  }
}
```

Static Binary Analysis



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?



Static Analysis



This is where static analysis finally comes to an end.

Multiple Languages



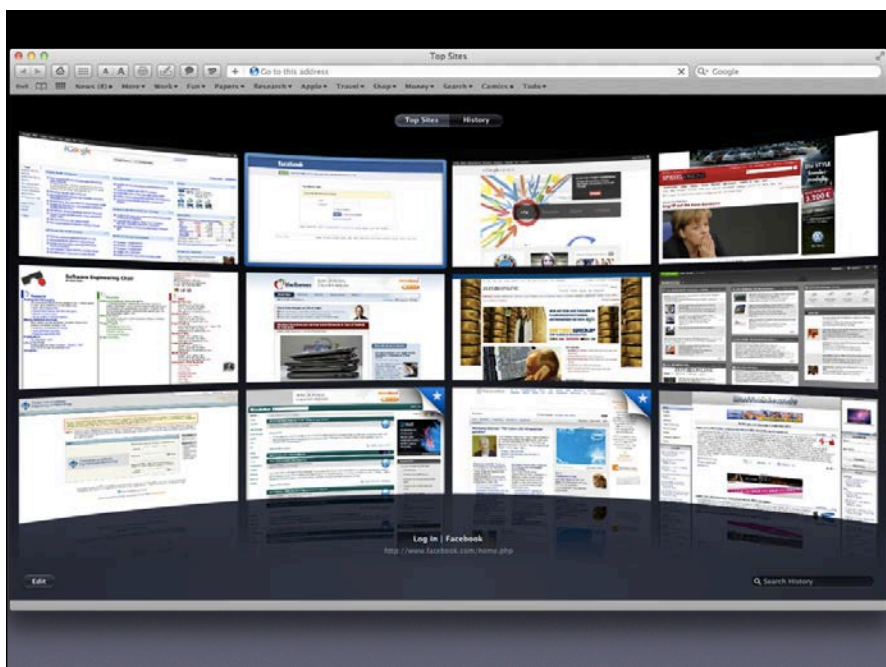
Obscure Code

```
double csqrt(double x, double eps) {  
    __asm {  
        fld x  
        fsqrt  
    }  
}
```

Remote Calls



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.



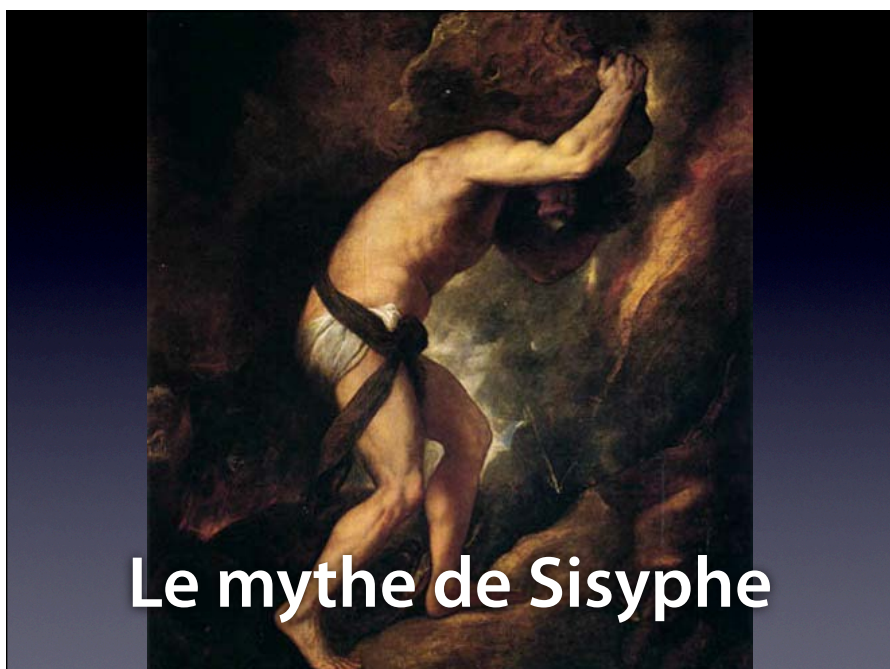
When you're doing static analysis these days, you're in some kind of dreamland. Everything is beautiful, everything is well-defined, and everything is under your control. (This is also called the academic bubble).

Picture © Myla Fox Productions
<http://mylafox.deviantart.com/art/My-Little-Pony-Rainbow-Dash-199094976>



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

Picture © Myla Fox Productions
<http://mylafox.deviantart.com/art/My-Little-Pony-Rainbow-Dash-199094976>



Le mythe de Sisyphe

In Greek mythology Sisyphus (/ 'sɪsəfəs/; Greek: Σίσυφος Sisyphos) was a king punished by being compelled to roll an immense boulder up a hill, only to watch it roll back down, and to repeat this throughout eternity.

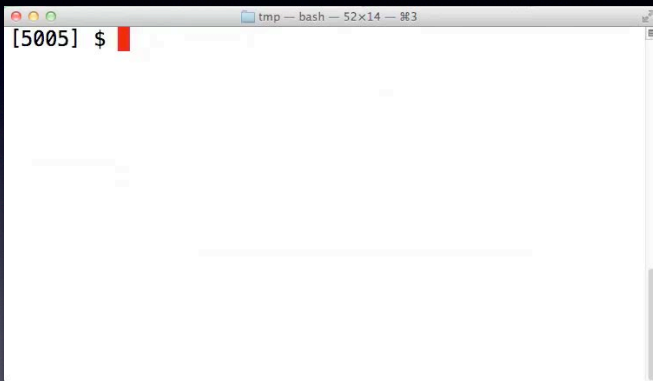
Titian(Tiziano Vecelli)

Sisyphos. During her stay in Augsburg 1547-1548, Queen Maria of Hungary, sister of Karl V., asked Titian to paint a series of "Condemned" or "Furies". Canvas, 237 x 216 cm Cat.426

Dynamic Analysis

- Originates from *execution monitoring*
- Considers (only) *actual* executions
- Covers all abstraction layers
- Tied to *run-time verification* techniques

Only need Execution



... and execution is normally the least we can do.

Multiple Languages



Obscure Code

