Software Component Protocol Inference

Tao Xie
General Examination Presentation

Dept. of Computer Science and Engineering
University of Washington
6 June 2003
Outline

- Background
- Overview of protocol inference
- Dynamic protocol inference framework
- Static protocol inference techniques
- Future work
- Conclusions
Background

• Software component
  – “defined as a unit of composition with contractually specified interfaces and explicit context dependencies only.” [Szyperski98]

• Component interface
  – Services that the component provides to and requests from other components

• Component interface protocol/component protocol
  – Sequencing constraints on the interface (bi-directional)
Focus

• Components written in OO languages
• Unidirectional protocol

Example: java.util.zip.ZipOutputStream

```java
public class ZipOutputStream
    extends DeflaterOutputStream implements ZipConstants {
    public ZipOutputStream(OutputStream out);
    public static final int DEFLATED;
    public static final int STORED;
    public void close() throw IOException;
    public void closeEntry() throw IOException;
    public void finish () throws IOException;
    public void putNextEntry(ZipEntry e) throws IOException;
    public void setComment(String comment);
    public void setLevel(int level);
    public void setMethod(int method);
    public synchronized void write(byte[] b, int off, int len) throws
        IOException;
}
```
**Informal Documentation**
- from *Java in a Nutshell* [Flanagan97]

| **Once** you have begun an entry with `putNextEntry()`, | you can write the contents of that entry with the `write()` methods. |
| **When** you reach the end of an entry, | you can begin a new one by calling `putNextEntry()` again, or you can close the current entry with `closeEntry()`, or you can close the stream itself with `close()`. |
| **Before** beginning an entry with `putNextEntry()`, | you can set the compression method and level with `setMethod()` and `setLevel()`. |

The constants **DEFLATED** and **STORED** are the two legal values for `setMethod()`. If you use **STORED**, the entry is stored in the ZIP file without any compression.

**If** you use **DEFLATED** [for `setMethod()`], you can also specify the compression speed/strength tradeoff bypassing a number from 1 to 9 to `setLevel()`.
Formal Protocol Specification
- Translated from [Butkevich et al. 00]

- In the form of Finite State Automaton (FSA)

<DEFLATED> putNextEntry, write*, closeEntry?
<DEFLATED>
Why Component Protocol Inference?

• Protocols are useful for correct component usage
  – Documentation
  – Static verification
  – Runtime verification

• But few components have accompanying protocols
Outline

- Background
- Overview of protocol inference
- Dynamic protocol inference framework
- Static protocol inference techniques
- Future work
- Conclusions
Protocol Inference

• Dynamic protocol inference
  – Inputs
    • Traces of method calls in the interface

• Static protocol inference
  – Inputs
    • Component code implementing the interface
    • Client code using the interface
# Overview of Previous Work

<table>
<thead>
<tr>
<th>Previous work</th>
<th>Target lang/sys</th>
<th>Analysis type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whaley et al. [WML02]</td>
<td>Java</td>
<td>Static and Dynamic</td>
<td>FSA</td>
</tr>
<tr>
<td>Reiss et al. [RR01]</td>
<td>Java, C++, and C</td>
<td>Dynamic</td>
<td>FSA</td>
</tr>
<tr>
<td>Ammons et al. [ABL02]</td>
<td>C</td>
<td>Dynamic</td>
<td>FSA</td>
</tr>
<tr>
<td>Cook et al. [CW98]</td>
<td>Software process</td>
<td>Dynamic</td>
<td>FSA</td>
</tr>
<tr>
<td>El-Ramly et al. [ESS02]</td>
<td>Interactive system</td>
<td>Dynamic</td>
<td>Frequently recurring usage patterns</td>
</tr>
<tr>
<td>Lie et al. [LCED01]</td>
<td>C protocol code</td>
<td>Static</td>
<td>FSA-like models to a model checker</td>
</tr>
</tbody>
</table>
Challenges

• Overgeneralization/over-restrictiveness
  – Overgeneralization: accept some illegal sequences
  – Over-restrictiveness: reject some legal sequences

• Separation/composition of constraints
  – e.g. DEFLATED and STORED groups
  – e.g. Concurrent FSAs

• Data-dependent transitions
  – e.g. setMethod(DEFLATED), setMethod(STORED)
  – e.g. pop() when currentSize>0

• Robustness to noise
  – Illegal sequences in traces or client code
  – Method calls without any sequencing constraints
Outline

