Mining for Software Reliability

How Data Mining Helps Improve Software Reliability

Chao Liu
Yahoo! Inc.
chaoliu@yahoo-inc.com

Tao Xie
North Carolina State Univ.
xie@csc.ncsu.edu

Jiawei Han
Univ. of Illinois at UC
hanj@cs.uiuc.edu

An up-to-date version of this tutorial is available at
http://www.ews.uiuc.edu/~chaoliu/tutorials.htm

Chao Liu
Research scientist, Data Mining Research Group, Yahoo! Inc.
Doing research in the confluence of data mining and software engineering
2007, Ph.D., UIUC
- Advisor: Prof. Jiawei Han
- Dissertation on statistical debugging and automated program failure triage
2005, M.S., UIUC
2003, B.S., Peking University, China

Tao Xie
Assistant Professor at North Carolina State University
2007, Ph.D., U. Washington (Advisor: David Notkin)
Leads the ASE research group at NCSU
Co-presented tutorials on Mining Software Engineering Data at KDD 06 and ICSE 07
Co-organizer of 2007 Dagstuhl Seminar on “Mining Programs and Processes”

http://ase.csc.ncsu.edu/

Jiawei Han
Professor in the University of Illinois at Urbana-Champaign
Leads a group of 15 Ph.D. students at CS, UIUC, working on data mining and database systems
ACM Fellow, 2003
Acknowledgments

- John Anvik
- David Andrezejewski
- Ahmed E. Hassan
- Sunghun Kim
- Stefan Leue
- Zhenmin Li
- Ben Liblit
- Jian Pei
- Andy Podgurski
- Muralikrishna Ramanathan
- Peter Rigby
- Alice X. Zheng
- Thomas Zimmermann
- Andreas Zeller

For their support to our tutorial

Tutorial Goals

- Learn about:
  - Recent and notable research and researchers in mining for software reliability
  - Data mining and data processing techniques and how to apply them to SE data for software reliability

- By end of tutorial, you should be able:
  - Prepare SE data for mining
  - Mine interesting information from SE data

Mining SE Data

MAIN GOAL

- Transform static record-keeping SE data to active data
- Make SE data actionable by uncovering hidden patterns and trends

Overview of Mining SE Data

<table>
<thead>
<tr>
<th>software engineering data</th>
<th>software engineering tasks helped by data mining</th>
</tr>
</thead>
<tbody>
<tr>
<td>code bases</td>
<td>programming</td>
</tr>
<tr>
<td>change history</td>
<td>defect detection</td>
</tr>
<tr>
<td>program states</td>
<td>testing</td>
</tr>
<tr>
<td>structural entities</td>
<td>debugging</td>
</tr>
<tr>
<td>bug reports/nl</td>
<td>maintenance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data mining techniques</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>classification</td>
<td>association/patterns</td>
</tr>
<tr>
<td>clustering</td>
<td></td>
</tr>
</tbody>
</table>

...
Overview of Mining SE Data

Chao Liu, Tao Xie, Jiawei Han, Mining for Software Reliability, ICDM, 2007
### Why Reliability?

- **Software is “full of bugs”**
  - Windows 2000, 35 million lines of code
  - 63,000 known bugs at the time of release, 2 per 1000 lines
- **Software failure costs**
  - Ariane 5 explosion due to “errors in the software of the inertial reference system” (Ariaen-5 flight 501 inquiry board report [http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf](http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf))
  - A study by the National Institute of Standards and Technology found that software errors cost the U.S. economy about $59.5 billion annually [http://www.nist.gov/director/prog-ofc/report02-3.pdf](http://www.nist.gov/director/prog-ofc/report02-3.pdf)
- **Testing and debugging are laborious and expensive**
  - “50% of my company employees are testers, and the rest spends 50% of their time testing!” —Bill Gates, in 1995

### Software Reliability Methods

- **Static Bug Detection**
  - Without running the code, detect bugs in code
- **Dynamic Bug Detection (aka. Testing)**
  - Run the code with some test inputs and detect failures/bugs
- **Debugging**
  - Given known test failures (symptoms), pinpoint the bug locations in the code

### Why Mining for Soft Reliability?

- **Finding bugs is challenging**
  - Require specifications/properties, which often don’t exist
  - Require substantial human efforts in analyzing data
  - We can mine common patterns as likely specifications/properties
  - Detect violations of patterns as likely bugs
  - We can mine huge data for patterns or locations to narrow down the scope of human inspection
  - E.g., code locations or predicates covered more in failing runs less in passing runs may be suspicious bug locations
Tutorial Outline

- Data Mining for Software Bug Detection
  - Frequent pattern mining
- Automated Debugging in Software Programs
  - From frequent patterns to software bugs
  - Statistical debugging
- Automated Debugging in Computer Systems
  - Automated diagnosis of system misconfigurations
  - Performance debugging
- Conclusion

Outline: Software Bug Detection

**Common approach:** mining rules/patterns from source code/revision histories and detecting bugs as rule/pattern violations

- **Mining rules from source code**
  - Bugs as deviant behavior [Engler et al., SOSP’01]
  - Mining programming rules with PR-Miner [Li et al., FSE’05]
  - Mining function precedence protocols [Ramanathan et al., ICSE’07]
  - Revealing neglected conditions [Chang et al., ISSTA’07]
- **Mining rules from revision histories**
  - DynaMine [Livshits& Zimmermann, FSE’05]
- **Mining copy-paste patterns from source code**
  - CP-Miner [Li et al., OSDI’04] to find copy-paste bugs

Bugs as Deviant Behavior [Engler et al., SOSP’00]

- Static verification tools need rules to check against program code
- Problem: what are the rules?!?!?
  - Does a follow b? Can foo fail? Does bar free its arguments? Does lock / protect variable x?
  - Manually finding rules is hard. So avoid it. Instead infer what code believes, cross check for contradiction

Bugs as Deviant Behavior

- Find errors without knowing truth?
  - **Contradiction** in belief. To find lies: cross-examine one witness or many witness. Any contradiction is an error (internal consistency)
  - **Deviation** from common behavior. To infer correct behavior: if 1 person does X, might be right or a coincidence. If 1000s do X and 1 does Y, probably an error (statistical analysis)
  - **Crucial:** we know contradiction is an error without knowing the correct belief!
Cross-Checking Program Beliefs

- **MUST beliefs**: inferred from acts that imply beliefs code *must* have
  
  ```c
  x = *p;    // MUST: belief: p not null
  // MAY: x, p protected by l.
  free(p);   // MUST: p heap allocated
  // MUST: p not used anymore.
  unlock(l); // MUST: l acquired
  ```

- **Check using internal consistency**: infer beliefs at different locations, then cross-check for contradiction

  ```c
  /* 2.4.1: drivers/isdn/svmb1/capidrv.c */
  if(!card)
    printk(KERN_ERR, "capidrv-%d: …", card->contrnr…)
  ```

- **MAY beliefs**: could be coincidental

  - Inferred from acts that imply beliefs code *may* have

  ```c
  lock(a);
  x++;  unlock(a);
  lock(a);
  x++;  unlock(a);
  ```

  - Check as MUST beliefs; rank errors by belief confidence.

Deriving “A must be followed by B”

- “a(); … b();” → MAY belief that a() follows b()

- **Algorithm**:
  
  - Assume every a-b is a valid pair
  - Emit “check” for each path that has a() then b()
  - Emit “error” for each path that has a() and no b()

  ```c
  foo(p, …);  "check foo-bar"  a();  "check a-b"  foo(p, …);  "error:foo, no bar!"
  ```

  - Rank errors for each pair using the test statistic

Some Results of Bugs as Deviant Behavior

<table>
<thead>
<tr>
<th>Checker</th>
<th>Bug</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-then-use</td>
<td>79</td>
<td>26</td>
</tr>
<tr>
<td>use-then-check</td>
<td>102</td>
<td>4</td>
</tr>
<tr>
<td>redundant checks</td>
<td>24</td>
<td>10</td>
</tr>
</tbody>
</table>

**Internal null checker on Linux 2.4.7**

- Rank errors for each pair using the test statistic

<table>
<thead>
<tr>
<th>OS</th>
<th>Errors</th>
<th>False</th>
<th>Applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenBSD 2.8</td>
<td>18</td>
<td>3</td>
<td>1615</td>
</tr>
<tr>
<td>Linux 2.4.1</td>
<td>12</td>
<td>16</td>
<td>4905</td>
</tr>
<tr>
<td>Linux 2.3.99</td>
<td>5</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**User-pointer checker**
Outline: Software Bug Detection

- **Mining rules from source code**
  - Bugs as deviant behavior [Engler et al., SOSP’01]
  - Mining techniques: Statistical analysis
  - Mining programming rules with PR-Miner [Li et al., FSE’05]
  - Mining function precedence protocols [Ramanathan et al., ICSE’07]
  - Revealing neglected conditions [Chang et al., ISSTA’07]

- **Mining rules from revision histories**
  - DynaMine [Livshits & Zimmermann, FSE’05]

- **Mining copy-paste patterns from source code**
  - CP-Miner [Li et al., OSDI’04] to find copy-paste bugs

Limitations of Bugs as Deviant Behavior

- Fixed rule templates
- Need specific knowledge about the software
- 2 elements

- PR-Miner [Li et al., FSE’05] (mining implicit programming rules) developed to address the limitations
  - General method (No prior knowledge; No templates)
  - General rules (Different types: function, variable, data type, etc.; Multiple elements)

Example 1. Variable Correlation

```c
void ll_stop(struct IsdnCardState *cs)
{
    isdn_ctrl ic;
    ic.command = ISDN_STAT_STOP;
    ic.driver = cs->myid;
    cs->iif.statcallb(&ic);
}
```

- It appears 98 times in Linux code.

