Mining Anomalies

Andrzej Wasylkowski

1

Why Mine Anomalies?

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

2

Overview

Automatic Defect Detection

Rule-based Techniques

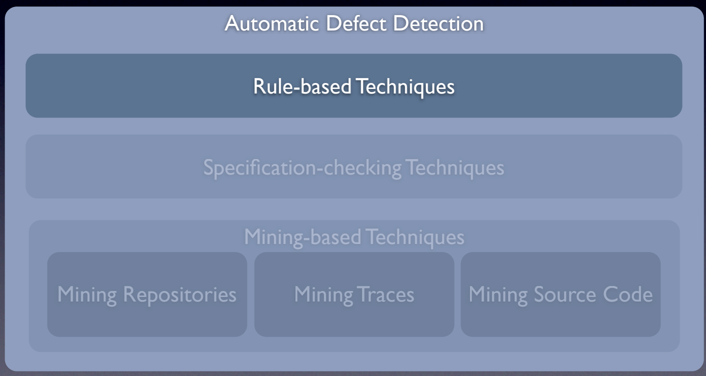
Specification-checking Techniques

Mining-based Techniques

Mining Repositories | Mining Traces | Mining Source Code

3

Overview



4

FindBugs

Hovemeyer, David, and William Pugh. 2004. Finding bugs is easy. *SIGPLAN Notices* 39, no. 12 (December): 92–106

5

FindBugs's Bug Patterns

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

Hovemeyer, David, and William Pugh. 2004. Finding bugs is easy. *SIGPLAN Notices* 39, no. 12 (December): 92–106

6

Rule-based Techniques

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

7

Rule-based Techniques

- Fixed "bug patterns" to check against
-
- "bug patterns"

Can we add our own rules?

