Experimental Program Analysis

Andreas Zeller
Saarland University

Experimental 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-

Static Analysis

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

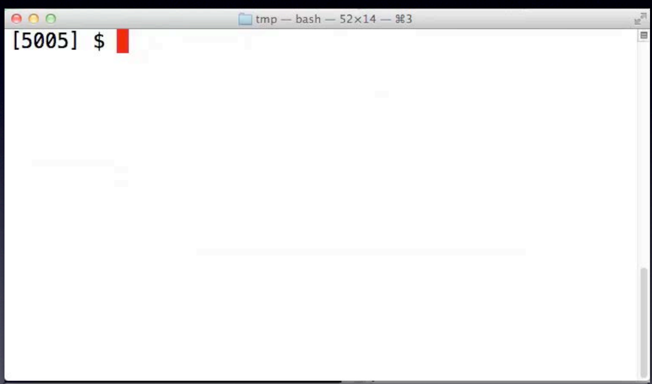
Keywords: program analysis, test case generation, specifications

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;
}
```

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

how do we validate this?

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

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                           – postcondition
   abs (Result ^ 2 – x) <= eps
end -- sqrt
```

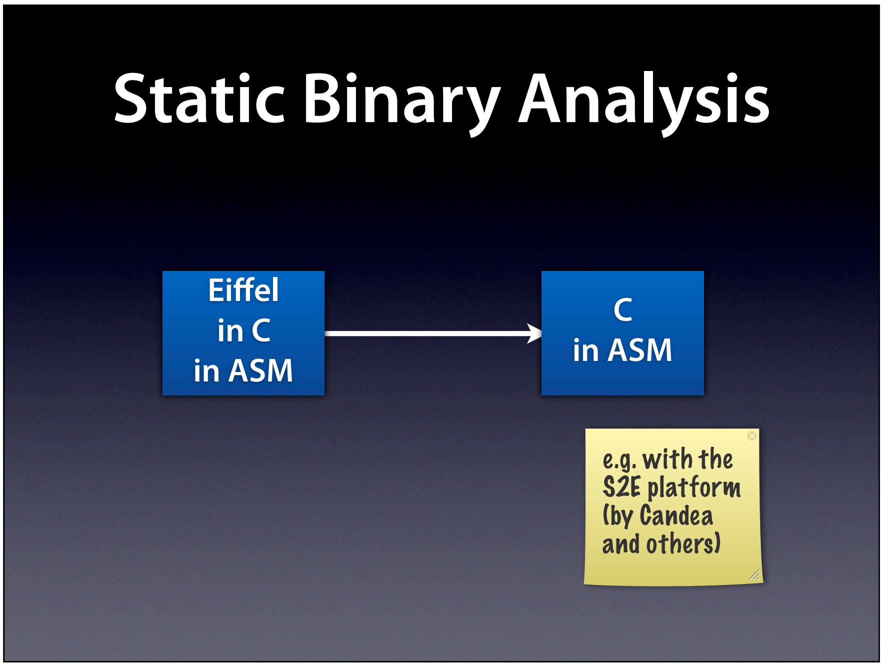
Here's an Eiffel implementation, coming with pre- and postconditions we can actually use for validation.

Static C Analysis

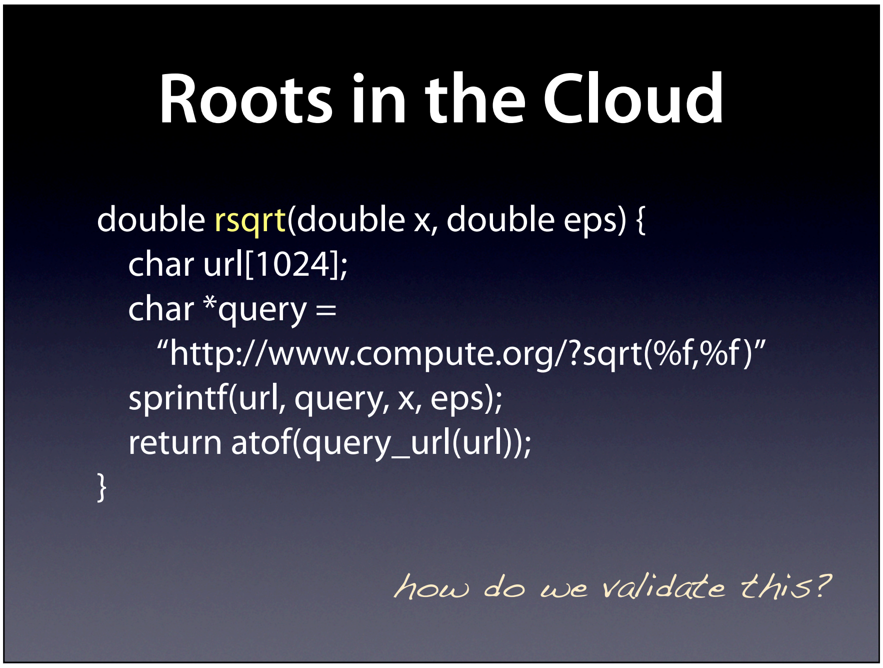
This is hard – but we can still map all languages to one and, for instance, analyze C programs.

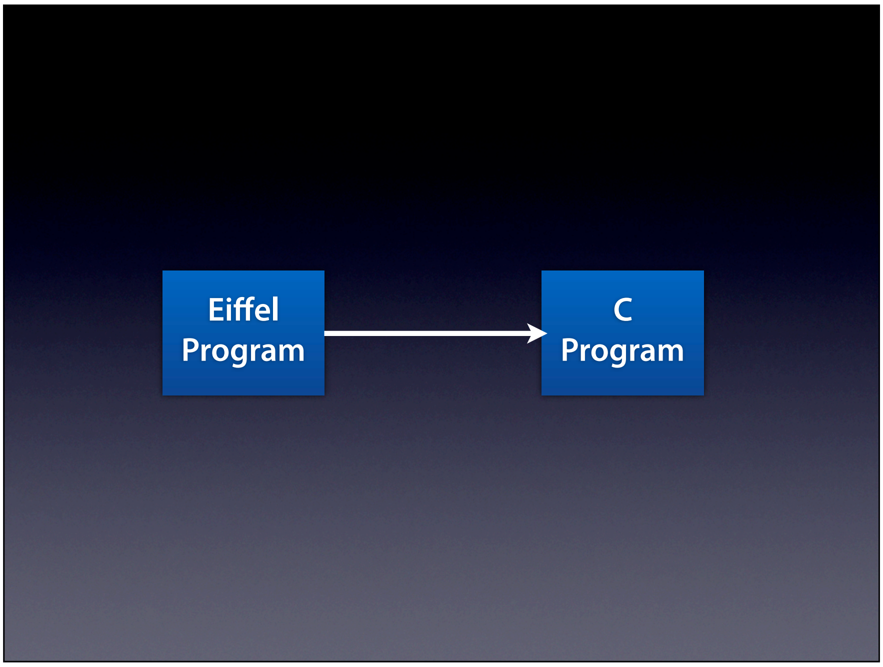
Real Square Roots

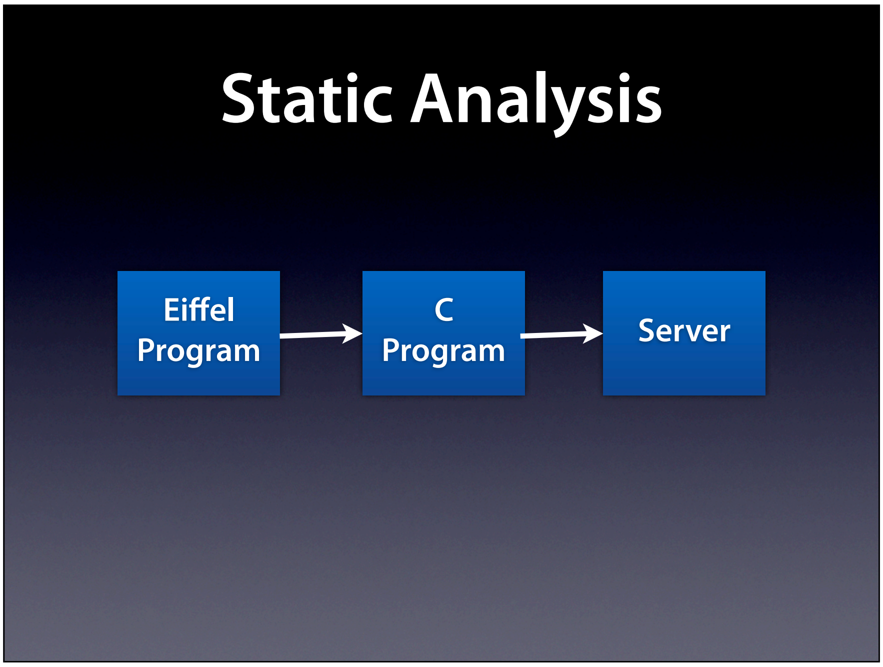
```
double asqrt(double x, double eps) {
    __asm {
      fld x
      fsqrt
    }
}
```



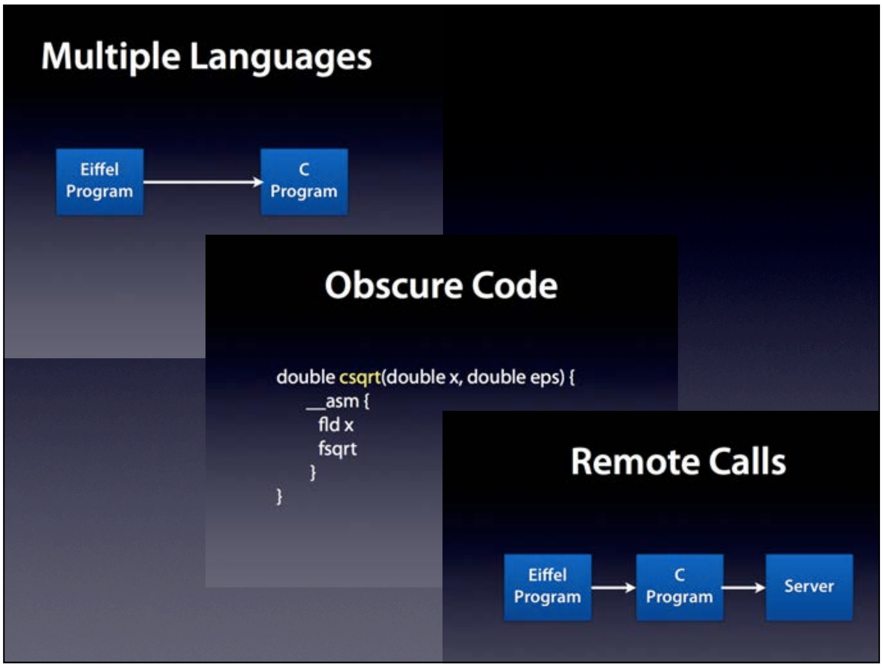
S2E does this nice job of going from concrete to symbolic and back again