• Background
• Overview of protocol inference
• Dynamic protocol inference framework
• Static protocol inference techniques
• Future work
• Conclusions
Dynamic Protocol Inference Framework

Trace Collection → Scenario Extraction → Protocol Inference → Protocol Usage

Traces → Scenarios → Protocols
Dynamic Protocol Inference Framework

Trace Collection → Scenario Extraction → Protocol Inference → Protocol Usage

Traces → Scenarios → Protocols
Dynamic Protocol Inference Framework

Trace Collection → Scenario Extraction → Protocol Inference → Protocol Usage

Traces → Scenarios → Protocols
Scenario Extraction

A component usage scenario consists of **interdependent** method calls to a component interface

Why scenario extraction?

- Interleaving independent calls

  Object1
  setMethod
  putNextEntry
  write
  closeEntry
  close

  Object2
  setMethod
  putNextEntry
  write
  closeEntry

- Neighboring independent calls

  Object1
  setMethod
  putNextEntry
  write
  closeEntry
  close

  Object2
  setMethod
  putNextEntry
  write
  closeEntry

- OO program traces

- C program traces
Scenario Extraction from OO Program Traces

• **Group by object** [Reiss et al.]
  – Method calls on the same object
  – A single FSA model for a class

• **Group by member fields** [Whaley et al.]
  – Method calls on the same object
  – Method calls that access the same field
  – \( n \) FSA submodels for a class with \( n \) fields

The **entry field**:
- putNextEntry, write, closeEntry

The **method field**:
- setMethod, putNextEntry
Scenario Extraction from C Program Traces-I

• Arguments and return values are used to group traces [Ammons et al.]

```
fopen()
fprintf(fp,......)
fscanf(fp,......)
fread(...,...,....,fp,......)
fwrite(...,...,....,fp,......)
fclose(fp)
```
Scenario Extraction from C Program Traces-II

- User-specified attributes of an abstract object
  - Definers: `fopen.return; fclose.fp`
  - Users: `fprintf.fp; fscanf.fp; fclose.fp; fread.fp; fwrite.fp`

- Flow dependency analysis

```
fopen():return=0x40, fprintf(fp=0x40), fscanf(fp=0x40), fclose(fp=0x40)
```
Scenario Extraction from C Program Traces-III

- A scenario is a set of function calls related by flow dependences.
  - User-specified scenario seeds and bounded size $N$
  - Scenario: ancestors and descendants of the seed function call

Seed: `fopen(); N=3`
Seed: `fclose(); N=3`
Dynamic Protocol Inference Framework

Trace Collection → Scenario Extraction → Protocol Inference → Protocol Usage

Traces → Scenarios → Protocols
Protocol Inference

• A learning activity
  – Find a protocol
    • explain the given scenarios
    • predict future scenarios.

• Inputs: positive or negative scenarios

• Algorithms
  – $k$-tails Algorithm [Reiss et al.][Ammons et al.][Cook et al.]
  – Separation of state-preserving methods [Whaley et al.]
  – Markov algorithm [Cook et al.]
  – IPM2 algorithm [El-Ramly et al.]
**$k$-tails Algorithm** [Biermann et al. 72]

- A state is defined by what future behavior can occur from it
  - The future (the $k$-tail): the next $k$ method calls
  - Merge two states
    - if they have a $k$-tail in common [Reiss et al.]
    - if one includes all the $k$-tails of the other one [Cook et al.]
**$k$-tails Algorithm Example ($k=2$ [Reiss et al.])**

- `setMethod, putNextEntry, write, write, closeEntry, putNextEntry, write, write, closeEntry, close`
- `setMethod, putNextEntry, write, write, write, closeEntry, close`

**Initial FSA**

**Merge 2-tail of p, w**

**Merge 2-tail of w, w**

**Noise:**
- States with low frequency [Cook et al.]
- Edges with low frequency [Ammons et al.]
Separation of State-Preserving Methods

[Whaley et al.]

• A submodel contains all the methods accessing the same field $f$.
  