8

Overview

Automatic Defect Detection

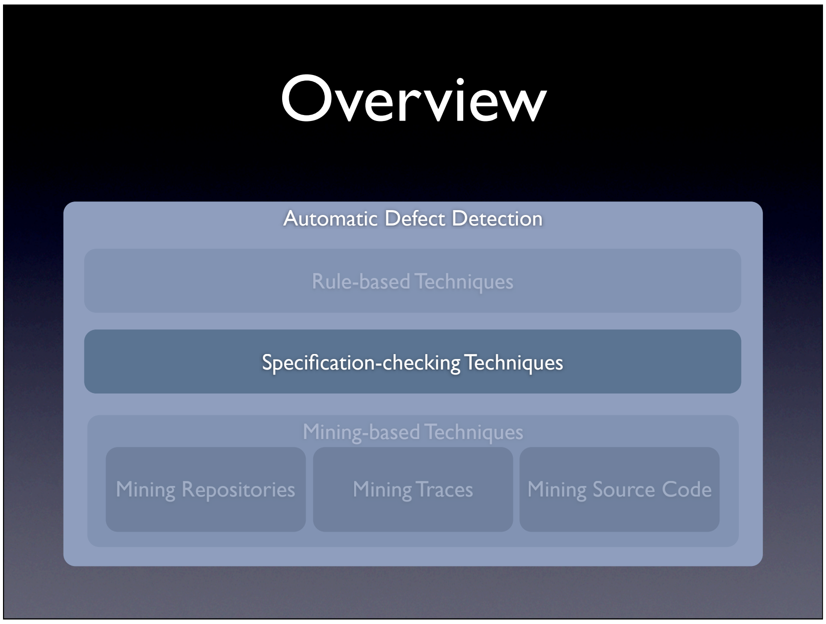
Rule-based Techniques

Specification-checking Techniques

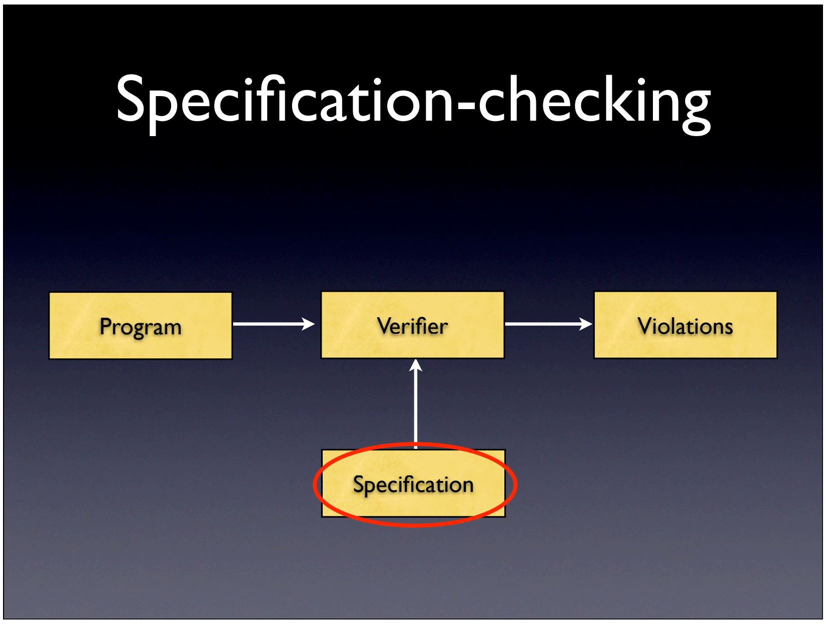
Mining-based Techniques

Mining Repositories | Mining Traces | Mining Source Code

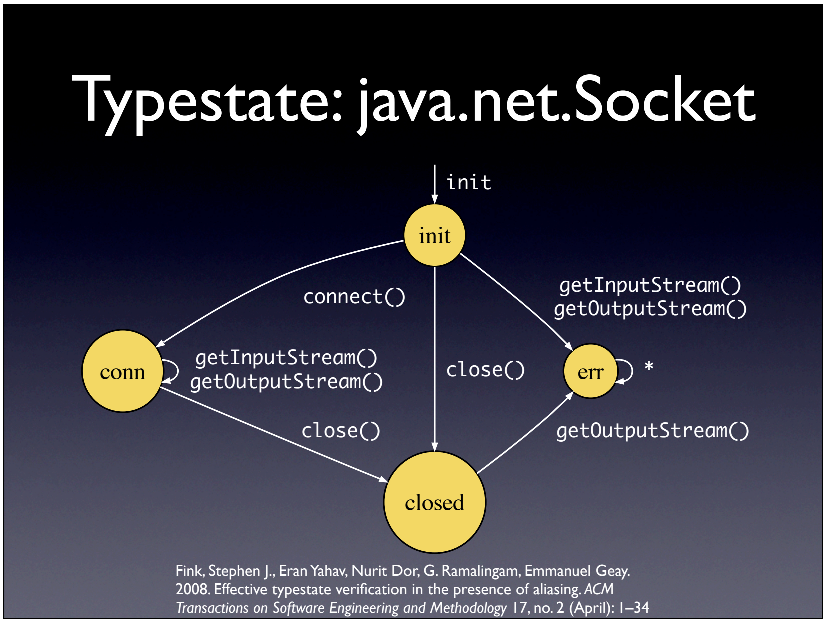
9


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

10

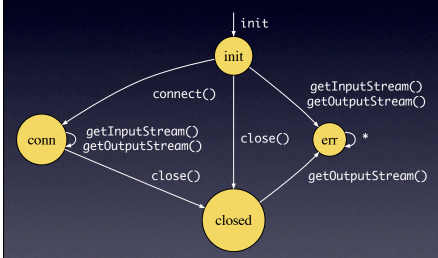

Specification-checking
Program
Verifier
Violations
Specification

11


Typestate: java.net.Socket
init
init
connect()
getInputStream()
getOutputStream()
conn
getInputStream()
getOutputStream()
close()
err
*
close()
getOutputStream()
closed
Fink, Stephen J., Eran Yahav, Nurit Dor, G. Ramalingam, Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. ACM Transactions on Software Engineering and Methodology 17, no. 2 (April): 1–34

12

Typestate Verification




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

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

Fink, Stephen J., Eran Yahav, Nurit Dor, G. Ramalingam, Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology* 17, no. 2 (April): 1–34