```
double csqrt(double x, double eps) {  
  __asm {  
    fld x  
    fsqrt  
  }  
}
```

Remote Calls



Indeed, none of the limitations of static analysis is an issue for dynamic analysis.

Checking Properties

```
double asqrt(double x, double eps) {  
    __asm {  
        fld x  
        fsqrt  
    }  
    assert abs(x * x - eps) < 0.0001;  
}
```

We can dynamically infer postconditions, for instance – and check them at runtime.

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

So, is there some sort of **middle ground** – *something that combines*

- * *the coverage of static analysis with*
- * *the applicability of dynamic analysis?*

Search-Based Program Analysis

The answer is – you guessed it – what I call “experimental analysis” or, suitable to this conference, “search-based analysis”.

Dynamic Analysis

limited to observed runs

- Originates from *execution monitoring*
- Considers (only) *actual* executions
- Covers all abstraction layers
- Tied to *run-time verification* techniques

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



Generate test cases
to systematically
explore behavior

Assess executions
to learn about
software behavior

Search-based Program Analysis

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



Generate test cases
to systematically
explore behavior

Assess executions
to learn about
software behavior

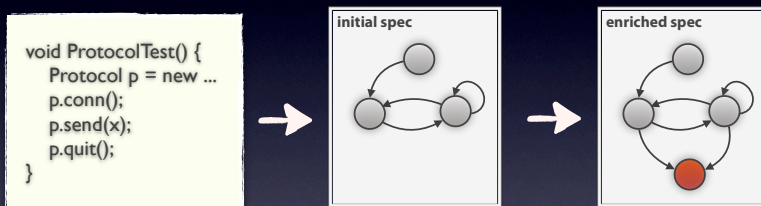


Challenges

We need to

1. Explore *complete* behavior
2. Restrict to *real usage*
3. Identify *relevant* behavior

Enriching specifications



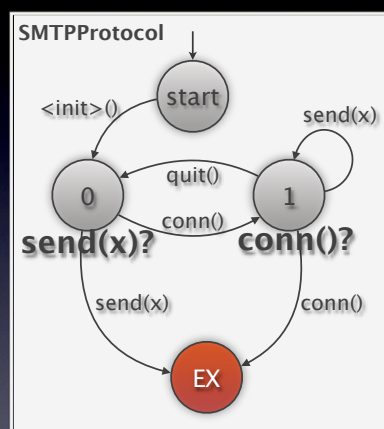
Execute and extract
initial spec

Generate test mutants
and enrich specs

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

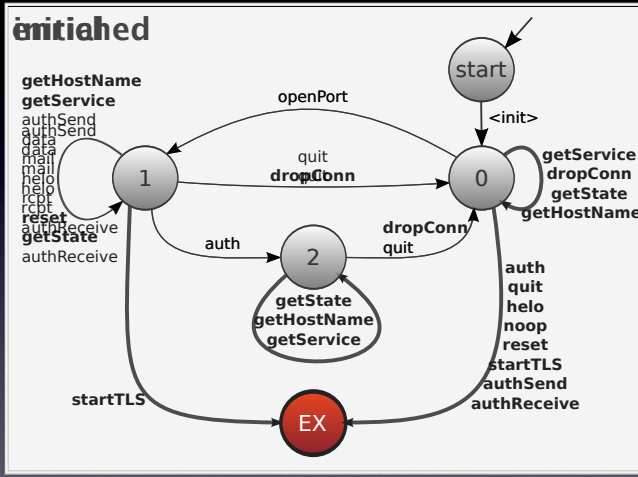
```
void ProtocolTest() {
  Protocol p = new ...
  p.conn();
  p.send(x);
  p.quit();
}
```