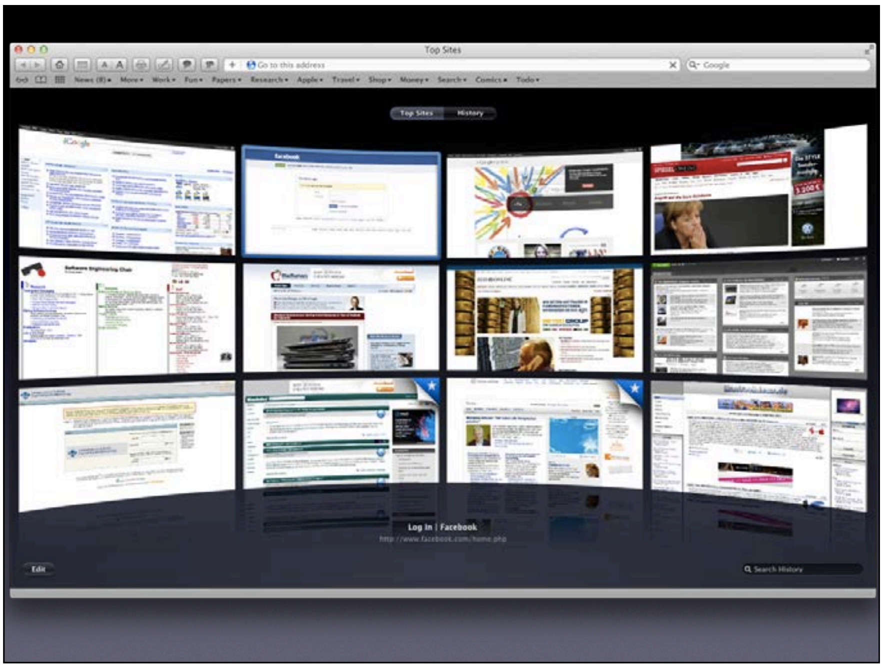




This is where static analysis finally comes to an end.



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).

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

Dynamic Analysis

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

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?

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

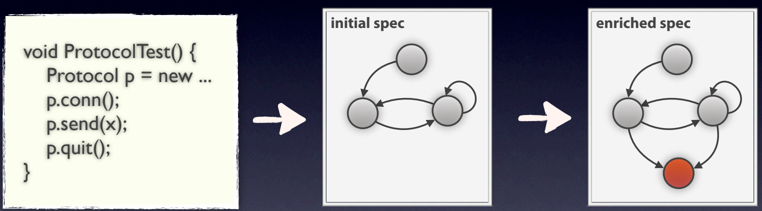
Generate test cases
to systematically
explore behavior

Assess executions
to learn about
software behavior

Experimental Program Analysis

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

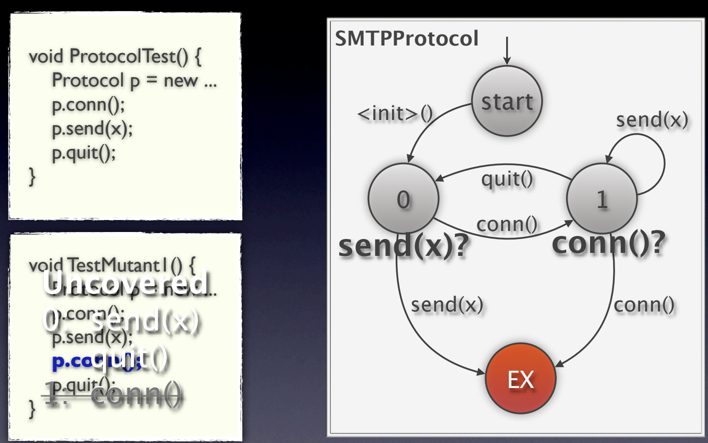
Enriching specifications



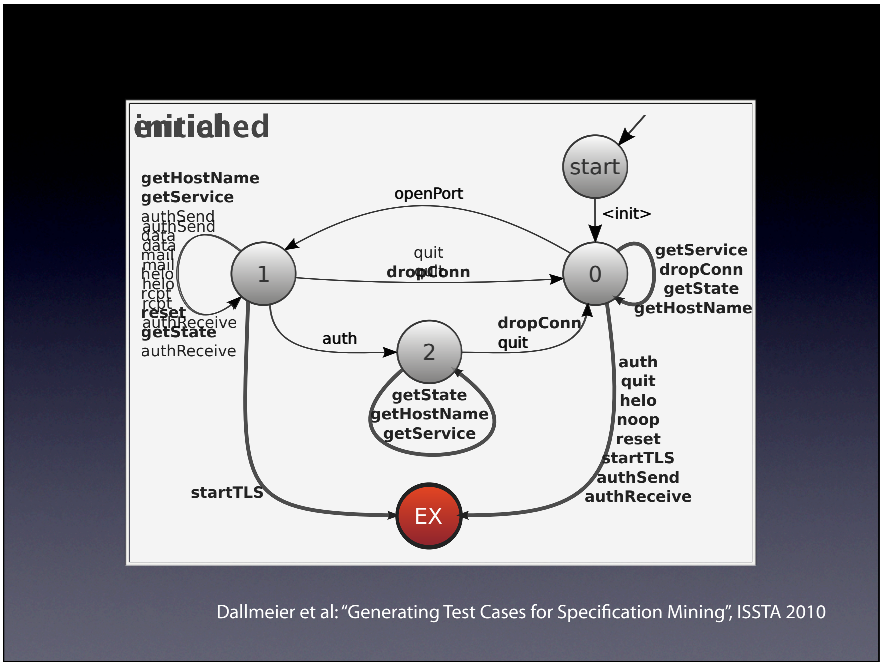
Execute and extract initial spec

Generate test mutants and enrich specs

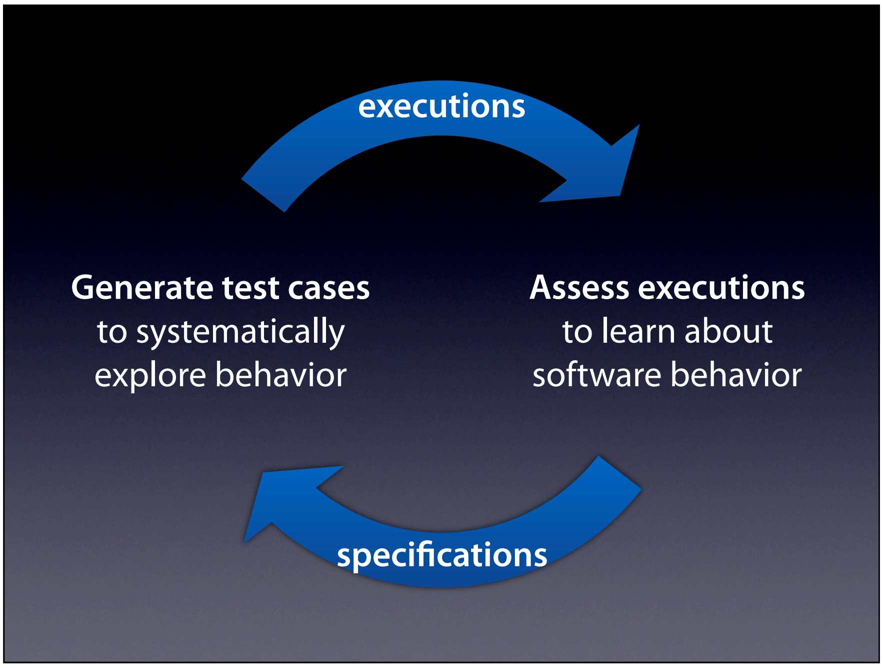
Dallmeier et al: "Generating Test Cases for Specification Mining", ISSTA 2010

