Mining Anomalies
Andrzej Wasylikowski

Why Mine Anomalies?

- How can we make programs more reliable?
- Testing, code inspection, etc.
- Mining anomalies, etc.
  - In general: automatic defect detection

Overview

Automatic Defect Detection

Rule-based Techniques

Specification-checking Techniques

Mining-based Techniques
  - Mining Repositories
  - Mining Traces
  - Mining Source Code
Overview

Automatic Defect Detection
  - Rule-based Techniques
  - Specification-checking Techniques
  - Mining-based Techniques
    - Mining Repositories
    - Mining Traces
    - Mining Source Code

FindBugs

Program \(\rightarrow\) FindBugs \(\rightarrow\) Violations
  - Bug Patterns

FindBugs’s Bug Patterns
  - Equal Objects Must Have Equal Hashcodes
  - Static Field Modifiable By Untrusted Code
  - Null Pointer Dereference
  - Return Value Should Be Checked
  - …

Rule-based Techniques

- Fixed “bug patterns” to check against
- Pros: Fully automatic, scalable
- Cons: Limited to occurrences of “bug patterns”

Can we add our own rules?

Overview

Automatic Defect Detection
  - Rule-based Techniques
  - Specification-checking Techniques
  - Mining-based Techniques
    - Mining Repositories
    - Mining Traces
    - Mining Source Code
Overview

Automatic Defect Detection
- Rule-based Techniques
- Specification-checking Techniques

Mining-based Techniques
- Mining Repositories
- Mining Traces
- Mining Source Code

Specification-checking

Program $\rightarrow$ Verifier $\rightarrow$ Violations

Specification

Typestate: java.net.Socket

Typestate Verification

Socket s1 = new Socket();
s1.connect (...);
inp = s1.getInputStream();
data = readData(inp);
s1.close();

Specification-checking Techniques

- Use external specification to check against
- Pros: adaptable, very precise
- Cons: need specification, may have scalability problems

Writing specifications is very difficult!
Overview

Automatic Defect Detection

Rule-based Techniques

Specification-checking Techniques

Mining-based Techniques
- Mining Repositories
- Mining Traces
- Mining Source Code

Mining Source Code

- Code is typically correct
- Deviant behavior can point to a bug
- We can learn what is common behavior…
- …and detect uncommon behavior

**ECC**

- **Program**
- **ECC**
- **Rules**
  - lock() is typically paired with unlock()
- **Violations**
  - In foo, lock() is not paired with unlock()

**Rule templates**

<a> must be paired with <b>

**ECC: Example**

```c
lock l;        // Lock
int a, b;      // Variables potentially // protected by l
void foo () {
    lock (l); // Enter critical section
    a = a + b; // MAY: a, b protected by l
    b = b + 1; // MUST: b not protected by l
}
void bar () {
    lock (l); // MAY: a protected by l
    a = a + 1; // MUST: a not protected by l
}
void baz () {
    a = a + 1; // MAY: a protected by l
    unlock (l); // MUST: b not protected by l
    b = b - 1;  // MUST: b not protected by l
    a = a / 5;  // MUST: a not protected by l
}
```

**Rule:**

lock l protects variable a

**Rule template:**

lock <l> protects variable <v>

**Rule:**

lock l protects variable b

**Violation:**

a is not protected by l in baz

**ECC: Example**

```c
lock l;        // Lock
int a, b;      // Variables potentially // protected by l
void foo () {
    lock (l); // Enter critical section
    a = a + b; // MAY: a, b protected by l
    unlock (l); // Exit critical section
    b = b + 1; // MUST: b not protected by l
}
void bar () {
    lock (l);
    a = a + 1;  // MAY: a protected by l
    unlock (l);
}
void baz () {
    a = a + 1;  // MAY: a protected by l
    unlock (l);
    b = b - 1;  // MUST: b not protected by l
    a = a / 5;  // MUST: a not protected by l
}
```

**Rule:**

lock l protects variable a

**Rule template:**

lock <l> protects variable <v>

**Rule:**

lock l protects variable b

**Violation:**

a is not protected by l in baz
ECC: Example

```c
lock l;        // Lock
int a, b;      // Variables potentially
// protected by l
void foo () {
    lock (l);   // Enter critical section
    a = a + b;  // MAY: a, b protected by l
    unlock (l); // Exit critical section
    b = b + 1;  // MUST: b not protected by l
}
void bar () {
    lock (l);
    a = a + 1;  // MAY: a protected by l
    unlock (l);
}
void baz () {
    a = a + 1;  // MAY: a protected by l
    unlock (l);
    b = b - 1;  // MUST: b not protected by l
    a = a / 5;  // MUST: a not protected by l
}
```

Rule template:
```
lock <l> protects variable <v>
```

Rule:
```
lock l protects variable a
```
Violation:
```
a is not protected by l in baz
```

Rule:
```
lock l protects variable b
```

Engler, Dawson, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring

ECC: Summary

- Mines rules based on templates
- Pros: fully automatic, project-specific
- Cons: templates are simple and have fixed size

Engler, Dawson, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring
Mines rules based on templates

Templates have a fixed number of slots.

Pros: fully automatic, project-specific

Cons: templates are simple and have fixed size

PR-Miner

static void
getRelationDescription (...) {
    HeapTuple relTup;
    ...
    relTup = SearchSysCache (...);
    if (!HeapTupleIsValid (relTup))
        elog (...);
    relForm = ...;
    ...
    ReleaseSysCache (relTup);
}
PR-Miner: Step 1

static void getRelationDescription (...) {
    HeapTuple relTup;
    ...
    relTup = SearchSysCache (...);
    if (!HeapTupleIsValid (relTup))
        elog (...);
    relForm = ...;
    ... ReleaseSysCache (relTup);
}


PR-Miner: Step 2

T: HeapTuple
F: SearchSysCache
F: HeapTupleIsValid
T: Form_pg_class
F: ReleaseSysCache
...