```
void TestMutant1() {
  Protocol p = new ...
  p.conn();
  p.send(x);
  p.quit();
  p.conn();
}
```

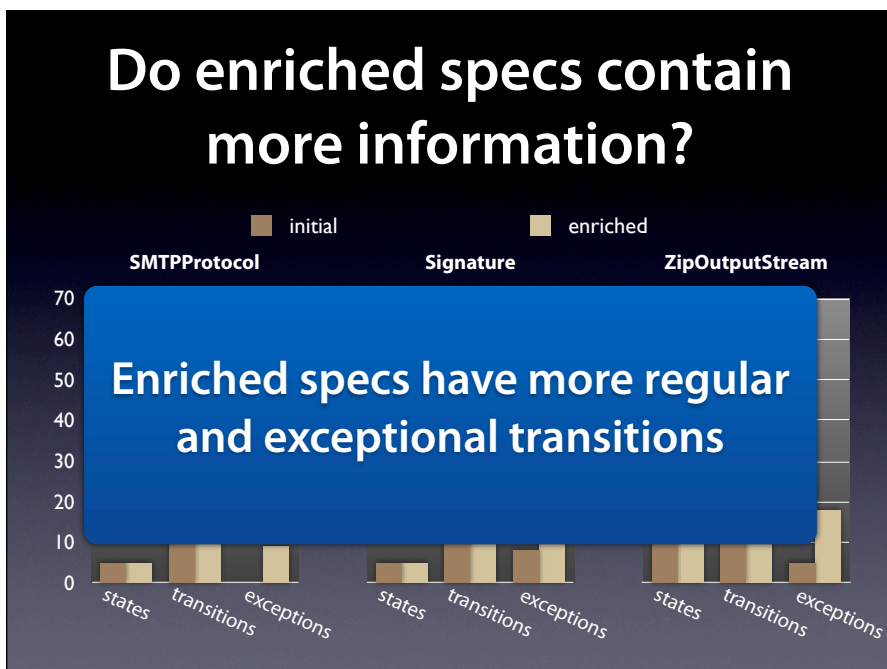
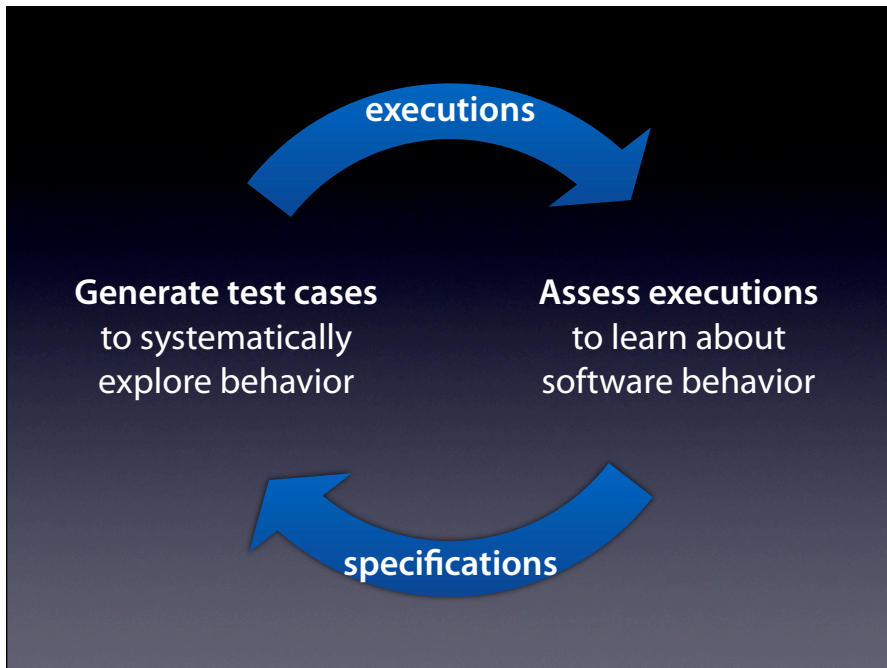


Dallmeier et al: "Generating Test Cases for Specification Mining", ISSTA 2010

“Enriched specs have more regular and exceptional transitions”;
 “Enriched specs can be almost as effective as manually crafted specs”

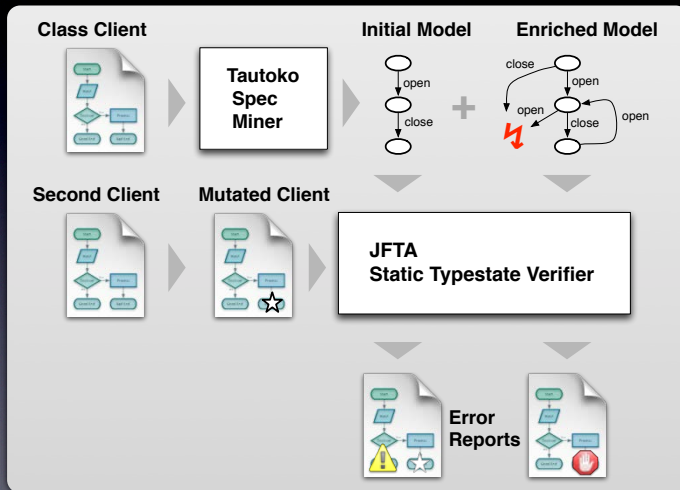


Dallmeier et al: “Generating Test Cases for Specification Mining”, ISSTA 2010

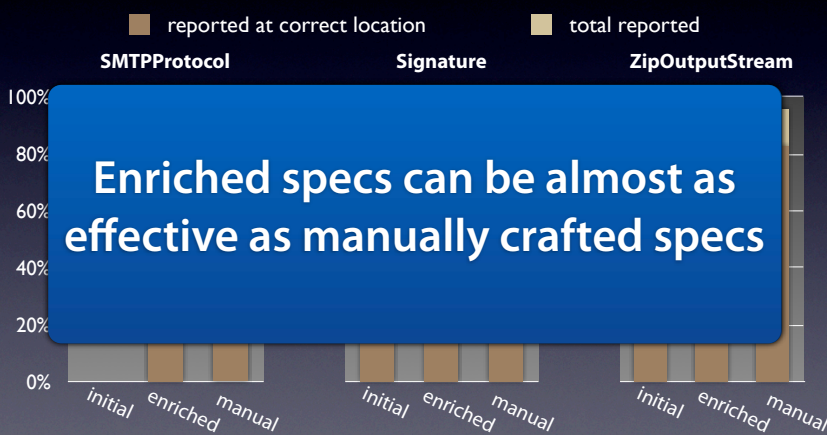


init vs enrich
 consistent for 3
 other subjects
 Enrich more
 trans. ALSO
**BETTER FOR
 VERIF?**

Evaluation

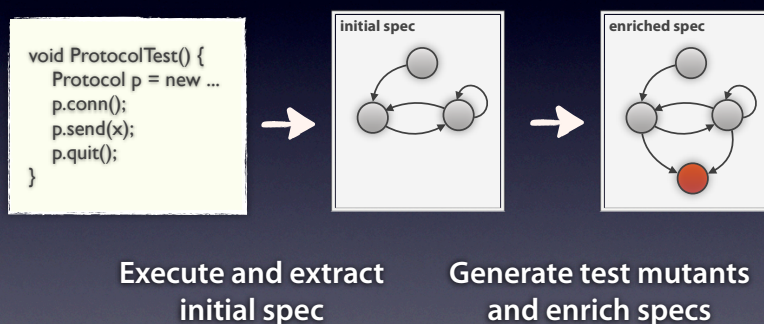