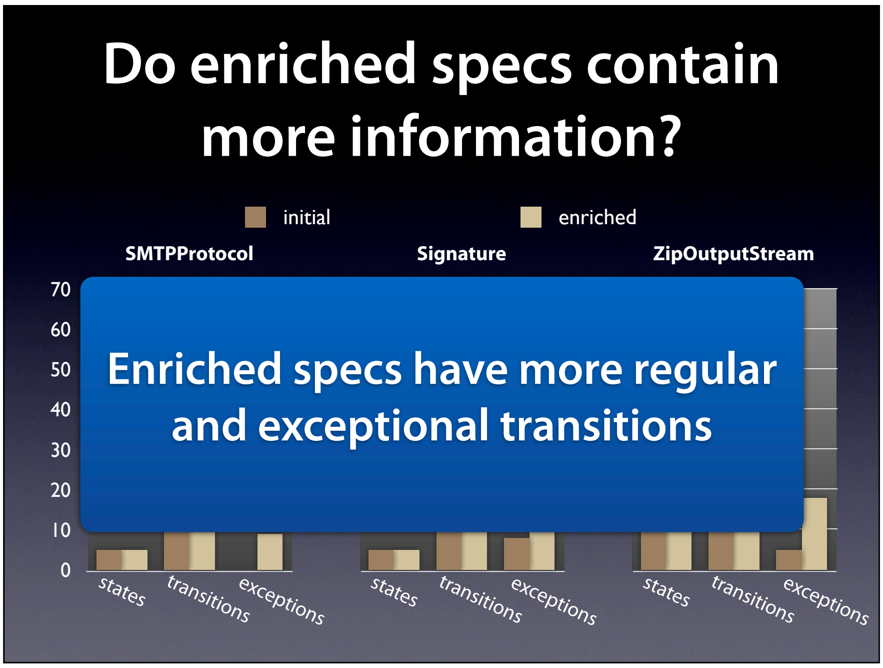


Dallmeier et al: "Generating Test Cases for Specification Mining", ISSTA 2010

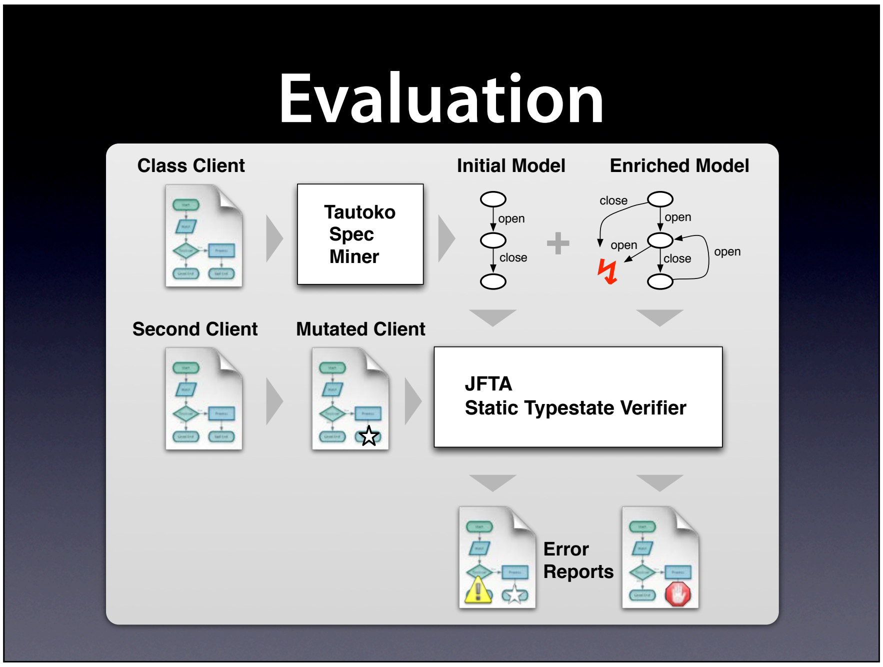


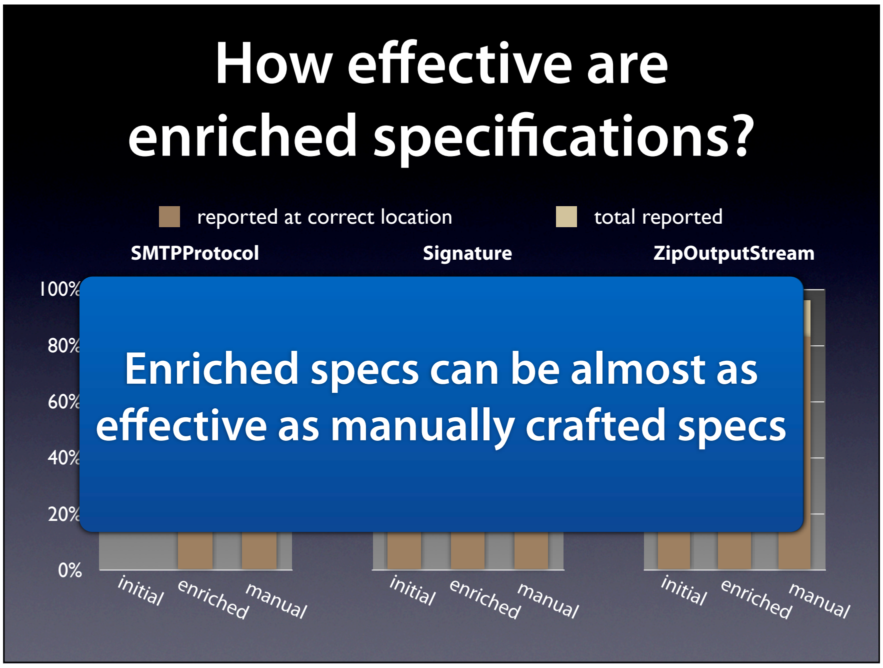
"Enriched specs have more regular and exceptional transitions";
"Enriched specs can be almost as effective as manually crafted specs"



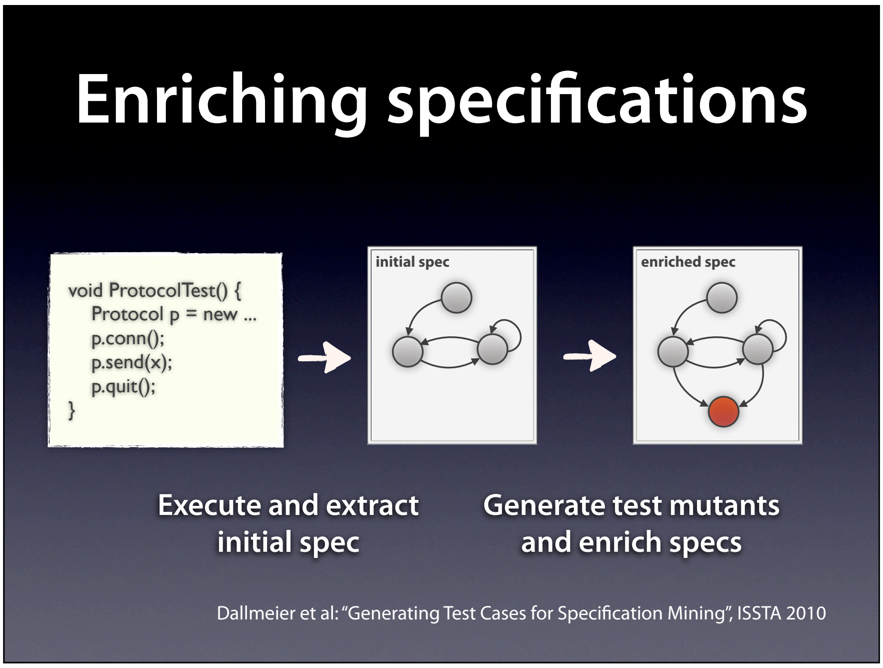


init vs enrich
consistent for 3 other subjects
Enrich more trans. ALSO BETTER FOR VERIF?



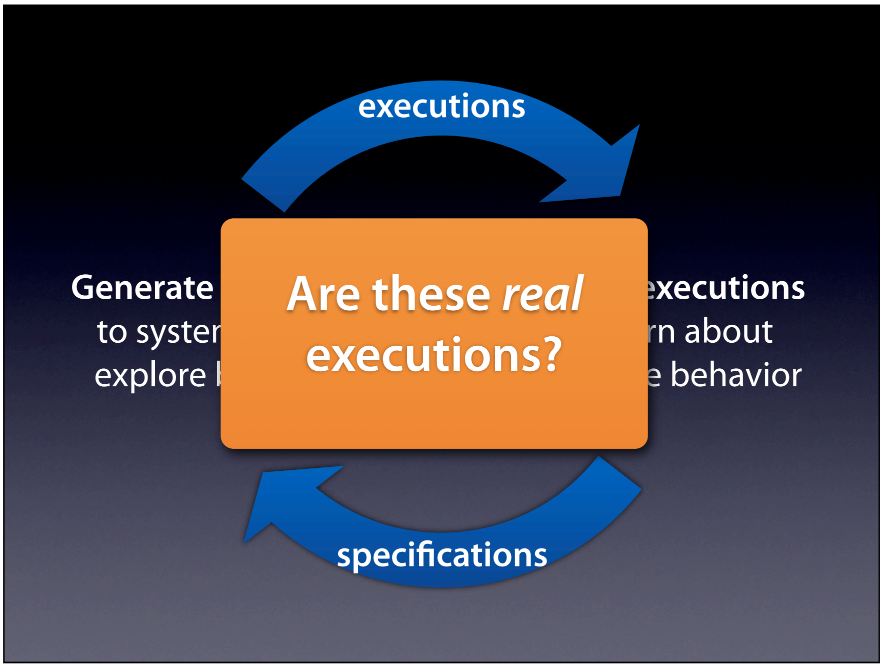


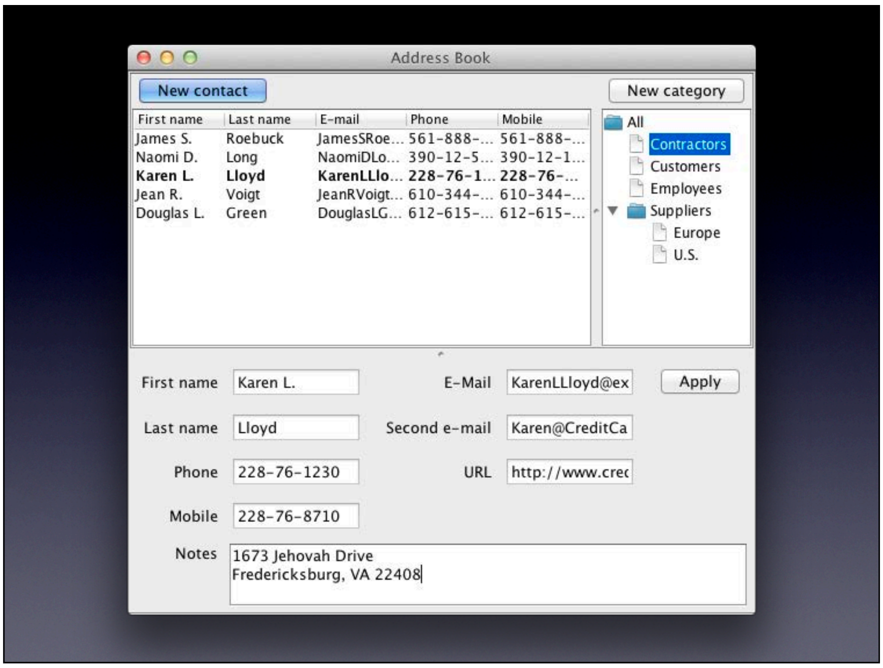
two types: report at correct call, at least report a violation
for comp, manually created model again consistent with other 3 test cases



A new kind of Analysis

- Static analysis – 0 runs
- Dynamic analysis – n given runs
- **Experimental analysis** – n generated runs





Here's a simple addressbook.

Random Testing

```
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();
```

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

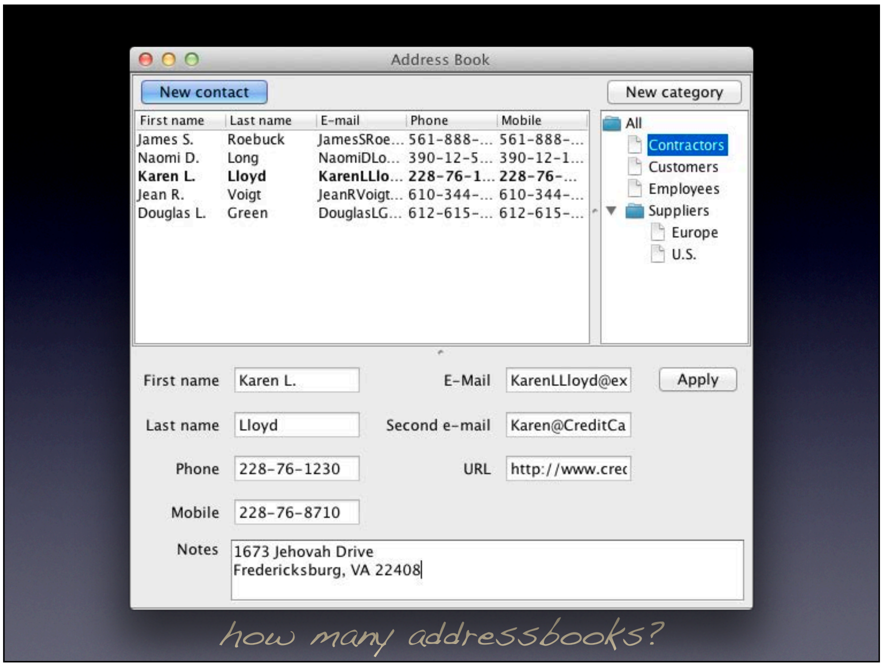
```
  AddressBook var65 = new AddressBook();
  EventHandler var66 = var65.getEventHandler();
  Category var67 = var65.getRootCategory();
  Contact var68 = new Contact();
  Category[] var69 = var68.getCategories();
  var65.removeContact(var68);
  java.util.List var71 = var65.getContacts();
  AddressBook 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

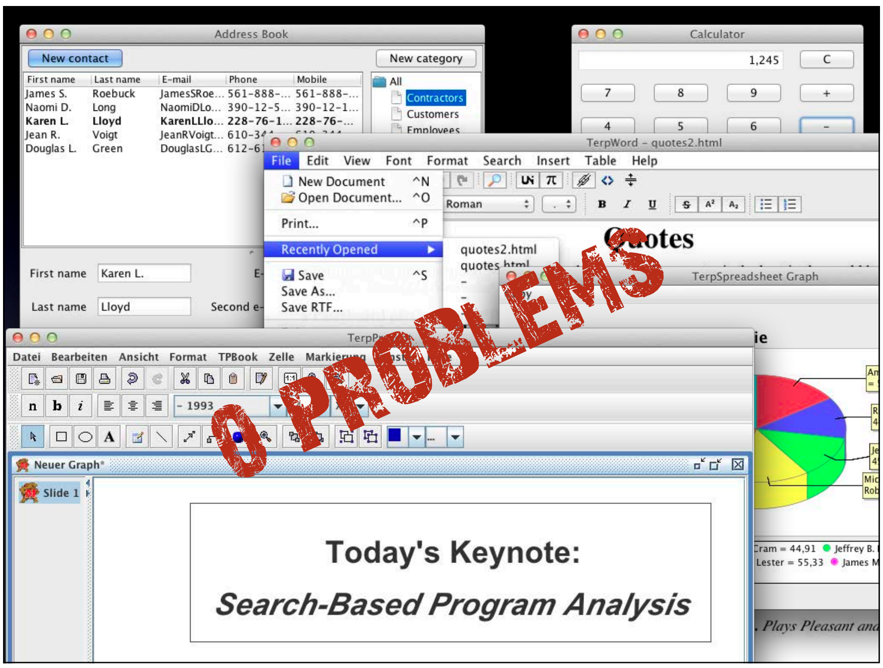
```
public class RandoopTest0 extends TestCase {
 public void test8() throws Throwable {
  if (debug) System.out.printf("%nRandoopTest0.test8");

