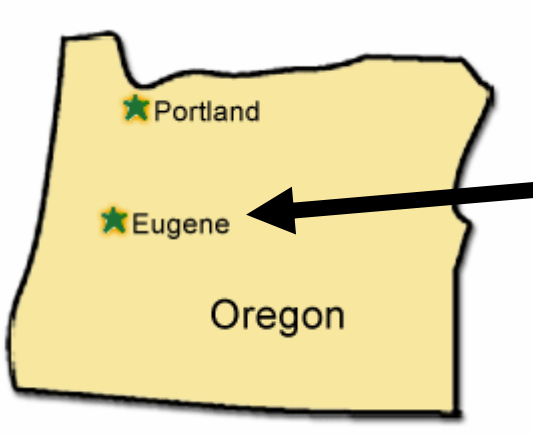
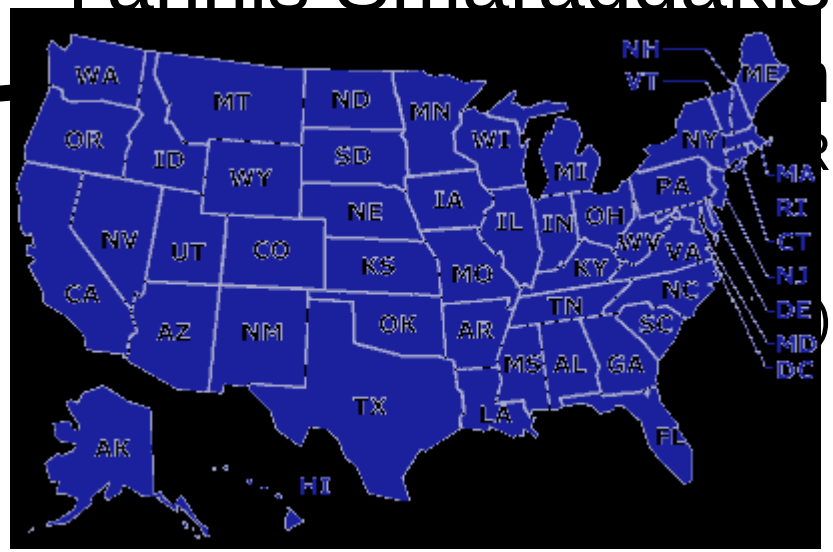
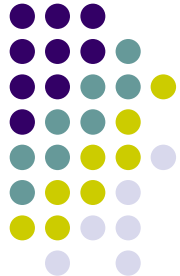
DSD-Crasher: A Hybrid Analysis Tool for Bug Finding

Yannis Smaragdakis





My Research



- ***The systems and languages end of SE***
  - **language tools for distributed computing**
    - NRMI, J-Orchestra, GOTECH
      [ICDCS'03, ECOOP'02, Middleware'04, ICSM'05, IEEE PervComp, ASE'03, ICSE'05, ...]
  - **automatic testing**
    - JCrasher, Check-n-Crash (CnC), DSD-Crasher
      [Softw.Prac.&Exp., ICSE'05, ICSE'06 ER, ISSTA'06 (best paper), ...]
  - **program generators and domain-specific languages**
    - cJ, Meta-AspectJ (MAJ), SafeGen, JTS, DiSTiL
      [GPCE'04 (best paper), ICSR'98, ICSE'06 ER, PEPM'04, GPCE'05, AOSD'07, ...]
  - **multiparadigm programming**
    - FC++, LC++
      [ICFP'00, JFP, Softw.Prac.&Exp., ...]
  - **software components**
    - mixin layers, layered libraries
      [ECOOP'98, ICSR'98, ICSR'02, TOSEM, ...]
  - **memory management**
    - EELRU, compressed VM, trace reduction, adaptive replacement
      [SIGMETRICS'99 (2x), Usenix'99 (best paper), TOMACS, Usenix'00, ISMM'04, MICRO'06, ...]

Find bugs in software

Testing widespread, for good reasons

Requires understanding the (mostly informal) spec

(–) Write test cases manually → Time-consuming

Generate test cases automatically (our Check 'n' Crash)

(–) Ignore informal spec → **False positive warnings**

Our goal: Generate better test cases (DSD-Crasher)

