Analyzing and Inferring the Structure of Code Changes

Miryung Kim
University of Texas at Austin

SNU Seminar, Nov 17th

Software evolution plays an ever-increasing role in software development

Motivating Scenarios

- “This program worked a month ago but is not working now. What changed since then? Which change led to a bug?”
- “Did Bob implement the intended changes correctly?”
- “There’s a merge conflict. What did Alice change?”

Diff Output

Changed Code		
File Name	Status	Lines
DummyRegistry	New	20 lines
AbsRegistry	New	133 lines
JRMPRegistry	Modified	123 lines
JeremieRegistry	Modified	52 lines
JacORBCosNaming	Modified	133 lines
IIOPCosNaming	Modified	50 lines
CmiRegistry	Modified	39 lines
NameService	Modified	197 lines
NameServiceManager	Modified	15 lines
Total Change: 9 files, 723 lines		

```
- public class CmiRegistry implements
NameService {
+ public class CmiRegistry extends
AbsRegistry implements NameService {
-     private int port = ...
-     private String host = null
-     public void setPort (int p) {
-        if (TraceCarol. isDebug()) { ...
-        }
-      }
-      public int getPort() {
-        return port;
-       }
-      public void setHost(String host)
{ ....
```

Check-In Comment

“Common methods go in an abstract class. Easier to extend/maintain/fix”

Changed Code		
File Name	Status	Lines
DummyRegistry	New	20 lines
AbsRegistry	New	133 lines
JRMPRegistry	Modified	123 lines
JeremieRegistry	Modified	52 lines
JacORBCosNaming	Modified	133 lines
IIOPCosNaming	Modified	50 lines
CmiRegistry	Modified	39 lines
NameService	Modified	197 lines
NameServiceManager	Modified	15 lines
Total Change: 9 files, 723 lines		

Why did all these files change together?
Is anything missing in this change?

Limitations

Diff

- Low-level

Natural Language Description (Check-In Comment)

- Often incomplete
- Difficult to trace back to code changes

Research Question

How do we ***automatically extract*** the differences between two versions into a ***concise*** and ***meaningful*** program change representation?

Research Question

How do we ***automatically extract*** the differences between two versions into a ***concise*** and ***meaningful*** program change representation?

- Help programmers reason about code changes **at a high level**
- Enable researchers to study software evolution better

Example Output

All `draw` methods take an additional `int` input argument.

All `setHost` methods in `Service's` subclasses deleted calls to `SQL` library except `NameService` class.

...

Concise
Easy to note inconsistent changes

Systematic Changes

- Refactoring [Opdyke 92, Griswold 92, Fowler 99...]

“Move related classes from one package to another package”

Systematic Changes

- Refactoring [Opdyke 92, Griswold 92, Fowler 99...]
- API update [Chow&Notkin 96, Henkel&Diwan 05, Dig&Johnson 05...]

“Update an API and all call sites of the API”

Systematic Changes

- Refactoring [Opdyke 92, Griswold 92, Fowler 99...]
- API update [Chow&Notkin 96, Henkel&Diwan 05, Dig&Johnson 05...]
- Crosscutting concerns [Kiczales et. al. 97, Tarr et. al. 99, Griswold 01...]

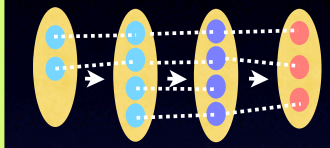
“Adding logging feature throughout code”

Systematic Changes

- Refactoring [Opdyke 92, Griswold 92, Fowler 99...]
- API update [Chow&Notkin 96, Henkel&Diwan 05, Dig&Johnson 05...]
- Crosscutting concerns [Kiczales et. al. 97, Tarr et. al. 99, Griswold 01...]
- Consistent updates on code clones [Miller&Myers 02, Toomim et. al. 04, Kim et. al. 05]

"Apply similar changes to syntactically similar code fragments"

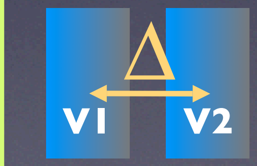
Thesis Overview



Analyses of Software Evolution

- Evolution of Code Clones

High-level changes are often systematic at a code level



Automatic Inference of High-Level Change Descriptions

- Rule-based Change Representations
- Rule Learning Algorithms

Outline

- Empirical Analyses of Code Clone Evolution [ISESE 04, ESEC/FSE 05]
- Automatic Inference of High-Level Change Descriptions
    - Changes to API Names and Signatures [ICSE 07]
    - Changes to Code Elements and Structural Dependencies
- Future Directions

Code Clones

Code clones are syntactically similar code fragments

```
public void updateFrom (Class c) {
    String cType = Util.makeType(c.Name
    ());
    if (seenClasses.contain(cType)) {
        return;
    }
    seenClasses.add(cType);
    if (hierarchy!=null) {
        ....
    }
    ...
```

```
public void updateFrom (ClassReader c) {
    String cType = CTD.convertType
    (c.Name());
    if (seenClasses.contain(cType)) {
        return;
    }
    seenClasses.add(cType);
    if (hierarchy!=null) {
        ....
    }
    ...
```

Found by a clone detector, CCFinder [Kamiya et al. 2002]

Conventional Wisdom about Code Clones

"Code clones must be aggressively refactored because they indicate poor software quality."
[Fowler 00, Beck 00, Nickell & Smith 03 ...]

```
public void updateFrom (Class c) {
    String cType = Util.makeType(c.Name
    ());
    if (seenClasses.contain(cType)) {
        return;
    }
    seenClasses.add(cType);
    if (hierarchy!=null) {
        ....
    }
    ...
```

```
public void updateFrom (ClassReader c) {
    String cType = CTD.convertType
    (c.Name());
    if (seenClasses.contain(cType)) {
        return;
    }
    seenClasses.add(cType);
    if (hierarchy!=null) {
        ....
    }
    ...
```

Found by a clone detector, CCFinder [Kamiya et al. 2002]

A Study of Copy and Paste Programming Practices at IBM

[Kim et al. ISESE 2004]

- To understand programmers' copy and paste coding behavior, I ***built an Eclipse plug-in that records edits*** and ***replays the captured edits***
- Programmers often ***create and manage*** code clones with clear intent