Example 2. More Complex Rule

```c
AlterTableCreateToastTable(…)
{
    Relation class_rel;
    ....
    class_rel = heap_openr(…);
    ....
    simple_heap_update(class_rel, …);
    ....
    CatalogUpdateIndexes(class_rel, …);
    ....
    heap_close(class_rel, …);
    ....
}
```

- It appears 68 times in PostgreSQL code.
Example 3: Violation

```c
struct scsi_id_instance_data *sbp2_alloc_device (...) {
    ......
    scsi_host = host_alloc (...);
    ......
    if (!add_host (scsi_host, &ud->device))
        ......
        // scan_host (scsi_host) is missing!
}
```

- Violations to this rule are detected in Linux code.

Overview: Extracting Rules

- Observation: elements are usually used together
- Idea: finding association among elements that are frequently used together in source code

Frequent itemset mining

Examples:
spin_lock_irqsave and spin_unlock_irqrestore appear together within the same function more than 3600 times.

Flowchart of Extracting Rules

- Source files
- Parsing & hashing
- Itemsets
- Mining
- Programming patterns
- Generating rules
- Programming rules

Parsing Source Code

- Purpose: building an itemset database
- What is mapped to a number?
  - Element: function call, variable, data type, etc.
- What is mapped to an itemset?
  - Basic block
  - File
  - Function definition
- What is mapped to an itemset database?
  - Source code
Example: Parsing A Function

Function definition:
```
int twa_probe(struct pci_dev *pdev, ...)
{
    struct Scsi_Host *host = NULL;
    ......
    host = host_alloc (...);
    ......
    add_host(host, &pdev->dev);
    ......
    scan_host (host);
    ......
}
```

Identifiers to be tokenized:
- Scsi_Host
- host_alloc
- add_host
- scan_host

Tokenized:
- 92
- 39
- 68
- 56
- 36

Fields are combined with structure type:
{92, 39, 68, 56, 36,...}

Mining Programming Patterns

- Apply frequent itemset mining algorithm on the itemset database
  - A frequent sub-itemset corresponds to a programming pattern
    - E.g., \{39, 68, 36, 92\}:27 corresponds to pattern
      \{Scsi_Host, host_alloc, add_host, scan_host\}
  - Tradeoff: consider order or not

Generating Programming Rules

- Programming patterns \(\rightarrow\) programming rules
  - E.g.,
    - Patterns: \(\{a, b, d\} : 3, \{a\} : 4\)
    - Rules:
      - \(a\) \(\Rightarrow\) \(\{b,d\}\) with confidence = \(\frac{3}{4} = 75\%\)
      - \(b\) \(\Rightarrow\) \(\{a,d\}\) with confidence = \(100\%\)
      - \(d\) \(\Rightarrow\) \(\{a,b\}\) with confidence = \(100\%\)
      - \(a,b\) \(\Rightarrow\) \(\{d\}\) with confidence = \(100\%\)
      - \(a,d\) \(\Rightarrow\) \(\{b\}\) with confidence = \(100\%\)
      - \(b,d\) \(\Rightarrow\) \(\{a\}\) with confidence = \(100\%\)

Rule Explosion Problem

- Exponential number of rules
- Solution: closed mining
  - Example: \(\{a,b,d\}:3, \{a\}:4\)
  - \(\{a,b\}:3, \{a,d\}:3, \{b,d\}:3\) are not closed
  - Close rules
    - \(\{a,b,d\}:3 \mid \{a\}:4\)
Detecting Violations

- For violations of a programming rule
  - The rule holds for most cases
    - Confidence > threshold
  - The rule is violated for a few cases
    - Confidence < 100%

Example: Detecting Violations

Programming patterns:
- \{Scsi\_Host, host\_alloc, add\_host, scan\_host\}: 27
- \{Scsi\_Host, host\_alloc, add\_host\}: 29

Programming rule:
- \{Scsi\_Host, host\_alloc, add\_host\} → \{scan\_host\}
  - with confidence 27/29 = 93%

Pruning False Violations

- Elements span across functions
- Inter-procedural analysis
  - Prune if missing elements appear in calling paths
  - E.g.: \{a\} → \{b\}

Some Results of Bug Detection

<table>
<thead>
<tr>
<th>Software</th>
<th>#C files</th>
<th>LOC</th>
<th>#functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>3,538</td>
<td>3,037,403</td>
<td>73,607</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>409</td>
<td>381,192</td>
<td>6,964</td>
</tr>
<tr>
<td>Apache</td>
<td>160</td>
<td>84,724</td>
<td>1,912</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software</th>
<th>Inspected (top 60)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bugs</td>
</tr>
<tr>
<td>Linux</td>
<td>16</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>6</td>
</tr>
<tr>
<td>Apache</td>
<td>1</td>
</tr>
</tbody>
</table>
Outline: Software Bug Detection

- **Mining rules from source code**
  - Bugs as deviant behavior [Engler et al., SOSP’01]
  - Mining techniques: Statistical analysis
  - Mining programming rules with PR-Miner [Li et al., FSE’05]
    - Mining techniques: Frequent itemset mining
  - Mining function precedence protocols [Ramanathan et al., ICSE’07]
  - Revealing neglected conditions [Chang et al., ISSTA’07]

- **Mining rules from revision histories**
  - DynaMine [Livshits & Zimmermann, FSE’05]

- **Mining copy-paste patterns from source code**
  - CP-Miner [Li et al., OSDI’04] to find copy-paste bugs

Limitations of PR-Miner

- Rules across multiple functions
  - Not using inter-procedural analysis

- False negatives of violations in control paths
  - Not using sophisticated analysis techniques

- Inter-procedural, path-sensitive inference of function precedence protocols to address the limitations [Ramanathan et al., ICSE’07]

Mining Function Precedence Protocols

- Precedence protocol:
  - A call to `fclose` is always preceded by a call to `fopen`

- Successor protocol:
  - A call to `fopen` is always succeeded by a call to `fclose`

Violation of Precedence Protocols

- Precedence protocol:
  - A call to `fclose` is always preceded by a call to `fopen`

- Successor protocol:
  - A call to `fopen` is always succeeded by a call to `fclose`
Motivating Example (from PostGRESql)

```java
181   RI_FKey_check(PG_FUNCTION_ARGS)
182   {
199      ri_CheckTrigger(...);
210      pk_rel = heap_open(...);
296      match_type = ri_DetermineMatchType(...);
303      ri_BuildQueryKeyFull(...);
437   }
```

ri_BuildQueryKeyFull not preceded by ri_DetermineMatchType
Leads to a potential crash

Mining Specifications Statically

- Analyze program source
- Programs are mostly well-written
  - call patterns that occur frequently are likely to be correct
- Detect common patterns with frequent subsequent mining
  - define protocols
- Flag deviant patterns as potential bugs

Path Sensitivity

```java
f() {
   init();
   ...
   access();
}
g() {
   a();
   if (cond)
      init();
   ...
   access();
}
```

init precedes access only in f
Inter-procedural Analysis

```c
h() {
    if (cond)
        lwrap();
    else
        lwrap();
    ...
    uwrap();
}
```

Tool Implementation/Evaluation

- **CHRONICLER** – tool implemented in C
- Tested on open source C programs
  - Apache, linux, openssh, gimp, postgresql
  - Lines of code varies from 66K to 2M
  - Number of call-sites varies from 10K to 110K

Some Results of Precedence-Related Bug Detection

Case Study: Linux

- **Hardware Bug**
  - Difficult to detect using traditional testing techniques
  - Platform dependent error
  - Transparently identified using CHRONICLER

- **Performance Bug**
  - Cache lookup operation was absent
  - Not easily specified as a bug for testing
  - Deviation delays data write flushes