  - e.g. putNextEntry, write, closeEntry (the entry field)

  ➢ State-modifying methods
    
    - write $f$; change the object state
    - e.g. putNextEntry, closeEntry

  ➢ State-preserving methods
    
    - only read $f$; not change the state of an object
    - e.g. write
### Submodel Extraction for the entry field

<table>
<thead>
<tr>
<th>Last state-modifying method history</th>
<th>Method call</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>putNextEntry()</td>
</tr>
<tr>
<td>putNextEntry()</td>
<td>write()</td>
</tr>
<tr>
<td>putNextEntry()</td>
<td>write()</td>
</tr>
<tr>
<td>putNextEntry()</td>
<td>closeEntry()</td>
</tr>
<tr>
<td>closeEntry()</td>
<td>putNextEntry()</td>
</tr>
<tr>
<td>putNextEntry()</td>
<td>write()</td>
</tr>
<tr>
<td>putNextEntry()</td>
<td>write()</td>
</tr>
<tr>
<td>putNextEntry()</td>
<td>closeEntry()</td>
</tr>
<tr>
<td>closeEntry()</td>
<td>END</td>
</tr>
</tbody>
</table>

![State Transition Diagram](image)

- Method call sequence:
  - setMethod()
  - putNextEntry(), write(), write(), write(), write(), closeEntry(), close()
Submodels for `zipOutputStream`

Submodel for the `entry` field

Submodel for the `closed` field

Submodel for the `crc` field

Submodel for the `written` field

A single FSA model by 2-tails algorithm
Challenges Revisited

• Overgeneralization/over-restrictiveness
  – Overgeneralization: accept some illegal sequences
  – Over-restrictiveness: reject some legal sequences

• Separation/composition of constraints
  – e.g. DEFLATED and STORED groups
  – e.g. Concurrent FSAs

• Data-dependent transitions
  – e.g. setMethod(DEFLATED), setMethod(STORED)
  – e.g. pop() when currentSize>0

• Robustness to noise
  – Illegal sequences in traces or client code
  – Method calls without any sequencing constraints

Interface: a, b, c, d, e
# Challenges Revisited

<table>
<thead>
<tr>
<th>Previous work</th>
<th>overgeneralization/over-restrictiveness</th>
<th>separation/composition of constraints</th>
<th>data-dependent transitions</th>
<th>robustness to noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whaley et al.</td>
<td>×</td>
<td>Separation</td>
<td>×</td>
<td>Handling unrelated methods by separation</td>
</tr>
<tr>
<td>Reiss et al.</td>
<td>×</td>
<td>Composition</td>
<td>×</td>
<td>Removing edges with low frequency</td>
</tr>
<tr>
<td>Ammons et al.</td>
<td>×</td>
<td>Composition</td>
<td>×</td>
<td>Removing states with low frequency</td>
</tr>
<tr>
<td>Cook et al.</td>
<td>×</td>
<td>Composition</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>

A single FSA model by 2-tails algorithm

### Submodel for the entry field
- States: S, E
- Transitions: putNextEntry
- Operation: closeEntry, write

### Submodel for the closed field
- States: S, E
- Transitions: putNextEntry
- Operation: closeEntry, close

-Glossary-  
- `setMethod(m) [m=STORED]`  
- `setMethod(m) [m=DEFLATED]`  
- `write`  
- `closeEntry`  
- `<DEFLATED>` putNextEntry, write*, closeEntry

---

-Challenges Revisited-  
- Removing states with low frequency
- Removing edges with low frequency
- Handling unrelated methods by separation
- Overgeneralization/over-restrictiveness
- Separation/composition of constraints
- Data-dependent transitions
- Robustness to noise
Dynamic Protocol Inference Framework

Trace Collection → Scenario Extraction → Protocol Inference → Protocol Usage

Evaluation: Cost-Benefit Analysis
Cost-Benefit Analysis - Cost

- **Trace collection**
  - Analysis scope [Ammons et al.][Cook et al.][Reiss et al.][Whaley et al.]

- **Scenario extraction**
  - Abstract object attributes [Ammons et al.]
  - Scenario seeds [Ammons et al.]
  - Scenario bounded size $N$ [Ammons et al.]

- **Protocol inference**
  - Algorithm parameters [Ammons et al.][Cook et al.][Reiss et al.]
  - Noise thresholds [Ammons et al.][Cook et al.]