  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

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.



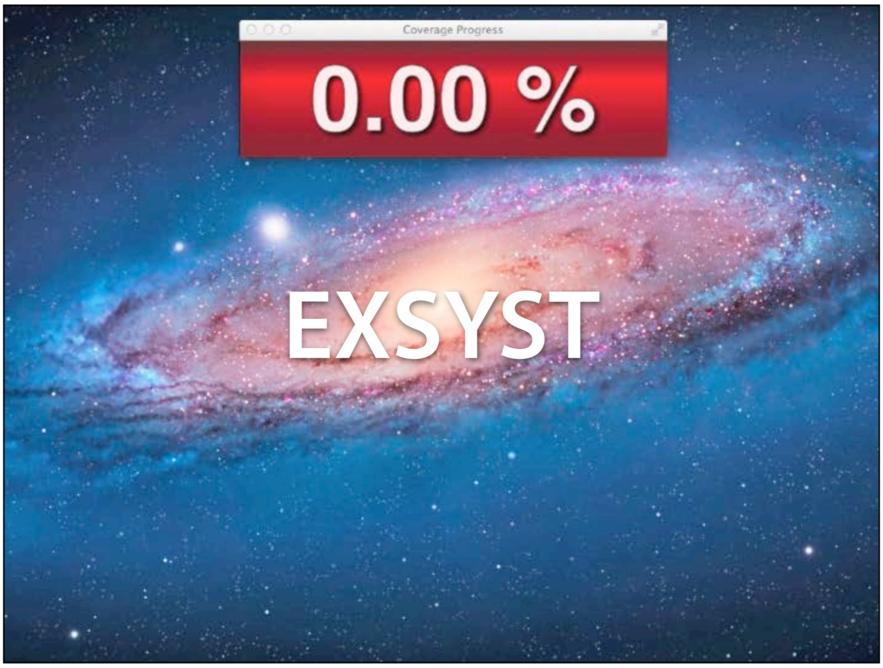
We examined a suite of five applications; overall, Randoop reported 181 failures, but all of them were false.

...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

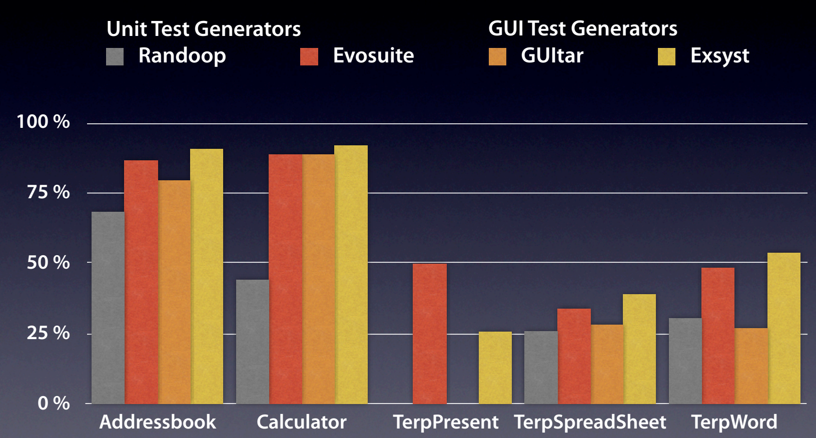
Joint work with Florian Gross and Gordon Fraser



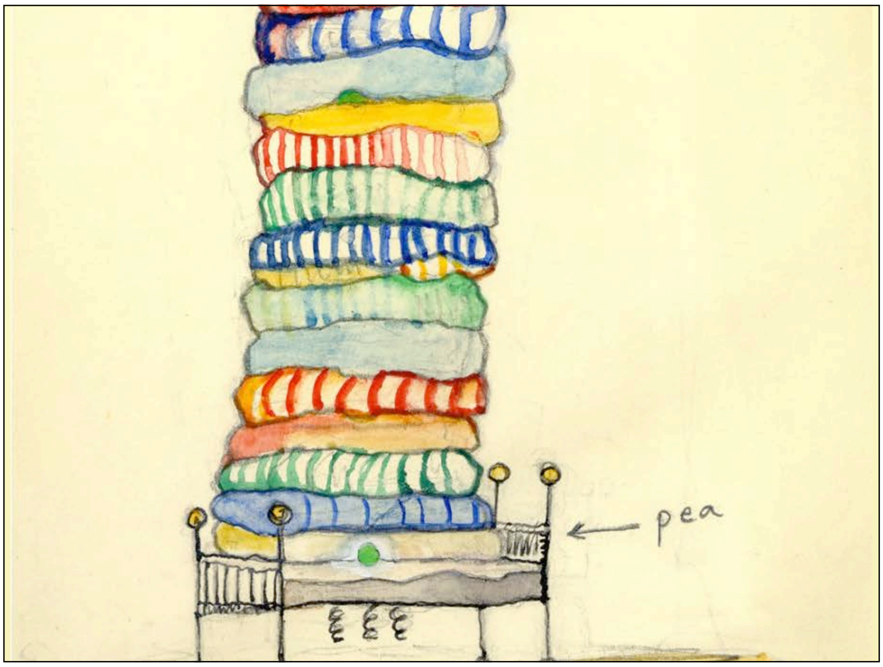
What I'm going to demo you now is our prototype called EXSYST, for Explorative SYStem Testing. EXSYST takes a Java program with a graphical user interface, such as our Addressbook example. It then generates user inputs such as mouse clicks or keystrokes and feeds them into the program. What you see here 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).

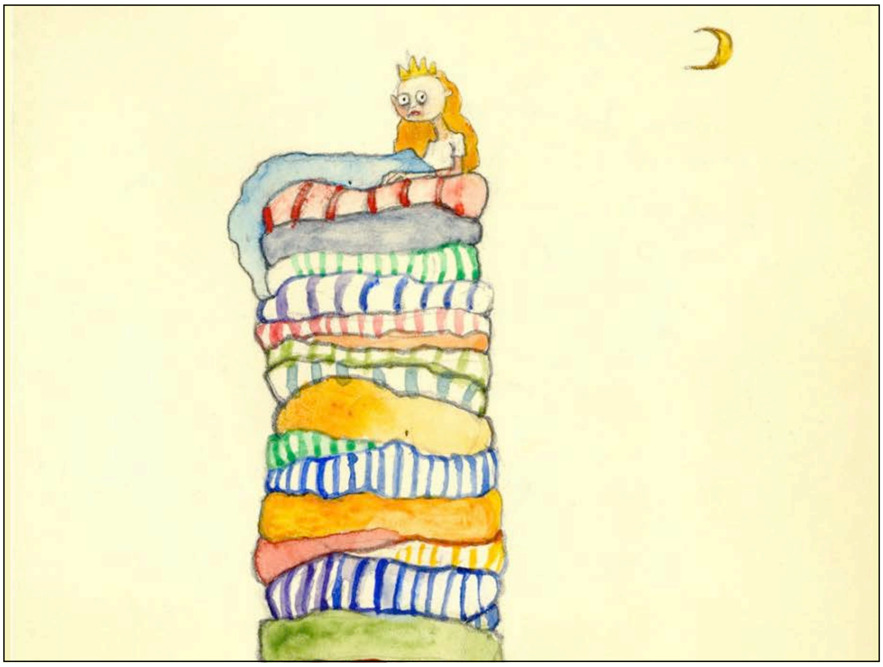
At first, these inputs are completely random, as you can see in these initial