*

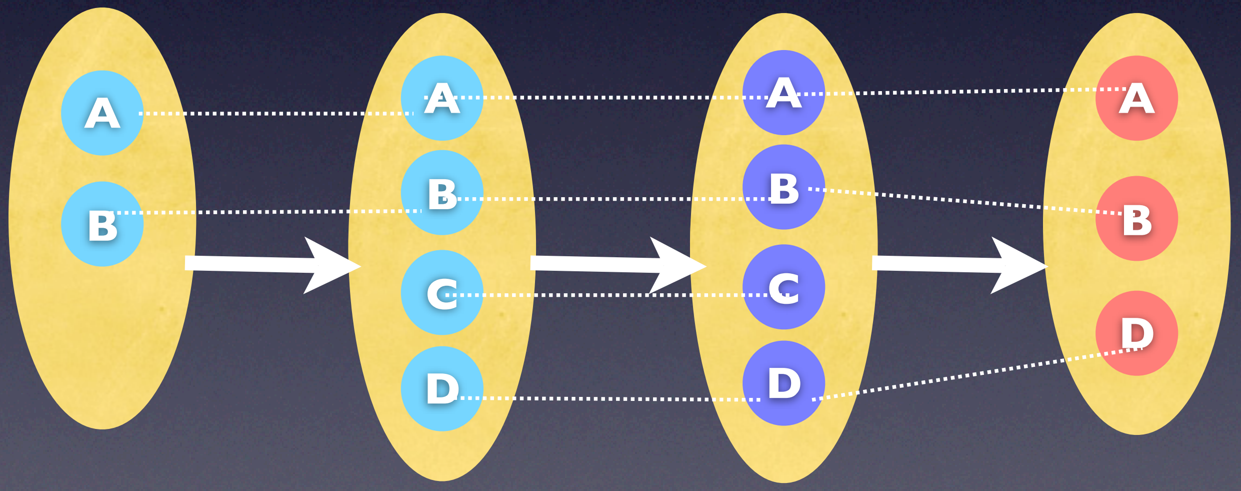
An Empirical Study of Code Clone Genealogies

[Kim et al. ESEC/FSE 2005]

- I developed an approach that ***automatically reconstructs*** the history of code clones ***from a source code repository***
- I studied clone evolution in two Java open source projects, *carol* and *dnsjava*

Clone Genealogy

Clone genealogy is a representation that captures clone change patterns over a sequence of program versions



