PROGRAMMING WITH MILLIONS OF EXAMPLES

Alon Mishne    Eran Yahav
Technion, Israel
Components are Prevalent

HIBERNATE

JFACE

Struts

eclipse

Apache Commons
http://commons.apache.org/
tapestry

SWT

jgraph
visualize everything

GRAILS
Component APIs are Complicated

There is only one thing more painful than learning from experience and that is not learning from experience.
– Archibald MacLeish
Temporal API Specifications

```java
java.nio.channels.SocketChannel {
    boolean connect(SocketAddress remote)
    int read(ByteBuffer dst)
    int write(ByteBuffer src)
    SelectableChannel configureBlocking(boolean block)
    boolean finishConnect()
    boolean isBlocking()
    void close()
    ...
}
```

- Legal interactions with a component
- What methods could be called at every internal state
Temporal API Specifications

java.nio.channels.SocketChannel (partial spec)

- Legal interactions with a component
- What methods could be called at every internal state
Examples are Prevalent
Examples are Prevalent

Retrieve column names from java.sql.ResultSet

With `java.sql.ResultSet` is there a way to get a column's name as a `String` by using the column's index? I had a look through the API doc but I can't find anything.

3 Answers

See ResultSetMetaData

```java
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM TABLE2");
ResultSetMetaData rsmd = rs.getMetaData();
String name = rsmd.getColumnNome(1);
```

and you can get the column name from there.
Challenge

Can we leverage the vast number of component usage examples to make it easier for programmers to write code using the component?
Connection c = new Conn();
???
ResultSet r = ???
while (?) {
    …
}

Connection c = new Conn();
Statement s = c.createStatement();
ResultSet r = s.executeQuery(...);
while (r.next()) {
    …
}
Mining Temporal Specifications

- Extract temporal specification from the program

Applications
- Program understanding
- Regression
- Deviant behaviors
- Specs for verification
- …
Approaches for Mining Temporal Specifications

- **Component-side mining**
  - Infer usage from component implementation
  - Relies on error conditions in component implementation

- **Client-side mining**
  - Infer usage from existing clients using the component

Real usage scenarios ↔ Permitted scenarios

- **Client-side mining**
  - connect; read; close;
  - connect; write; close;
  - connect; write; write; close;

- **Component-side mining**
  - connect; close;
  - close;
  - close

...
Dynamic vs. Static Specification Mining

- **Dynamic**
  - Mine specification from representative executions
  - Requires running the program (with varying inputs)
  - Incomplete coverage of behaviors

- **Static**
  - Analyze the program without running it
  - Covers all client behaviors

- Reality: the amount of code available for inspection vastly exceeds the amount of code amenable to automated dynamic analysis
PRIME Approach

- Static client-side specification mining
- Bad news: this is hard
- Good news: we can still make it work
Dimensions of Synthesis

- **Program**
- **Schema**
- **Input** "program"
- **Examples**
- **specification**
  - none
  - assertion
- **Synthesis Algorithm**
- **static analysis and machine learning**
- **code snippet or API spec**
- **correctness guarantee**
Example

How should I use a `java.nio.channels.SocketChannel`?
example1() {

    SocketChannel sc = SocketChannel.open();

    sc.configureBlocking(false);

    sc.connect();

    sc.finishConnect();

    ByteBuffer dst = ...;

    sc.read(dst);
}

Analyzing a Single Code Sample
void example2() {
    Collection<SocketChannel> chnls = createChannels();
    for (SocketChannel sc : chnls){
        sc.connect(new ...);
        while (!sc.finishConnect()) { /* ... wait for connection ... */ }
        if (?) { receive(sc); } else { send(sc); }
    }
    closeAll(channels);
}

Collection<SocketChannel> createChannels() {
    List<SocketChannel> list = new LinkedList<SocketChannel>();
    list.add(createChannel(" ", 80));
    //... more channels added to list ...
    return list;
}