(+) Infer informal spec → Fewer false positive warnings

False positives: Show-stopper

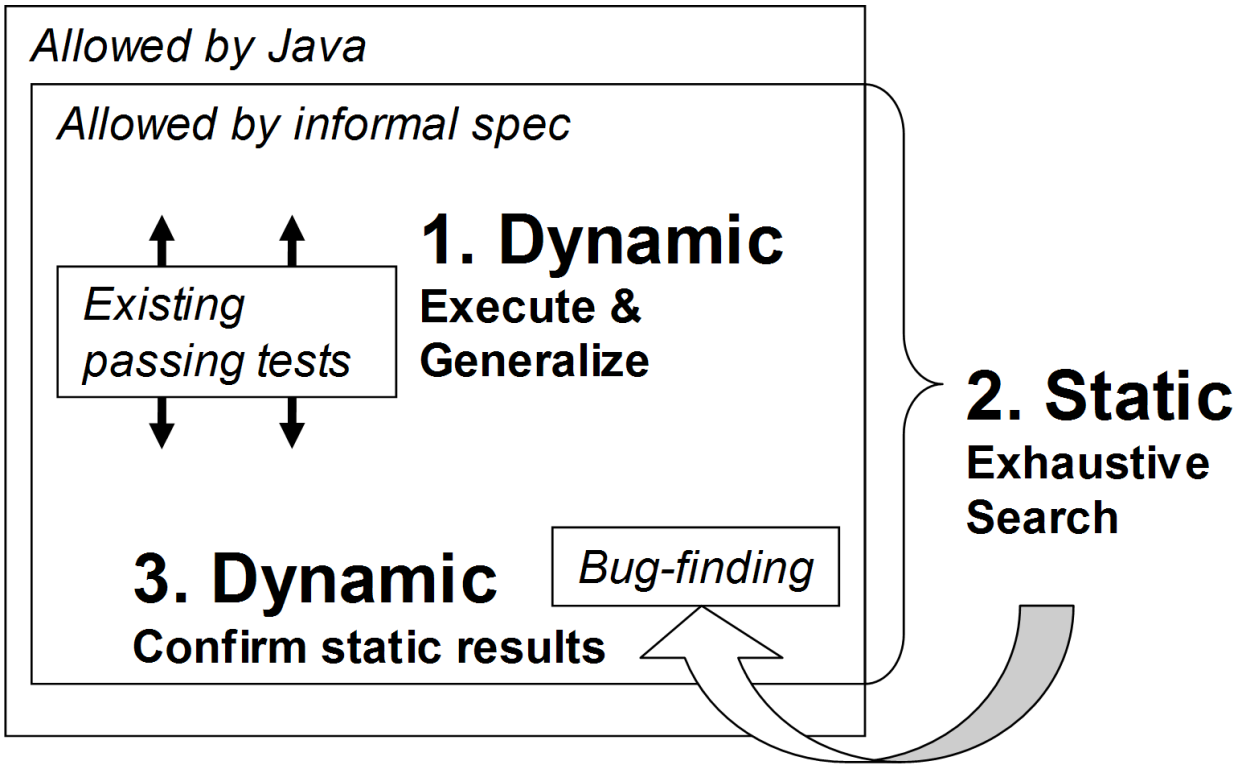
Flanagan et al. (ESC/Java people), 2002:

> "[T]he tool has not reached the desired level of cost effectiveness. In particular, users complain about an annotation burden that is perceived to be heavy, and about **excessive warnings about non-bugs**, particularly on unannotated or partially-annotated programs."

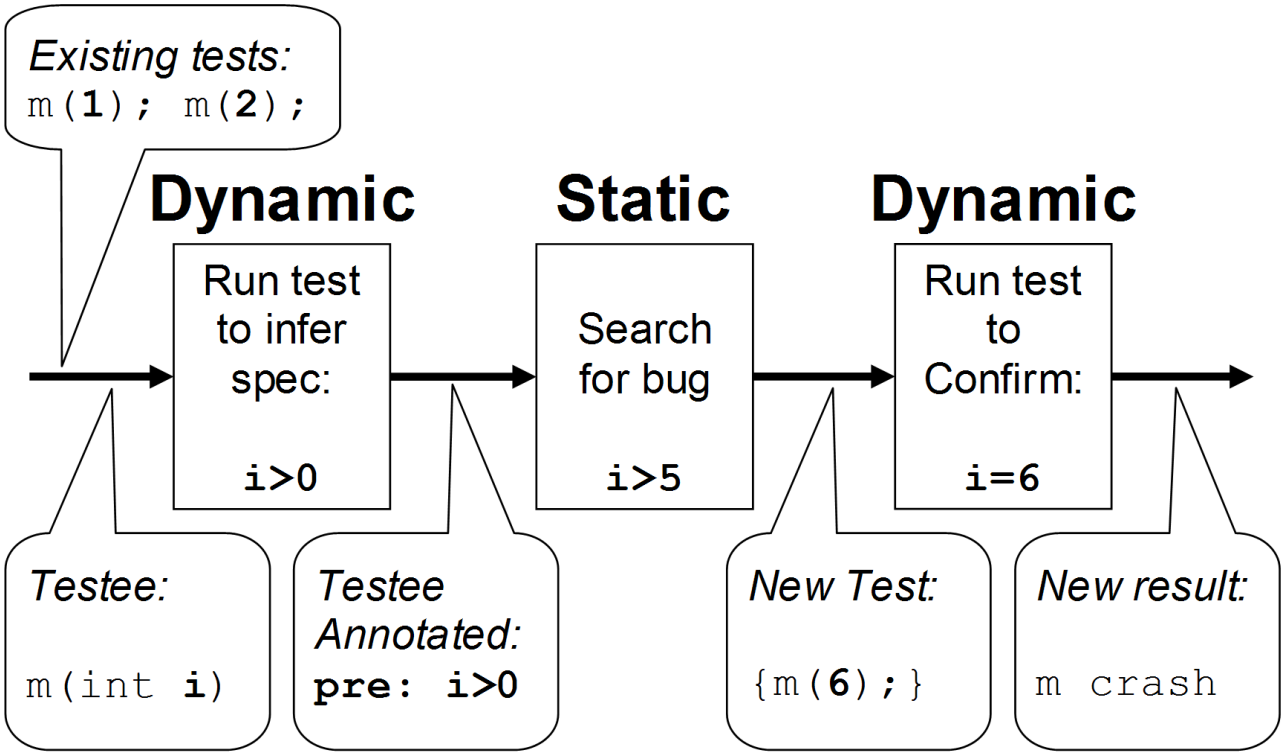
Rutar et al., 2004:
$>$ 9k NPE warnings in 170k non commented source stmt

