Mining Programs
LASER 2006

Andreas Zeller
Saarland University
zeller@acm.org

Laser 2006 – Summer School on Software Engineering

Isolating Causes
- Actual world
- Alternate world
- Mixed world
- Test

Locating Defects
- Failing run
- Passing run

Detecting Anomalies
- Properties
- Differences correlate with failure

Mining Models
- Failing Tests
- Passing Tests
- Differences correlate with failures

An F-16
(northern hemisphere)
An F–16
(southern hemisphere)

F–16 Landing Gear

The First Bug
September 9, 1947
More Bugs

Facts on Debugging

- Software bugs cost ~60 bln US$/yr in US
- Improvements could reduce cost by 30%
- Validation (including debugging) can easily take up to 50-75% of the development time
- When debugging, some people are three times as efficient than others
A Sample Program

$ sample 9 8 7
Output: 7 8 9

$ sample 11 14
Output: 0 11

How to Debug
(Sommerville 2004)

Locate error → Design error repair → Repair error → Re-test program

The Traffic Principle

T rack the problem
R eproduce
A utomate
F ind Origins
F ocus
I solate
C orrect
The Traffic Principle

Tack the problem
R eproduce
A utomate
F ind Origins
F ocus
I solate
C orrect

From Defect to Failure

1. The programmer creates a defect – an error in the code.
2. When executed, the defect creates an infection – an error in the state.
3. The infection propagates.
4. The infection causes a failure.

This infection chain must be traced back – and broken.

Search in Space + Time

variables

Search in Space + Time
The Defect

variables

time

Search in Time

• In shell_sort, the state must have become infected.

• Basic idea: Observe a transition from sane to infected.

A Program State
Isolating Causes

Locating Defects

Alternate world
Actual world

Mixed world

Isolating Causes

Detecting Anomalies

Mining Models

What is the cause of this failure?

Causality

What does it mean that “A causes B”?

- Counterfactual approach: If A had not occurred, then B would not have occurred
- Statistical approach: A results in an increase in the probability of B
Counterfactual Causality

Actual world

Effect does not occur

Effect does occur

Alternate world

Causes

bug.c

double bug(double z[], int n) {
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}

empty.c

double bug(double z[], int n) {
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
Causes as Differences

Actual world

empty.c: GCC works fine

bug.c: GCC crashes

Alternate world

Cause: bug.c

Actual Causes

“The” cause (actual cause) is a minimal difference

Actual cause

Isolating Causes

double bug(double z[], int n) {
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
Isolating Causes

double bug(double z[], int n) {
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}

Actual cause narrowed down
Isolating Causes

```c
double bug(double z[], int n) {
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

Isolating Causes

```c
double bug(double z[], int n) {
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