Version i

Version i+1

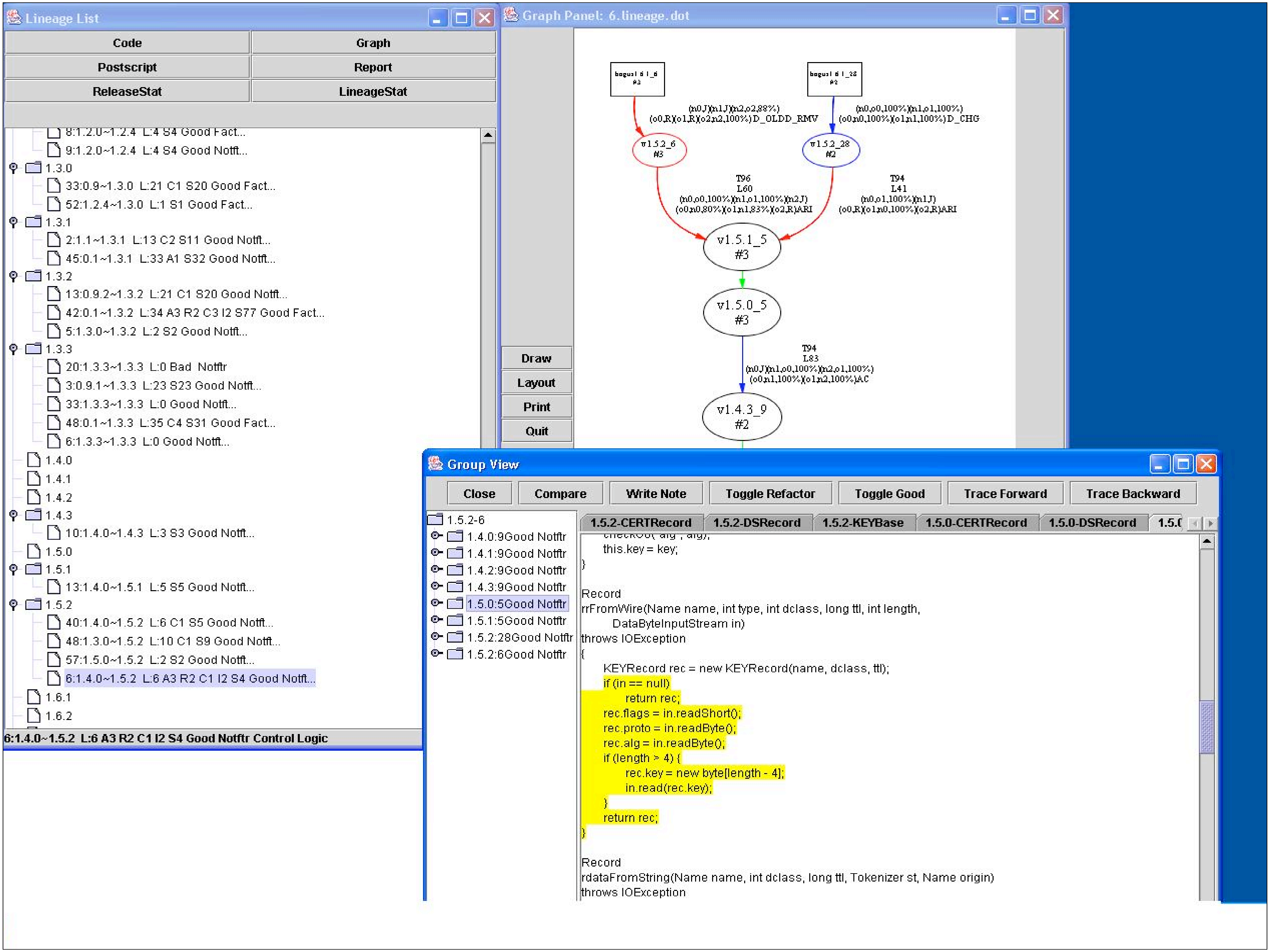
Version i+2

Version i+3

ADD

CONSISTENT_ CHANGE

INCONSISTENT_ CHANGE



```
8:1.2.0~1.2.4  L:4 S4 Good Fact...
9:1.2.0~1.2.4  L:4 S4 Good Notft...
1.3.0
   33:0.9~1.3.0  L:21 C1 S20 Good Fact...
   52:1.2.4~1.3.0  L:1 S1 Good Fact...
1.3.1
   2:1.1~1.3.1  L:13 C2 S11 Good Notft...
   45:0.1~1.3.1  L:33 A1 S32 Good Notft...
1.3.2
   13:0.9.2~1.3.2  L:21 C1 S20 Good Notft...
   42:0.1~1.3.2  L:34 A3 R2 C3 I2 S77 Good Fact...
   5:1.3.0~1.3.2  L:2 S2 Good Notft...
1.3.3
   20:1.3.3~1.3.3  L:0 Bad  Notftr
   3:0.9.1~1.3.3  L:23 S23 Good Notft...
   33:1.3.3~1.3.3  L:0 Good Notft...
   48:0.1~1.3.3  L:35 C4 S31 Good Fact...
   6:1.3.3~1.3.3  L:0 Good Notft...
1.4.0
1.4.1
1.4.2
1.4.3
   10:1.4.0~1.4.3  L:3 S3 Good Notft...
1.5.0
1.5.1
   13:1.4.0~1.5.1  L:5 S5 Good Notft...
1.5.2
   40:1.4.0~1.5.2  L:6 C1 S5 Good Notft...
   48:1.3.0~1.5.2  L:10 C1 S9 Good Notft...
   57:1.5.0~1.5.2  L:2 S2 Good Notft...
   6:1.4.0~1.5.2  L:6 A3 R2 C1 I2 S4 Good Notft...
1.6.1
1.6.2
```

6:1.4.0~1.5.2 L:6 A3 R2 C1 I2 S4 Good Notftr Control Logic

```
1.5.2-6
   1.4.0:9Good Notftr
   1.4.1:9Good Notftr
   1.4.2:9Good Notftr
   1.4.3:9Good Notftr
   1.5.0:5Good Notftr
   1.5.1:5Good Notftr
   1.5.2:28Good Notftr
   1.5.2:6Good Notftr
```

```
    checkOut( alg , alg);
    this.key = key;
}

Record
rrFromWire(Name name, int type, int dclass, long ttl, int length,
        DataByteInputStream in)
throws IOException
{
    KEYRecord rec = new KEYRecord(name, dclass, ttl);
    if (in == null)
        return rec;
    rec.flags = in.readShort();
    rec.proto = in.readByte();
    rec.alg = in.readByte();
    if (length > 4) {
        rec.key = new byte[length - 4];
        in.read(rec.key);
    }
    return rec;
}

Record
rdataFromString(Name name, int dclass, long ttl, Tokenizer st, Name origin)
throws IOException
```

Contradicting Evidence to Conventional Wisdom

- *Many clones are short-lived, diverging clones*
  - 48-72% of clone genealogies lasted less than 8 check-ins out of over 160 check-ins
  - 26-34% of these clones disappeared due to divergent changes
- *Refactoring cannot remove many long-lived clones*
  - 65-73% of long-lived, consistently changing clones are not easy to refactor using standard refactoring techniques [Folwer 00]

<u>example</u>

Summary of
Studies on Code Clones

By focusing on the ***evolutionary aspects of clones***, I found

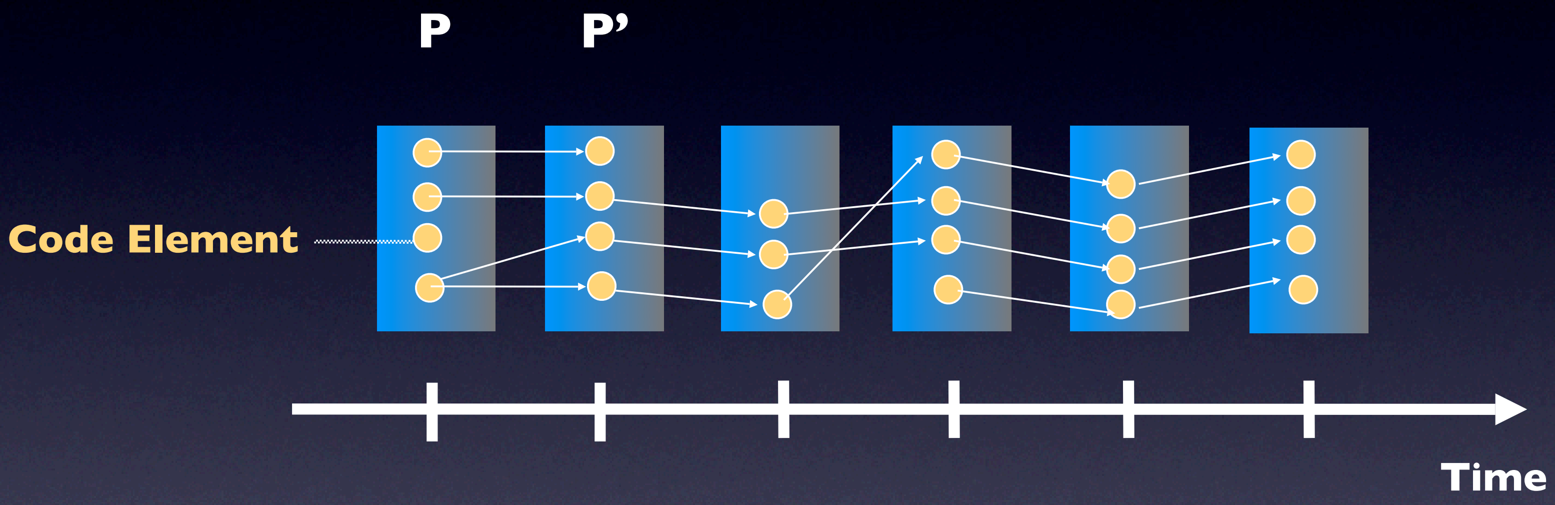
- Clones are inevitable parts of software evolution
- Refactoring may not be applicable to or beneficial for many code clones

My studies shifted research efforts from automatic clone detection to code clone management support (e.g., [Duala-Ekoko & Robillard 07, Krinke 07, Aversano et al. 07, Lozano et al. 07, etc.])