Outline: Software Bug Detection

- **Mining rules from source code**
  - Bugs as deviant behavior [Engler et al., SOSP’01]
  - Mining techniques: Statistical analysis
  - Mining programming rules with PR-Miner [Li et al., FSE’05]
    - Mining techniques: Frequent itemset mining
  - Mining function precedence protocols [Ramanathan et al., ICSE’07]
    - Mining techniques: Frequent subsequence mining
  - Revealing neglected conditions [Chang et al., ISSTA’07]

- **Mining rules from revision histories**
  - DynaMine [Livshits & Zimmermann, FSE’05]

- **Mining copy-paste patterns from source code**
  - CP-Miner [Li et al., OSDI’04] to find copy-paste bugs
Limitation of Precedence-Related Bug Detection

- Does not take data flow or data dependency into account

- A new approach to discovering neglected conditions [Chang et al., ISSTA’07] addresses the issue
  - Based on dependence analysis, frequent itemset, and frequent subgraph mining

Neglected Conditions

- AKA missing conditions, cases, or paths
- Long known to be important and difficult-to-find
- Often exploited by attackers
  - Buffer overflow
  - SQL injection
  - Cross-site scripting
  - Format string attacks

Approach to Detecting Neglected Conditions

- User indicates minimal constraints on rules (e.g., condition involving function call)
- Rules abstracted as graph minors of system dependence graphs
- Candidate rules identified using frequent itemset and frequent subgraph mining
- Must be confirmed manually by developers
- Possible rule violations identified using heuristic graph matching
- Must be confirmed manually

Dependence Graph Representation

- Wide variety of rules can be described in terms of:
  - Control dependences
  - Data dependences
  - Statements attributes
    - Parameter types
    - ASTs
- Insensitive to non-essential variations in statement ordering
  - Permits rules to be matched in more contexts
- Employ Grammatech’s CodeSurfer to generate a system dependence graph
Example

APU_DECLARE(apr_status_t) apr_sdbm_fetch(apr_sdbm_t *db, ...
..., apr_sdbm_datum_t key) {

if (db == NULL || bad(key)) return APR_EINVAL;
...
if ((status = apr_sdbm_lock(db, ...)) != APR_SUCC) return ...

if ((status = getpage(db, (exhash(key))) == APR_SUCC) { ... }

(void) apr_sdbm_unlock(db);

Courtesy of Andy Podgurski

Frequent Subgraph Mining

- Given a graph dataset $D = \{G_1, G_2, \ldots, G_n\}$
  FSM algorithms find any subgraph whose support is $\geq$ to a given threshold $t$
- FSM algorithms currently handle only small graphs
- Use algorithm based on Monkey [Zhang et al., ICDE’07 ]

SDG Minors

- Isomorphic to graph derived from SDG subgraph by edge contractions
- Model transitivity of dependences
- Needed because relationship between statements in rule can be realized in different instances by different chains of direct dependences

Courtesy of Andy Podgurski
### Example Minor

```
if (parent_err) {
    rv = apr_file_dup(&attr->parent_out, parent_err, attr->pool);
}
```

```
if (parent_err) {
    if (attr->parent_err == NULL)
        rv = apr_file_dup(&attr->parent_err, parent_err, attr->pool);
    else
        rv = apr_file_dup2(attr->parent_err, parent_err, attr->pool);
}
```

### Rule Mining Issues
- Graph set size
- Edge direction
- Multigraphs
- Non-maximal subgraphs
- Node labeling
- Frequency threshold

### Transforming the Graph Set
1. Identify candidate nodes
2. Compute dependency sphere of radius \( r \) around each candidate node
3. Remove all nodes that can’t be reached from (or to) the center via a directed path
4. Use the MAFIA frequent itemset mining algorithm to identify and remove nodes whose labels are too infrequent to occur in a frequent minor
5. Compute the near transitive closure of each reduced dependency sphere

### Node Labeling
- Nodes are labeled with their AST
- Variables names are ignored; types are retained
- Some control point ASTs are converted to canonical form
  - \(!=\) is converted to \(==\)
  - \(<\) is converted to \(>\)
  - \(\leq\) is converted to \(\geq\)
User Evaluation of Rules

- User employs tool to review candidate rules to confirm they are of interest
- Tool permits user to:
  - Select candidate rule to view from prioritized list
  - Highlight source code lines of rule
  - Modify rule to remove irrelevant elements or dependences
  - Select rule whose uses will be checked for violations
  - Indicate *key nodes* to be used in searching for rule violations

Detecting Rule Violations

- All PDGs in code base searched
- Violation of rule $R$ is PDG minor similar to $R$ but lacking nodes or dependences in $R$
- Each PDG $G$ is searched for node $n$ with same label as any *key node*
- $n$ is made center of a dependence sphere $S$
- $S$ is searched for rule violations
  - Heuristic graph matching algorithm used to find minor $M$ of $S$ that best matches $R$
  - Rule violations ranked by differences between $M$ and $R$

Some Results of Neglected-Condition Bug Detection

- 9 subprojects of the *Apache* HTTP server project (2.2.2) were analyzed
  - ~ 400 files and $\geq$ 25,000 SLOC
  - SDG generated by *CodeSurfer* contained ~ 500,000 nodes
- The 443 functions called in at least two different functions were chosen for further analysis

<table>
<thead>
<tr>
<th>Rules used to detect violations</th>
<th>88</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reported rule violations</td>
<td>69</td>
</tr>
<tr>
<td>Potential neglected conditions</td>
<td>8</td>
</tr>
</tbody>
</table>

Outline: Software Bug Detection

- Mining rules from source code
  - Bugs as deviant behavior [Engler et al., SOSP’01]
    - Mining techniques: Statistical analysis
  - Mining programming rules with PR-Miner [Li et al., FSE’05]
    - Mining techniques: Frequent itemset mining
  - Mining function precedence protocols [Ramanathan et al., ICSE’07]
    - Mining techniques: Frequent subsequence mining
  - Revealing neglected conditions [Chang et al., ISSTA’07]
    - Mining techniques: Frequent itemset mining+Frequent subgraph mining

- Mining rules from revision histories
  - DynaMine [Livshits & Zimmermann, FSE’05]

- Mining copy-paste patterns from source code
  - CP-Miner [Li et al., OSDI’04] to find copy-paste bugs
Outline: Software Bug Detection

- **Mining rules from source code**
  - Bugs as deviant behavior [Engler et al., SOSP’01]
    - Fixed rule templates, need knowledge about programs, pairs of elements
  - Mining programming rules with PR-Miner [Li et al., FSE’05]
    - Not inter-procedural, not path sensitive
  - Mining function precedence protocols [Ramanathan et al., ICSE’07]
    - No data flow or dependence
  - Revealing neglected conditions [Chang et al., ISSTA’07]

- **Mining rules from revision histories**
  - DynaMine [Livshits & Zimmermann, FSE’05]

- **Mining copy-paste patterns from source code**
  - CP-Miner [Li et al., OSDI’04] to find copy-paste bugs

---

Crucial Observation

**Things that are frequently changed together often form a pattern**

...also known as co-change

**Co-changed items = patterns**

---

Co-added Method Calls

```java
public void createPartControl(Composite parent) {
    ...
    // add listener for editor page activation
    getSite().getPage().addPartListener(partListener);
}

public void dispose() {
    ...
    // remove listener for editor page activation
    getSite().getPage().removePartListener(partListener);
}
```

---

Co-added Method Calls

```java
public void createPartControl(Composite parent) {
    ...
    // add listener for editor page activation
    getSite().getPage().addPartListener(partListener);
}

public void dispose() {
    ...
    // remove listener for editor page activation
    getSite().getPage().removePartListener(partListener);
}
```
How DynaMine Works

Revision history mining

1. Mine CVS histories
2. Patterns
3. Rank and filter
4. Instrument relevant method calls
5. Run the application
6. Post-process
7. Reporting
8. Usages
9. Error patterns
10. Unlikely patterns

Mining Patterns

Revision history mining

1. Mine CVS histories
2. Patterns
3. Rank and filter
4. Instrument relevant method calls
5. Run the application
6. Post-process
7. Reporting
8. Error patterns
9. Unlikely patterns

Mining Method Calls

Foo.java
- o1.addListener()
- o1.removeListener()
- o2.addListener()
- o2.removeListener()
- System.out.println()

Bar.java
- o2.addListener()
- o2.removeListener()

Baz.java
- o3.addListener()
- o3.removeListener()
- list.iterator()
- iter.hasNext()
- iter.next()

Qux.java
- o4.addListener()
- System.out.println()
- o4.removeListener()

Finding Pairs

Foo.java
- o1.addListener()
- o1.removeListener()
- 1 Pair

Bar.java
- o2.addListener()
- o2.removeListener()
- System.out.println()
- 1 Pair