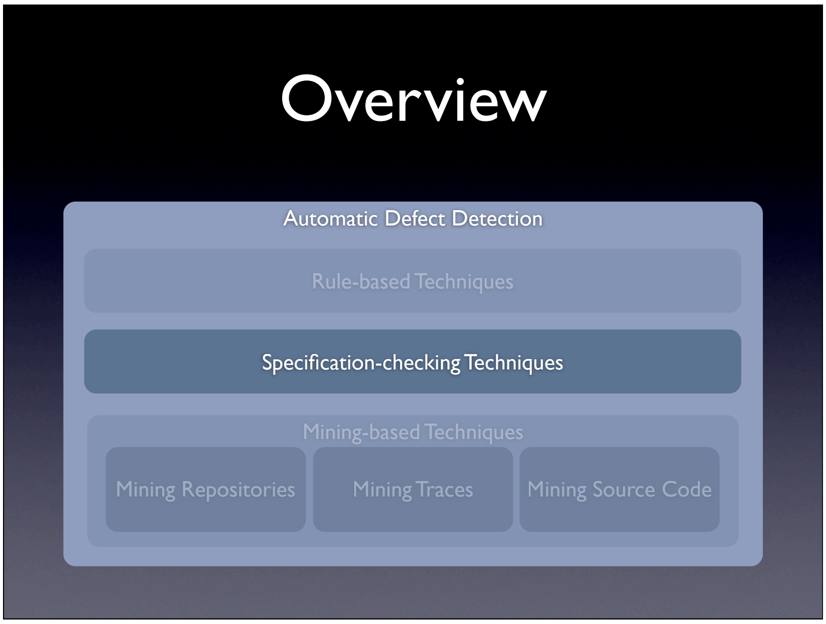
Specification-checking Techniques

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

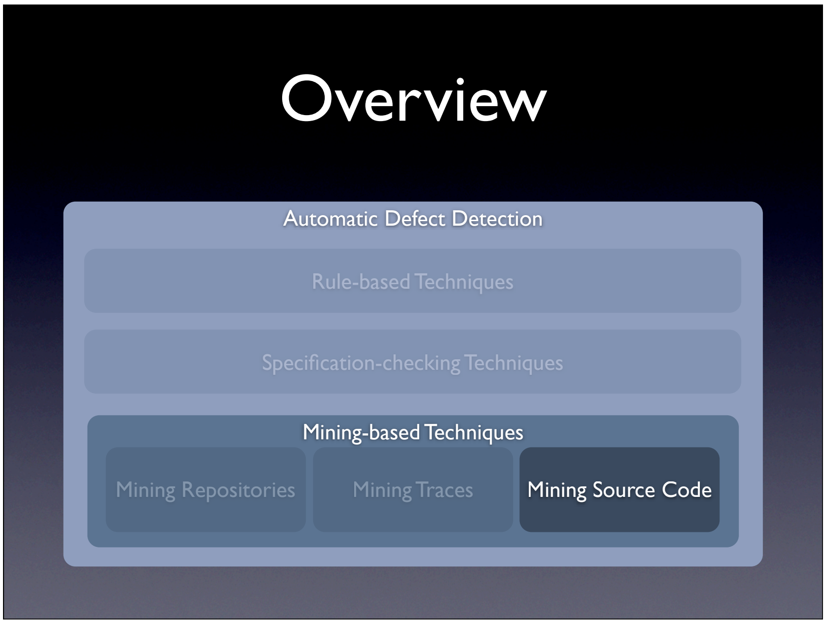
Specification-checking Techniques

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

Writing specifications is very difficult!



16



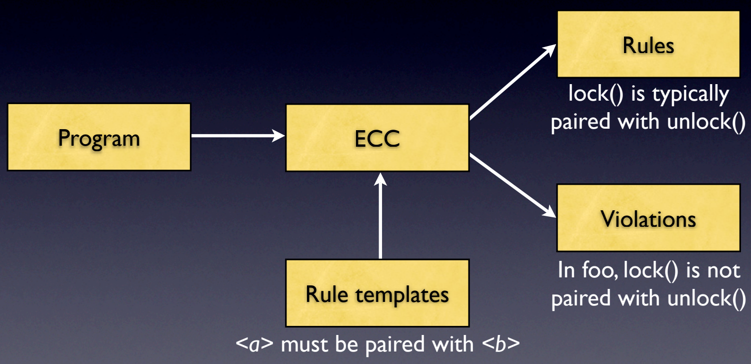
17

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

18

ECC



Engler, Dawson, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP 2001*, 57–72. New York, NY: ACM.