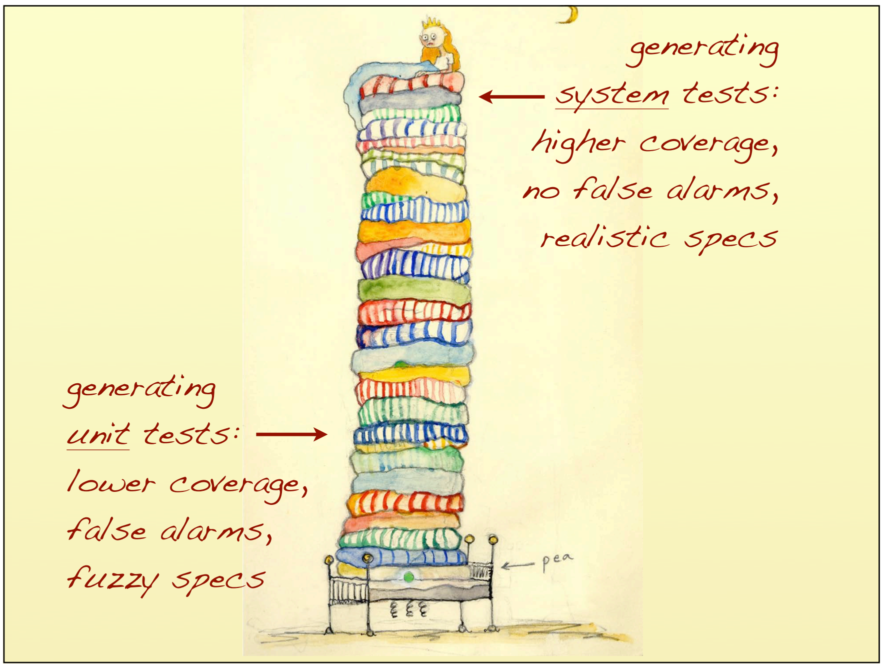
Coverage Compared

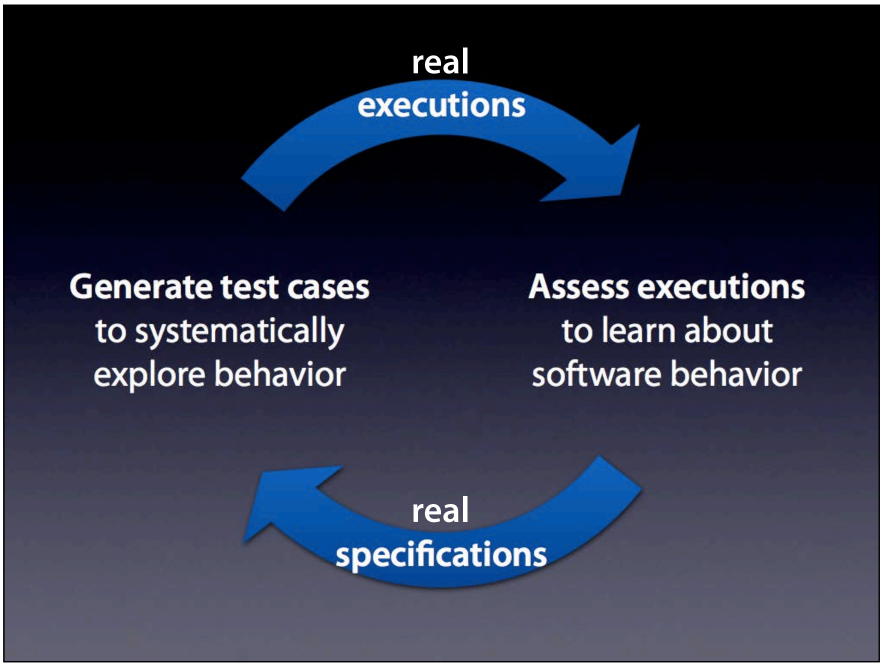


The results are clear. Although it's going through the GUI, EXSYST achieves a far higher coverage than Randoop. Here are the results for * Addressbook and

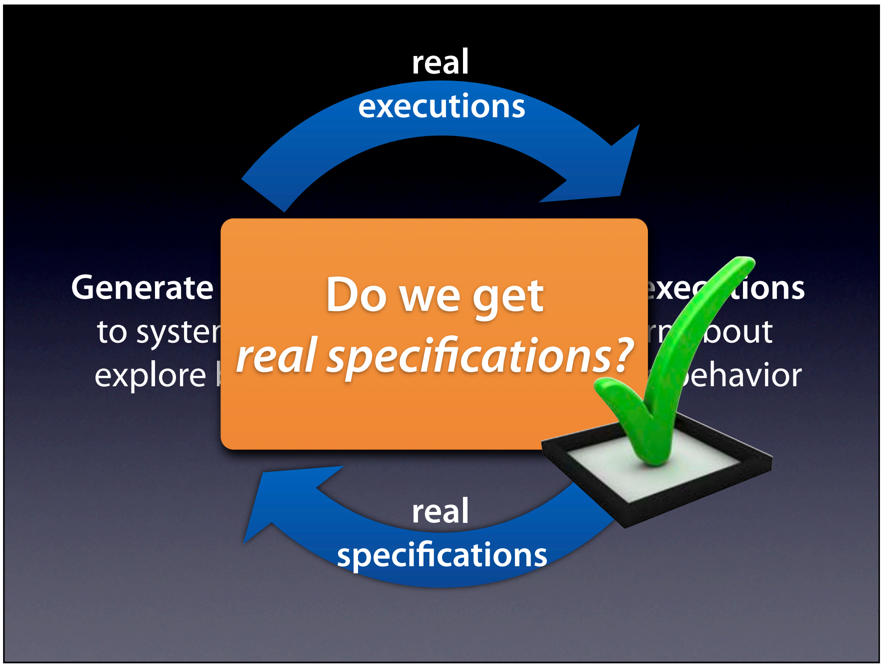

pea

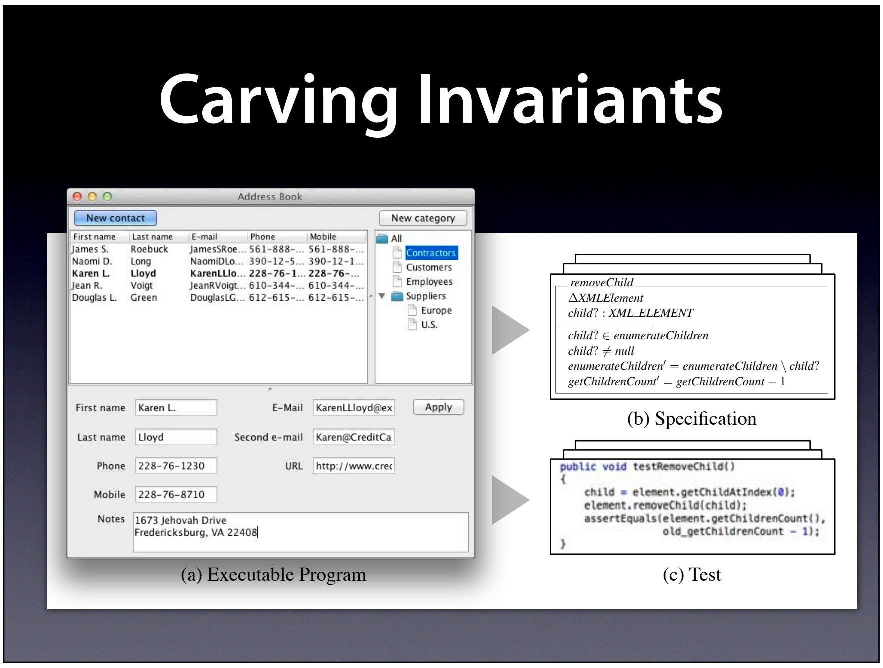



generating
system tests:
higher coverage,
no false alarms,
realistic specs
generating
unit tests:
lower coverage,
false alarms,
fuzzy specs
pea



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



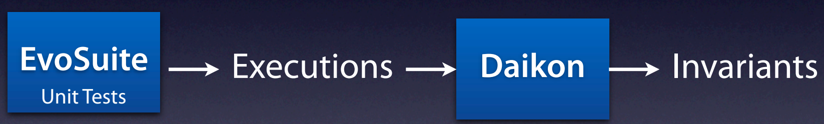


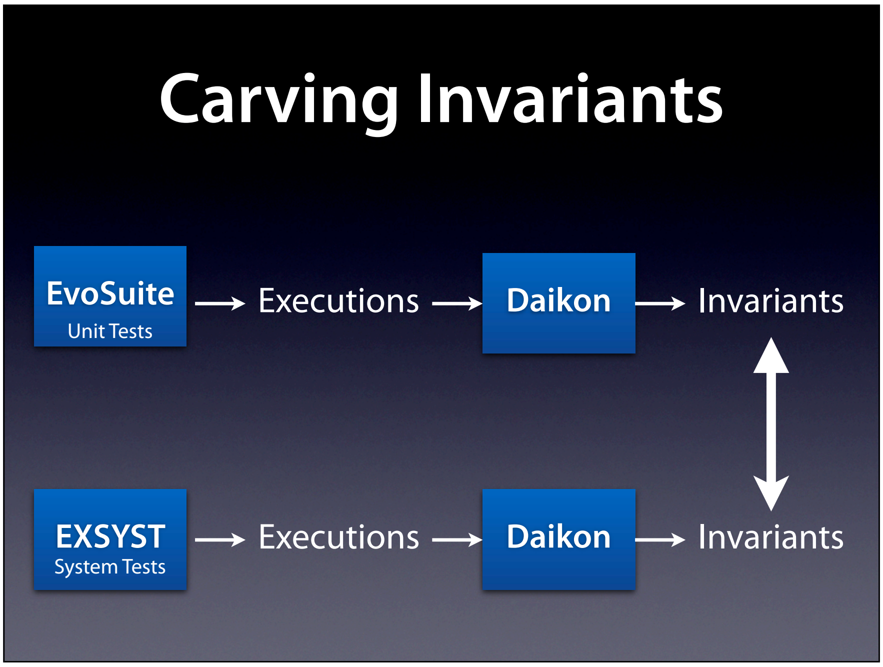
We map the pre- and postconditions, as implemented in the system interface, down to the code – and thus down to the extracted specs.

