

CS699/IT803  
Professor Yutao Zhong

**Automatic Extraction of Object-  
Oriented Component Interfaces**  
by Whaley J, Martin, M, and Lam M

Chien-Chih Lin  
clin3@gmu.edu  
April 20, 2006

## Outline

- Introduction
- Motivation
- Background
- Proposed Approach
- Implementation & Evaluation
- Related & Future Work
- Conclusion

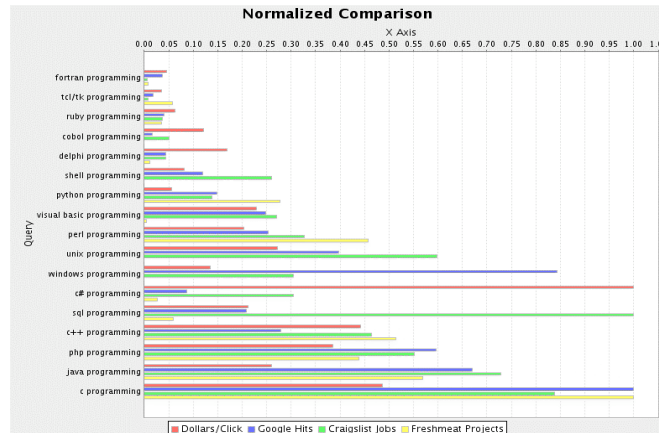
## Acknowledgement

- Thanks for Professor Zhong for her feedback.
- Thanks for John Whaley's PowerPoint of this paper.  
<http://www.stanford.edu/~jwhaley/papers/issta02.ppt>
- **David N. Welton** ([davidw@dedasys.com](mailto:davidw@dedasys.com)). Programming Language Popularity. 2004-09-24 (updated 2005-10-20).  
[http://www.dedasys.com/articles/language\\_popularity.html](http://www.dedasys.com/articles/language_popularity.html)

## Introduction

- The increasing use of **Component-based** software.

# Introduction



Programming Language Popularity  
Each color represents each group.  
Length of bar shows the rank of each survey.

# Introduction

- **High-Level Software Performance Monitoring (SPM)**
- **Interface specifications** are important!
  - Misunderstanding the API is a common source of error
- Ideally, we **want formal specifications**.
  - However, many components don't have any specifications, formal or informal!

## Motivation

- Our goal: **automatic generation of interface specifications**
  - For large, object-oriented programs
  - Partial specifications

## Motivation

### Why Automatic Extraction?

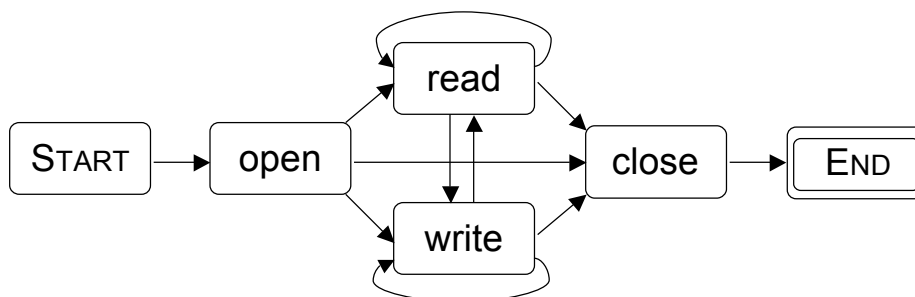
- **Documentation**
  - Based on the actual code, so no divergence
- **Rules for static or dynamic checkers**
  - Find errors in API usage
- **Find API bugs**
  - Discrepancy between code & intended API
- **Dynamic extraction**
  - Evaluation of test coverage

## Background

- Example: File
- A Simple OO Component Model
- Problem 1 -- Splitting by fields
- Problem 2 -- State-preserving methods
- Summary of Model

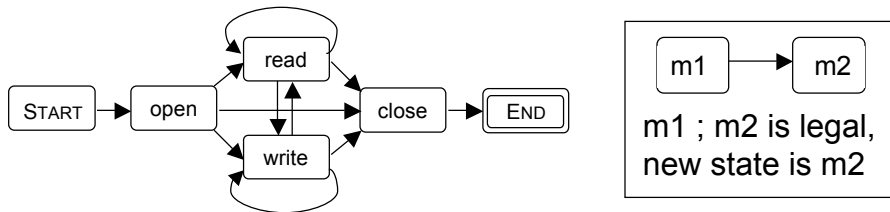
## Example: File

- Use a *Finite State Machine (FSM)* to express ordering constraints.