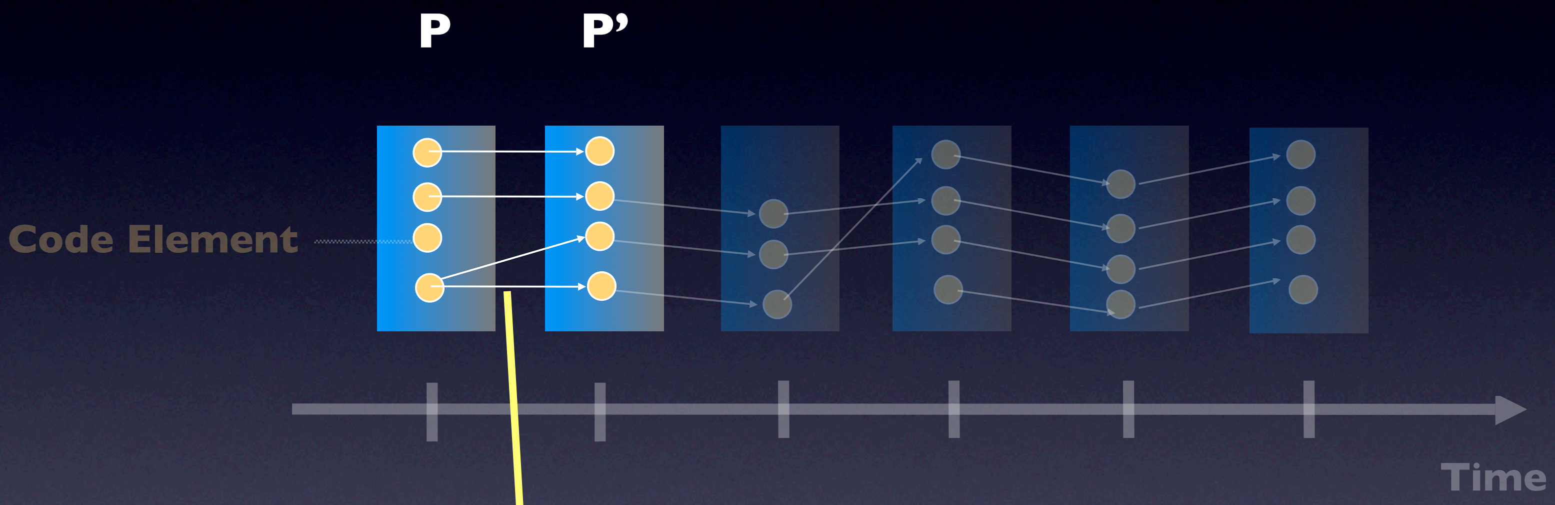
Outline

- Empirical Analyses of Code Clone Evolution
- Automatic Inference of High-Level Change Descriptions
  - Changes to API Names and Signatures
  - Changes to Code Elements and Structural Dependencies
- Future Directions

Motivation:
Code Evolution Analyses



Research Question



"How do we automatically match corresponding code elements between two program versions?"

Existing Approaches

[Kim et al. MSR 2006]

diff, Syntactic Diff (CDiff), Semantic Diff, JDiff, BMAT, origin analysis, refactoring reconstruction tools, clone detectors, etc.

Individually compare code elements
at particular granularities
using similarity measures

Limitations of Existing Approaches

Limitations of Existing Approaches

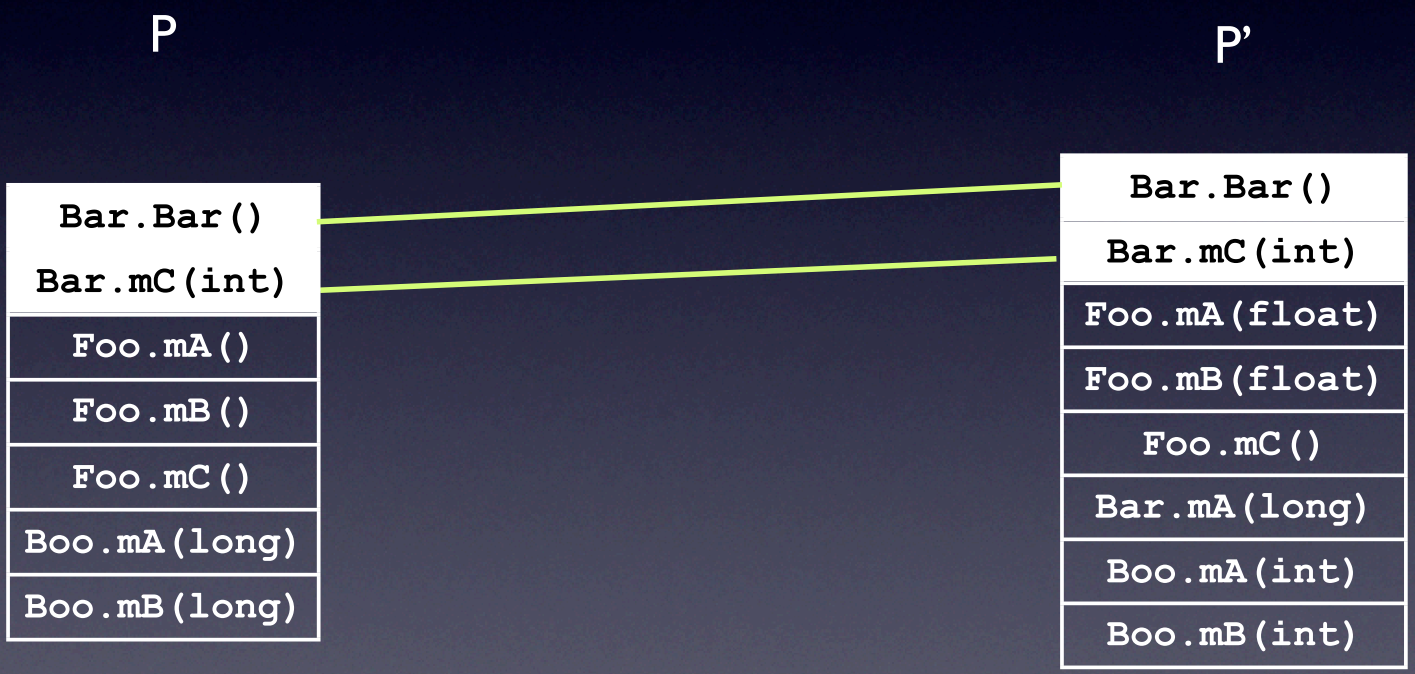
P

```
Bar.Bar()
Bar.mC(int)
Foo.mA()
Foo.mB()
Foo.mC()
Boo.mA(long)
Boo.mB(long)
```