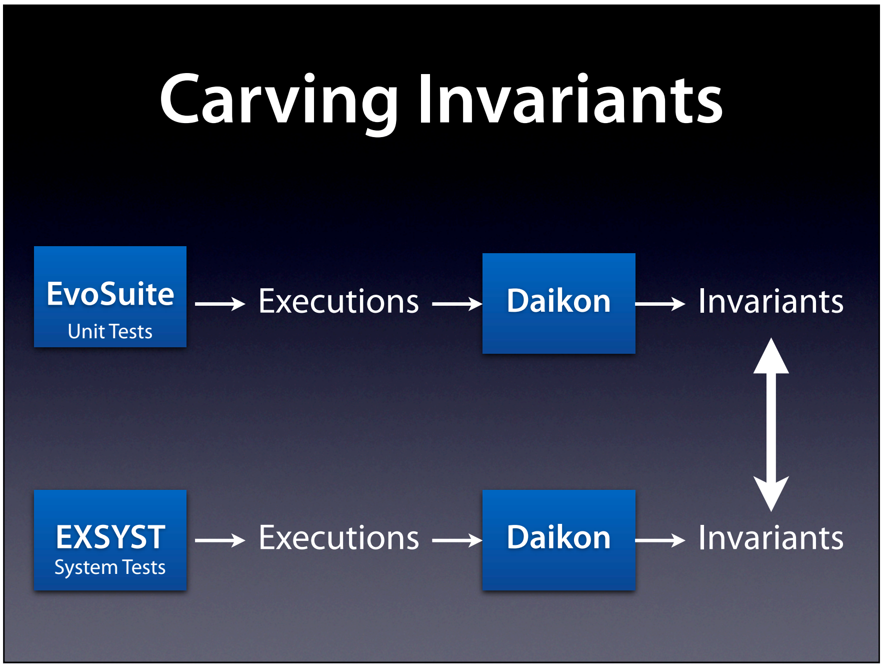
Carving Invariants

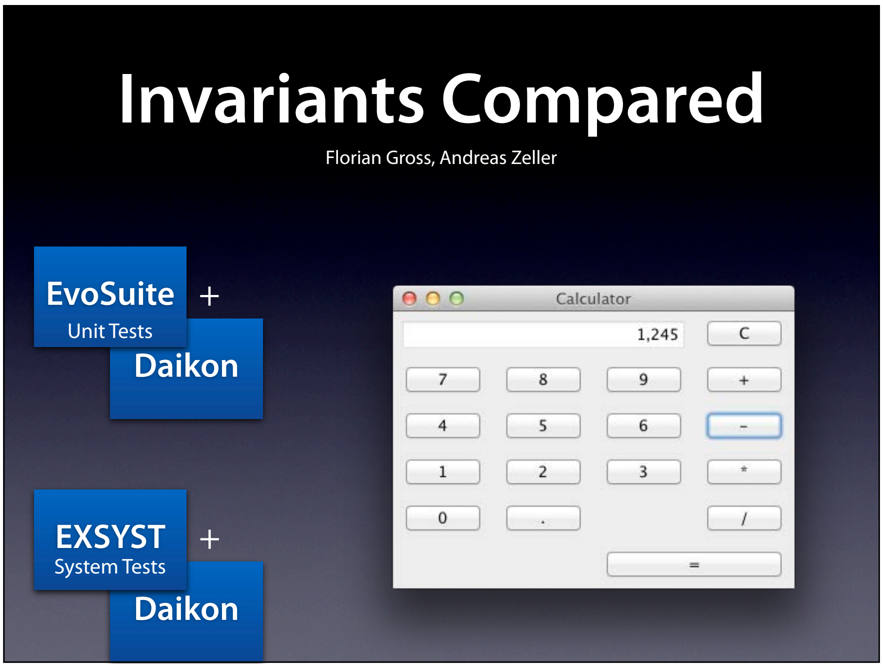
Carving Invariants

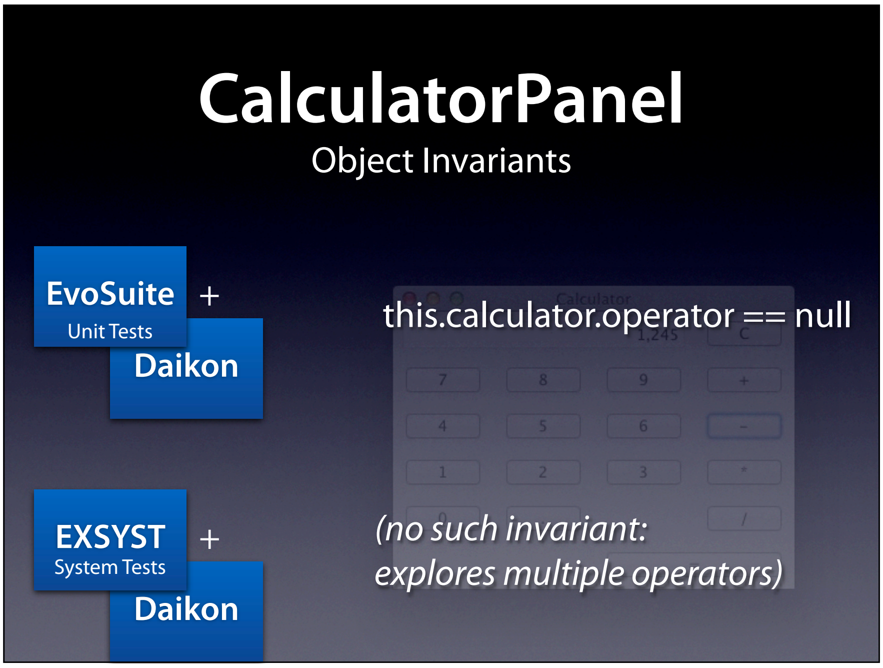
Carving Invariants

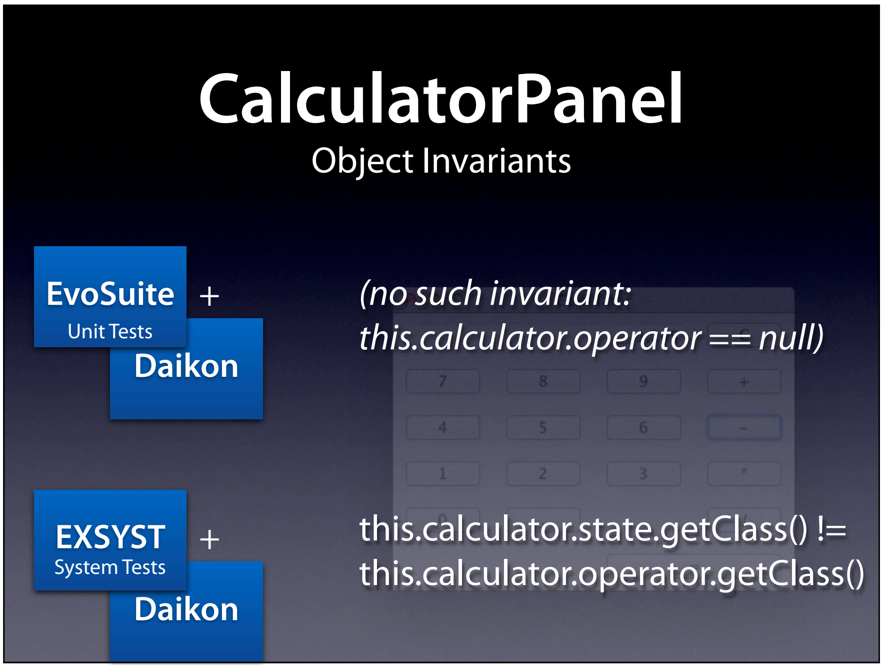


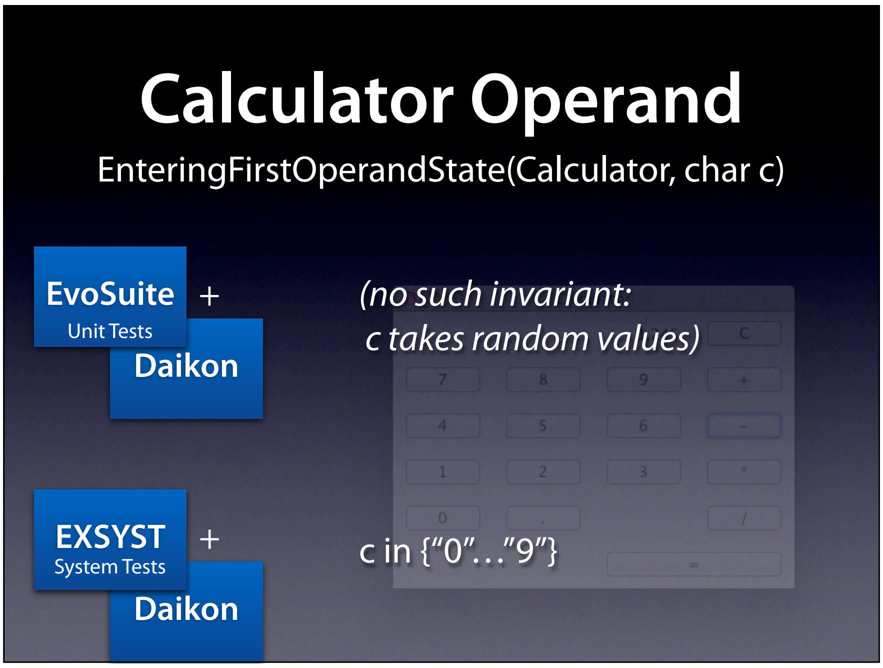

Carving Invariants
EvoSuite
Unit Tests
Executions
Daikon
Invariants
EXSYST
System Tests
Executions
Daikon
Invariants

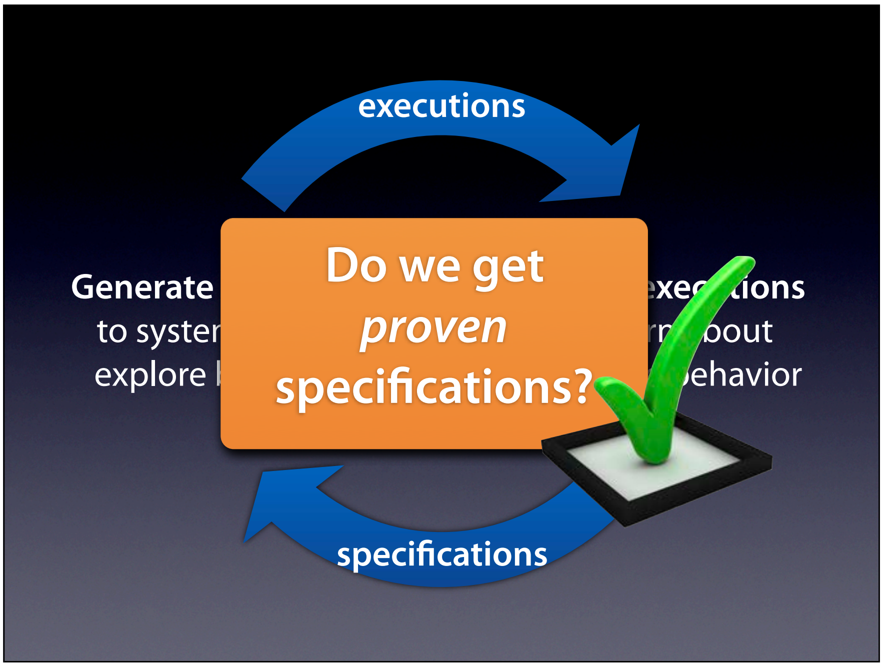

Carving Invariants
EvoSuite
Unit Tests
Executions
Daikon
Invariants
EXSYST
System Tests
Executions
Daikon
Invariants

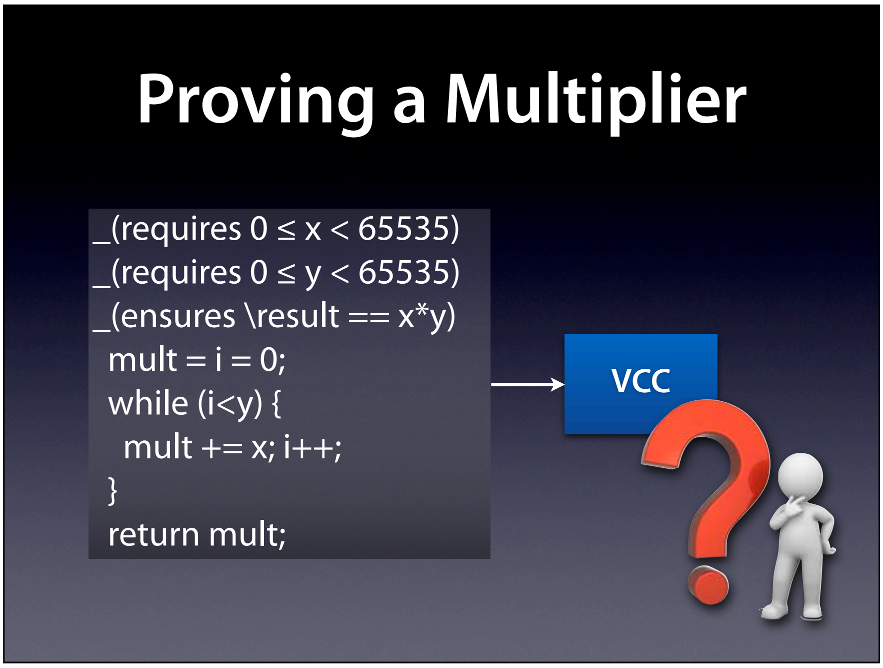

Invariants Compared
Florian Gross, Andreas Zeller
EvoSuite
Unit Tests
+
Daikon
EXSYST
System Tests
+
Daikon
Calculator
1,245
C
7
8
9
+
4
5
6
-
1
2
3
*
0
.
/
=