Baz.java
- o3.addListener()
- o3.removeListener()
- list.iterator()
- iter.hasNext()
- iter.next()
- 2 Pairs

Qux.java
- o4.addListener()
- System.out.println()
- o4.removeListener()
- 0 Pairs
- 1.42
Mining Method Calls

Co-added calls often represent a usage pattern

Finding Patterns

Find “frequent itemsets” (with Apriori)

Ranking Patterns

Support count = #occurrences of a pattern
Confidence = strength of a pattern, P(A|B)

Ranking Patterns

This is a fix!
Rank removeListener() patterns higher
### Investigated Projects

<table>
<thead>
<tr>
<th></th>
<th>JEDIT</th>
<th>ECLIPSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>since</td>
<td>2000</td>
<td>2001</td>
</tr>
<tr>
<td>developers</td>
<td>92</td>
<td>112</td>
</tr>
<tr>
<td>lines of code</td>
<td>700,000</td>
<td>2,900,000</td>
</tr>
<tr>
<td>revisions</td>
<td>40,000</td>
<td>400,000</td>
</tr>
</tbody>
</table>

*Courtesy of Thomas Zimmermann*

---

### Simple Method Pairs

#### GUIs & Listener

- `addWidget()`, `removeWidget()`, `addPropertyChangeListener()`, `removePropertyChangeListener()`

#### Usage pattern

#### Error pattern

*Courtesy of Thomas Zimmermann*

---

### State Machines in Eclipse

#### Locking of Resources

- `HLock()`, `HUnlock()`

*Not hit at runtime*

*Courtesy of Thomas Zimmermann*

---

### Pretty-printing

- `redoAlignment()`, `editAlignment()`, `enterAlignment()`

*Usage pattern*

*Courtesy of Thomas Zimmermann*
State Machines in Eclipse

- Memory context manipulation

State Machines in JEdit

- Compound edits (for undo/redo)

Dynamic Validation

- Find and count matches and mismatches.
  - o.register(d)
  - o.deregister(d)
  - Static vs dynamic counts.

Matches and Mismatches

- revision history mining
- mine CVS histories
- patterns
- rank and filter
- dynamic analysis
- reporting
- report patterns
- report bugs

Chao Liu, Tao Xie, Jiawei Han, Mining for Software Reliability, ICDM, 2007

Courtesy of Thomas Zimmermann
Pattern classification

post-process
v validations, e violations

usage patterns

error patterns

unlikely patterns

e=v/10
v/10<=e<=2v
otherwise

Results of Mining Patterns

optional captions

Outline: Software Bug Detection

Mining rules from source code
- Bugs as deviant behavior [Engler et al., SOSP’01]
  - Mining techniques: Statistical analysis
- Mining programming rules with PR-Miner [Li et al., FSE’05]
  - Mining techniques: Frequent itemset mining
- Mining function precedence protocols [Ramanathan et al., ICSE’07]
  - Mining techniques: Frequent subsequence mining
- Revealing neglected conditions [Chang et al., ISSTA’07]
  - Mining techniques: Frequent itemset mining+Frequent subgraph mining

Mining rules from revision histories
- DynaMine [Livshits & Zimmermann, FSE’05]
  - Mining techniques: Frequent itemset mining

Mining copy-paste patterns from source code
- CP-Miner [Li et al., OSDI’04] to find copy-paste bugs

Mining Copy-Paste Bugs

- Copy-pasting is common
  - 12% in Linux file system [Kasper 03]
  - 19% in X Window system [Baker 95]
- Copy-pasted code is error prone
  - Among 35 errors in Linux drivers/i2o, 34 are caused by copy-paste [Chou et al. 01]

(Simplified example from linux-2.6.6/arch/sparc/prom/memory.c)
Overview of Copy-Paste Bug Detection

- Parse source code & build a sequence database
- Mine for basic copy-pasted segments
- Compose larger copy-pasted segments
- Prune false positives

Parsing Source Code

- Purpose: building a sequence database
- Idea: statement $\rightarrow$ number
  - Tokenize each component
  - Different operators/constant/key words $\rightarrow$ different tokens
- Handle identifier renaming:
  - same type of identifiers $\rightarrow$ same token

Parsing Source Code:

- old = 3;
- new = 3;

Building Sequence Database

- Program $\rightarrow$ a long sequence
  - Need a sequence database
- Cut the long sequence
  - Naïve method: fixed length
  - Method: basic block

Final sequence DB:

- (65)
- (16, 16, 71)
- ...
- (65)
- (16, 16, 71)

Mining for Basic Copy-pasted Segments

- Apply frequent sequence mining algorithm on the sequence database
- Modification
  - Constrain the max gap
**Composing Larger Copy-Pasted Segments**

- Combine the neighboring copy-pasted segments repeatedly

```c
for (i=0; i<n; i++) {
    total[i].adr = list[i].addr;
    total[i].bytes = list[i].size;
    total[i].more = &total[i+1];
}
```

**Hash values**

<table>
<thead>
<tr>
<th>copy-pasted</th>
<th>65</th>
<th>16</th>
<th>16</th>
<th>71</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>combine</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pruning False Positives**

- Unmappable segments
  - Identifier names cannot be mapped to corresponding ones

```
f (a1);
f (a2);
f (a3);
```

- Tiny segments

```
f1 (b1);
f1 (b2);
f2 (b3);
```

**Detecting “Forget-to-change” Bugs**

- Method: measure *unchanged ratio*
  - Example: 1 out of 4 “total” is unchanged, *unchanged ratio = 0.25*
  ```c
  for (i=0; i<n; i++) {
      total[i].adr = list[i].addr;
      total[i].bytes = list[i].size;
      total[i].more = &total[i+1];
  }
  ```

- If $0 < \text{unchanged ratio} < \text{threshold}$, it is reported as a bug.

```
for (i=0; i<n; i++) {
    taken[i].adr = list[i].addr;
    taken[i].bytes = list[i].size;
    taken[i].more = &total[i+1];
}
```

**Some Results of C-P Bug Detection**

<table>
<thead>
<tr>
<th>Software</th>
<th>Verified Bugs</th>
<th>Potential Bugs (careless programming)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>28</td>
<td>21</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>23</td>
<td>8</td>
</tr>
<tr>
<td>Apache</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software</th>
<th># LOC</th>
<th>Time</th>
<th>Space(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>4.4 M</td>
<td>20 mins</td>
<td>527</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>3.3 M</td>
<td>20 mins</td>
<td>459</td>
</tr>
<tr>
<td>Apache</td>
<td>224 K</td>
<td>15 secs</td>
<td>30</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>458 K</td>
<td>38 secs</td>
<td>57</td>
</tr>
</tbody>
</table>
Outline: Software Bug Detection

Mining rules from source code
- Bugs as deviant behavior [Engler et al., SOSP’01]
  - Mining techniques: Statistical analysis
- Mining programming rules with PR-Miner [Li et al., FSE’05]
  - Mining techniques: Frequent itemset mining
- Mining function precedence protocols [Ramanathan et al., ICSE’07]
  - Mining techniques: Frequent subsequence mining
- Revealing neglected conditions [Chang et al., ISSTA’07]
  - Mining techniques: Frequent itemset mining + Frequent subgraph mining

Mining rules from revision histories
- DynaMine [Livshits & Zimmermann, FSE’05]
  - Mining techniques: Frequent itemset mining

Mining copy-paste patterns from source code
- CP-Miner [Li et al., OSDI’04] to find copy-paste bugs
  - Mining techniques: Frequent subsequence mining

Static Bug Detection ➔ Dynamic

Issues faced by static bug detection
- False positives: reported warnings that are not real bugs
  - Static analysis may produce a high percentage of false positives
- False negatives: real bugs being missed
  - Violation of frequent code patterns can be used to detect only a small (yet important) subset of real bugs

Need to complement static bug detection with dynamic bug detection (testing) and debugging

Software Reliability Methods

Static Bug Detection
- Without running the code, detect bugs in code

Dynamic Bug Detection (aka. Testing)
- Run the code with some test inputs and detect failures/bugs

Debugging
- Given known test failures (symptoms), pinpoint the bug locations in the code

Tutorial Outline

Data Mining for Software Bug Detection
- Frequent pattern mining

Automated Debugging in Software Programs
- From frequent patterns to software bugs
- Statistical debugging

Automated Debugging in Computer Systems
- Automated diagnosis of system misconfigurations
- Performance debugging

Conclusion
Software Bugs Are Costly

- Account for 40% of computer system failures [Marcus00]
- Cost $60B/year (0.6% GDP) [NIST02]
- Debugging and testing account for 50~75% of development cost [Hailpern02]

Debugging is HARD

- Debugging is hard because of
  - Bulky code implementing convoluted logic
  - Bug by bug, experience hard to carry over mechanically
- Crashing bugs
  - Relatively easy to debug
  - Can be extremely hard as well
    - Crash on using heap data that was corrupted a long time ago
- Semantic bugs
  - Unexpected behaviors but no crashes
  - Generally harder to debug

Non-crashing Failures

Non-crashing Failures
### Automated Debugging

**Semantic Bugs Dominate**

**Memory-related Bugs:**
- Many are detectable

**Concurrency bugs**
- Only few detectable
- Mostly require annotations or specifications

**Semantic Bugs:**
- Application specific
- Only few detectable
- Mostly require annotations or specifications

**Bug Distribution [Li et al., ASID'06]**
- 264 bugs in Mozilla and 98 bugs in Apache manually checked
- 29,000 bugs in Bugzilla automatically checked

**Automated Debugging**

- A grand challenge
  - Can computers help developers debug?