How effective are enriched specifications?



two types: report at correct call, at least report a violation for comp, manually created model

Enriching specifications



Challenges

We need to

1. Explore *complete* behavior
2. Restrict to *real usage*
3. Identify *relevant* behavior

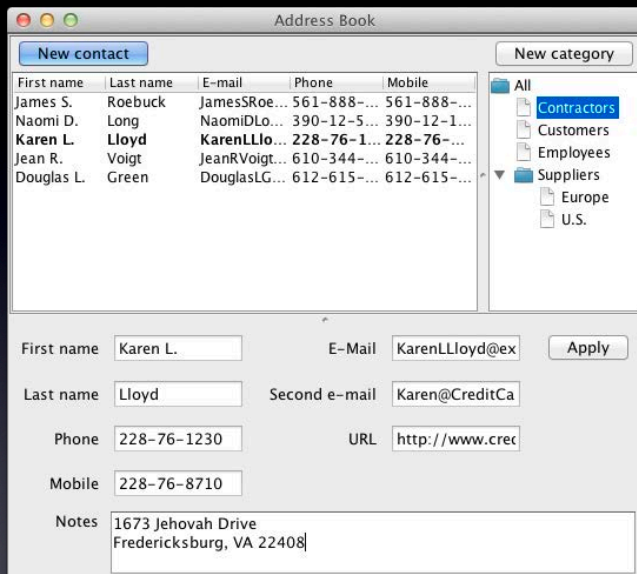
Challenges

We need to

1. Explore *complete* behavior
2. Restrict to *real usage*
3. Identify *relevant* behavior

For analysis, we not only want to be as complete as possible, but we'd also like to get rid of all the nonsensical behavior - that is, keep **real executions** only.

Here's a simple addressbook.



Random Testing

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