Isolating Causes

```c
double bug(double z[], int n) {
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

Actual cause of the GCC crash
Isolating Causes

Actual world

Alternate world

Test

Mixed world

✔

✘

Test

✔

✘

Test

"+ 1.0"

Input

Failure Cause

✔

✘

✔

✘

✔

✔

Failure Cause
Configuration

Circumstance $\delta$

All circumstances
$C = \{\delta_1, \delta_2, \ldots\}$

Configuration $c \subseteq C$
$c = \{\delta_1, \delta_2, \ldots \delta_n\}$

Tests

Testing function
$test(c) \in \{\text{✔, ✘, ?}\}$

Initial configurations
$test(c_{\text{✔}}) = \text{✔}$
$test(c_{\text{✘}}) = \text{✘}$

Minimal Difference

Goal: Subsets $c_\text{✔}$ and $c_\text{✘}$
$\emptyset = c_{\text{✔}} \subseteq c_{\text{✘}} \subseteq c_\text{✘} \subseteq c_{\text{x}}$

Difference
$\Delta = c_\text{✘} \setminus c_\text{✔}$

Difference is 1-minimal
$\forall \delta_i \in \Delta \cdot test(c_\text{✘} \cup \{\delta_i\}) \neq \text{✔} \land test(c_\text{✔} \setminus \{\delta_i\}) \neq \text{✘}$
Algorithm Sketch

- Extend \( dd_{\text{min}} \) such that it works on two sets at a time – \( c_x' \) and \( c_x'' \).
- Compute subsets
  \[ \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n = \Delta = c_x' \setminus c_x'' \]
- For each subset, test
  - the addition \( c_x' \cup \Delta_i \)
  - the removal \( c_x' \setminus \Delta_i \)

Test Outcomes

| \( \text{test}(c_x' \setminus \Delta_i) \) | \( c_x':=c_x' \setminus \Delta_i \) | \( \checkmark \) |
| \( \text{test}(c_x' \cup \Delta_i) \) | \( c_x':=c_x' \cup \Delta_i \) | \( \checkmark \) |
| otherwise | increase granularity |

most valuable outcomes

\[ dd(c_x, c_x) = dd(c_x, c_x, 2) \]
\[ dd(c_x', c_x', n) = \]
\[
\begin{cases}
(c_x', c_x') & \text{if } |\Delta| = 1 \\
\text{if } \exists i \in \{1..n\} \cdot \text{test}(c_x' \setminus \Delta_i) = \checkmark \\
\text{if } \exists i \in \{1..n\} \cdot \text{test}(c_x' \cup \Delta_i) = \checkmark \\
\text{else if } \exists i \in \{1..n\} \cdot \text{test}(c_x' \cup \Delta_i) = \times \\
\text{else if } \exists i \in \{1..n\} \cdot \text{test}(c_x' \setminus \Delta_i) = \checkmark \\
\text{else if } \exists i \in \{1..n\} \cdot \text{test}(c_x' \setminus \Delta_i) = \times \\
\text{else if } n < |\Delta| \text{ (“increase granularity”)} \\
\text{otherwise}
\end{cases}
\]

dd in a Nutshell

\[ dd(c_x, c_x) = (c_x', c_x') \quad \Delta = c_x' \setminus c_x'' \text{ is } 1\text{-minimal} \]

Laser 2006 – Summer School on Software Engineering
def dd(c_pass, c_fail):
    n = 2
    while 1:
        delta = listminus(c_fail, c_pass)
        deltas = split(delta, n); offset = 0; j = 0
        while j < n:
            i = (j + offset) % n
            next_c_pass = listunion(c_pass, deltas[i])
            next_c_fail = listminus(c_fail, deltas[i])
            if test(next_c_fail) == FAIL and n == 2:
                c_fail = next_c_fail; n = 2; offset = 0; break
            elif test(next_c_fail) == PASS:
                c_pass = next_c_fail; n = 2; offset = 0; break
            elif test(next_c_pass) == FAIL:
                c.fail = next_c_pass; n = 2; offset = 0; break
            elif test(next_c_pass) == PASS:
                c_pass = next_c_pass; n = max(n - 1, 2); offset = i; break
            else:
                j = j + 1
                if j >= n:
                    if n >= len(delta):
                        return (delta, c_pass, c_fail)
                    else:
                        n = min(len(delta), n * 2)
                        break

Properties

number of tests $t$ – worst case:

$$t = |\Delta|^2 + 7|\Delta|$$

where $\Delta = c_x \setminus c_y$

number of tests $t$ – best case