ECC: Example

```
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

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 errors in systems code. In *SOSP 2001*, 57–72. New York, NY: ACM.

ECC: Example

```
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 errors in systems code. In *SOSP 2001*, 57–72. New York, NY: ACM.

ECC: Example

```
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 errors in systems code. In *SOSP 2001*, 57–72. New York, NY: ACM.

ECC: Example

```
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 errors in systems code. In *SOSP 2001*, 57–72. New York, NY: ACM.

ECC: Summary

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

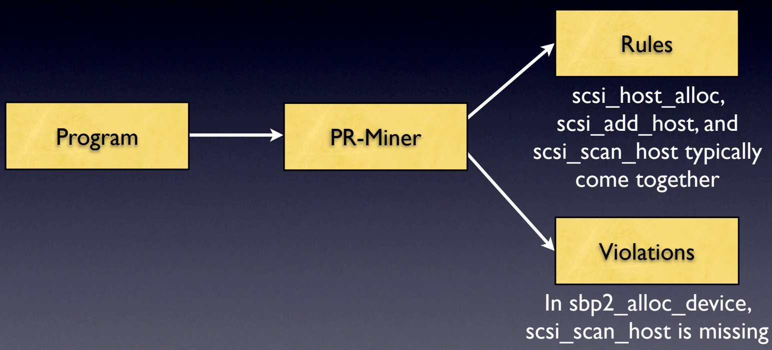
Engler, Dawson, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP 2001*, 57–72. New York, NY: ACM.

ECC: Summary

- Mines rules based on templates
- Templates have a fixed number of slots.
- fixed size

Engler, Dawson, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP 2001*, 57–72. New York, NY: ACM.

PR-Miner



Li, Zhenmin, and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, 306–315, New York, NY: ACM

PR-Miner: Step 1

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

Li, Zhenmin, and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, 306–315, New York, NY: ACM

PR-Miner: Step 1

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

```
T: HeapTuple
…
F: SearchSysCache
F: HeapTupleIsValid
F: elog
T: Form_pg_class
…
F: ReleaseSysCache
```

Li, Zhenmin, and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, 306–315, New York, NY: ACM

28

PR-Miner: Step 2

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

```
T: Form_pg_class
T: HeapTuple
F: SearchSysCache
F: elog
F: ReleaseSysCache
...
```

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

```
T: HeapTuple
F: SearchSysCache
F: HeapTupleIsValid
F: elog
T: Form_pg_class
...
```

Li, Zhenmin, and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, 306–315, New York, NY: ACM

29

PR-Miner: Step 2

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

```
T: Form_pg_class
T: HeapTuple
F: SearchSysCache
F: elog
F: ReleaseSysCache
...
```

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

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

```
T: HeapTuple
F: SearchSysCache
F: HeapTupleIsValid
F: elog
T: Form_pg_class
...
```

Li, Zhenmin, and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, 306–315, New York, NY: ACM

30

PR-Miner: Step 2

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

```
T: Form_pg_class
T: HeapTuple
F: SearchSysCache
F: elog
F: ReleaseSysCache
...
```

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

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

```
T: HeapTuple
F: SearchSysCache
F: HeapTupleIsValid
F: elog
T: Form_pg_class
...
```

Violation:
F: ReleaseSysCache is
missing

Li, Zhenmin, and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, 306–315, New York, NY: ACM

31

PR-Miner: Summary

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

Li, Zhenmin, and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, 306–315, New York, NY: ACM