```
public class RandoopTest0 extends TestCase {
    ...

    public void test8() throws Throwable {
        if (debug) System.out.printf("%nRandoopTest0.test8");

        AddressBook var0 = new AddressBook();
        EventHandler var1 = var0.getEventHandler();
        Category var2 = var0.getRootCategory();
        Contact var3 = new Contact();
        AddressBook var4 = new AddressBook();
        EventHandler var5 = var4.getEventHandler();
        Category var6 = var4.getRootCategory();
        String var7 = var6.getName();
        var0.addCategory(var3, var6);
        SelectionHandler var9 = new SelectionHandler();
        AddressBook var10 = new AddressBook();
        EventHandler var11 = var10.getEventHandler();
        Category var12 = var10.getRootCategory();
    }
}
```

```
MainWindow var64 = new MainWindow(var6);
AddressBook var65 = new AddressBook();
EventHandler var66 = var65.getEventHandler();
Category var67 = var65.getRootCategory();
Contact var68 = new Contact();
Category[] var69 = var68.getCategories();
var65.removeContact(var68);
java.util.List var71 = var65.getContacts();
AddressBook var72 = new AddressBook();
EventHandler var73 = var72.getEventHandler();
Category var74 = var72.getRootCategory();
EventHandler var75 = var72.getEventHandler();
SelectionHandler var76 = new SelectionHandler();
actions.CreateContactAction var77 = new actions.CreateContactAction(var72, var76);
boolean var78 = var77.isEnabled();
AddressBook var79 = new AddressBook();
EventHandler var80 = var79.getEventHandler();
Category var81 = var79.getRootCategory();
String var82 = var81.getName();
var77.categorySelected(var81);
Category var85 = var65.createCategory(var81, "hi!");
String var86 = var85.toString();
Category var88 = var0.createCategory(var85, "exceptions.NameAlreadyInUseException");
}

```



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

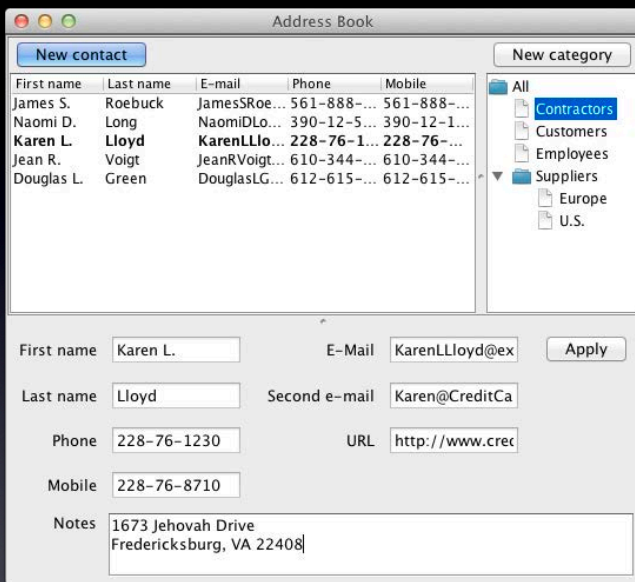
Simplified Test Case

```
public class RandoopTest0 extends TestCase {
    public void test8() throws Throwable {
        if (debug) System.out.printf("%nRandoopTest0.test8");

        AddressBook a1 = new AddressBook();
        AddressBook a2 = new AddressBook();
        Category a1c = a1.createCategory(a1.getRootCategory(), "a1c");
        Category a2c = a2.createCategory(a1c, "a2c");
    }
}
```

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

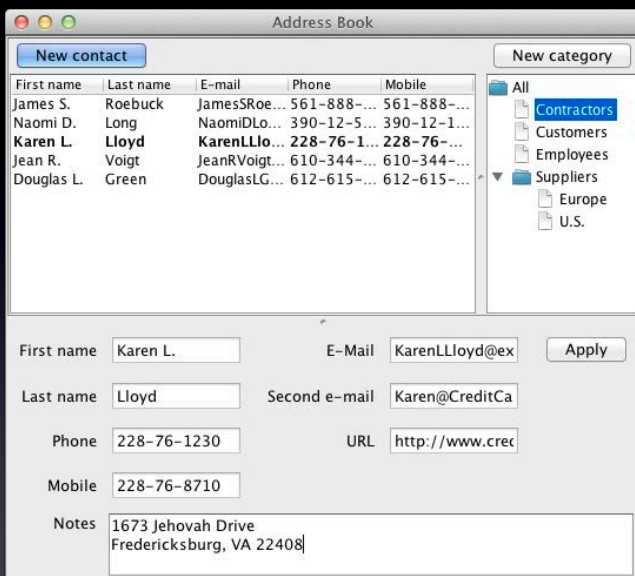


how many addressbooks?

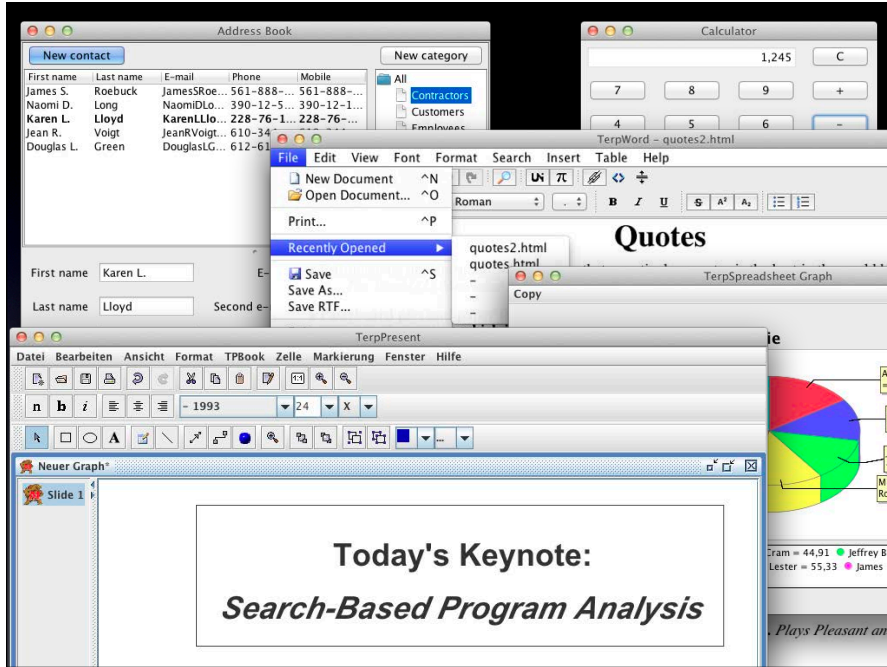
Search-based System Testing

- 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



So we generate one input after
another...



...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

Initial Results

- Found real bugs in examined applications
- Every bug found is triggered by real input
- Higher coverage than GUItar / Randoop
- No nonsensical tests

Joint work with Florian Gross and Gordon Fraser

In one example, our code coverage is lower – but that’s because Randoop covers dead code unreachable from the GUI.

Challenges

We need to

1. Explore *complete* behavior
2. Restrict to *real usage*
3. Identify *relevant* behavior

Challenges

We need to

1. Explore *complete* behavior
2. Restrict to *real usage*
3. Identify *relevant* behavior

Note, though, that the tests we have generated do not contain assertions – they are still only executions, but not actual tests. How do we find out what these executions should do? What is their relevant effect?

What is relevant?

- Features that *clients rely upon* can be determined from usage
- Features that *characterize correct behavior* in other words: specifications that *detect bugs*

To address these issues, let me take you a bit into our recent work in test case generation.