P’

```
Bar.Bar()
Bar.mC(int)
Foo.mA(float)
Foo.mB(float)
Foo.mC()
Bar.mA(long)
Boo.mA(int)
Boo.mB(int)
```

Limitations of Existing Approaches



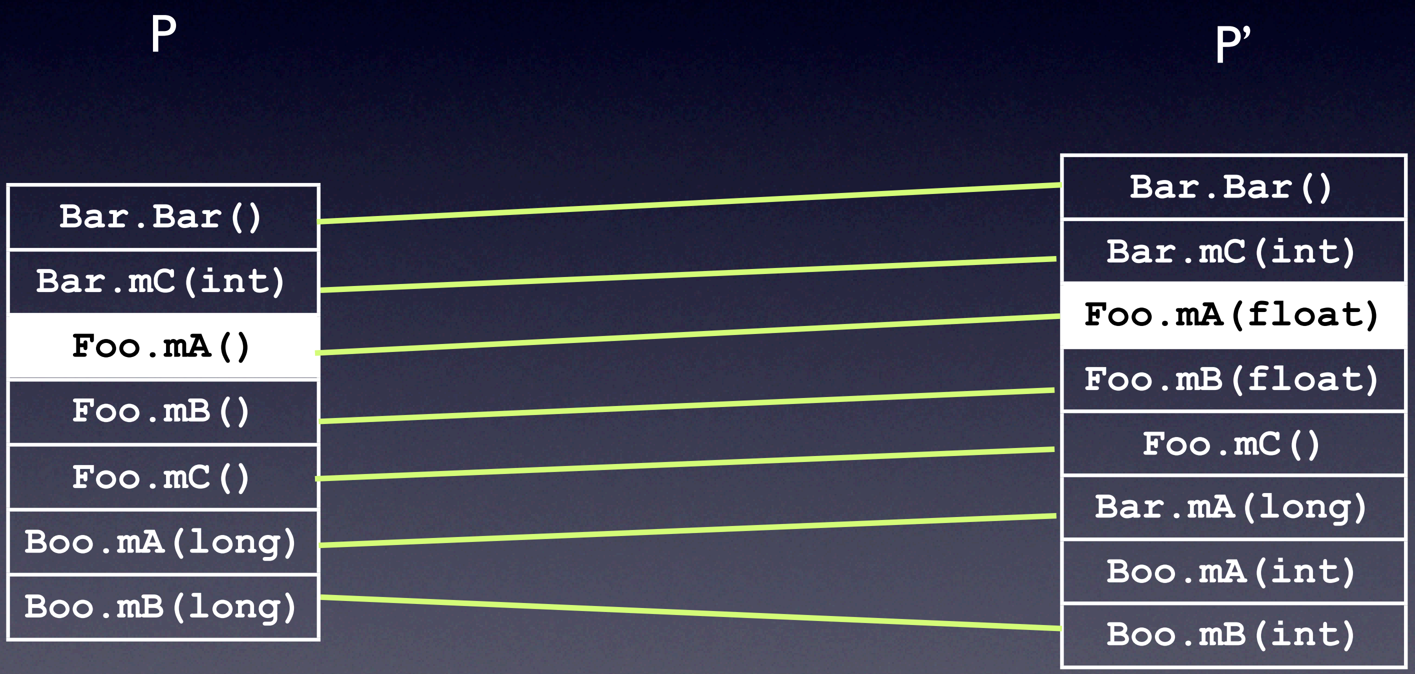
P

Bar.Bar()
Bar.mC(int)
Foo.mA()
Foo.mB()
Foo.mC()
Boo.mA(long)
Boo.mB(long)

P’

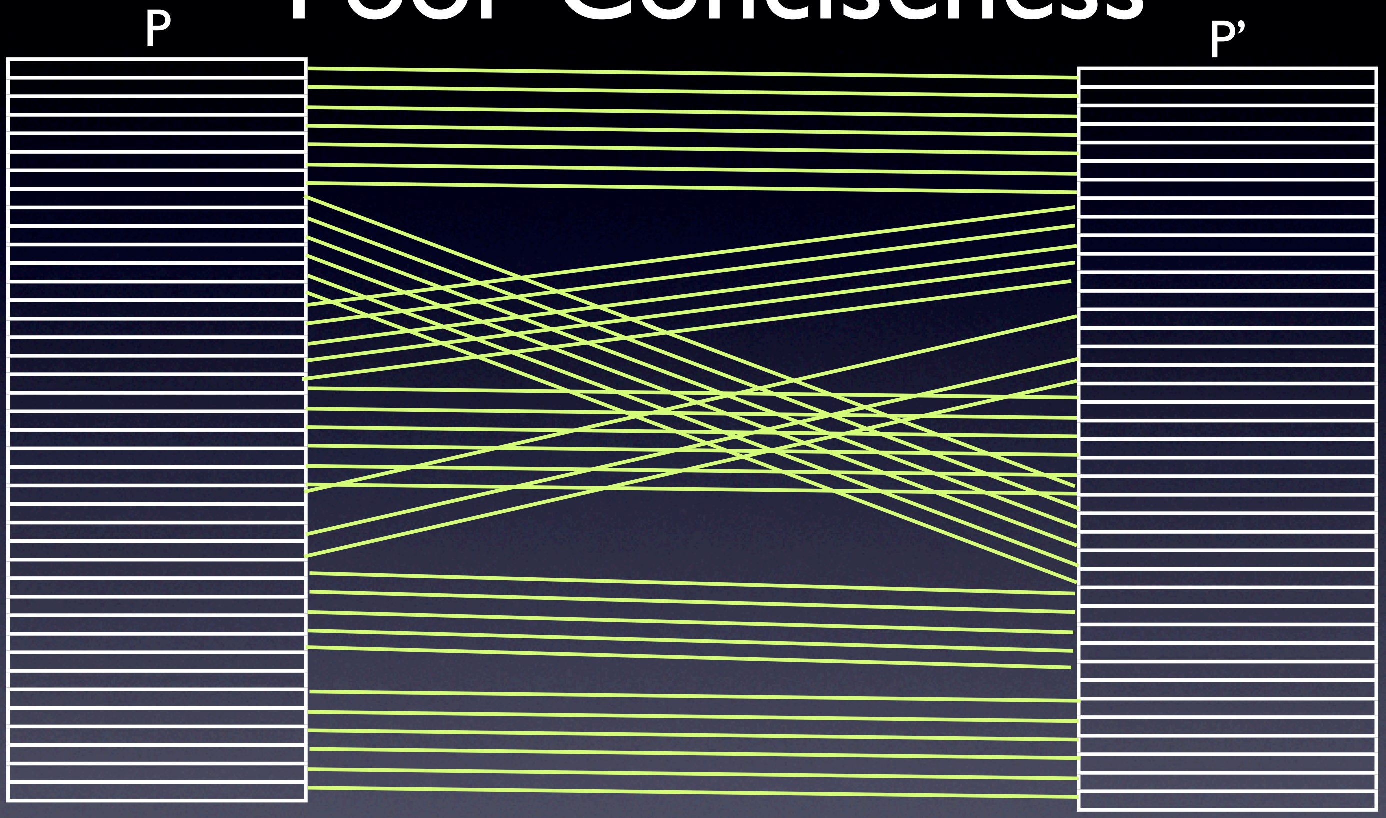
Bar.Bar()
Bar.mC(int)
Foo.mA(float)
Foo.mB(float)
Foo.mC()
Bar.mA(long)
Boo.mA(int)
Boo.mB(int)