> "[T]here are **too many warnings to be easily useful by themselves**."

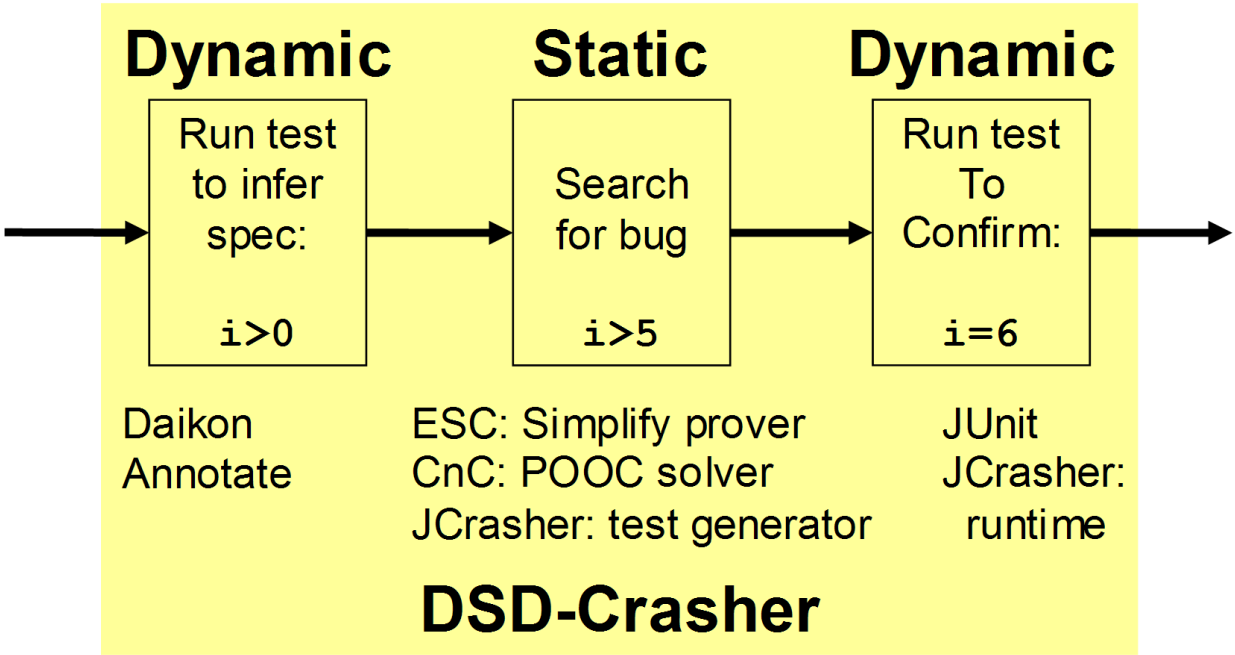
How we treat testee input



DSD-Crasher overview



DSD-Crasher overview



ESC/Java: Extended Static Checking

★ Compile-time program checker

★ Recognizes invariants stated in Java Modeling Language

★ Knows pre- and postconditions of language operations

Pointer dereference, class cast, array access, etc.