SocketChannel createChannel(String hostName, int port) {
    SocketChannel sc = SocketChannel.open();
    sc.configureBlocking(false);
    return sc;
}
void receive(SocketChannel x) {
    FileInputStream fos = new ...
    ByteBuffer dst = ...
    int numBytesRead = 0;
    while (numBytesRead >= 0) {
        numBytesRead = x.read(dst);
        fos.write(dst.array());
    }
    fos.close();
}

void send(SocketChannel x) {
    for (?) {
        ...
    }
}

void example2() {
    Collection<SocketChannel> chnls = createChannels();
    for (SocketChannel sc : chnls) {
        sc.connect(new ...);
        while (!sc.finishConnect()) { /* ... wait */ }
        if (?) { receive(sc); } else { send(sc); }
    }
    closeAll(channels);
}

void closeAll(Collection<SocketChannel> chnls) {
    for (SocketChannel sc : chnls) {
        sc.close();
    }
}
SocketChannel createChannel (...) {
    SocketChannel sc = SocketChannel.open();
    sc.configureBlocking(false);
    return sc;
}

void example() {
    Collection<SocketChannel> chns = createChannels();
    for (SocketChannel sc : chns) {
        sc.connect(new ...);
        while (!sc.finishConnect()) {
            if (?) { receive(sc); }
            else { send(sc); }
        }
        closeAll(channels);
    }
}

void receive(SocketChannel x) {
    ...
    while (numBytesRead >= 0) {
        numBytesRead = x.read(dst);
        fos.write(dst.array());
    }
} ...
SocketChannel Specification

(Partial specification)
Challenges

- **Partial programs**
  - Program fragments
  - Missing information
  ➔ Support mining specifications with partial information

- **Dynamically allocated objects**
  - Unbounded number of objects
  - Aliasing
  - Objects flow through complex heap-allocated data structures
  ➔ Heap abstraction

- **Unbounded length of histories**
  - History (event sequence) observed for an object might be unbounded
  ➔ History abstraction

- **Noise**
  - Analysis imprecision and/or incorrect client programs
  ➔ Noise reduction
Overview

Code search

Abstract Trace Collection

Summarization

Overapproximation of API usage in the search results

Library

Candidate API specifications
Abstract Trace Collection

- The French Recipe for Abstract Interpretation [Cousot&Cousot77]

- Abstraction
  - Abstract state provides a bounded description of possible program states at a program point

- Abstract Transformers
  - Conservatively represent the effect of statements on abstract states

- Exploration
  - Compute the possible abstract states at each program point by fixed-point iteration
Abstraction

- Abstract state is a set of abstract values (disjunction)
- Abstract value is a pair (conjunction)
  - Heap abstraction: abstracts unbounded heap
  - History abstraction: abstracts unbounded sequences of operations
Abstraction

- Abstract state is a set of abstract values (disjunction)
- Abstract value is a pair (conjunction)
  - Heap abstraction: abstracts unbounded heap
  - History abstraction: abstracts unbounded sequences of operations
Heap Abstraction – Take I

- Divide the heap into a **fixed partition** based on allocation site
- All objects **allocated at the same** program point represented by a single “abstract object”
Abstract Transformers – Take I

```
sc.configureBlocking();
```

...
SocketChannel `createChannel` (...)
{
    SocketChannel `sc = SocketChannel.open(); // AS1`
    sc.configureBlocking(false);
    return sc;
}

void `example()` {
    Collection<SocketChannel> `chnls = createChannels();`
    for (SocketChannel `sc : chnls`){
        sc.connect(new ...);
        while (!sc.finishConnect()) {...}
        if (?) { receive(sc); } else { send(sc); }
    }
    closeAll(channels);
}

void `receive(SocketChannel x)` {
    ...
    while (numBytesRead >= 0) {
        numBytesRead = x.read(dst);
        fos.write(dst.array());
    }
    ...
}}
Refined Heap Abstraction

- Heap data for an “abstract object” o
  - unique = true
    - abstract value represents a single object
  - must = \{x.f\}
    - the access path x.f must point to o
  - mustNot = \{y.g\}
    - the access path y.g must not point to o
  - ...

- Dynamic partition

- Must points-to information allows strong updates

```
sc=open()
<AS1, must: \{ sc \},
  ➔

sc.cfg
<AS1, must: \{ sc \},
  ➔
  ➖
  ➔
  <AS1, ➖
```
Objects Are Related

```java
public ResultSet realLifeCreateResultSet(name) {
    DataSource ds = ConnectionFactory.createConnectionFactory();
    Connection c = ds.getConnection();
    PreparedStatement p = c.prepareStatement("select * from "+ name);
    return p.executeQuery();
}
```
Maintaining (some) Object Relations

public void method1(A a) {
    a.connect();
    B b = a.createB();
    b.disconnect();
}
Maintaining (some) Object Relations

public ResultSet realLifeCreateResultSet(String name) {
    DataSource ds = ConnectionFactory.createConnectionFactory();
    Connection c = ds.getConnection();
    PreparedStatement p = c.prepareStatement("select * from " + name);
    return p.executeQuery();
}
History Abstraction