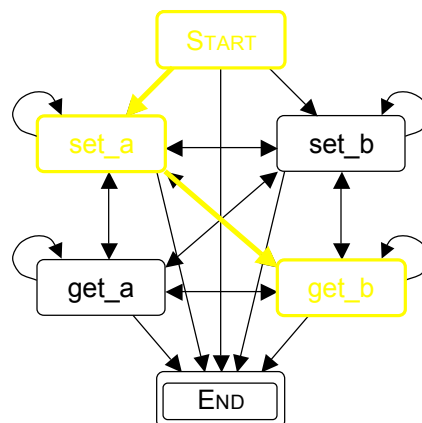
## A Simple OO Component Model

- Each object follows an FSM model.
- One state per method, plus START & END states.
- Method call causes a transition to a new state.

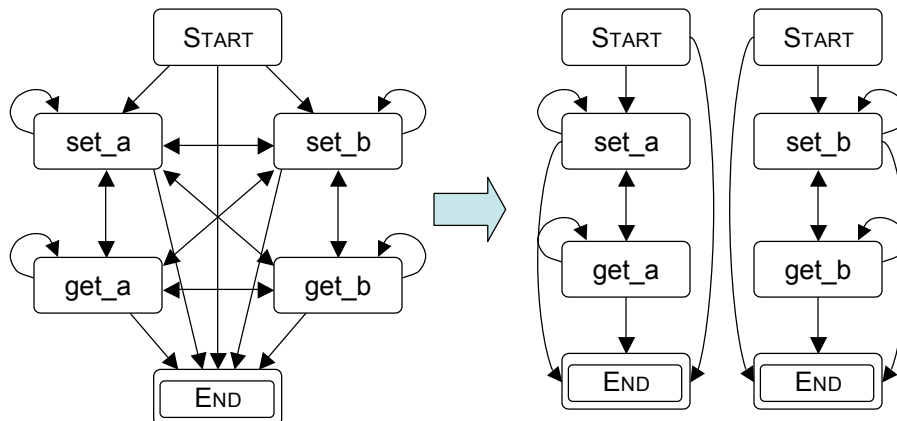


## Problem 1

- An object has two fields, a and b.
- Each field must be set before being read.
- Solution:  
a product of FSMs,  
one for each field.



## Splitting by fields

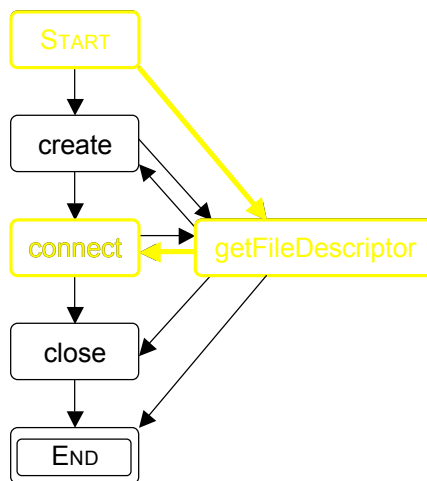


Separate by fields into different, independent submodels.

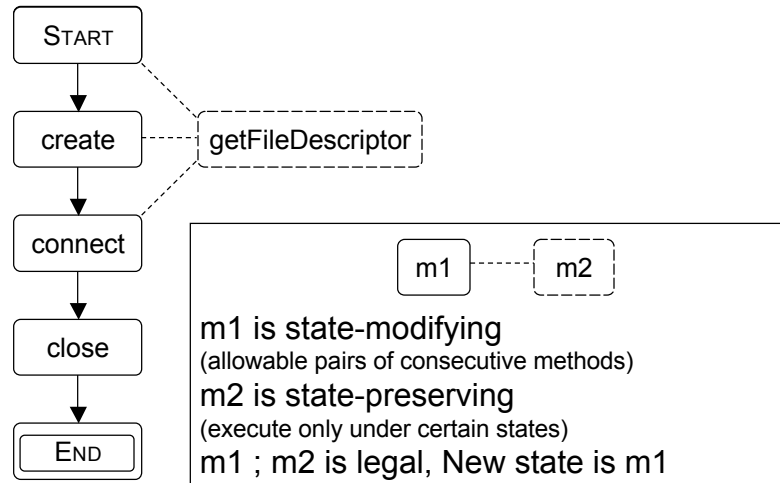
## Problem 2

Model for Socket

- `getFileDescriptor` is *state-preserving*.
- Solution: distinguish between *state-modifying* and *state-preserving*.