CalculatorPanel
Object Invariants
EvoSuite
Unit Tests
+
Daikon
this.calculator.operator == null
EXSYST
System Tests
+
Daikon
(no such invariant:
explores multiple operators)

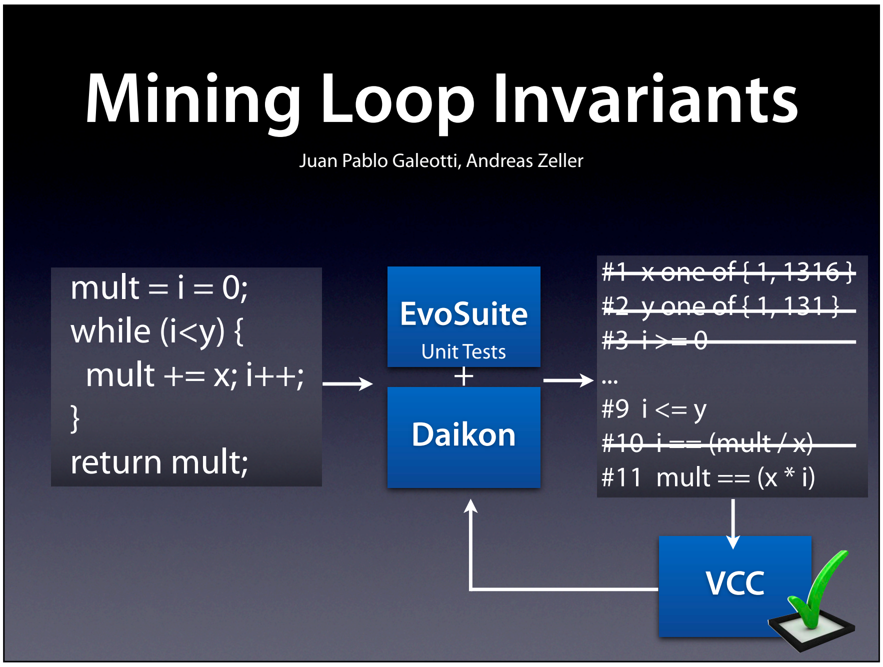

CalculatorPanel
Object Invariants
EvoSuite
Unit Tests
+
Daikon
(no such invariant:
this.calculator.operator == null)
EXSYST
System Tests
+
Daikon
this.calculator.state.getClass() !=
this.calculator.operator.getClass()


Calculator Operand
EnteringFirstOperandState(Calculator, char c)
EvoSuite
Unit Tests
+
Daikon
(no such invariant:
c takes random values)
EXSYST
System Tests
+
Daikon
c in {"0"…"9"}



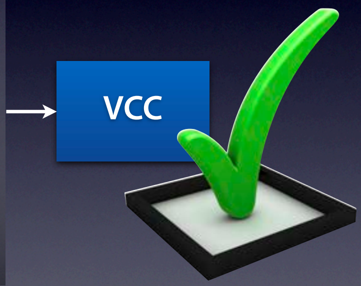


Automated program proving requires loop and recursion invariants



Proven Specifications

```
_(requires 0 ≤ x < 65535)
_(requires 0 ≤ y < 65535)
_(ensures \result == x*y)
 mult = i = 0;
 while (i<y) {
   mult += x; i++;
 }
 return mult;
```



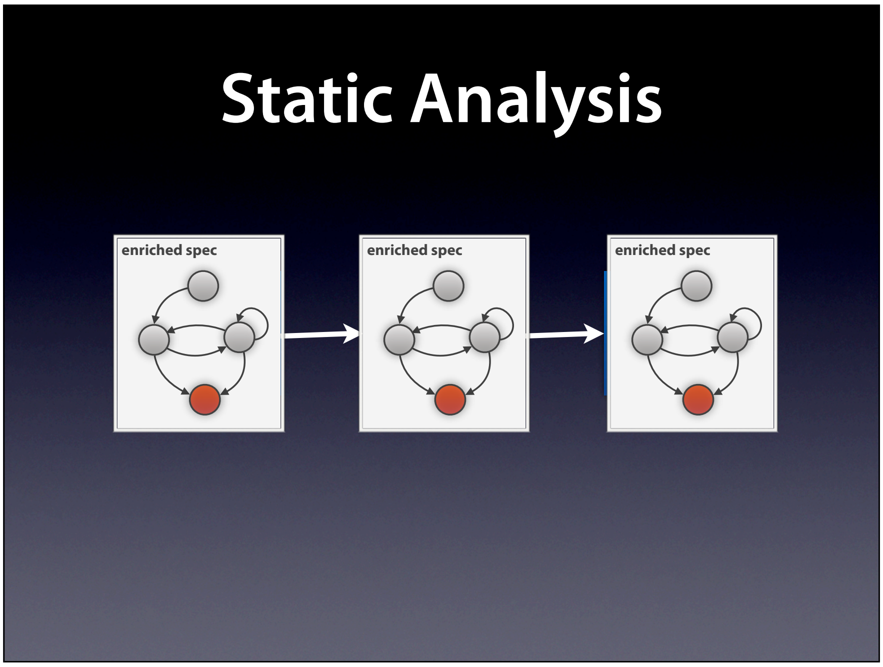
Challenges

Mine specifications that are

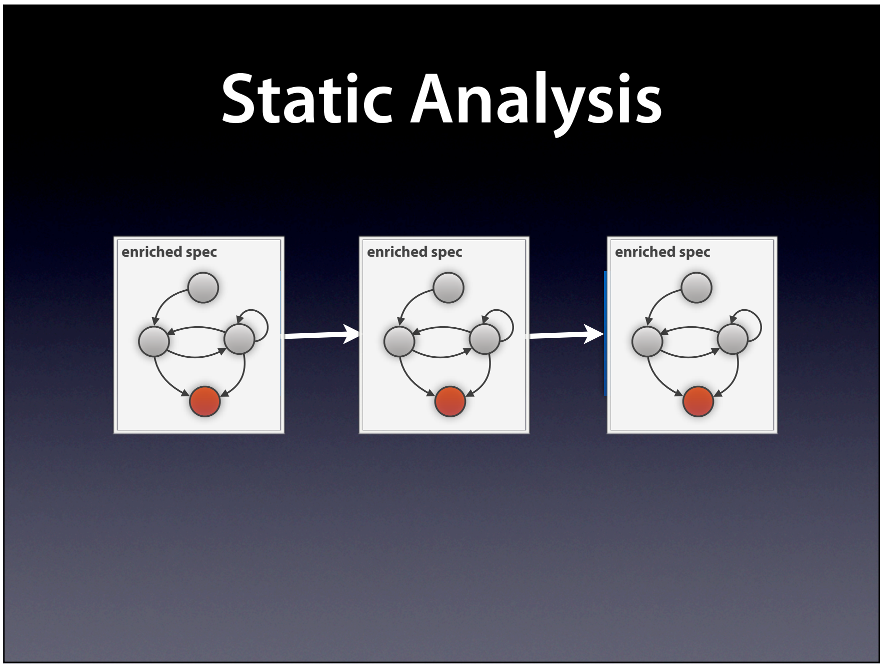
complete	**real**	**proven**

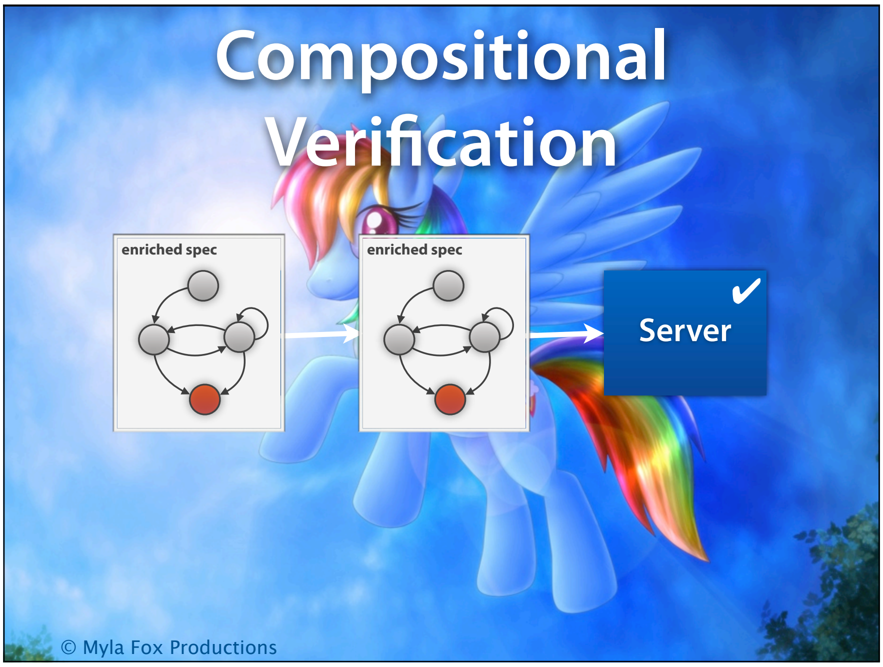
But then, remember: all of this build on a finite number of executions. Will we ever be able to reach the completeness of static and symbolic techniques?

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

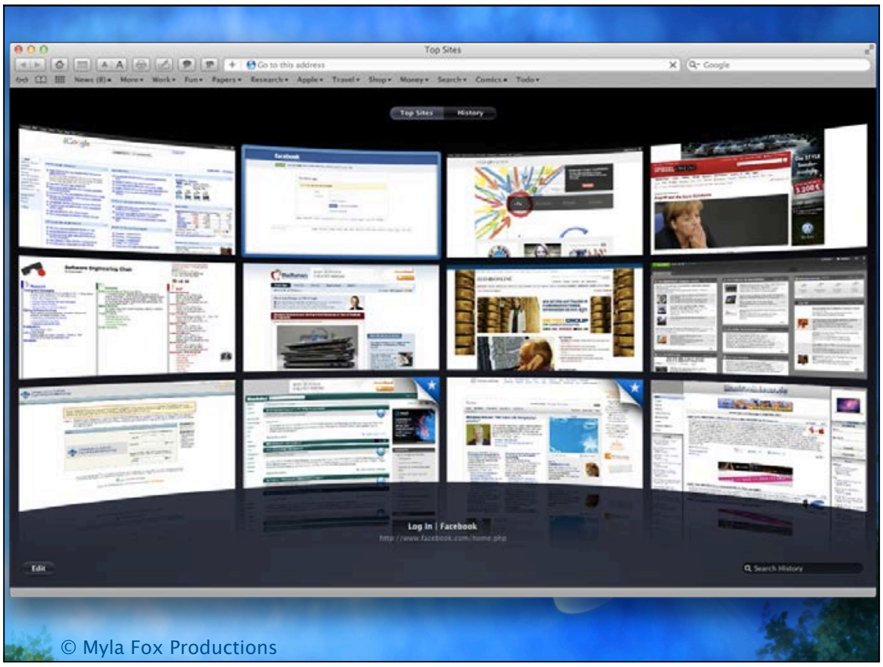


The maybe best part of experimental analysis, however, is that it **smoothly blends with all sorts of static analysis and verification.** That's because we can use the mined specifications as **surrogates** for individual components, allowing for **local verification and analysis.**





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…



…scalable to the challenges that await for us, every day, everywhere in the wide world of software.