★ Detects potential invariant violations

We concentrate on runtime exceptions

★ Analyzes each method in isolation

(–) No inter-procedural analysis

ESC/Java: Analyzing a method

1. In the body, replace any call to any entity `m` with:

```
check precondition(m);
assume postcondition(m)
```

2. Compute weakest precondition:

   `wp(method body, true)`

   States from which execution terminates normally

   Remaining states lead execution to an error

   Violate some precondition (or postcondition)

   - Dereferencing null
   - Illegal type cast
   - Illegal array allocation or access
   - Division by zero

False positives (violate Java semantics)

Imprecise = Unsound: ESC produces spurious error reports

Cases that cannot occur (Java semantics approximated)

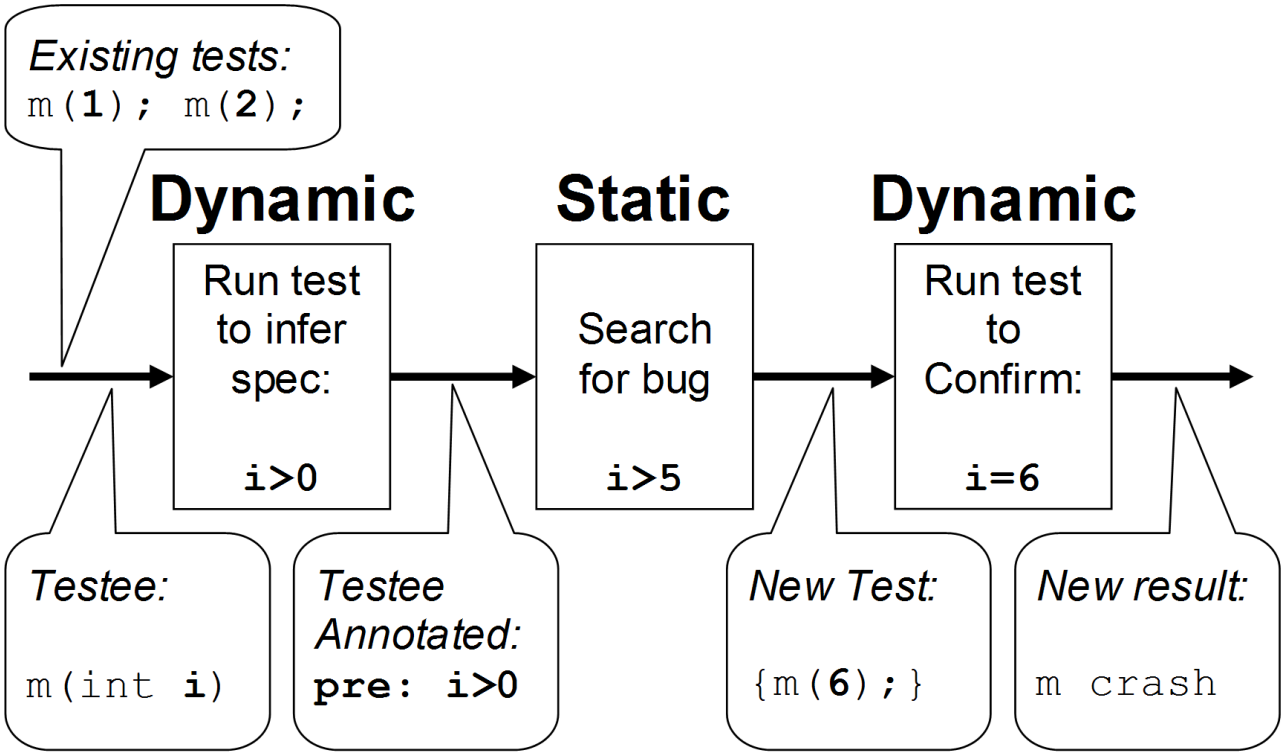
```
public int get10() {return 10;}

public int meth(int p) {return p / get10();}
```

ESC ignores implementation of `get10()`

ESC warns of a division by zero in `meth`

Our solution: Generate test, confirm behavior: DS**D**

DSD-Crasher overview



False positives (violate user spec)

ESC cannot access informal spec

```
public int forPosInt(int i) {
  if (i<0) throw new MyRuntimeException();
  //..
```