- Abstract history
  - Automaton over-approximating unbounded event sequences

- Quotient-based abstractions for history
  - Automata states which are equivalent w.r.t. a given equivalence relation $R$ are merged
History Abstraction

- Past-Future Abstraction

\[(q_1, q_2) \in R[k_{in}, k_{out}] \text{ if } q_1 \text{ and } q_2 \text{ share both an incoming sequence of length } k_{in}\text{ and an outgoing sequence of length } k_{out}\]

Past 1 Examples – R[1,0]

Future 1 Example – R[0,1]
Abstract Semantics

- Initial abstract history
  - empty sequence automaton
- When an API method is invoked
  - history extended: append event and construct quotient

```java
sc = open
sc.config
sc.connect
while (!sc.finCon) {
  // Past 1 equivalent
  // endof while
} //endof while
```
Are We Done?

- Bounded is great, but not enough

- Merge histories at control flow join points
  - Speed up convergence

- Merge all histories that
  - have identical heap-data, and
  - satisfy a given merge criterion

- Merge: union construction followed by quotient construction
Example: Past Abstraction with Exterior Merge
Dealing with the Unknown

```java
FileComponent fc = new FileComponent();
fc.open();
while(?) {
    fc.read();
}
fc.close();
```

```
example1
FileComponent fc = new FileComponent();
fce.open();
while(?) {
    fc.read();
}
fce.close();
```

```
example2
FileComponent fc = new FileComponent();
fce.open();
foo(fc);
fce.close();
```
Recap: Abstraction Dimensions

Third dimension: different history abstraction, not shown here
Using the Analyzed Samples
Merge Same Type Together

\[ H_0 \xrightarrow{f()} H_1 \xrightarrow{g()} H_2 \xrightarrow{h()} H_3 \]

\[ H_0 \xrightarrow{f()} H_1 \xrightarrow{w()} H_2 \xrightarrow{h()} H_3 \]
Merge All Samples of Same Type
Merging all Samples of Same Type
Merge by Use Case

Use case 1

Use case 2

Use case 3

But how?
Clustering

We define a distance function between samples, then use classic clustering techniques from data mining.
Clustering: Distance Function

Different use-cases typically use different methods

Distance = \sqrt{(1-1)^2 + (1-0)^2 + (0-1)^2 + (1-1)^2} = \sim 1.41
## Clustering Results

<table>
<thead>
<tr>
<th></th>
<th># samples</th>
<th>% samples</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>java.sql.Statement</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>396</td>
<td>18</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>239</td>
<td>60</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>77</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>8</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td><strong>java.sql.Timestamp</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>62</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>47</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>
Sample collection is multithreaded
Specialized partial compiler for analyzing fragments
Weka for clustering
Inter-procedural analysis:
- Dealing with parameters and return values
- Dealing with recursion (both direct and indirect)
Optimizing history representation for scalability
Integrating with Eclipse to provide GUI
Results

- FileOutputStream is the common way to write to a file in Java.
- Even a few samples generate this cluster:
“Apache Commons CLI” is a library used for command-line parsing.

80% of samples belong to the CommandLine class, and 74% of CommandLine’s samples got clustered together into:
Results

- “JDBC”, our pet example, is used for accessing SQL databases.
- 88% of samples from java.sql.DatabaseMetaData got clustered together into:
Results

- “Apache Commons Net”, client implementations for many net protocols
- Analysis results for org.apache.commons.net.time.TimeTCPClient class:
Results

- “JDBC” once again
- Analysis results for java.sql.ResultSet class:
Future Applications

```java
public void method1() {
    File f = new File();
    f.;
    f.write(buffer);
    f.?
}
```

```java
public void method1() {
    File f = new File();
    f.open();
    f.write(buffer);
    f.flush();
    f.close();
}
```

WARNING: java.io.File is only used this way in 0.5% of samples
Future Applications

- Stack-overflow for API usage
  - Questions in English
  - Representative code samples mined automatically
  - Programmers can add results, and rank them

- “Scene Completion”
Opportunities

- Improving the analysis
  - Dealing with pure methods
  - Removing spurious ordering constraints
- Adding other sources for examples
  - Krugle
  - Coders
  - Stackoverflow
  - ...
- Applying to other programming languages
  - e.g., javascript
- Many more...
Summary

- Client-side static specification mining
  - Partial programs
  - Based on flow-sensitive, context-sensitive abstract interpretation
  - Combined domain abstracting both aliasing and event sequences
- Family of abstractions to represent unbounded event sequences
- Summarization algorithms
  - Stitching of partial specifications with unknowns
- Preliminary experimental results
The End