```
void concrete_test()
{
    YearMonthDay var0 = new YearMonthDay();
    TimeOfDay var1 = new TimeOfDay(var0);
    CopticChronology var2 = (CopticChronology)
        org.joda.time.Chronology.getCopticUTC();
    FixedDateTimeZone var3 =
        (FixedDateTimeZone) var2.getZone();
    DateTime var4 = var0.toDateTime(var1);
    DateTime var5 = var4.withZone(var3);
}
```

Welcome to the wonderful world of test case generation. Tremendous progress in the last years: symbolic, search-based, concolic... But the first thing you notice: No assertions.

μTest

```
class Foo {  
  int bar(int x) {  
    return 2 * x;  
  }  
}
```



```
class Foo {  
  int bar(int x) {  
    return 2 + x;  
  }  
}
```



```
void test() {  
  f = new Foo();  
  y = f.bar(10);  
  assert(y==20);  
}
```

- generates test cases *with oracles*
- retains assertions that *find most mutants*

Fraser, Zeller: "Mutation-driven Generation of Unit Tests and Oracles", ISSTA 2010

Instead of this...

```
void concrete_test()  
{  
  YearMonthDay var0 = new YearMonthDay();  
  TimeOfDay var1 = new TimeOfDay(var0);  
  CopticChronology var2 = (CopticChronology)  
    org.joda.time.Chronology.getCopticUTC();  
  FixedDateTimeZone var3 =  
    (FixedDateTimeZone) var2.getZone();  
  DateTime var4 = var0.toDateTime(var1);  
  DateTime var5 = var4.withZone(var3);  
}
```

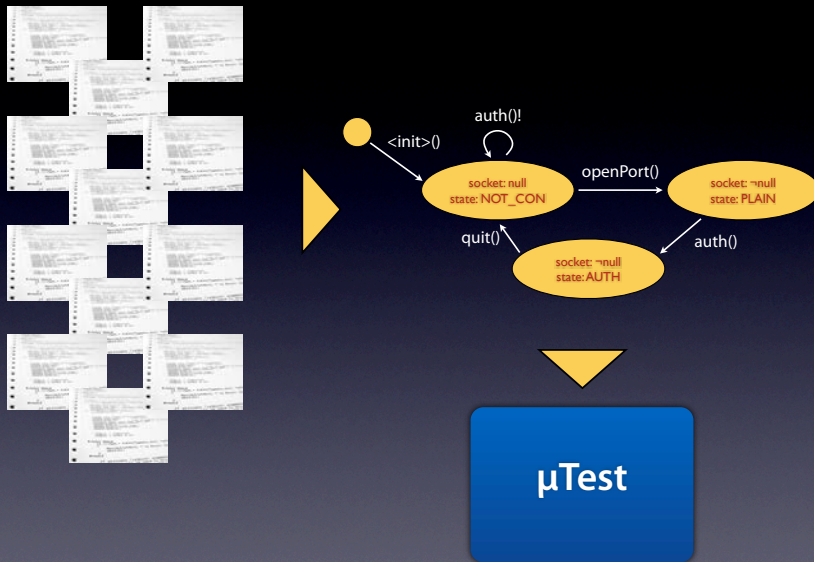
Fraser, Zeller: "Mutation-driven Generation of Unit Tests and Oracles", ISSTA 2010

```
void concrete_test()  
{  
  YearMonthDay var0 = new YearMonthDay();  
  TimeOfDay var1 = new TimeOfDay(var0);  
  CopticChronology var2 = (CopticChronology)  
    org.joda.time.Chronology.getCopticUTC();  
  FixedDateTimeZone var3 =  
    (FixedDateTimeZone) var2.getZone();  
  DateTime var4 = var0.toDateTime(var1);  
  DateTime var5 = var4.withZone(var3);  
  
  assertFalse (var4.equals(var5));  
  assertNotNull (var5); Catch the most mutants  
}
```

Fraser, Zeller: "Mutation-driven Generation of Unit Tests and Oracles", ISSTA 2010

we thus get this (with oracles). Much better, because it tells you what is expected - but still unreadable.

Next thing we need to do: We need to make these more readable. And for this, we mine existing usage examples.



Fraser, Zeller: "Mutation-driven Generation of Unit Tests and Oracles", ISSTA 2010