(no unresolved outcomes):

$$t \leq \log_2(\Delta)$$

size of difference – no unresolved outcomes

$$|c'_x \setminus c'_y| = 1$$

Applications

<table>
<thead>
<tr>
<th>Input</th>
<th>Code Changes</th>
<th>Schedules</th>
</tr>
</thead>
</table>

Laser 2006 – Summer School on Software Engineering
Isolating Input

Failure Cause

Isolation: 5 tests
Simplification: 48 tests

Code Changes

From: Brian Kahne <bkahne@ibmoto.com>
To: DDD Bug Report Address <bug-ddd@gnu.org>
Subject: Problem with DDD and GDB 4.17

When using DDD with GDB 4.16, the run command correctly uses any prior command-line arguments, or the value of "set args". However, when I switched to GDB 4.17, this no longer worked: If I entered a run command in the console window, the prior command-line options would be lost. [...]
Version Differences

New version

Program works

Old version

Program fails

Causes

What was Changed

$ diff -r gdb-4.16 gdb-4.17
diff -r gdb-4.16/COPYING gdb-4.17/COPYING
5c5
< 675 Mass Ave, Cambridge, MA 02139, USA
---
> 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
282c282
< Appendix: How to Apply These Terms to Your New Programs
---
> How to Apply These Terms to Your New Programs

...and so on for 178,200 lines (8,721 locations)

Challenges

• Granularity – within some large change, only a few lines may be relevant
• Interference – some (later) changes rely on other (earlier) changes
• Inconsistency – some changes may have to be combined to produce testable code

Delta debugging handles all this
General Plan

- Decompose diff into changes per location (= 8,721 individual changes)
- Apply subset of changes, using PATCH
- Reconstruct GDB; build errors mean unresolved test outcome
- Test GDB and return outcome

Isolating Changes

- Result after 98 tests (= 1 hour)

The Failure Cause

diff -r gdb-4.16/gdb/infcmd.c gdb-4.17/gdb/infcmd.c
1239c1278
< "Set arguments to give program being debugged when it is started."\n---
> "Set argument list to give program being debugged when it is started."\n
- Documentation becomes GDB output
- DDD expects Arguments, but GDB outputs Argument list
## Optimizations

- History – group changes by creation time
- Reconstruction – cache several builds
- Grouping – according to scope
- Failure Resolution – scan error messages for possibly missing changes

---

## Thread Schedules

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>open(&quot;.htpasswd&quot;)</td>
<td>open(&quot;.htpasswd&quot;)</td>
</tr>
<tr>
<td></td>
<td>read(...)</td>
<td>read(...)</td>
</tr>
<tr>
<td></td>
<td>modify(...)</td>
<td>modify(...)</td>
</tr>
<tr>
<td></td>
<td>write(...)</td>
<td>write(...)</td>
</tr>
<tr>
<td></td>
<td>close(...)</td>
<td>close(...)</td>
</tr>
</tbody>
</table>

- A’s updates get lost!
**Record + Replay**

**Schedules as Input**

**Finding Differences**

- We start with runs ✓ and X
- We determine the differences $\Delta_i$ between thread switches $t_i$:
  - $t_1$ occurs in ✓ at “time” 254
  - $t_1$ occurs in X at “time” 278
  - The difference $\Delta_1 = |278 - 254|$ induces a statement interval: the code executed between “time” 254 and 278
  - Same applies to $t_2$, $t_3$, etc.

The schedule difference causes the failure!
Isolating Relevant Differences

We use Delta Debugging to isolate the relevant differences. Delta Debugging applies subsets of differences to

- The entire difference \( \Delta_1 \) is applied
- Half of the difference \( \Delta_2 \) is applied
- \( \Delta_3 \) is not applied at all

DEJAVU executes the debuggee under this generated schedule; an automated test checks if the failure occurs.
The Failure Cause