## State-preserving methods



## Summary of Model

- Product of FSMs
  - Per-thread, per-instance
- One submodel per field
  - Interprocedural mod-ref analysis
    - Identifies methods belonging to submodel
    - Separates state-modifying and state-preserving methods.
- One submodel per Java interface
  - Implementation not required.

## Proposed Approach

- Based on a simplified version of **path expressions** that treat objects as a resource shared between different sections of an application.
- **Static Analysis**  
to deduce illegal call sequences in a program
- **Dynamic Analysis**  
to extract models from execution runs
- **Dynamic Model Checker**  
to ensure that the code conforms to the model

## Extraction Techniques

Static	Dynamic
For all possible program executions	For one particular program execution
Conservative	Exact (for that execution)
Analyze implementation	Analyze component usage
Detect illegal transitions	Detect legal transitions
Superset of ideal model (upper bound)	Subset of ideal model (lower bound)

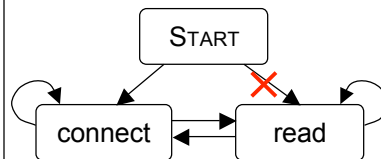
## Implementation & Evaluation

- Implemented for Java
- Analyzed >1 million lines of code
  - Java class libraries
  - Java 2 Enterprise Edition
  - Java network libraries
  - joeq virtual machine

## Static Model Extractor

- Defensive programming
  - Implementation throws exceptions (user or system defined) on illegal input.

```
public void connect() {  
    connection = new Socket();  
}  
public void read() {  
    if (connection == null)  
        throw new IOException();  
}
```



## Detecting Illegal Transitions

- Only support simple predicates
  - Comparisons with constants, implicit null pointer checks
- Find <source, target> pairs such that:
  - Source must execute:
    - *field = const* ;
  - Target must execute:
    - if (*field == const*)  
throw exception;

## Algorithm

- Source method: Constant propagation
  - Constant at exit node
- Target method: Control dependence
  - Throw of exception is control dependent on predicate

## Dynamic Extractor

- Goal: find the legal transitions that occur during an execution of the program
- Java bytecode instrumentation
- For each thread, each instance of a class:
  - Track last state-modifying method for each submodel.
- Same mechanism for dynamic checking
  - Instead of adding to model, flag exception.

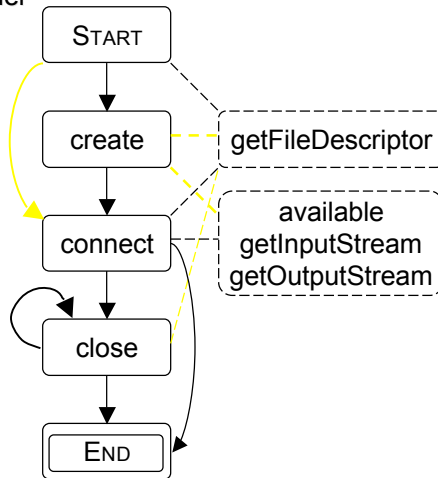
## Experiences

- We applied our tool to several real-life applications.

Program	Description	Lines of code	Analyses	Category
Java.net 1.3.1	Networking library	12,000	Both	-- Upper/lower bound (static/dynamic)
Java libraries 1.3.1	General purpose library	300,000	Static	-- Automatic Documentation
J2EE 1.2.1	Business platform	900,000	Dynamic	-- Automatic Documentation -- Test coverage
joeq	Java virtual machine	65,000	Both	-- Finding API bugs

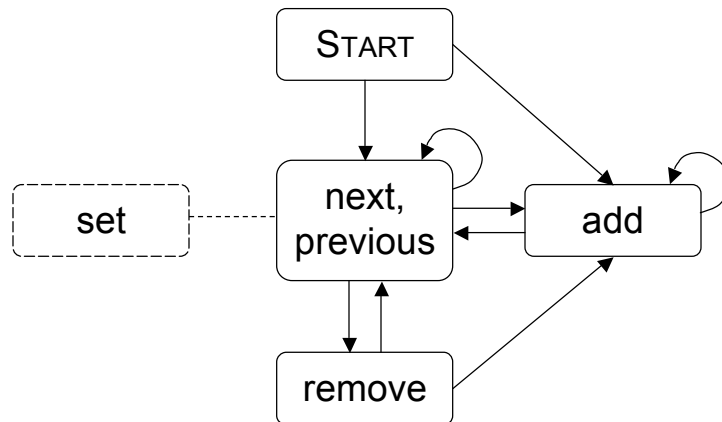
# Upper/lower bound of model

SocketImpl model  
(dynamic)  
(+static)