```
void concrete_test()
{
    YearMonthDay var0 = new YearMonthDay();
    TimeOfDay var1 = new TimeOfDay(var0);
    CopticChronology var2 = (CopticChronology)
        org.joda.time.Chronology.getCopticUTC();
    FixedDateTimeZone var3 =
        (FixedDateTimeZone) var2.getZone();
    DateTime var4 = var0.toDateTime(var1);
    DateTime var5 = var4.withZone(var3);

    assertFalse (var4.equals(var5));
    assertNotNull (var5);
}
```

Fraser, Zeller: "Mutation-driven Generation of Unit Tests and Oracles", ISSTA 2010

```
void concrete_test()
{
    YearMonthDay var0 = new YearMonthDay();
    TimeOfDay var1 = new TimeOfDay(var0);
    DateTimeZone var3 = DateTimeZone.UTC;
    DateTime var4 = var0.toDateTime(var1);
    DateTime var5 = var4.withZone(var3);

    assertFalse (var4.equals(var5));
    assertNotNull (var5);
}
```

Fraser, Zeller: "Mutation-driven Generation of Unit Tests and Oracles", ISSTA 2010

By mining usage information, the test case already makes more sense. But we can even do better.

We can split this test into a part that sets up the input, and the actual methods we want to test.

```
void concrete_test()
{
    - Input
    YearMonthDay var0 = new YearMonthDay();
    TimeOfDay var1 = new TimeOfDay(var0);
    DateTimeZone var3 = DateTimeZone.UTC;

    - Methods under test
    DateTime var4 = var0.toDateTime(var1);
    DateTime var5 = var4.withZone(var3);

    assertFalse (var4.equals(var5));
    assertNotNull (var5);
}
```

Fraser, Zeller: "Generating Parameterized Unit Tests", ISSTA 2011

We then turn the test into a **parameterized** test – a unit test that can be executed with arbitrary values. For a start, though, we assume that every parameter has still a concrete value

```
void parameterized_test(TimeOfDay input1,
    DateTimeZone input2, YearMonthDay input3)
{
    - Turn input into parameters
    assume (input3.equals(new YearMonthDay()));
    assume (input1.equals(new TimeOfDay(input3)));
    assume (input2.equals(DateTimeZone.UTC));

    DateTime var4 = var0.toDateTime(input1);
    DateTime var5 = var4.withZone(input2);

    assertFalse (var4.equals(var5));
    assertNotNull (var5);
}
```

Fraser, Zeller: "Generating Parameterized Unit Tests", ISSTA 2011

Let us focus on one of these preconditions. This is a property of input2. But again, we have the choice between several such properties.

```
void parameterized_test(TimeOfDay input1,
    DateTimeZone input2, YearMonthDay input3)
{
    assume (input3.equals(new YearMonthDay()));
    assume (input1.equals(new TimeOfDay(input3)));
    assume (input2.equals(DateTimeZone.UTC));

    DateTime var4 = var0.toDateTime(input1);
    DateTime var5 = var4.withZone(input2);

    assertFalse (var4.equals(var5));
    assertNotNull (var5);
}
```

Fraser, Zeller: "Generating Parameterized Unit Tests", ISSTA 2011

```
assume (input2 != null);
assume (input2.isFixed());
assume (input2.getID().equals("UTC"));
assume (input2.hashCode() == 0xb0b0feed);
assume (input2.equals(DateTimeZone.UTC));
```

```
DateTime var4 = var0.toDateTime(input1);
DateTime var5 = var4.withZone(input2);
```

```
assertFalse (var4.equals(var5));
assertNotNull (var5);
}
```

Fraser, Zeller: "Generating Parameterized Unit Tests", ISSTA 2011

But which preconditions shall we retain? We retain those preconditions where a **change affects the postcondition**. We systematically negate the preconditions, generate test cases,

```
void parameterized_test(TimeOfDay input1,
    DateTimeZone input2, YearMonthDay input3)
{
```

```
    assume (input2 != null);
    assume (input2.isFixed());
    assume (input3 != null);
    assume (input3.size() == 3);
```

```
    DateTime var4 = var0.toDateTime(input1);
    DateTime var5 = var4.withZone(input2);
```