T: StringInfoData
T: HeapTuple
F: SearchSysCache
F: HeapTupleIsValid
F: elog
F: ReleaseSysCache
...

T: HeapTuple
F: SearchSysCache
F: Form_pg_class
...

Rule:
T: HeapTuple, F: SearchSysCache, and F: ReleaseSysCache typically come together

PR-Miner: Step 2

**Rule:**
T: HeapTuple
F: SearchSysCache
F: HeapTupleIsValid
T: Form_pg_class
F: ReleaseSysCache
...

**Violation:**
F: ReleaseSysCache is missing

T: StringInfoData
T: HeapTuple
F: SearchSysCache
F: NameStr
F: ReleaseSysCache
...

**Rule:**
T: HeapTuple
F: SearchSysCache
F: elog
F: ReleaseSysCache
...

**Violation:**
F: ReleaseSysCache is missing

---

PR-Miner: Summary

- Mines rules being sets of entities
- Pros: scalable, project-specific, flexible rule size
- Cons: no ordering of entities

---

**Ordering is not taken into account.**
Creating an Object Usage Model: Example 1

```java
public List getList(Set ps) {
    List l = new ArrayList();
    createList(this.cl, l);
    Iterator it = ps.iterator();
    while (it.hasNext()) {
        Property p = it.next();
        addProperty(p, l);
    }
    reapList(l);
    return l;
}
```
Creating an Object Usage Model: Example 2

```java
public List getList (Set ps) {
    List l = new ArrayList ();
    createList (this.cl, l);
    Iterator it = ps.iterator ();
    while (it.hasNext ()) {
        Property p = it.next ();
        addProperty (p, l);
    }
    reapList (l);
    return l;
}
```

Example OUMs: StringTokenizer

```
st.<init> ()
st.hasMoreTokens ()
st.nextToken ()
st.<init> ()
st.countTokens ()
st.nextToken ()
```
Extracting Temporal Properties

```
RETVALL: Set.iterator()
it.hasNext() it.next()
```

```
RETVALL: Set.iterator() < Iterator.hasNext() @ this
RETVALL: Set.iterator() < Iterator.next() @ this
Iterator.hasNext() @ this < Iterator.next() @ this
```

```
RETVALL: Set.iterator() < Iterator.hasNext() @ this
RETVALL: Set.iterator() < Iterator.next() @ this
Iterator.hasNext() @ this < Iterator.next() @ this
```

```
RETVALL: Set.iterator() < Iterator.hasNext() @ this
RETVALL: Set.iterator() < Iterator.next() @ this
Iterator.hasNext() @ this < Iterator.next() @ this
```

```
st.<init>()
st.countTokens()
st.nextToken()
```

```
StringTokenizer.<init>() @ this < StringTokenizer.countTokens() @ this
StringTokenizer.<init>() @ this < StringTokenizer.nextToken() @ this
StringTokenizer.countTokens() @ this < StringTokenizer.nextToken() @ this
StringTokenizer.nextToken() @ this < StringTokenizer.nextToken() @ this
```

Extracting Temporal Properties: Summary

```
Object Usage Model #1

Temporal Properties #1
```

```
Object Usage Model #2

Temporal Properties #2
```

```
Object Usage Model #n

M's Temporal Properties
```

```
```

40
```

```
```

41
```

```
```

42
```
### Methods vs. Temporal Properties

<table>
<thead>
<tr>
<th>Temporal Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>a&lt;b  c&lt;d  a&lt;c  d&lt;a  ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th>Temporal Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td></td>
</tr>
<tr>
<td>M3</td>
<td></td>
</tr>
<tr>
<td>M4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

This forms a pattern.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Temporal Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td></td>
</tr>
<tr>
<td>M3</td>
<td></td>
</tr>
<tr>
<td>M4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Another pattern.
Methods vs. Temporal Properties

Methods

<table>
<thead>
<tr>
<th>Temporal Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
</tr>
</tbody>
</table>

Yet another pattern

Detecting Violations

Example Violation (1)

```java
private boolean verifyNIAP (...) {
    Iterator iter = ...;
    while (iter.hasNext()) {
        ... = iter.next();
        ... = iter.next();
        return verifyNIAP (...); // should be fixed
    }
    return true;
}
```
Example Violation (2)

```java
public String getRetentionPolicy () {
    for (Iterator it = ...; it.hasNext();)
        ... = it.next();
    ... return retentionPolicy;
}
```

should be fixed

Example Violation (3)

```java
public void visitCALOAD (CALOAD o) {
    Type arrayref = stack().peek(1);
    Type index = stack().peek(0);
    indexOfInt(o, index);
    arrayrefOfArrayType(o, arrayref);
}
```

should check the elements’ type, too

JADET: Summary

- Mines rules being sets of temporal properties
- Pros: fully automatic, scalable, project specific
- Cons: quite complicated, many false positives
JADET: Summary

- Mines rules being sets of temporal properties
- Pros: fully automatic, scalable, project specific
- Cons: quite complicated, many false positives

All problems solved? Of course not!

Summary

- Three main approaches:
  - Rule-based Techniques
  - Specification-checking techniques
  - Mining-based techniques

Code-mining Techniques

- “Learn” rules from source code
- Rule violation = potential defect
- Can find project-specific bugs
- Many different rules types