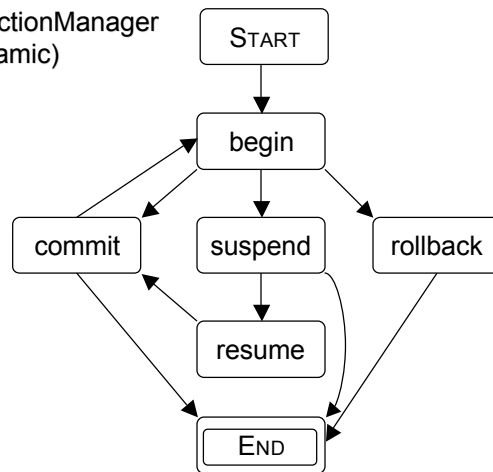
# Automatic documentation

- java.util.AbstractList.ListItr  
slice on lastRet field (static)



# Automatic documentation

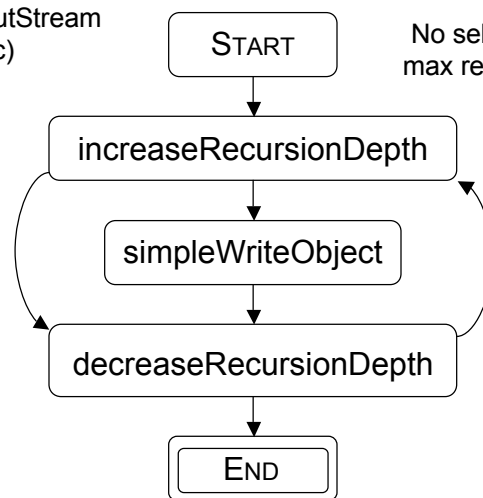
J2EE TransactionManager  
(dynamic)



# Test coverage

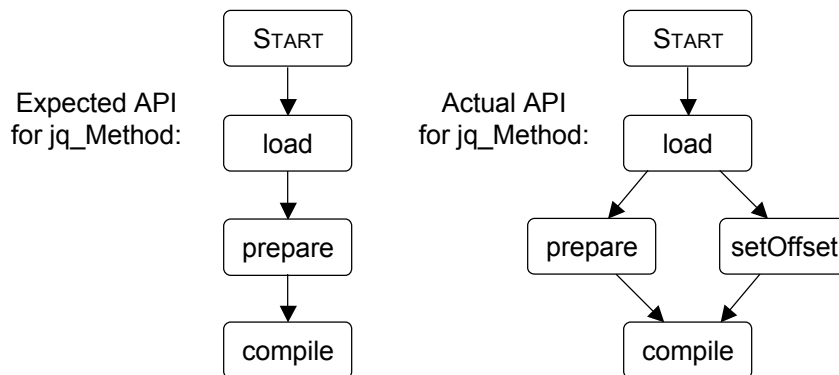
J2EE IIOPOutputStream  
(dynamic)

No self-edges implies a  
max recursion depth of 1



## Finding API bugs

- Applied our tool to the jq virtual machine



## Related Work

- Dynamic
  - Daikon (Ernst99)
  - DIDUCE (Hangal02)
  - K-limited FSM extraction (Reiss01)
  - Machine-learning (Ammons02)
- Static
  - Metal (Engler00)
  - Vault (DeLine01), NIL, Hermes (Strom86)
  - SLAM toolkit (Ball01)
  - ESC (Dettefs98)
- ESC + Daikon (Flanagan01, Nimmer02)

## Future Work

- For Information Security or Intrusion Detection
  - Check the Behavior of the Program (Signatures)
  - Check Boundary Safety
  - Check Resource Safety (API bugs)
- Working on **High-Level Software Performance Monitoring (SPM)**

## Question

?  
???  
?????  
????????  
??????????  
????????????  
??????????????  
????????????????  
??????????????????

## Conclusion

- Product of FSM
  - Model is simple, but useful
- Upper/lower bound: static/dynamic
- Useful for:
  - Documentation generation
  - Test coverage
  - Rules for automatic checkers
  - Finding API bugs