- Automated debugging
  - Seeing is believing
  - Analyze what went wrong in failures

- A multi-disciplinary endeavor
  - Software engineering
    - [Zeller, FSE'00, Renieris and Reiss, ASE'03, Cleve and Zeller, ICSE'05, Liu et al., FSE'05…]
  - Programming language
    - [Liblit et al., PLDI'03, PLDI'05, …]
  - Data mining and machine learning
    - [Zheng et al., NIPS'03, Liu et al., SDM'06, Zheng et al., ICML'06, Liu et al., ICDM'06, Andzejewski et al., ECML'07]

- A practically important and intellectually challenging problem

### Automated Debugging: An Overview

**Instrumentation**

- Function call graphs
- Variable evaluations
- Branchings
- Memory graphs
- …

**Executions**

- Vectors of behaviors that distinguish failing executions from passing ones
- Positive for failing and negative for passing
- Usually imbalanced

**Data Mining**

- Function call graphs
- Variable evaluations
- Branchings
- Memory graphs
- …

**Developer fixes the defect**

**Objective**

- Locate behaviors that distinguish failing executions from passing ones
- Nothing else but feature selection!
- Unfortunately, direct applications of existing techniques do not compare to the state of the art

**Automated Debugging as Feature Selection**

- Vector space representation
  - Each execution behavior as one feature
    - E.g., branchings, function calls, return values
  - Executions as vectors
    - Positive for failing and negative for passing
    - Usually imbalanced

- Instrumented code as feature vectors
  - E.g., branchings, function calls, return values
Tutorial Outline

- Data Mining for Software Bug Detection
  - Frequent pattern mining

- Automated Debugging in Software Programs
  - From frequent patterns to software bugs
  - Statistical debugging

- Automated Debugging in Computer Systems
  - Automated diagnosis of system misconfigurations
  - Performance debugging

- Conclusion

Execution Representations

- Executions represented at different granularities
  - Instruction level
  - Statement level
  - Block level
  - Function level
  - Files, modules, components …

- Representation forms
  - Sets of executed instructions, statements, functions, …
  - Sequences of executed instructions, statements, functions, …
  - Graphs of folded sequences
    - Control flow graphs in SE terminology, the most convenient to handle

Function call graphs

- Function-level abstraction of program behaviors
- Function calls and transitions
- First-order sequential information about function interactions

```c
int main()
{
    ...;
    A();
    ...;
    B();
}
int A(){ ... } int B(){ ... C() ... } int C(){ ... }
```

Discriminative Subgraph

- Function call graphs from passing and failing executions
- Debugging Question: What functions are fault relevant?

- Basic idea
  - Find distinct subgraphs that tell failing from passing executions

- Approaches
  - Indirect approach: Deriving function distinction from classification accuracies [Liu et. al, SDM’05]
  - Direct approach: Measure function distinction directly [Di Fatta et. al, SOQUA’06]
Distinction Derived from Classification
[Liu et al., SDM’05]

- Idea and Intuition
  - Both passing and failing executions start mostly the same
  - So low classification precision
  - Until some time abnormal behaviors manifest, which induce a much higher precision
  - So incremental classification along executions

- Methodology
  - Call graph expands as execution goes on
  - Classify graphs accumulated by the first-time entrance and the last-time exit of each function, getting $P_{\text{exit}} - P_{\text{exit}}$ for each function
  - The higher the precision boost, the more likely this function relates to the bug

An Illustration of Incremental Classification

A Running Example

```c
void subline(char *lin, char *pat, char *sub)
{
    int i, lastm, m;
    lastm = -1;
    i = 0;
    while((lin[i] != ENDSTR)) {
        m = amatch(lin, i, pat, 0);
        if (m >= 0){
            lastm = m;
        }
        if ((m == -1) || (m == i)) {
            i = i + 1;
        } else
            i = m;
    }
}
```

- A program called “replace”
  - Regular expression matching and substitution
  - 17 functions
  - 563 lines of C code

- Execution behaviors
  - 130 of 5542 test cases give incorrect outputs
  - But NO crashes

Backtrace for Noncrashing Failures

- Tag precision pairs for each function
- Locate the trace of precision boosts

We wish to know the backtrace for such noncrashing failures!
Analogy to Backtraces for Crashes

backtrace from GDB

# 0x0067a2 in __libc_start_elf () from /lib64-linux.so.2
# 1 0x0072e59 in RAISE () from /lib64/libc.so.6
# 2 0x00730862 in abort () from /lib64/libc.so.6
# 3 0x008048b0 in omatch at file c:343
# 4 0x00804f29 in amatch at file c:446
# 5 0x00804906 in subline at file c:492
# 6 0x008049139 in change at file c:514
# 7 0x00804923c in main at file c:552

Discriminative Subgraphs as Indicator of Faults [Di Fatta et. al, SOQUA’06]

- Find discriminative subgraphs directly
- Discriminative subgraphs
  - Subgraphs frequent in failing executions, but not that frequent in passing ones

Tutorial Outline

- Data Mining for Software Bug Detection
  - Frequent pattern mining
- Automated Debugging in Software Programs
  - From frequent patterns to software bugs
  - Statistical debugging
- Automated Debugging in Computer Systems
  - Automated diagnosis of system misconfigurations
  - Performance debugging
- Conclusion

Bug Isolation Architecture

Courtesy of Ben Liblit
A Running Example

```c
void subline(char *lin, char *pat, char *sub)
{
    int i, lastm, m;
    lastm = -1;
    i = 0;
    while((lin[i] != ENDSTR)) {
        m = amatch(lin, i, pat, 0);
        if (m >= 0) {
            lastm = m;
        }
        if ((m == -1) || (m == i)) {
            i = i + 1;
        } else
            i = m;
    }
}
```

130 of 5542 test cases fail, no crashes

Profile Executions as Vectors

<table>
<thead>
<tr>
<th>Predicate</th>
<th># of true</th>
<th># of false</th>
</tr>
</thead>
<tbody>
<tr>
<td>lin[i] != ENDSTR == true</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Ret_amatch &lt; 0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Ret_amatch == 0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Ret_amatch &gt; 0</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Two passing executions

| 5 | 1 | 1 | 5 | 1 | 5 | 4 | 2 | 1 | 5 | 2 | 4 | 1 | 5 |

One failing execution

| 9 | 1 | 1 | 9 | 1 | 9 | 8 | 2 | 8 | 2 | 2 | 8 | 1 | 9 |

- Extreme case
  - Always false in passing and always true in failing ...
- Generalized case
  - Different true probability in passing and failing executions

Instrumentation in General

- Predicates: Propositions about any program properties
  - E.g., idx < BUFSIZE, a + b == c, foo() > 0 ...
  - Every evaluation gives either true or false
  - Each can be evaluated multiple times during one execution

- Execution report
  - Vector of predicate counters (fold along time dimension)
  - Success/failure outcome label

- Statistical debugging
  - Locate fault-relevant predicates

Bug Isolation Using Predicate Elimination
[Liblit et al., PLDI’03]

- Locate PERFECT predicates
  - Ever true only in failing executions
  - Perfectly correlated with failing executions

- Elimination strategy
  - Disregard counters that are zeros on all runs
  - Disregard counters that are zero on all failed runs
  - Disregard any counter that has a non-zero value on any successful run

- Capture a category of easy bugs
  - When a predicate is true, it always fails
Winnowing Down to the Culprits

- Ccrypt 1.2 crashing Bug
  - XXX lines of C code
  - Crash on receiving EOF when asking confirming overwriting an existing file
- 570 call sites, $3 \times 570 = 1710$ counters
  - 1569 are always zero
  - 141 remain
  - 139 are nonzero on some successful run
  - Not much left!

```
file_exists() > 0
xreadline() == 0
```

Not much left!

What if no perfect predicates exist?