1. User will consider report a **false positive**

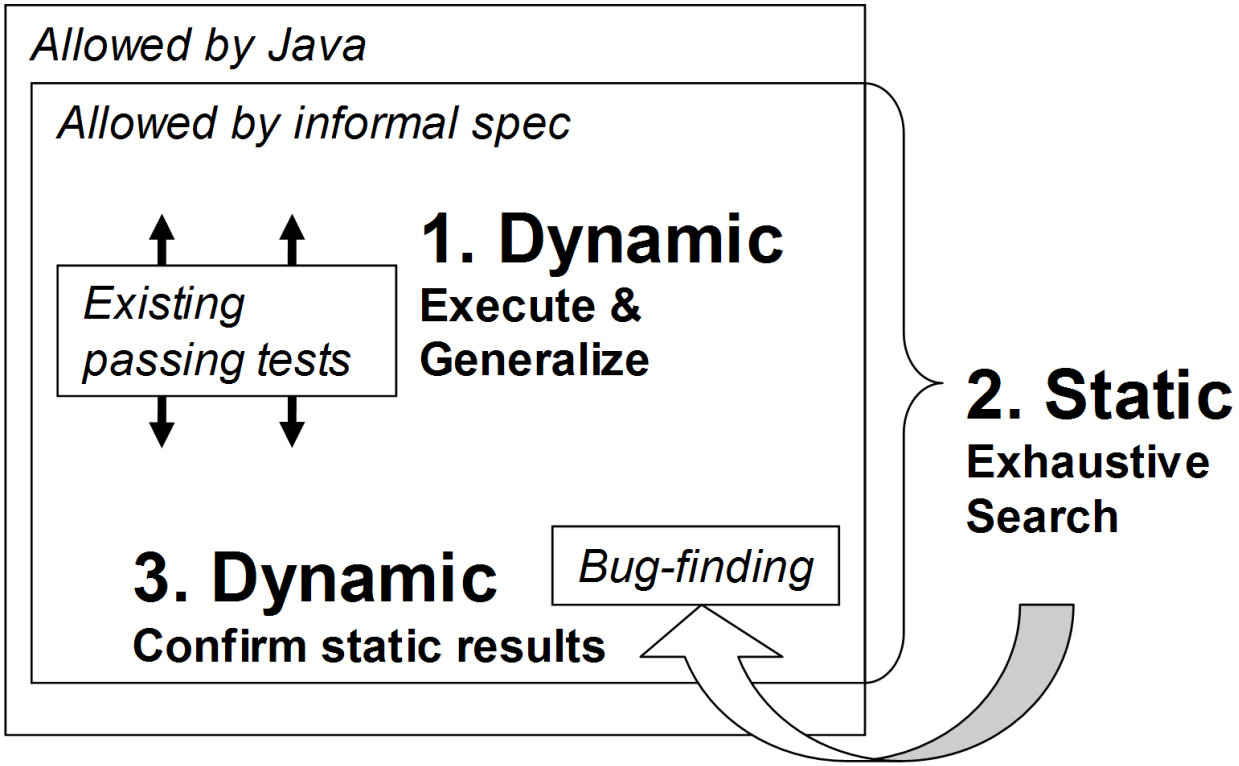
   Need to infer implicit spec (first dynamic step)

2. May also terminate test execution path to real problem

```
public int caller(int p) {
  int j = forPosInt(p);
  //bug
```

(–) May suppress true positives

Idea


Allowed by Java
Allowed by informal spec
Existing passing tests
1. Dynamic
Execute & Generalize
2. Static
Exhaustive Search
3. Dynamic
Confirm static results
Bug-finding

Daikon: Infer invariants from execution

1. Instrument testee
2. Execute testee
   Dump variable values at each method entry & exit
3. Analyze execution traces
   (a) Instantiate invariant templates with variable values
   (b) Invariant invalidated by sample $\rightarrow$ Drop invariant
4. Invariant held for some samples and never invalidated
   $\rightarrow$ Assume: true invariant
5. Annotate testee's source code (with JML):
   Preconditions, postconditions, class invariants