- **Protocol usage**

Ammons et al.  
Cook et al.  
Reiss et al.  
Whaley et al.
Cost-Benefit Analysis - Benefit

• Accuracy
• Usefulness in particular applications
• Case studies
  – Whaley et al.
    • J2EE (50 “very interesting” models/657 classes)
    • 1 method in joeq program
  – Ammons et al.
    • 1 documented rule for X11 windowing sys (2000 functions)
    • 17 X11 clients (96 scenarios), 5 violating programs (2 buggy)
    • 72 clients (90 traces), 17 inferred “useful” specs, 2/3 detect 199 true bugs [Ammons 03]
  – Cook et al.
    • A change request process, 159 traces* 32 events, reflect 65% vs. 40%
Outline

• Background
• Overview of protocol inference
• Dynamic protocol inference framework
• Static protocol inference techniques
• Future work
• Conclusions
Static Protocol Inference Techniques

- **Static analysis of client code** [Lie et al. 01]

  ![Diagram showing the process of static protocol inference]

  - Trace Collection
  - Scenario Extraction
  - Protocol Inference
  - Protocol Usage

- **Static analysis of component code** [Whaley et al.]

34
Static Analysis of Component Code [Whaley et al.]

– Defensive programming

```java
public void closeEntry() throws IOException {
    entry = null;
}
```

```java
public synchronized void write(byte[] b, int off, int len) throws IOException {
    // (no writes of entry)
    if (entry == null) {
        throw new ZipException("no current ZIP entry");
    }
    // ...
}
```

• closeEntry(), write() is not allowed

• Select exception-guarding predicates and related fields in m
• Find method \( m' \) to set the fields to constants
• Identify illegal sequences from \( m \) to \( m' \)

Experimental results:
Java standard class library (81/914 classes, 24 listed)
Outline

• Background
• Overview of protocol inference
• Dynamic protocol inference framework
• Static protocol inference techniques
• Future work
  – Component testing
  – Inference improvement
• Conclusions
Component testing-I
Negative samples from component tests

• Component tests provide negative samples
  – Test case: write, putNextEntry

• Automatic test generation for a submodel
  – Submodel for the entry field:
    putNextEntry, write, closeEntry

Generate call sequences:
  putNextEntry, write √
  write, putNextEntry ×
  putNextEntry, closeEntry √
  closeEntry, putNextEntry √
  write, closeEntry ×
  closeEntry, write ×
Component testing-II
Feedback loop between component testing and protocol inference

• Better protocols $\leftrightarrow$ better tests

Spec-based test generation
(likely) Specs $\rightarrow$ Tests

Dynamic spec inference

A single FSA model by 2-tails algorithm
Inference Improvement-I
Composition and separation of constraints

- Concept analysis [Wille 82] to compose constraints

<table>
<thead>
<tr>
<th>methods</th>
<th>entry</th>
<th>entries</th>
<th>crc</th>
<th>written</th>
<th>locoff</th>
<th>closed</th>
<th>method</th>
<th>names</th>
</tr>
</thead>
<tbody>
<tr>
<td>putnextEntry</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>write</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>R</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>closeEntry</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>R</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>close</td>
<td></td>
<td>W</td>
<td></td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>setMethod</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

c0=all methods

c1={putnextEntry,setMethod}

c2={close,closeEntry,putnextEntry,write}

c3={closeEntry,putnextEntry,write}

c4={closeEntry,write}

c5={putnextEntry}

- Cluster analysis [Anderberg 73] to separate constraints
Inference Improvement-II
Data-dependent transition inference

• Data-dependent transitions
  – e.g. setMethod(DEFLATED), setMethod(STORED)
  – e.g. pop() when currentSize>0

• Heuristics to identify the data related to a component mode
  – Side-effect-free boolean methods
    • isEmpty(), isFull() in Stack class
  – Member fields in conditionals
    • if (currentSize>0), if (currentSize==MAXSIZE)
    • switch (method)
      { case DEFLATED:... case STORED:... }
New Problem: Argument Object Sequencing
Constraint Inference

• Problem: before calling `putNextEntry(ZipEntry e)` with argument `e`,
  – What method calls in `ZipEntry` need to be invoked on object `e`?
  – What method calls in `ZipOutputStream` need to be invoked by passing `e`?
• Related to bi-directional protocols for collaboration
Outline

• Background
• Overview of protocol inference
• Dynamic protocol inference framework
• Static protocol inference techniques
• Future work
• Conclusions
Conclusions

• Discussed component protocol inference problems and identified challenges
• Proposed a dynamic inference framework to compare previous work
• Discussed static inference techniques
• Suggested future work in the area
Trace Collection - I

Collected data types for a method call

• Method signature. [Whaley et al.][Reiss et al.][Ammons et al.]
  – Software process [Cook et al.]
  – Screen ID [El-Ramly et al.]