32

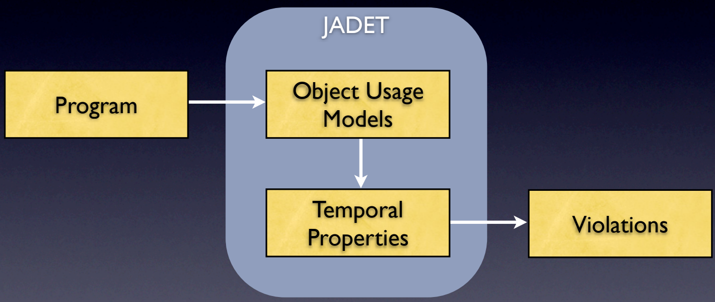
PR-Miner: Summary

- Mines rules being sets of entities
- Ordering is not taken into account.
- Cons: no ordering of entities

Li, Zhenmin, and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, 306–315, New York, NY: ACM

33

JADET



Wasylkowski, Andrzej, Andreas Zeller, and Chrisitan Lindig. 2007. Detecting object usage anomalies. In *ESEC-FSE 2007*, 35–44, New York, NY: ACM

34

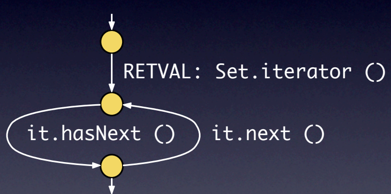
Creating an Object Usage Model: Example I

```
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;
}
```

35

Creating an Object Usage Model: Example I

```
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;
}
```



36

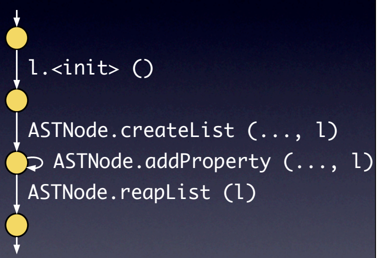
Creating an Object Usage Model: Example 2

```
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;
}
```

37

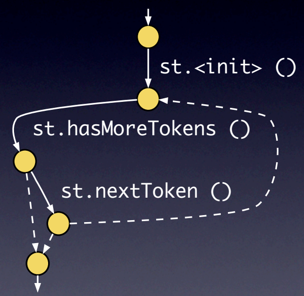
Creating an Object Usage Model: Example 2

```
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;
}
```