```java
public class Scene {
    private static int ScenesLoaded = 0;
    (more methods...)

    private int LoadScene(String filename) {
        int OldScenesLoaded = ScenesLoaded;
        (more initializations...)

        infile = new DataInputStream(…);
        (more code...)

        ScenesLoaded = OldScenesLoaded + 1;
        System.out.println("" + ScenesLoaded + " scenes loaded.");
        ...
        ...
    }
```

Detecting Anomalies

- Properties
- Differences correlate with failure

Mining Models

- isEmpty
- add()
- clear()
- isEmpty()
- <init>

Failing Tests

- add() isEmpty add() clear() clear() isEmpty() <init>

Passing Tests

- Differences correlate with failures
Isolating Causes

Actual world

Alternate world

Mixed world

Locating Defects

Failing run

Passing run

Detecting Anomalies

Mining Models

What is the cause of this failure?

Causes in State

Infected state

Sane state

• 41,000 variables

• 42,000 references

• 1 reference is wrong.

• Which one?

The difference causes GCC to crash!
Search in Time

Failing run

Passing run

\[ \text{link} \rightarrow \text{fld[0].rtx} \rightarrow \text{fld[0].rtx} = \text{link} \]

Why Transitions?

- Each failure cause in the program state is caused by some statement
- These statements are executed at cause transitions
- Cause transitions thus are statements that cause the failure
All GCC Transitions

<table>
<thead>
<tr>
<th>Location</th>
<th>New cause at transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Start&gt;</td>
<td>argv[3]</td>
</tr>
<tr>
<td>toplev.c:4755</td>
<td>name</td>
</tr>
<tr>
<td>toplev.c:2909</td>
<td>dump_base_name</td>
</tr>
<tr>
<td>c-lex.c:187</td>
<td>finpbuf= _IO_buf_base</td>
</tr>
<tr>
<td>c-lex.c:1213</td>
<td>nextchar</td>
</tr>
<tr>
<td>c-lex.c:1213</td>
<td>yysa[41]</td>
</tr>
<tr>
<td>c-typeck.c:3615</td>
<td>yysa[42]</td>
</tr>
<tr>
<td>c-lex.c:1213</td>
<td>last_insn=fld[1].rtx=...=fld[1].rtx.code</td>
</tr>
<tr>
<td>c-decl.c:1213</td>
<td>sequence_result[2]=...=fld[1].rtx.code</td>
</tr>
<tr>
<td>combine.c:4271</td>
<td>x=fld[0].rtx=fld[0].rtx</td>
</tr>
</tbody>
</table>

combine.c

if (GET_CODE (XEXP (x, 0)) == PLUS {
    x = apply_distributive_law
    (gen_binary (PLUS, mode,
                 gen_binary (MULT, mode,
                             XEXP (XEXP (x, 0), 0),
                             XEXP (x, 1)),
                 XEXP (XEXP (x, 0), 1),
                 XEXP (x, 1))));

if (GET_CODE (x) != MULT)
    return x;
}

Implementations

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Web service + command line</td>
<td>24 months</td>
</tr>
<tr>
<td></td>
<td>Eclipse plug-in</td>
<td>12 months</td>
</tr>
<tr>
<td></td>
<td>Module</td>
<td>2 days</td>
</tr>
</tbody>
</table>
Isolating Causes

Actual world

Alternate world

Mixed world

Locating Defects

Failing run

Passing run

Detecting Anomalies

Properties

Properties

Differences correlate with failure

Mining Models

Failing Tests

Passing Tests

Differences correlate with failures

Detecting Anomalies

Run

Run

Properties

Properties

Differences correlate with failure

Mining Models

¬isEmpty

add()

remove()

isEmpty()

<init>

Failing Tests

add()

¬isEmpty

add()

clear()

clear()

isEmpty()

<init>

Passing Tests

Differences correlate with failures

Locating Defects

Failing run

Passing run

Differences correlate with failures
Properties

Data properties that hold in all runs:
- “At f(), x is odd”
- “0 ≤ x ≤ 10 during the run”

Code properties that hold in all runs:
- “f() is always executed”
- “After open(), we eventually have close()”

Techniques

Dynamic Invariants  Value Ranges  Sampled Values
Dynamic Invariants

At f(), x is odd

At f(), x = 2

Daikon

• Determines invariants from program runs
• Written by Michael Ernst et al. (1998–)
• C++, Java, Lisp, and other languages
• analyzed up to 13,000 lines of code

Daikon

public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}

• Run with 100 randomly generated arrays of length 7–13

Precondition
n == size(b[])  
b != null  
n <= 13  
n >= 7

Postcondition
b[] = orig(b[])  
return == sum(b)
Daikon

- get trace
- filter invariants
- report results

Postcondition
b[i] = orig(b[i])
return == sum(b)

Getting the Trace

- Records all variable values at all function entries and exits
- Uses VALGRIND to create the trace

Filtering Invariants

- Daikon has a library of invariant patterns over variables and constants
- Only matching patterns are preserved
**Method Specifications**

*using primitive data*

<table>
<thead>
<tr>
<th>(x = 6)</th>
<th>(x \in {2, 5, \ldots, 30})</th>
<th>(x &lt; y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y = 5x + 10)</td>
<td>(z = 4x + 12y + 3)</td>
<td>(z = fn(x, y))</td>
</tr>
</tbody>
</table>

*using composite data*

| A subseq B | \(x \in A\) | sorted(A) |

checked at method entry + exit

**Object Invariants**

| string.content[string.length] = '\0' |
| node.left.value \(\leq\) node.right.value |
| this.next.last = this |

checked at entry + exit of public methods

**Matching Invariants**