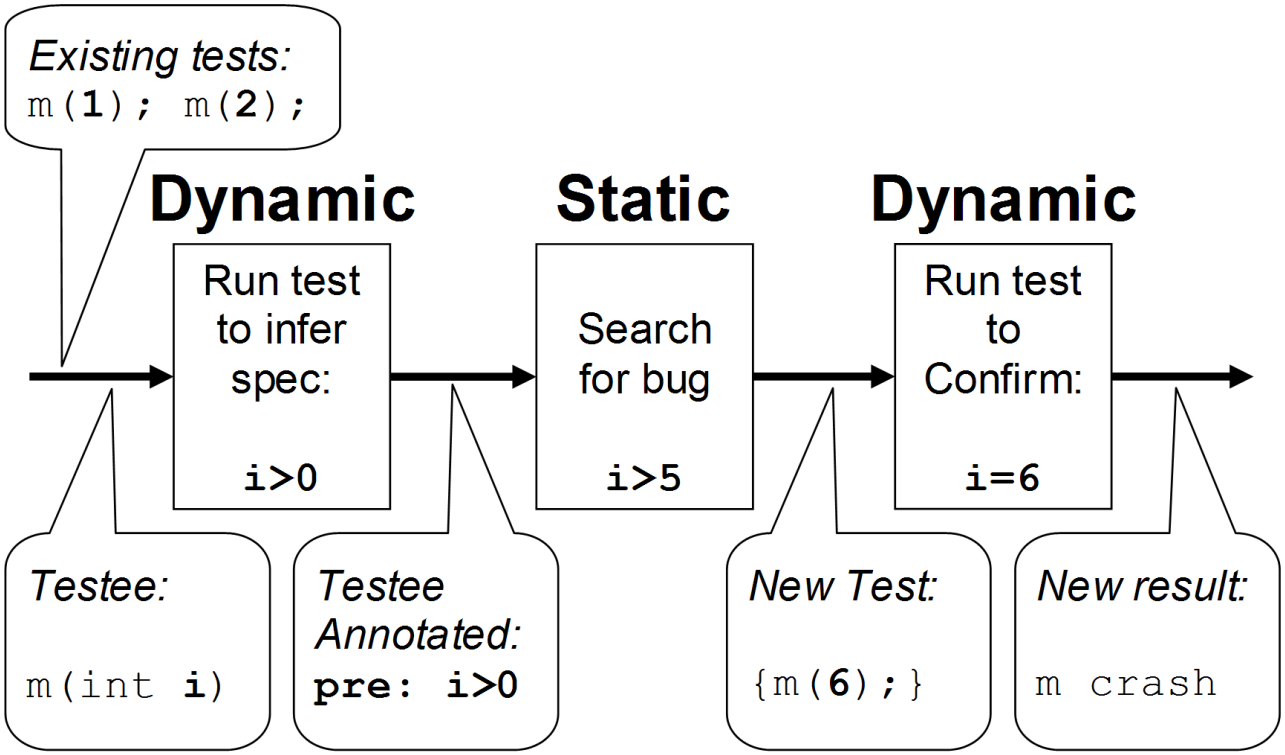
Daikon configuration

- Concentrate on simple invariants, e.g.
  - intVariable $\{==, >=, >, <=, <\}$ intConstant
  - intVariable $\{==, >=, >, <=, <\}$ intVariable
- Complex invariants even harder to infer correctly
- Ignore most complex invariants that involve
  - variable is one of $\{$const1, const2, ..$\}$
  - Elements of container structures
  - float, double, String

Daikon expresses invariants relative to methods

Wrote JML spec for frequently used methods, e.g.:
`daikon.Quant.size`

DSD-Crasher overview



Inferred invariants as assumptions

```
//inferred pre-condition: (i != 0)
//inferred post-condition: (\result < 0)
public int foo(int i) {..}
```

- Daikon invariants generalize observed behavior
- Body of `foo`: Assume input to be `(i != 0)`
  - (+) Exclude probably unwanted input
  - (–) May miss bugs caused by `(i == 0)`
- Calling `foo`: Assume `(\result < 0)`
  - (+) Exclude probably impossible output
  - (–) May miss bugs when `foo (\returns >= 0)`

Inferred invariants not requirements

```
//inferred pre-condition: (i != 0)
//inferred post-condition: (\result < 0)
public int foo(int i) {..}
```

- Very likely that Daikon has missed valid executions

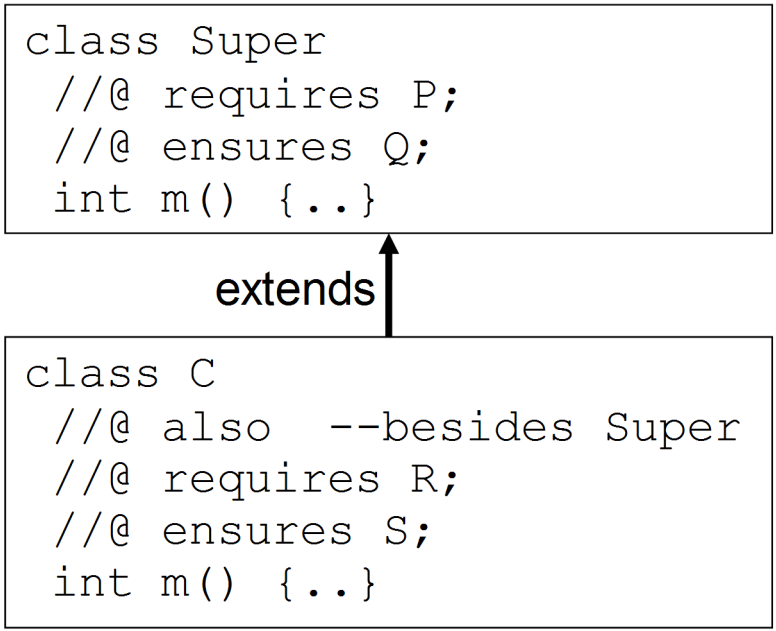
  Calling `foo` with `(i == 0)` may be ok