Limitations of Existing Approaches



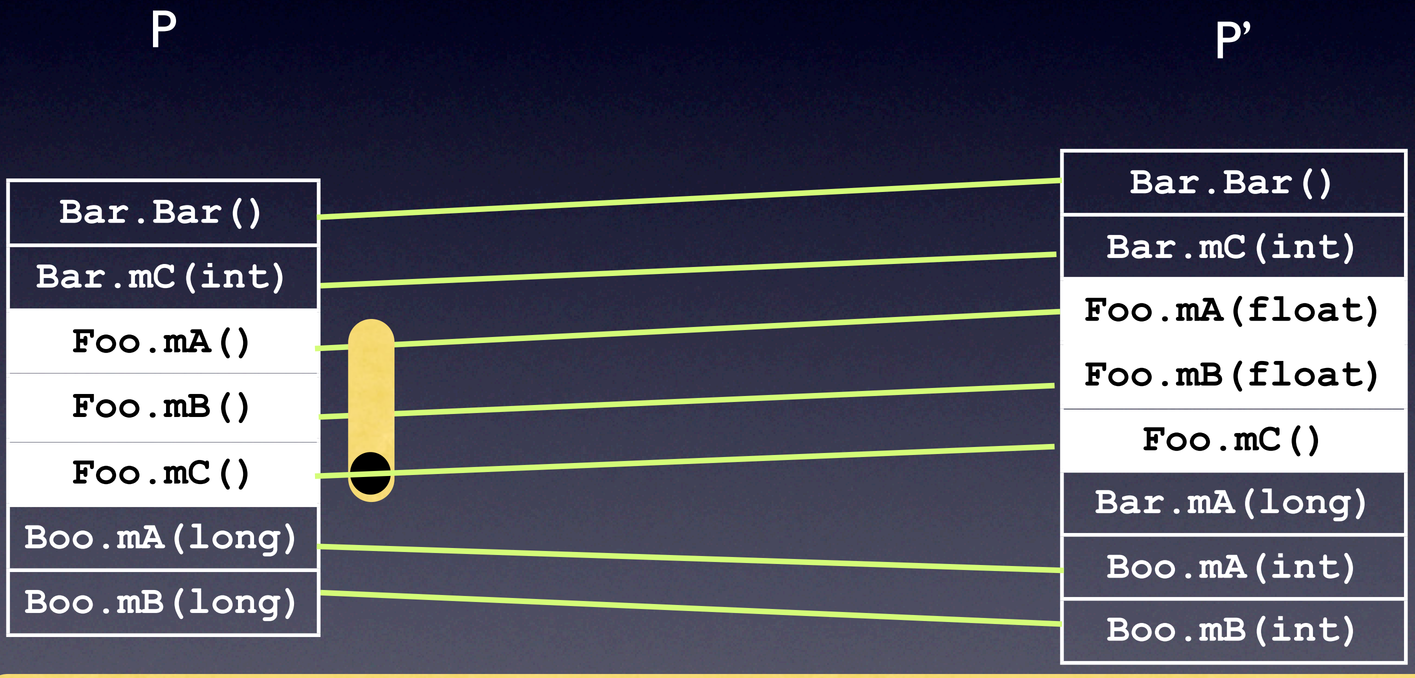
P	P’
Bar.Bar()	Bar.Bar()
Bar.mC(int)	Bar.mC(int)
Foo.mA()	Foo.mA(float)
Foo.mB()	Foo.mB(float)
Foo.mC()	Foo.mC()
Boo.mA(long)	Bar.mA(long)
	Boo.mA(int)
Boo.mB(long)	Boo.mB(int)