- Nothing else but a feature selection problem
  - Logistic regression with L1 normal regularization
    [Liblit et al., PLDI'03]
  - Predict success/failure as function of counters
  - Penalty factor forces most coefficients to zero
    - Large coefficient $\Rightarrow$ highly predictive of failure

Feature Selection in One-sided classification [Zheng et al., NIPS'03]

- Limitations of logistic regression-based debugging
  - It places equal weight to false positives and false negatives
  - In debugging, we care more about failures
  - More interested in predicates that accurately predict failures

- Solution
  - Different penalties for false positives and false negatives

- Result
  - Similar debugging power, but more flexible
  - Improved theoretical justification
Feature Ranking through Simple Statistics

[Liblit et al, PLDI’05]

- Correlation analysis
  - Context(P) = Prob(fail | P ever evaluated)
  - Failure(P) = Prob(fail | P ever evaluated as true)
  - Increase(P) = Failure(P) – Context(P)

How more likely the program fails when a predicate is ever evaluated true

Liblit05 in Illustration

Context(P) = Prob(fail | P ever evaluated)
= 4/10 = 2/5

Failure(P) = Prob(fail | P ever evaluated as true)
= 3/7

Increase(P) = Failure(P) – Context(P)
= 3/7 – 2/5 = 1/35

More Information, More Effective [Liu et al., FSE’05]

- SOBER: Statistical Model-Based Fault Localization
  - What is probability for a predicate to be true?
  - Liblit05: Ever true?
- Evaluation bias
  - Estimated head probability from every execution
  - Specifically,
    \[ X = \frac{n_t}{n_t + n_f} \]
    where \( n_t \) and \( n_f \) are the number of true and false evaluations in one execution.
  - Defined for each predicate and each execution

Divergence in Head Probability

- Multiple evaluation biases from multiple executions
  - \( P = (p_1, p_2, ..., p_n) \Rightarrow S_P = (X_1', X_2', ..., X_n') \sim f(X | \theta_P) \)
  - \( F = (f_1, f_2, ..., f_m) \Rightarrow S_F = (X_1, X_2, ..., X_m) \sim f(X | \theta_f) \)
- Evaluation bias as generated from models
  - \[ f(X | \theta_P) \]
  - \[ f(X | \theta_f) \]
**Major Challenges**

- No closed form of either model
- No sufficient number of failing executions to estimate $f(X | \theta_f)$

**SOBER: A Hypothesis Testing-Based Approach**

- Null hypothesis $\mathcal{H}_0: f(X | \theta_p) = f(X | \theta_f)$, where $\theta = (\mu, \sigma^2)$
- From $S_T = (X_1, X_2, \ldots, X_m)$,
  
  $Y = \sum_{i=1}^{m} \frac{X_i}{m} \sim N(\mu, \sigma^2/m) = N(\mu_p, \sigma^2_p/m)$

- The likelihood of observing $S_T$ under $\mathcal{H}_0$ is
  
  $f(Y | \theta_p) = \frac{\sqrt{m}}{\sigma_p} \varphi(Z)$, where $Z = \frac{Y - \mu_p}{\sigma_p/\sqrt{m}} \sim N(0, 1)$

- Smaller $f(Y | \theta_p) \Rightarrow$ More likely $\mathcal{H}_0$ is not true $\Rightarrow$ larger divergence between $f(X | \theta_p)$ and $f(X | \theta_f) \Rightarrow P$ is more fault-relevant

- Fault relevance score of predicate $P$ is
  
  $s(P) = \log \left( \frac{\sigma_p}{\sqrt{m} \varphi(Z)} \right)$

**SOBER in Illustration**

**Comparison at a Glance**

Siemens program suite
- 130 buggy versions of 7 small ($<$700LOC) programs
- All semantic bugs

What percentage bugs can be located with no more than $\alpha \%$ of code examination

T-Score $\leq 20\%$ is meaningful
BayesDebug: How Bayesians Debug [Liu et al., ICDM’06]

- Motivation
  - SOBER is essentially a frequentist approach
  - What is the Bayesian counterpart?

- Practical concern
  - What if there is only one failing case
    - In development for example
  - Can we localize the bug with only one passing and one failing execution?
    - The practical setting for manual debugging
    - Neither Liblit05 nor SOBER work well due to the small sample size

How Bayesian Debug: Belief Contrast

- Manual debugging
  - Contrast what is seen against what it should be
  - “What it should be” is in the mind

- Principles
  - A notion of correctness: “what it should be”
  - A notion of incorrectness: “What is observed wrong”
  - Notion contrast!
  - So can we emulate the notions of correctness and incorrectness, and then contrast them in computers?

Notion Emulation and Contrast

Notions: The belief in head probability

- Each evaluation is a Bernoulli trial with head probability \( \theta \)
- \( X = (X_1, X_2, \ldots, X_n) \), where \( X_i \sim Bernoulli(\theta) \) and \( \theta \in [0, 1] \)
- What is the belief in \( \theta \) after observing \( X \)?

A Bayesian approach to notion emulation

- Before program starts, \( Pr(\theta) \) is a uniform distribution
- When program terminates,

\[
Pr(\theta|X) = \frac{Pr(X|\theta)Pr(\theta)}{Pr(X)}
\]

Notion contrast

What is the belief divergence between \( Pr(\theta|X) \) and \( Pr(\theta|X_p) \)?

An Illustrative Example

<table>
<thead>
<tr>
<th>Time</th>
<th>A Passing Run</th>
<th>A Failing Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 true, 0 false</td>
<td>0 true, 0 false</td>
</tr>
<tr>
<td>1</td>
<td>3 true, 1 false</td>
<td>1 true, 3 false</td>
</tr>
<tr>
<td>2</td>
<td>8 true, 2 false</td>
<td>Time</td>
</tr>
</tbody>
</table>

A Bayesian

Notion Contrast
**Posterior at Program Termination**

- Prior at program start: $\text{Beta}(1, 1)$, a uniform distribution
- Posterior at the end of the passing execution

$$
\theta_p \sim f(\theta | X_p) = \text{Beta}(1 + \sum_{i=1}^n X_i, n + 1 - \sum_{i=1}^n X_i),
$$

where $X_p = (X_1, X_2, \cdots, X_n)$.
- Posterior at the end of the failing execution

$$
\theta_f \sim f(\theta | X_f) = \text{Beta}(1 + \sum_{i=1}^m X'_i, m + 1 - \sum_{i=1}^m X'_i),
$$

where $X_f = (X'_1, X'_2, \cdots, X'_m)$.

---

**An Analytical Form of the Notion Divergence**

**Theorem (Divergence between Beta Posteriors)**

The KL-divergence between $f(\theta_p | X_f)$ and $f(\theta_f | X_p)$, denoted as $KL(\theta_f || \theta_p)$, is

$$
\ln \frac{B(a, b)}{B(c, d)} + (c - a)[\Psi(c) - \Psi(c + d)] + (d - b)[\Psi(d) - \Psi(c + d)],
$$

where $\Gamma(x) = \int_0^{\infty} t^{x-1}e^{-t}dt$, and $\Psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$.

Computational efficient although looks complex

- Both $\Gamma(x)$ and $\Psi(x)$ can be computed in $O(1)$ time

---

**A Glimpse of the Comparison**

- Generally, BayesDebug is better than Liblit05
- But sometimes, Liblit05 can be better than BayesDebug
- No uniform “good” test cases

---

**Multiple bugs [Zheng et al., ICML’06]**

- **Motivation**
  - Unknown number of bugs hidden in release programs
  - About 63,000 known bugs in Windows 2000 at first release
  - Parallel debugging

- **Challenges**
  - Methods discussed above were designed with single-bug situation in mind
  - Iterative approach is an OK approach, but not parallelizable
  - Super-bug and sub-bug predictors are mixed together

- **Idea**
  - Simultaneous identification of multiple bugs through co-clustering on the predicate x failing runs matrix
  - Predicates should group by the runs that they predict
  - Runs should group by the predicates that predict them
An Illustration of the Problem

- Debugging MOSS via Zheng04
  - MOSS is a plagiarism detection tool, 6001 LOC
  - Re-instate 9 historical bugs for testing purpose

Predicates Selected by the Single-Bug Algorithm
1. \((p+\text{passage\_index})\rightarrow\text{last\_line} < 4\)
2. \((p+\text{passage\_index})\rightarrow\text{first\_line} < 1\)
3. \(i > 20\)
4. \(i > 26\)
5. \((p+\text{passage\_index})\rightarrow\text{last\_line} < 1\)
6. \(i > 23\)
7. \((p+\text{passage\_index})\rightarrow\text{last\_line} == \text{next}\)
8. \(i > 22\)
9. \(i > 25\)
10. \(i > 28\)

MOSS Results

Predicates Selected by the Single-Bug Algorithm
1. \((p+\text{passage\_index})\rightarrow\text{last\_line} < 4\)
2. \((p+\text{passage\_index})\rightarrow\text{first\_line} < 1\)
3. \(i > 20\)
4. \(i > 26\)
5. \((p+\text{passage\_index})\rightarrow\text{last\_line} < 1\)
6. \(i > 23\)
7. \((p+\text{passage\_index})\rightarrow\text{last\_line} == \text{next}\)
8. \(i > 22\)
9. \(i > 25\)
10. \(i > 28\)

K-means clusters of MOSS failures

- Outcome: runs are clustered by program usage modes, not failures modes

Sub-bug predictors

Super-bug predictors
Simultaneous Identification of Multiple Bugs via Co-clustering

- Co-clustering: Clustering along two dimensions simultaneously
  - Co-clustering of words and documents [Dhillon KDD’01]
  - Let $A$ be the conventional term-document matrix
  - Apply spectral clustering with the following matrix as the adjacency matrix
    $\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$

- What is the $A$ in the debugging setting?