• Sequencing order (all)

• Class/Object ID [Whaley et al.][Reiss et al.] or arguments and
  return values [Ammons et al.]
Trace Collection - II

Summary of data collection mechanisms

<table>
<thead>
<tr>
<th>Previous work</th>
<th>Source code instrumentation</th>
<th>Bytecode/executable instrumentation</th>
<th>Execution environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whaley et al.</td>
<td></td>
<td>√ (component code)</td>
<td></td>
</tr>
<tr>
<td>Reiss et al.</td>
<td></td>
<td></td>
<td>√ (JVMPI)</td>
</tr>
<tr>
<td>Ammons et al.</td>
<td></td>
<td>√ (client code)</td>
<td></td>
</tr>
<tr>
<td>Cook et al.</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>El-Ramly et al.</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Trace Collection - III

Comparison of data collection mechanisms

- **Component code instrumentation** [Whaley et al.]
  + does it once for all (clients)
  + without requiring the availability of the client code

- **Client code instrumentation** [Ammons et al.]
  + better control of the instrumentation scope
  + without requiring the availability of the component code

- **Execution environment using Java Virtual Machine Profiling Agent (JVMPI)** [Reiss et al.]
  + Combine the above two
Internal usage of component

- Methods in the interface are called by component itself
- Internal usage needs to be identified and filtered out
  - Whaley et al. maintain knowledge of the local call stack
  - Reiss et al. post-process the collected traces.

```java
public void putNextEntry(ZipEntry e) throws IOException {
    ensureOpen();
    if (entry != null) {
        closeEntry(); // close previous entry
    }
    ..... 
}
```
Online vs. Offline Analysis

- **Online analysis** - Whaley et al.
  - Performed while the system is running

- **Offline analysis** - Reiss et al., Ammons et al., Cook et al., and El-Ramly et al.
  - Performed after the system has terminated
IPM2 algorithm [El-Ramly et al.]

• Given two scenarios: 1,3,2,3,4,3 and 2,3,2,4,1,3
• Infer two patterns: 2,3,4 and 3,2,4,3
Protocol Usage

• Without tool supports
  – Characterizing test suite [Whaley et al.]
  – Understanding systems [Whaley et al.]
  – Assisting spec construction [Whaley et al.]
  – Tuning algorithm parameters [Reiss et al.]

• With tool supports
  – Auditing applications [Whaley et al.]
  – Debugging specifications [Ammons et al.]
## Summary of Dynamic Inference Techniques

<table>
<thead>
<tr>
<th>Previous work</th>
<th>Trace collection</th>
<th>Scenario extraction</th>
<th>Protocol inference</th>
<th>Protocol usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whaley et al.</td>
<td>Method calls, Class/Object Ids</td>
<td>Object-based, Slicing by member fields</td>
<td>Separation of state modifying and state preserving methods</td>
<td>Test suite characterization, Software auditing</td>
</tr>
<tr>
<td>Reiss et al.</td>
<td>Method calls, Class/Object Ids</td>
<td>Object-based</td>
<td>k-tails algorithm</td>
<td>Alg parameter tuning</td>
</tr>
<tr>
<td>Ammons et al.</td>
<td>Method calls, Argument/return values</td>
<td>Flow dependence, Simplification, Standardization</td>
<td>sk-strings algorithm</td>
<td>Trace verification, Specification debugging</td>
</tr>
<tr>
<td>Cook et al.</td>
<td>Process events</td>
<td>n/a</td>
<td>k-tails algorithm, Markov algorithm</td>
<td>Process validation</td>
</tr>
<tr>
<td>El-Ramly et al.</td>
<td>Screen Ids</td>
<td>Interaction-based</td>
<td>IPM2 algorithm</td>
<td>Legacy system reengineering</td>
</tr>
</tbody>
</table>
Static Analysis of Client Code

- Scenarios can be extracted from code statically as inputs to protocol inference algorithms.
  - Model checking:
    - models extracted from code by using pattern matching and program slicing [Lie et al. 01].
  - Intrusion detection
    - an FSA for system calls inferred from application code [Wagner et al. 01].
  - Bug detection
    - temporal rules inferred from the Linux code [Engler et al. 01]