  `foo` producing `(\result >= 0)` may be ok

- Do not confuse user with violations of guessed invariants

  Do not enforce inferred invariants as requirements

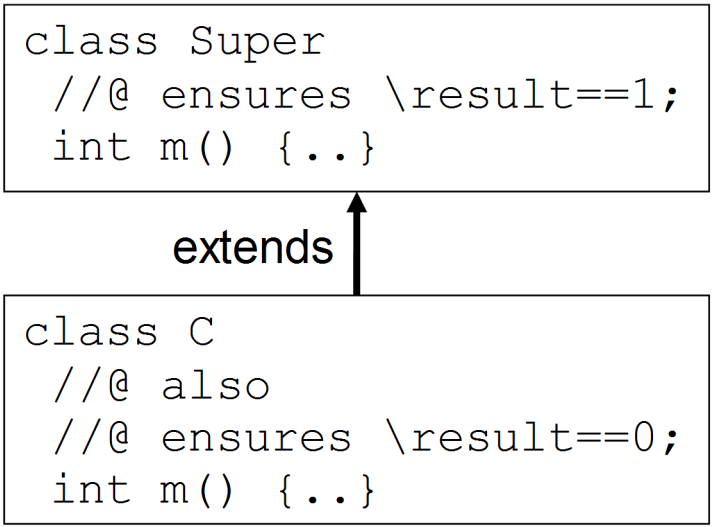
What is behavioral sub-typing?



- Pre-condition(`C.m`): `(P or R)`
- Post-condition(`C.m`): `(P -> Q) and (R -> S)`

Example of contradicting invariants

Daikon associates values & invariants with executed body



Derive postcondition for `C.m`:

```
((result==1) & (result==0)) <-> false
```

Evaluation

Goal is to find bugs, not to cover code

(+) Static analysis already covers code

Nimmer & Ernst 2002: Precision and recall

(+) Great if you know perfect results (here: all bugs)

JBoss JMS, part of JBoss 4.0 RC1

5k non-comment source statements

Groovy 1.0 beta 1 version, excluded low-level classes

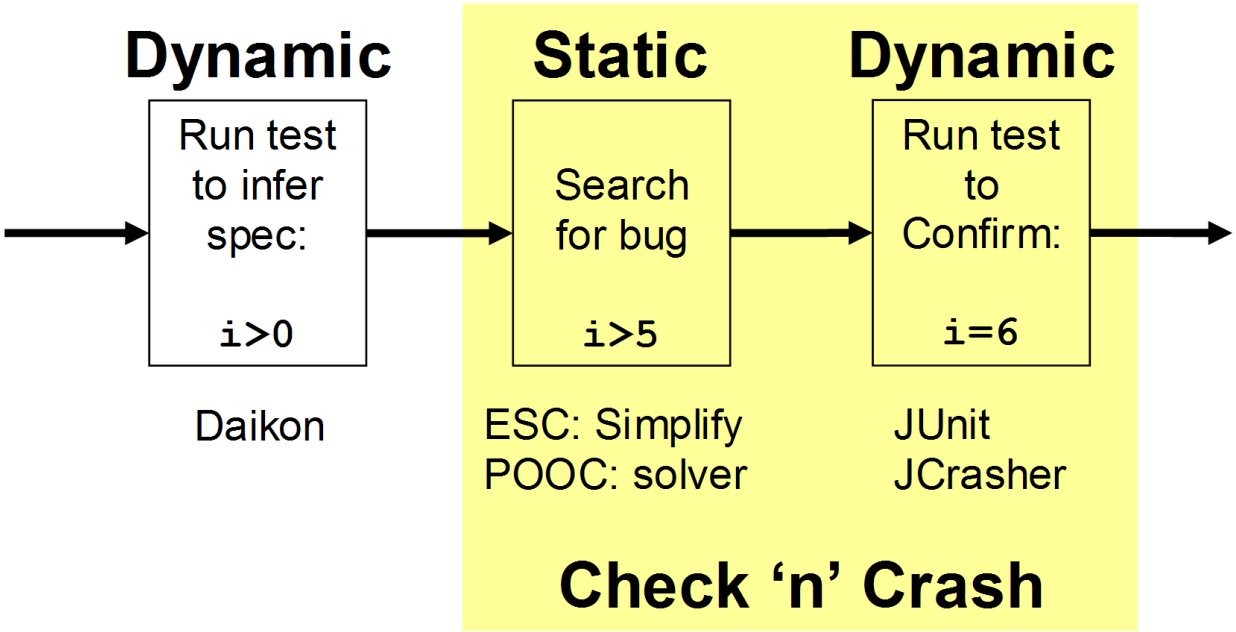
2k non-comment source statements

Used 603 of its unit test cases

Daikon produced 1.5 MB compressed invariants

`http://groovy.codehaus.org/`

Check 'n' Crash (SD)



More precise than Check 'n' Crash (SD)

JBoss JMS: Check 'n' Crash reports false positive warning of `NegativeArraySizeException`

```
public void setBytes(String name, byte[] value,
  int offset, int length) throws JMSException
{
  byte[] bytes = new byte[length];
  //..
}
```

Used test case that calls `setBytes` three times

Daikon infers

```
requires length == daikon.Quant.size(value)
```

DSD-Crasher suppresses false positive

More precise than Check 'n' Crash (SD)