Limitation 1.
Poor Conciseness



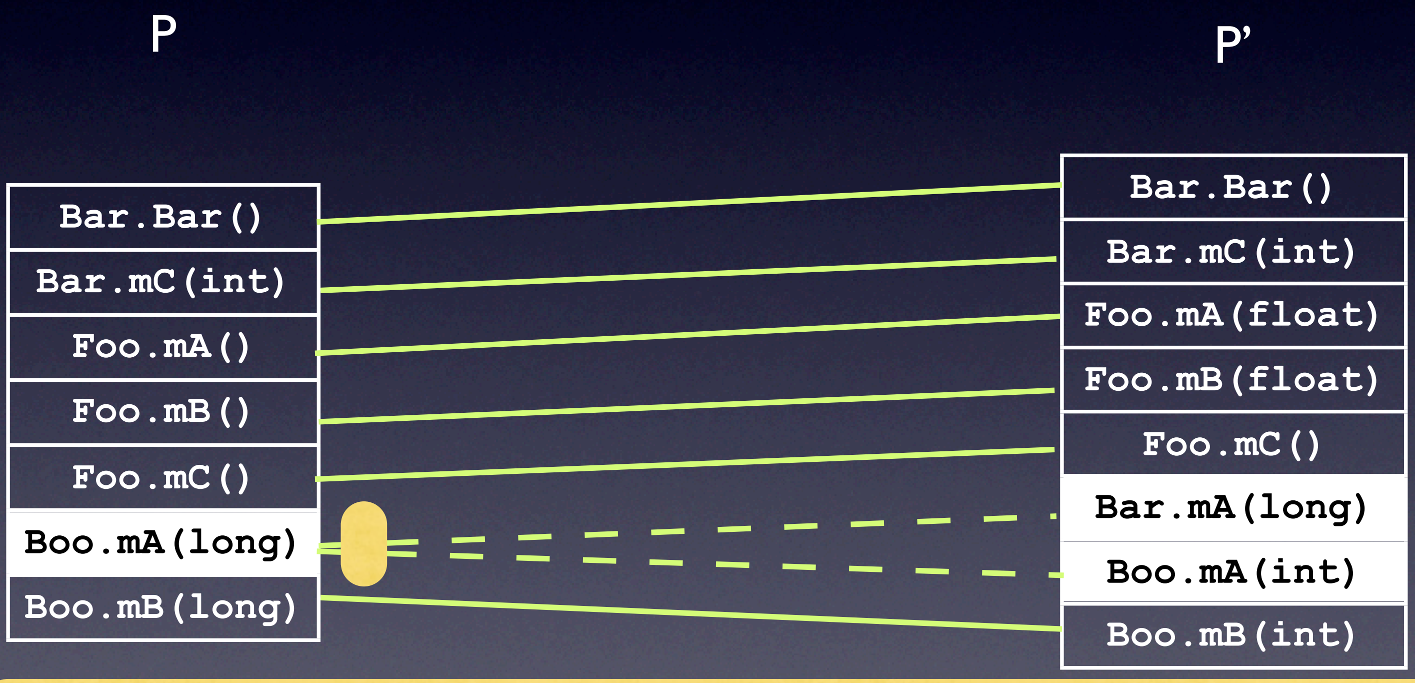
Output is an unstructured, usually lengthy list of matches

Limitation 2.
Hard to Identify Exception



Difficult to spot inconsistent changes

Limitation 3.
Low Recall

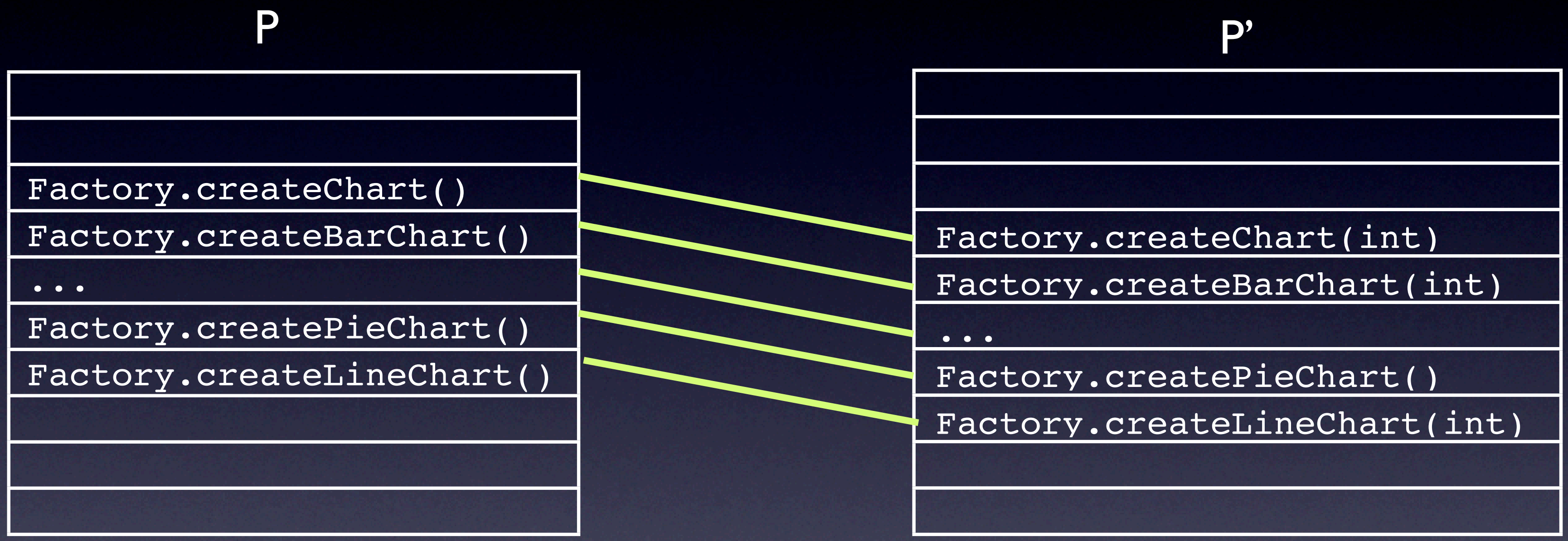


Difficult to disambiguate among many potential matches

What is the Core Question?