```java
public int ex1511(int[] b, int n) {
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

**Pattern**

A == B

**Variables**

s size(b[]) sum(b[]) n orig(n) return ...
### Matching Invariants

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>n</th>
<th>size(b[])</th>
<th>sum(b[])</th>
<th>orig(n)</th>
<th>ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>size(b[])</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>sum(b[])</td>
<td></td>
<td></td>
<td></td>
<td>x x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>orig(n)</td>
<td></td>
<td></td>
<td></td>
<td>x x</td>
<td>x x</td>
<td>x x</td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

**Pattern**

A == B

**Variables**

s size(b[]) sum(b[]) orig(n) n return ...
Matching Invariants

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>n</th>
<th>size(b[])</th>
<th>sum(b[])</th>
<th>orig(n)</th>
<th>ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>size(b[])</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>sum(b[])</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>orig(n)</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

```
public int ex1511(int[] b, int n) {
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

Enhancing Relevance

- Handle polymorphic variables
- Check for derived values
- Eliminate redundant invariants
- Set statistical threshold for relevance
- Verify correctness with static analysis
Daikon Discussed

- As long as some property can be observed, it can be added as a pattern
- Pattern vocabulary determines the invariants that can be found ("sum()", etc.)
- Checking all patterns (and combinations!) is expensive
- Trivial invariants must be eliminated

Techniques

<table>
<thead>
<tr>
<th>Dynamic Invariants</th>
<th>Value Ranges</th>
<th>Sampled Values</th>
</tr>
</thead>
</table>

Dynamic Invariants

- Can we check this on the fly?

<table>
<thead>
<tr>
<th>Invariant</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>At f(), x is odd</td>
<td>At f(), x = 2</td>
</tr>
</tbody>
</table>
Diduce

- Determines *invariants* and *violations*
- Written by Sudheendra Hangal and Monica Lam (2001)
- Java bytecode
- analyzed > 30,000 lines of code

---

**Training Mode**

- Start with empty set of invariants
- Adjust invariants according to values found during run
Invariants in Diduce

For each variable, Diduce has a pair \((V, M)\)

- \(V\) = initial value of variable
- \(M\) = range of values: i-th bit of \(M\) is cleared if value change in i-th bit was observed
- With each assignment of a new value \(W\), 
  \(M\) is updated to \(M := M \land \neg (W \land V)\)
- Differences are stored in same format

Training Example

<table>
<thead>
<tr>
<th>Code</th>
<th>i</th>
<th>Values</th>
<th>Differences</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i = 10)</td>
<td>1010</td>
<td>1010 1111</td>
<td>–</td>
<td>(i = 10)</td>
</tr>
<tr>
<td>(i += 1)</td>
<td>1011</td>
<td>1010 1110</td>
<td>0001 1111</td>
<td>(10 \leq i \leq 11 \land</td>
</tr>
<tr>
<td>(i += 1)</td>
<td>1100</td>
<td>1010 1000</td>
<td>0001 1111</td>
<td>(8 \leq i \leq 15 \land</td>
</tr>
<tr>
<td>(i += 1)</td>
<td>1101</td>
<td>1010 1000</td>
<td>0001 1111</td>
<td>(8 \leq i \leq 15 \land</td>
</tr>
<tr>
<td>(i += 2)</td>
<td>1111</td>
<td>1010 1000</td>
<td>0001 1101</td>
<td>(8 \leq i \leq 15 \land</td>
</tr>