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

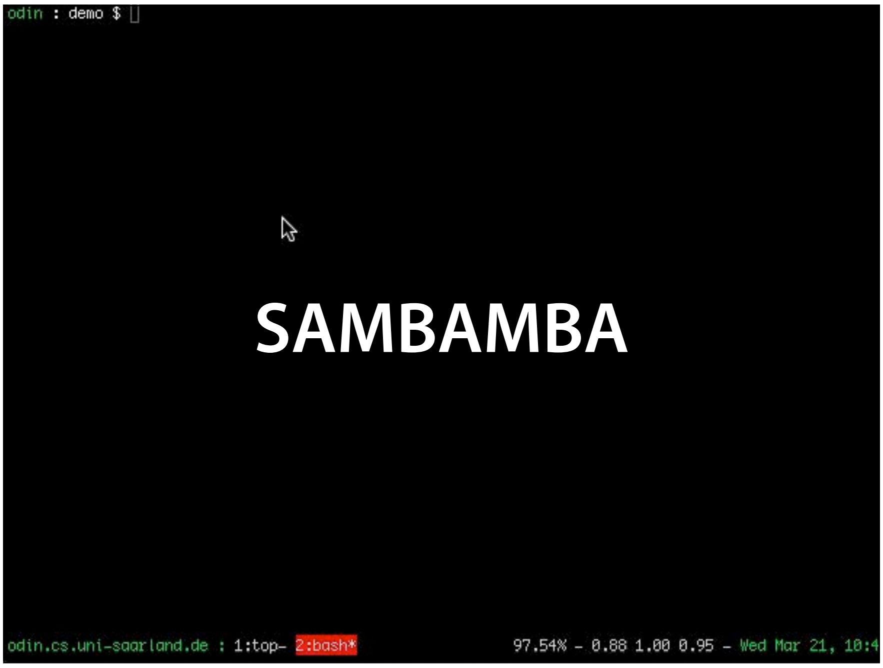
- Originates from *execution monitoring*
- Considers (only) *actual* executions
- Covers all abstraction layers
- Tied to *run-time verification* techniques

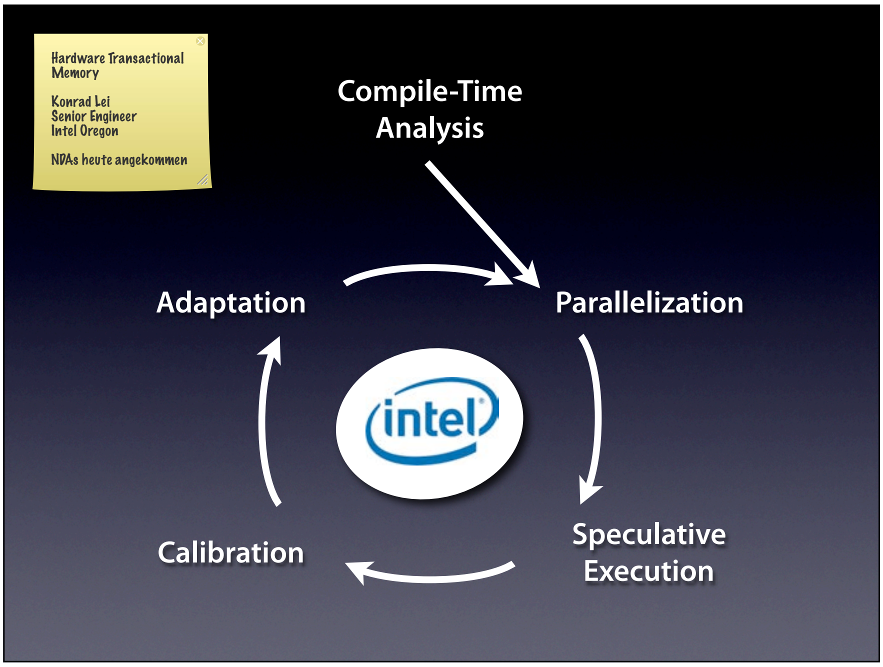
SAMBAMBA

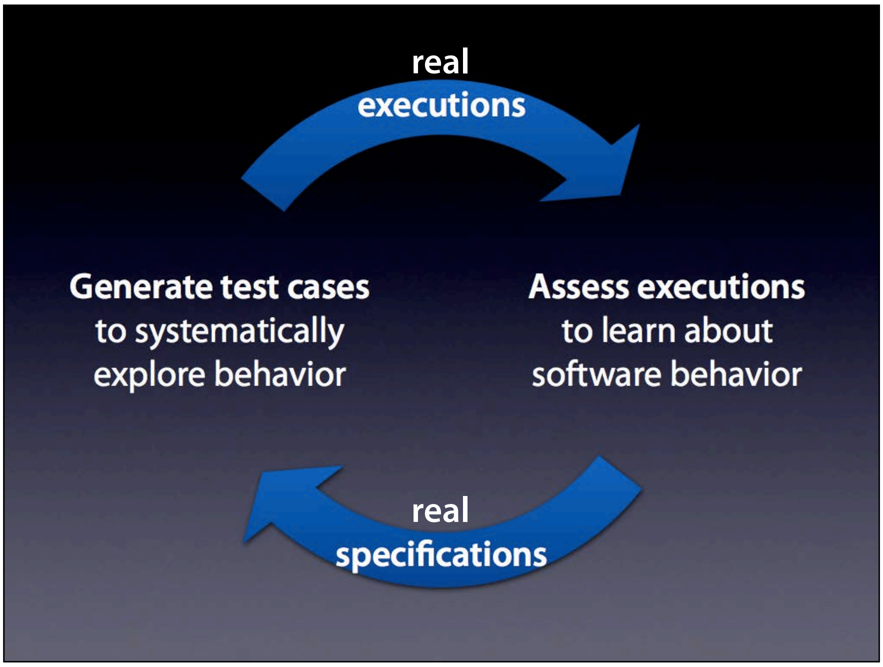
- Compiler and runtime system for online adaptive parallelization
- Based on LLVM
- Target: Common C/C++ programs

```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);
}
```

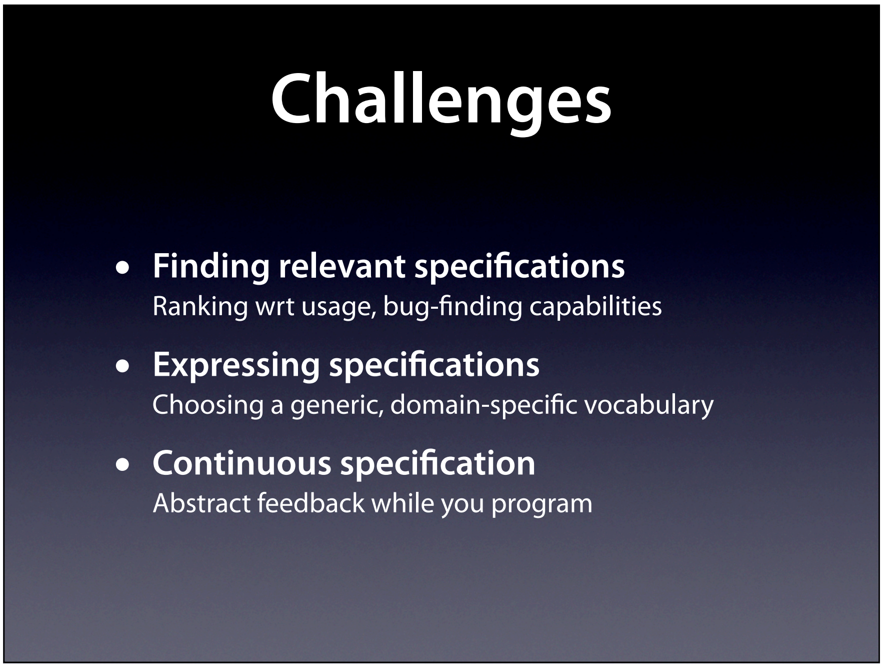
Demo 1:
- gcc
- execute gcc version
- run sambamba (parallelized both functions)
- execute sambamba version

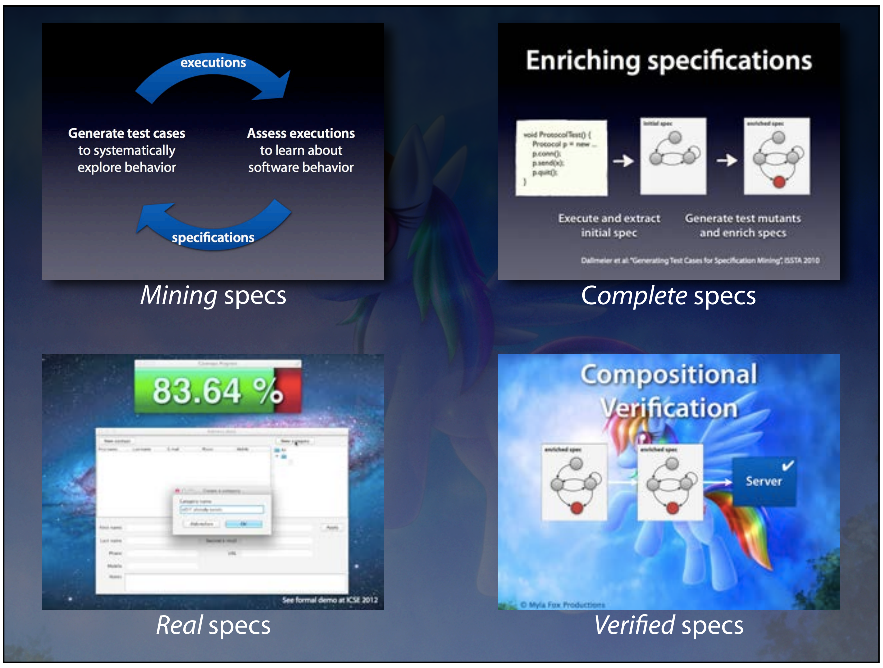






* The more we can cover behavior, the more we learn about the system
* In presence of obscure code, search-based techniques are first choice





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