The Term-Doc Matrix in Debugging

- $A_{ij}$ is the truth probability of predicate $i$ in run $j$
- This is not what is observed, but a hidden variable to be estimated.

\[
\begin{align*}
N &= \text{# times reached} \\
X &= \text{# times true} \\
M &= \text{# times observed} \\
Y &= \text{# times obs. and true} \\
N &\sim \beta \text{Pois}(\lambda) + (1-\beta)\delta(N=0) \\
X &\sim \gamma_1 \text{Bin}(\alpha, N) + \gamma_2 \delta(X=0) + (1-\gamma_1 - \gamma_2)\delta(X=N) \\
M &\sim \text{Bin}(d, N) \\
Y | N, M, X &\sim \text{Hypergeom}(N, M, X)
\end{align*}
\]

Estimate $\lambda, \beta, \alpha, \gamma$

Compute $P(X_{ij} > 0 | M_{ij}, Y_{ij})$ for pred $i$ in run $j$

Bi-Clustering Voting Scheme

A good bug predictor accounts for failed runs, not successful runs; its complement accounts for successful runs, not failed runs.

Quality of pred $i$ and its complement: $Q_i = \frac{F_i}{S_i}, \quad \bar{Q}_i = \frac{1}{Q_i}$

Inferred truth probability: $A_{ij} = P(X_{ij} > 0 | Y_{ij}, M_{ij})$

Votes from F runs: $F_i = \sum_{j \in F} A_{ij} \frac{R_{ij}}{\sum_h R_{ih}}, \quad \bar{F}_i = \sum_{j \in F} A_{ij} \frac{R_{ij}}{\sum_h R_{ij}}$

Votes from S runs: $S_i = \sum_{j \in \bar{F}} A_{ij} \frac{R_{ij}}{\sum_h R_{ij}}, \quad \bar{S}_i = \sum_{j \in \bar{F}} A_{ij} \frac{R_{ij}}{\sum_h R_{ij}}$

Vote from run $j$: $R_{ij} = \begin{cases} A_{ij} Q_i & \text{if } j \in F \\ A_{ij} \bar{Q}_i & \text{if } j \in \bar{F} \end{cases}, \quad \bar{R}_{ij} = \begin{cases} A_{ij} Q_i & \text{if } j \in \bar{F} \\ A_{ij} \bar{Q}_i & \text{if } j \in F \end{cases}$

Attempt II: Cluster Predicates

Spectral clusters of mass predicates

Outcome: sampling decimates predicate correlations; super-bug predictors blur clusters
Multi-Bug Algorithm

1. Estimate truth probabilities
2. Initialize all predicate quality scores to one
3. Iterate update equations until convergence
4. Run \( j \) votes for predicate \( i^* = \arg\max_i R_{ij} \)
5. Rank predicates by their votes

Result on MOSS: Top-9 Predicates

Debugging with Latent Topic Models
[Andrzejewski et al., ECML’07]

- Analogy between executions and documents
  - Predicates \( \leftrightarrow \) Vocabulary
  - Predicate counts \( \leftrightarrow \) Word counts
  - Program run \( \leftrightarrow \) Bag-of-words document
  - Bug manifestation \( \leftrightarrow \) Latent topic analysis

- Ideas
  - Passing executions are generated from usage models
  - Failing executions are generated from a mixture of usage models and bug models
  - Can we extract bug models by modeling passing and failing executions simultaneously

Latent Dirichlet Allocation (LDA)

- Extracting bug topics from failing executions directly?
  - Not very successful
    - Most of failing executions are still correct
    - Most failing executions quickly exit after failure occurs
    - Usage patterns overwhelm bug patterns
  - Need usage models to account for the dominant usage patterns
\[ \Delta \text{LDA} \]

- **w**: words
- **z**: topics
- \( \theta \): topic weights
- \( \phi^u \), \( \phi^s \): usage topic multinomials
- \( \beta^u, \alpha^s \): hyperparameters
- \( \beta^b, \alpha^f \): bug topic multinomials
- \( \phi^b \): bug topic multinomials
- \( \beta^b, \alpha^f \): hyperparameters

**Outcome Flag**

- \( \alpha^o \): outcome flag
- \( \phi^o \): outcome multinomials
- \( \beta^o, \alpha^f \): hyperparameters

**Diagrams**

1. Diagram showing the process of \( \Delta \text{LDA} \) with labeled nodes and edges.
2. Diagram showing the process of \( \Delta \text{LDA} \) with labeled nodes and edges.
3. Diagram showing the process of \( \Delta \text{LDA} \) with labeled nodes and edges.
4. Diagram showing the process of \( \Delta \text{LDA} \) with labeled nodes and edges.
Outputs from ΛLDA

- Topic weight vector for each failure run
  - Cluster failure runs
- Bug topics \( P(w|z) \)
  - Find top predicates characterizing each bug topic for debugging purpose
    - Calculate \( P(z|w) \) via Bayesian theorem
    - For each bug topic \( z \), sort predicates \( w \) by \( P(z|w) \)
- Usage topics
  - Uncovers usage modes for program understanding

Group Failing Runs by Topic Weights

Exif is an open-source image manipulation tool, 10,588 LOC

| Rank | \( p(z|w) \) | Predicate \( w \) | Expert opinion |
|------|-------------|----------------|----------------|
| 1    | 0.99977     | jpeg-data.c:434  jpeg_data_set_exif_data() | Direct result |
| 2    | 0.69517     | jpeg-data.c:436  jpeg_data_set_exif_data() | Direct result |
| 3    | 0.56748     | jpeg-data.c:207  jpeg_data_load_data() | Smoking gun |

Unresolved Problems

- How to set the number of bug topics
  - Set equal to the number of known bugs
- How to set the number of usage topics
  - Set equal to the number of use cases, i.e., in how many ways the program is run
- Use non-parametric Bayesian methods to estimate the number of bugs?
Summary on Automated Debugging

- Automated debugging, statistical debugging in particular, becomes more and more alike a machine learning problem.
- Dependencies are still barely modeled so far
  - Predicates are apparently not independent
  - The dependencies are readily available from SE and PL researchers
  - Debugging on PDGs
- Executions as more regular documents
  - Like documents with many repeated statements and paragraphs
  - We DO know the underlying generation model for executions while not for documents
  - Essentially, modeling executions should be easier than modeling documents, but unfortunately not the case so far.

Tutorial Outline

- Data Mining for Software Bug Detection
  - Frequent pattern mining
- Automated Debugging in Software Programs
  - From frequent patterns to software bugs
  - Statistical debugging
- Automated Debugging in Computer Systems
  - Automated diagnosis of system misconfigurations
  - Performance debugging
- Conclusion
Motivation

- Technical support contributes 17% of the total cost of ownership (TCO) of today’s desktop PCs, according to an independent study by the Tolly Group in 2000. [http://www.nocomsoftware.se/p9138/files/products_tolly_tco.pdf](http://www.nocomsoftware.se/p9138/files/products_tolly_tco.pdf)
- Most application malfunctions stem from misconfigurations
  - Conflicting changes of the shared configuration data
  - Uninstallation is unclean
  - …
- The system is fine, but just too slow with a long response time
- Automated diagnosis through mining computer system data
  - Computer configuration data
    - More than 200,000 registry entries Windows XP
    - Huge volumes of configuration files in *nix systems
  - Computer running state data
    - CPU usage, disk usage, network usage, temperature …

Tutorial Outline

- Data Mining for Software Bug Detection
  - Frequent pattern mining
- Automated Debugging in Software Programs
  - From frequent patterns to software bugs
  - Statistical debugging
- Automated Debugging in Computer Systems
  - Automated diagnosis of system misconfigurations
  - Performance debugging
- Conclusion