Groovy experiments

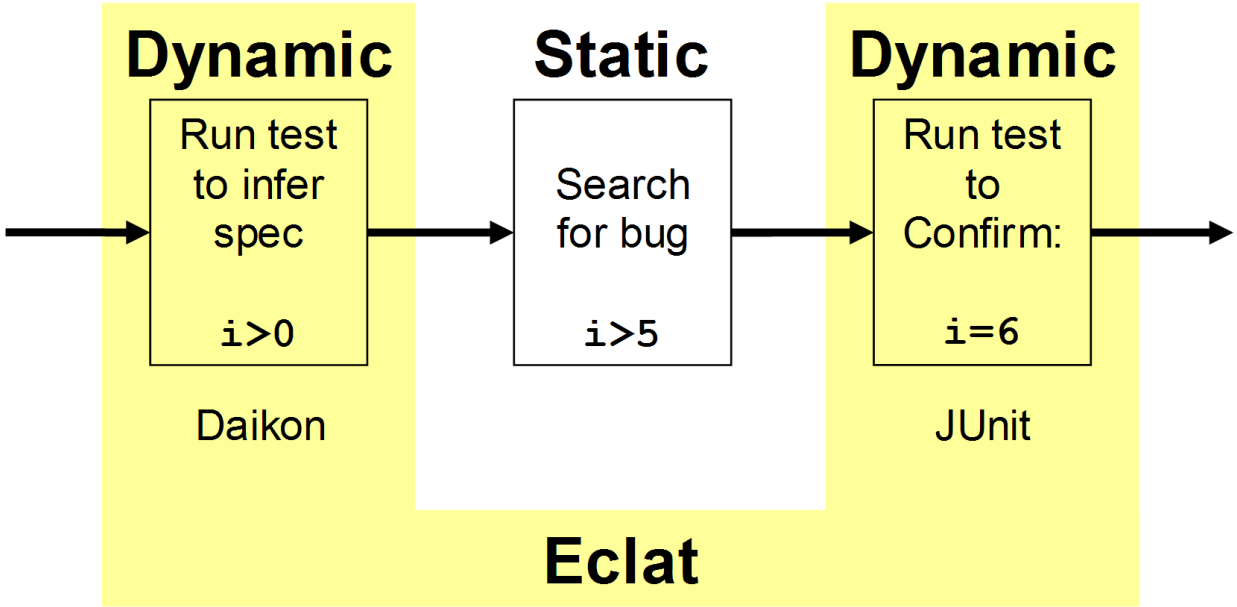
Tool	Runtime [min:s]	Exception reports
CnC-relaxed	10:43	19
DSD-Crasher	30:32	11

Using Daikon-inferred invariants

12..18: ESC could statically rule out false positives

19: ESC produces more complicated error condition, threw off constraint solver (easy to fix)

Eclat (DD)



Does more than we look for
Static analysis mostly random

Deeper than Eclat (DD)

JBoss JMS: `ClassCastException` reports

Tool	Reports	Runtime [min:s]
Eclat-default	0	1:20
Eclat-hybrid, 4 rounds	0	2:37
Eclat-hybrid, 5 rounds	0	3:34
Eclat-hybrid, 10 rounds	0	16:39
Eclat-exhaustive, 500 s timeout	0	13:39
Eclat-exhaustive, 1000 s timeout	0	28:29
Eclat-exhaustive, 1500 s timeout	0	44:29
Eclat-exhaustive, 1750 s timeout	0	1:25:44
DSD-Crasher	3	1:59

Deeper than Eclat (DD)

JBoss JMS example

```
public static byte[] getBytes(Object value)
  throws MessageFormatException
{
  if (value == null) {return null;}
  else if (value instanceof Byte[]) {
    return (byte[]) value;
  } //..
}
```

Deeper than Eclat (DD)

Groovy experiments

Tool	Exception reports	Runtime [min:s]
Eclat-default	0	7:01
Eclat-hybrid, 4 rounds	0	8:24
Eclat-exhaustive, 2 rounds	2	10:02
Eclat-exhaustive, 500 s timeout	2	16:42
Eclat-exhaustive, 1200 s timeout	2	33:17
DSD-Crasher	4	30:32

DSD-Crasher benefits from deeper static analysis

Pointers

JCrasher: An automatic robustness tester for Java
Software–Practice & Experience, 34(11):1025-1050, September 2004
`http://www.cc.gatech.edu/jcrasher/`

Check 'n' Crash: Combining static checking and testing
27th International Conference on Software Engineering (ICSE 2005), pp. 422–431, May 2005
`http://www.cc.gatech.edu/cnc/`

Dynamically discovering likely interface invariants
28th International Conference on Software Engineering (ICSE 2006), Emerging Results Track, May 2006

Download DSD-Crasher
`http://www.cc.gatech.edu/cnc/`

Conclusions