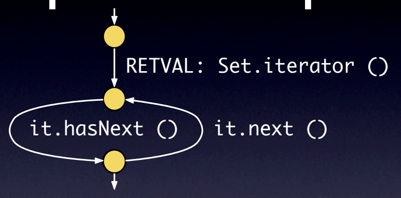
38

Example OUMs: StringTokenizer



39

Extracting Temporal Properties



Temporal properties

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

40

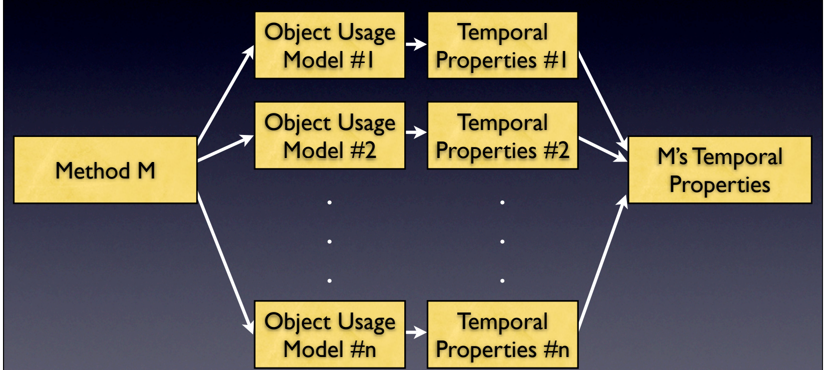
Extracting Temporal Properties

Temporal properties

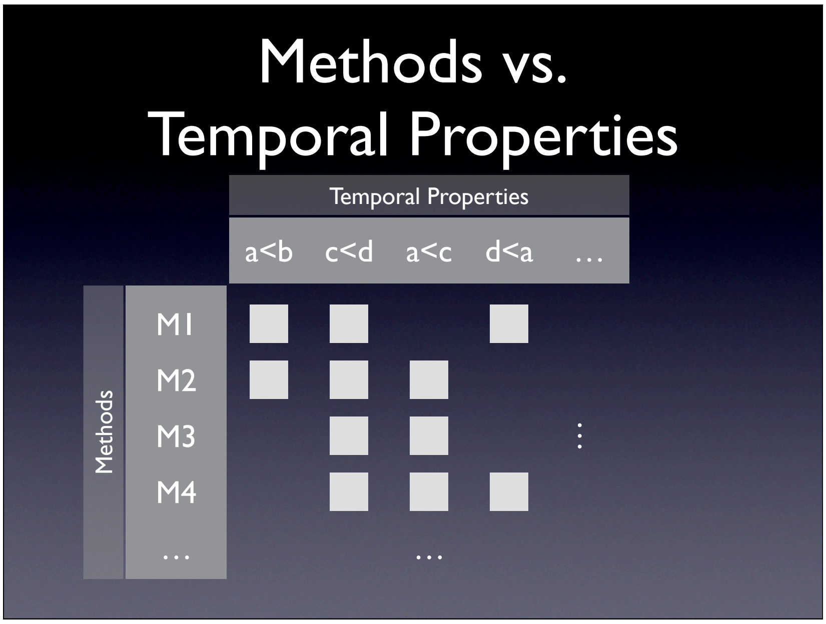
```
     StringTokenizer.<init>() @ this < StringTokenizer.countTokens() @ this
     StringTokenizer.<init>() @ this < StringTokenizer.nextToken() @ this
StringTokenizer.countTokens() @ this < StringTokenizer.nextToken() @ this
  StringTokenizer.nextToken() @ this < StringTokenizer.nextToken() @ this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

41

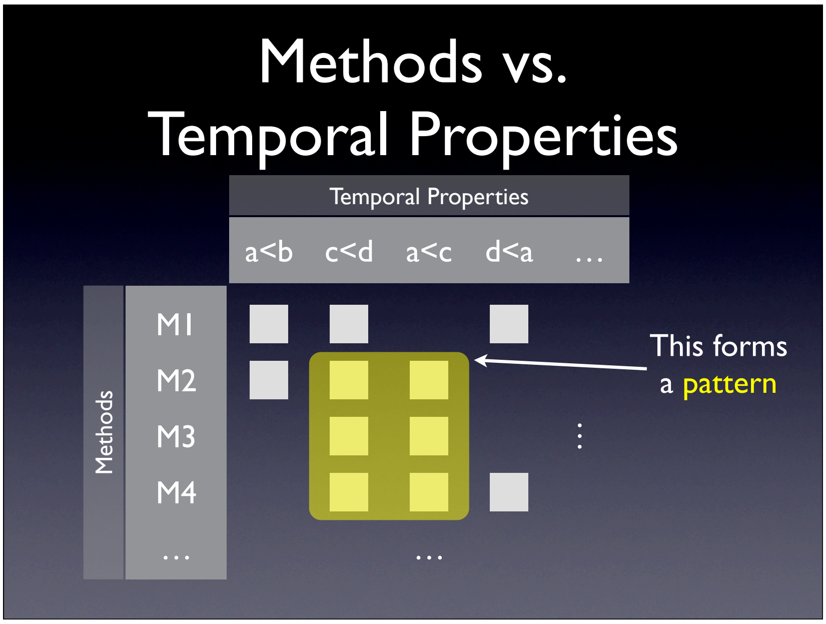
Extracting Temporal Properties: Summary



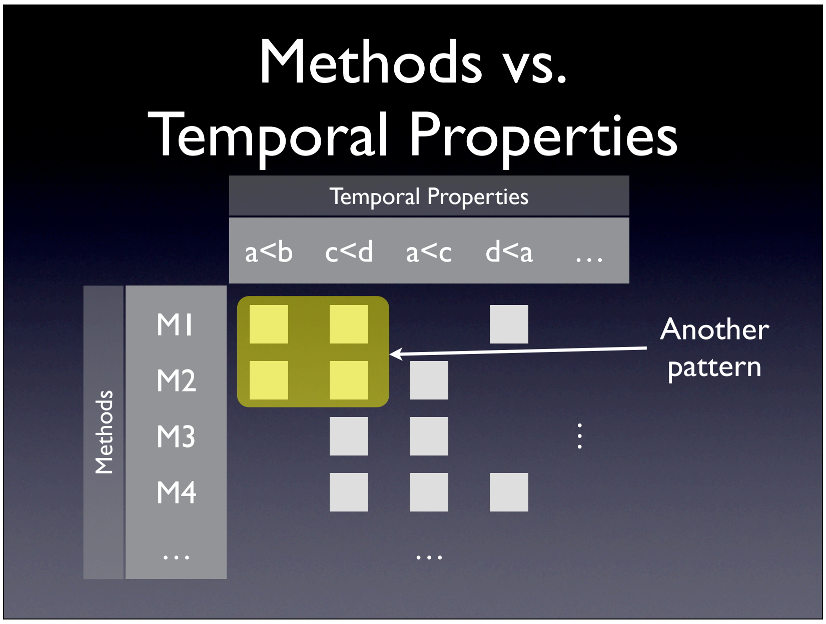
42


Methods vs.
Temporal Properties
Temporal Properties
a<b c<d a<c d<a …
Methods
M1
M2
M3
M4
…

43

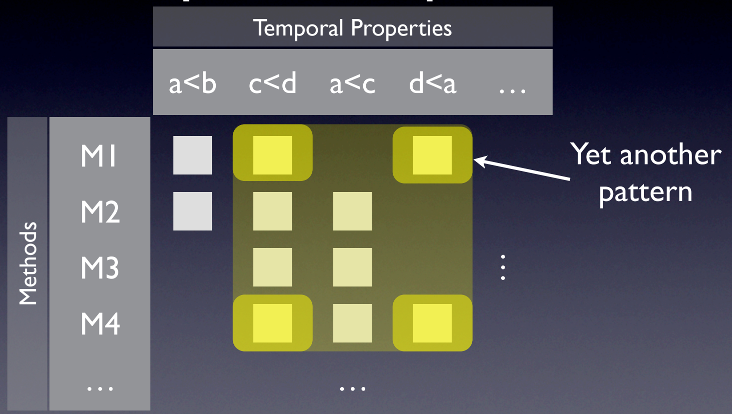

Methods vs.
Temporal Properties
Temporal Properties
a<b c<d a<c d<a …
Methods
M1
M2
M3
M4
…
This forms a pattern

44


Methods vs.
Temporal Properties
Temporal Properties