</tbody>
</table>

During checking, clearing an M-bit is an anomaly

Diduce vs. Daikon

- Less space and time requirements
- Invariants are computed on the fly
- Smaller set of invariants
- Less precise invariants
**Techniques**

- Dynamic Invariants
- Value Ranges
- Sampled Values

**Detecting Anomalies**

- How do we collect data in the field?
- Differences correlate with failure

**Liblit’s Sampling**

- We want properties of runs in the field
- Collecting all this data is too expensive
- Would a sample suffice?
- Sampling experiment by Liblit et al. (2003)
Return Values

• Hypothesis: function return values correlate with failure or success
• Classified into positive / zero / negative

CCRYPT fails

• CCRYPT is an interactive encryption tool
• When CCRYPT asks user for information before overwriting a file, and user responds with EOF, CCRYPT crashes
• 3,000 random runs
• Of 1,170 predicates, only `file_exists() > 0` and `xreadline() == 0` correlate with failure

Liblit’s Sampling

• Can we apply this technique to remote runs, too?
• 1 out of 1,000 return values was sampled
• Performance loss <4%
Web Services

- Sampling is first choice for web services
- Have 1 out of 100 users run an instrumented version of the web service
- Correlate instrumentation data with failure
- After sufficient number of runs, we can automatically identify the anomaly

Techniques

<table>
<thead>
<tr>
<th>Dynamic Invariants</th>
<th>Value Ranges</th>
<th>Sampled Values</th>
</tr>
</thead>
</table>

After 3,000 runs, only five predicates are left that correlate with failure.
Anomalies and Causes

- An anomaly is not a cause, but a correlation
- Although correlation ≠ causation, anomalies can be excellent hints
- Future belongs to those who exploit
- Correlations in multiple runs
- Causation in experiments

Locating Defects

- NN (Renieris + Reiss, ASE 2003)
- CT (Cleve + Zeller, ICSE 2005)
- SD (Liblit et al., PLDI 2005)
- SOBER (Liu et al, ESEC 2005)

Results obtained from Siemens test suite; cannot be generalized

Differences correlate with failures
Isolating Causes

Alternate world
Actual world
Isolating Causes

Locating Defects

Mixed world

Detecting Anomalies

Properties

Differences correlate with failure

Mining Models

¬isEmpty
add()
remove()
isEmpty()
<init>

Failing Tests

Properties

Differences correlate with failures

Mining Models

¬isEmpty
add()
clear()
clear()
isEmpty()
<init>

Passing Tests

Differences correlate with failures

Object States

add(1)
isEmpty()
remove(1)
firstElement()

Mutators
change state

Inspectors
return state

Use static analysis to differentiate
Building Models

- After each mutator call, we invoke the inspectors to extract object state
- States form finite state machine
Correlations

Differences correlate with failures

Christian Lindig
Valentin Dallmeier
Laser 2006 – Summer School on Software Engineering

**Isolating Causes**

- Actual world
- Alternate world
- Mixed world

**Locating Defects**

- Failing run
- Passing run

**Detecting Anomalies**

- Properties
- Differences correlate with failure

**Mining Models**

- Failing Tests
- Passing Tests

Differences correlate with failures

---

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit http://creativecommons.org/licenses/by/1.0 or send a letter to Creative Commons, 559 Abbott Way, Stanford, California 94305, USA.