STRIDER [Wang et al., LISA’03]

Solution-query phase

- The program keeps failing
- It was working
- Now it doesn’t work
- Support Articles
- Config Action UI
- App Info Doc
- Support Database Lookup
- Ownership Mapping

Filtered & Ranked Candidate Set

Narrow-down phase

- Tracing
- State Diff
- Noise Filtering
- Intersection
- State Ranking

PC Genomics Database

Courtesy of Yi-Min Wang
PeerPressure Troubleshooting [Wang et al., OSDI’04]

- Golden state in the mass
  - An application runs well on most machines
  - Let friends help you out
- Problem solved
  - Diagnose app failures due to single-entry problem
  - Eliminate the need of manual identification of healthy state
- Not fully automated yet
  - Users still need to reproduce the failure

PeerPressure Architecture

Intuition of Registry Ranking

<table>
<thead>
<tr>
<th>EOI</th>
<th>Mine</th>
<th>S1’s</th>
<th>S2’s</th>
<th>S3’s</th>
<th>S4’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e3</td>
<td>57</td>
<td>4</td>
<td>0</td>
<td>100</td>
<td>34</td>
</tr>
</tbody>
</table>

- Is e1 a misconfiguration entry?
- Is e2 a misconfiguration entry?
- Is e3 a misconfiguration entry?
- The ranking is: e1 > e3 > e2

Intuitive judgement

- e1 is likely correct
- e2 is likely wrong
- e3 is likely not wrong

Ranking algorithm should implement such intuition
Ranking Algorithm

- t suspected registry entries, how to rank them according to their sickness?

\[
P(S) = \frac{1}{t} \quad P(H) = 1 - \frac{1}{t}
\]

\[
P(S | V) = \frac{P(V | S)P(S)}{P(V | S)P(S) + P(V | H)P(H)}
\]

\[
P(V | S) = \frac{1}{c} \quad P(V | H) = \frac{m + n}{N + cn}
\]

Registry Ranking

- Calculate the ranking score for each entry

\[
P(S | V) = \frac{N + cn}{N + cnt + cm(t - 1)}
\]

- Rank them in descending order

- Let operators examine the ranking list from the top down

A Close Look at Selected Problems

<table>
<thead>
<tr>
<th>Case</th>
<th>Rank</th>
<th>Ties</th>
<th># of Suspects</th>
<th>Corroboration</th>
<th># of Matches</th>
<th># of Snippets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>System Restore</td>
<td>1</td>
<td>0</td>
<td>1350</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>JPG</td>
<td>16</td>
<td>0</td>
<td>1779</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Outlook</td>
<td>1</td>
<td>0</td>
<td>37</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>IE Passwords</td>
<td>1</td>
<td>0</td>
<td>135</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Media Player</td>
<td>1</td>
<td>0</td>
<td>182</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>IE</td>
<td>1</td>
<td>0</td>
<td>1777</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>IE Proxy</td>
<td>1</td>
<td>0</td>
<td>1777</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>IE Offline</td>
<td>1</td>
<td>0</td>
<td>1230</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Taskbar</td>
<td>1</td>
<td>0</td>
<td>64</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>Network Connections</td>
<td>2</td>
<td>0</td>
<td>551</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>Folder Details-Click</td>
<td>2</td>
<td>1</td>
<td>26308</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>Outlook Express</td>
<td>3</td>
<td>0</td>
<td>482</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>Counted Start-Executables</td>
<td>1</td>
<td>0</td>
<td>237</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>Speedcontrol</td>
<td>1</td>
<td>0</td>
<td>105</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>IE Menu bar</td>
<td>1</td>
<td>2</td>
<td>3990</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>IE Favorites</td>
<td>2</td>
<td>0</td>
<td>3200</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>Sound Problem</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>IE New Window</td>
<td>1</td>
<td>0</td>
<td>853</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>Yahoo Tool bar</td>
<td>1</td>
<td>0</td>
<td>853</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>Media Player in IE</td>
<td>9</td>
<td>0</td>
<td>5483</td>
<td>65</td>
<td>0</td>
</tr>
</tbody>
</table>

Localization Effectiveness and How Hard the Problem is

Tutorial Outline

- Data Mining for Software Bug Detection
  - Frequent pattern mining

- Automated Debugging in Software Programs
  - From frequent patterns to software bugs
  - Statistical debugging

- Automated Debugging in Computer Systems
  - Automated diagnosis of system misconfigurations
  - Performance debugging

- Conclusion

Performance Debugging

- Motivation
  - The server has a longer response time than expected, why?

- Correlating low-level system states to high-level performance compliance [Cohen et al., OSDI'04]
  - System state: CPU utilization, disk reads/writes, swap space, …
  - Performance compliance: is response time short enough

- Approaches
  - Induce an interpretable classifier
  - Diagnosis as interpreting the classifier
  - Performance prediction as applying the classifier

Tree-Augmented Bayesian Network

- Naïve Bayes Classifier
  - Independent attributes conditioned on class labels

- Tree-Augmented Bayesian Network
  - Every attribute has at most one more parent besides class variable
  - Account for dependencies between attributes
  - Tree-structure renders efficient search of dependence structures
  - Great interpretability

Performance Debugging Framework

- Caveats: Correlation, but not causality
- Indexing of performance states for automated diagnosis [Cohen et al., SOSP'05]
Mining into Computer Systems

- Huge volume of data from computer systems
  - Persistent state interactions, event logs, network logs, CPU usage, ...
- Mining system data for ...
  - Reliability
  - Performance
  - Manageability
  - ...
- Challenges in data mining
  - Statistical modeling of computer systems
  - Online, scalability, interpretability ...

Tutorial Outline

- Data Mining for Software Bug Detection
  - Frequent pattern mining
- Automated Debugging in Software Programs
  - From frequent patterns to software bugs
  - Statistical debugging
- Automated Debugging in Computer Systems
  - Automated diagnosis of system misconfigurations
  - Performance debugging
- Conclusion

Conclusion

- Ubiquitous computing demands reliable software
- Mining for software reliability
  - Mining program source code/version histories to find bugs
  - Mining program runtime data to locate why an execution fails ...
  - Mining system snapshots to diagnose misconfigurations and performance problems
- An active and rewarding research area
  - International Workshop on Mining Software Repositories since 2004
  - SIGCOMM Workshop on Mining Network Data since 2005
  - Systems and Machine Learning Workshop since 2006

Software-Informatics

- Software systems are similar to bio-systems
  - Various components working together to act normally
  - But much simpler!
    - Software is programmed by us in years
    - Bio-system is programmed by God in billions of years
- Software-informatics
  - Data is easier to collect
    - Software is easier to instrument and measure
    - Software is easier to manipulate
  - Computer scientist is well equipped for software-informatics
    - Most computer scientists know software
    - Easier to devise appropriate mining algorithm since the process of data generation is explicit
    - Mining result is easier to interpret
Rep Publications (Static Bug Detection)

- Benjamin Livshits and Thomas Zimmermann, DynaMine: Finding Common Error Patterns by Mining Software Revision Histories, ESEC/FSE 2005

Rep Publications (Automated Debugging)

- Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan, Scalable Statistical Bug Isolation, PLDI 2005
- Chao Liu, Long Fei, Xifeng Yan, Jiawei Han and Samuel Midkiff, Statistical Debugging: A Hypothesis Testing-Based Approach, IEEE TSE 2006
- Chao Liu, Zeng Lian and Jiawei Han, How Bayesians Debug, ICDM 2006
- Chao Liu, Xifeng Yan and Jiawei Han, Mining Control Flow Abnormality for Logic Error Isolation, SDM 2006
- Chao Liu, Xifeng Yan, Long Fei, Jiawei Han and Samuel Midkiff, SOBER: Statistical Model-Based Bug Localization, ESEC/FSE 2005
- Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken, Statistical Debugging: Simultaneous Identification of Multiple Bugs, ICML 2006
- Alice X. Zheng, Michael I. Jordan, Ben Liblit, and Alex Aiken, Statistical Debugging of Sampled Programs, NIPS 2003

Rep Publications (System Debugging)

- Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, Correlating instrumentation data to system states: A building block for automated diagnosis and control, OSDI 2004
- Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, Armando Fox, “Capturing, indexing, clustering and retrieving system history,” SOSP 2005
- Emre Kiciman, Dave Maltz, John Platt, and Moises Goldszmidt, Mining Web Logs to Debug Distant Connectivity Problems, In ACM SIGCOMM Workshop on Mining Network Data (MineNet-06), Pisa, Italy, Sep 15, 2006
- John Platt, Emre Kiciman and Dave Maltz, Fast Variational Inference for Large-scale Internet Diagnosis, NIPS 2007

Q & A

Thank You!