```
    assertFalse (var4.equals(var5));
    assertNotNull (var5);
}
```

- This is a full specification!

Fraser, Zeller: "Generating Parameterized Unit Tests", ISSTA 2011

Consequently, we get a full set of preconditions, postconditions, execution – all search-based, all generated. This is what we'd like to show to the programmer, who'd check whether this is

Challenges

We need to

1. Explore *complete* behavior ✓
2. Restrict to *real usage* ✓
3. Identify *relevant* behavior ✓

We have the building blocks in place for doing better analysis!

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

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



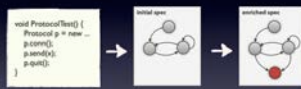
Generate test cases to systematically explore behavior

Assess executions to learn about software behavior



The more we can cover behavior, the more we learn about the system – and this gives us great opportunities to finally deal with obscure, complex systems.
All of this flows together. It's searching for behavior. It's a Yin and Yang thing.

Enriching specifications

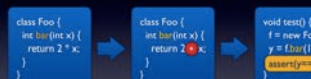


Execute and extract initial spec Generate test mutants and enrich specs

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

Explore *complete* behavior

μTest



- generates test cases *with oracles*
- retains assertions that *find most mutants*

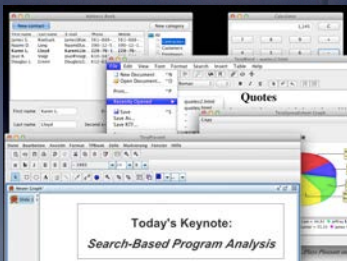
Frazer, Zeller: "Mutation-driven Generation of Unit Tests and Oracles", SSTa 2010

Identify *relevant* behavior

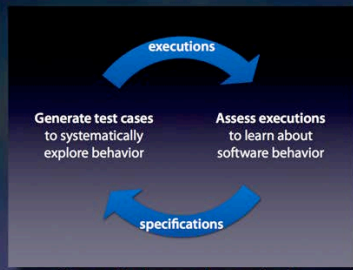
Generate test cases to systematically explore behavior



Search-based analysis

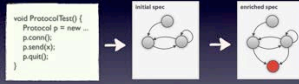


Restrict to *real usage*



Search-based analysis

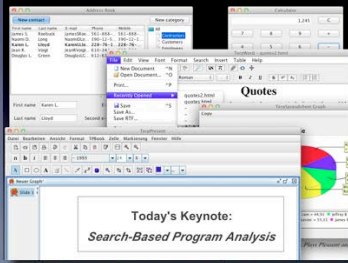
Enriching specifications



Execute and extract initial spec Generate test mutants and enrich specs

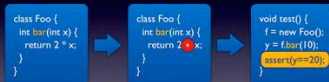
Dallmeier et al: "Generating Test Cases for Specification Mining", ISSITA 2010

Explore *complete* behavior



Restrict to *real usage*

μTest



- generates test cases *with oracles*
- retains assertions that *find most mutants*

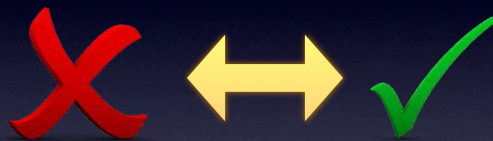
Fraser, Zeller: "Mutation-driven Generation of Unit Tests and Oracles", ISSITA 2010

Identify *relevant* behavior

Applications

- Exhaustively test and explore real systems
- Obtain specifications for functional testing
- Ease understanding and debugging

Debugging

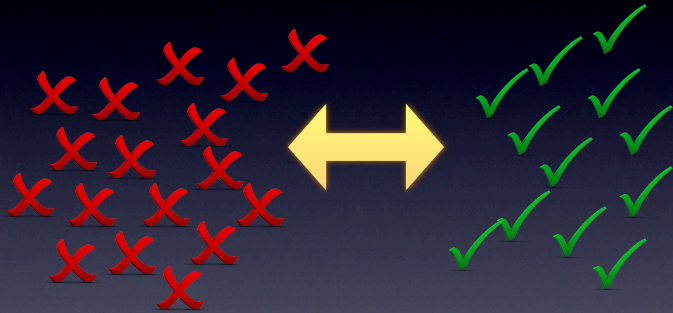


need more runs

You may be aware of statistical fault localization. Little may you know that all these techniques only work if you have tens of thousands of tests. Ben Liblit (who invented this) was very clear about this (and did his best to collect some); later folks weren't.

Directed Test Generation

Our idea: generate test cases to narrow down the diagnosis...



Diagnosis

... in terms of features that are relevant, real, and demonstrated by real test cases, just as shown before.



- Branches taken
- Values assigned
- Invariants violated
- and more...

Rößler: "Understanding Failures Through Facts", ESEC/FSE DS 2011

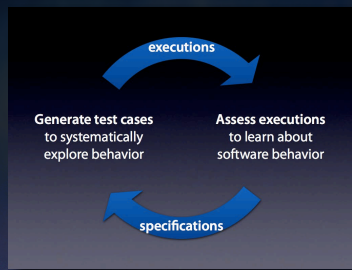
Diagnosis

This is our current achievement. Debugging (almost) solved.

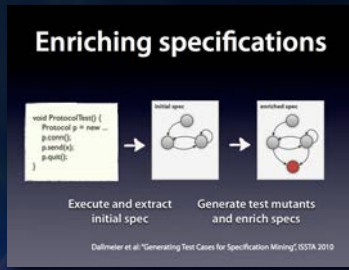


0.2% of source code

Burger, Zeller: "Minimizing Failure Reproduction", ISSTA 2011



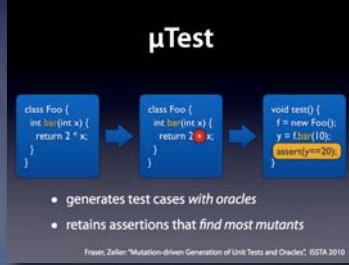
Search-based analysis



Explore *complete* behavior



Restrict to *real usage*



Identify *relevant* behavior

The more we can cover behavior, the more we learn about the system – and this gives us great opportunities to finally deal with obscure, complex systems. And this is not only what we **should** do – this is something we **must** do. Thank you!