a<b c<d a<c d<a …
Methods
M1
M2
M3
M4
…
Another pattern

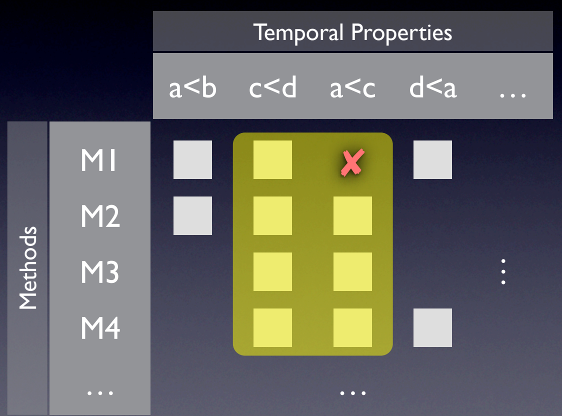
45

Methods vs. Temporal Properties



46

Detecting Violations



47

Example Violation (1)

```
private boolean verifyNIAP (…) {
    …
    Iterator iter = …;
    while (iter.hasNext()) {
        … = iter.next();
        …
        return verifyNIAP (…);
    }
    return true;
}
```

should be fixed

48

Example Violation (2)

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

should be fixed

49

Example Violation (3)

```
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

50

JADET: Summary

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

51

JADET: Summary

- Mines rules being sets of temporal properties
- All problems solved? Of course not!
- Cons: quite complicated, many false positives

52

Summary

★ Three main approaches:

  ★ Rule-based Techniques

  ★ Specification-checking techniques

  ★ Mining-based techniques

53

Code-mining Techniques

★ "Learn" rules from source code

★ Rule violation = potential defect

  ★ Can find project-specific bugs

★ Many different rules types

54