Given two program versions (P, P’),
with respect to a particular vocabulary of changes,
find changes from P to P’

Example Change



P

```
Factory.createChart()
Factory.createBarChart()
...
Factory.createPieChart()
Factory.createLineChart()
```

P’

```
Factory.createChart(int)
Factory.createBarChart(int)
...
Factory.createPieChart()
Factory.createLineChart(int)
```

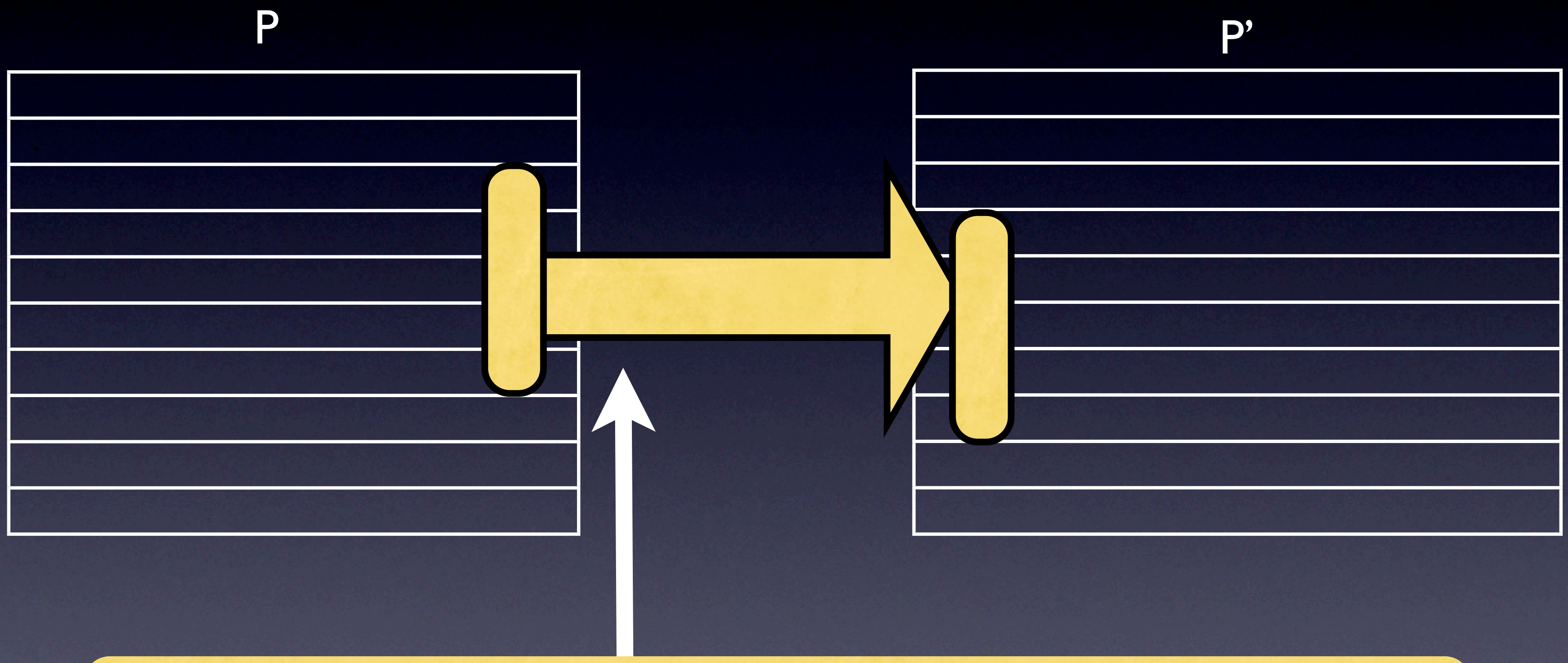
“Add `int` input argument to all chart creation APIs”

Our Rule-based Matching Approach

[Kim et al. ICSE 2007]

- Our ***change-rules*** can concisely describe ***a set of related API-level changes***.
- Our tool ***automatically infers*** a set of change rules between two versions of a program.

Change-Rule Syntax



```
FOR ALL x:method-header IN
scope
transformation(x)
```

Scope

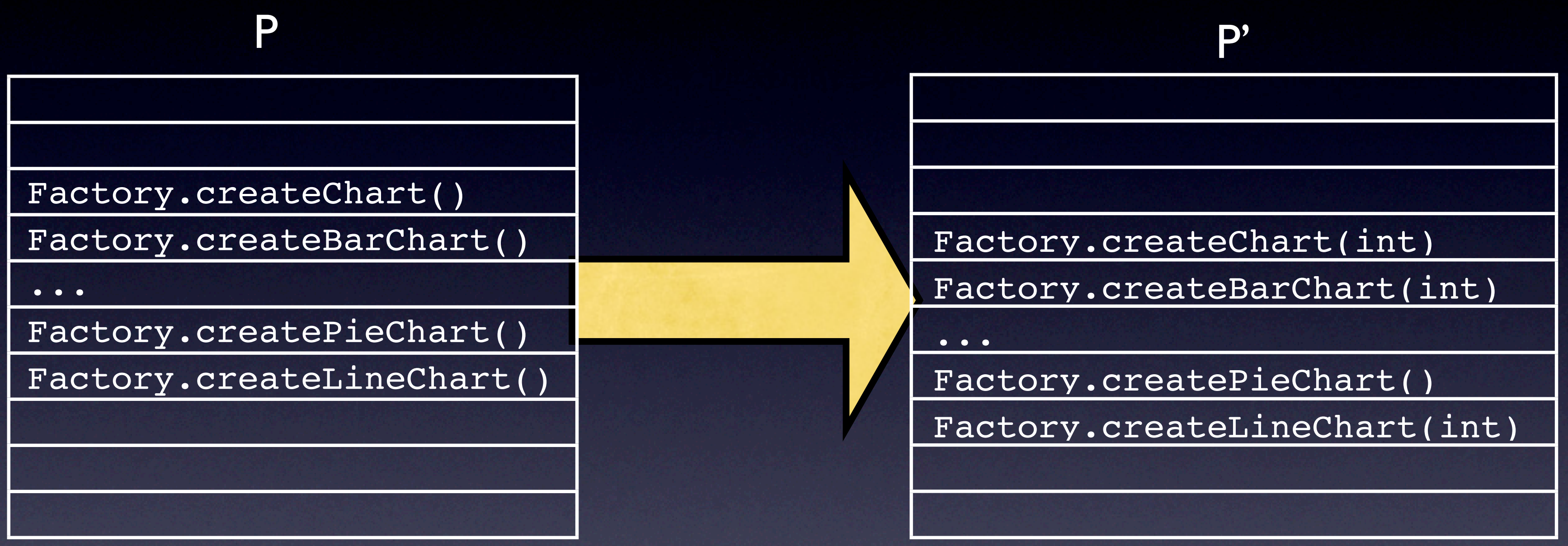
- We use a regular expression to denote a set of methods

  e.g. chart.Factory.create*Chart(*)

API-Level Transformations

- Replace the name of package, class, and method
- Replace the return type
- Modify the input signature, etc.

Example Change-Rule



P

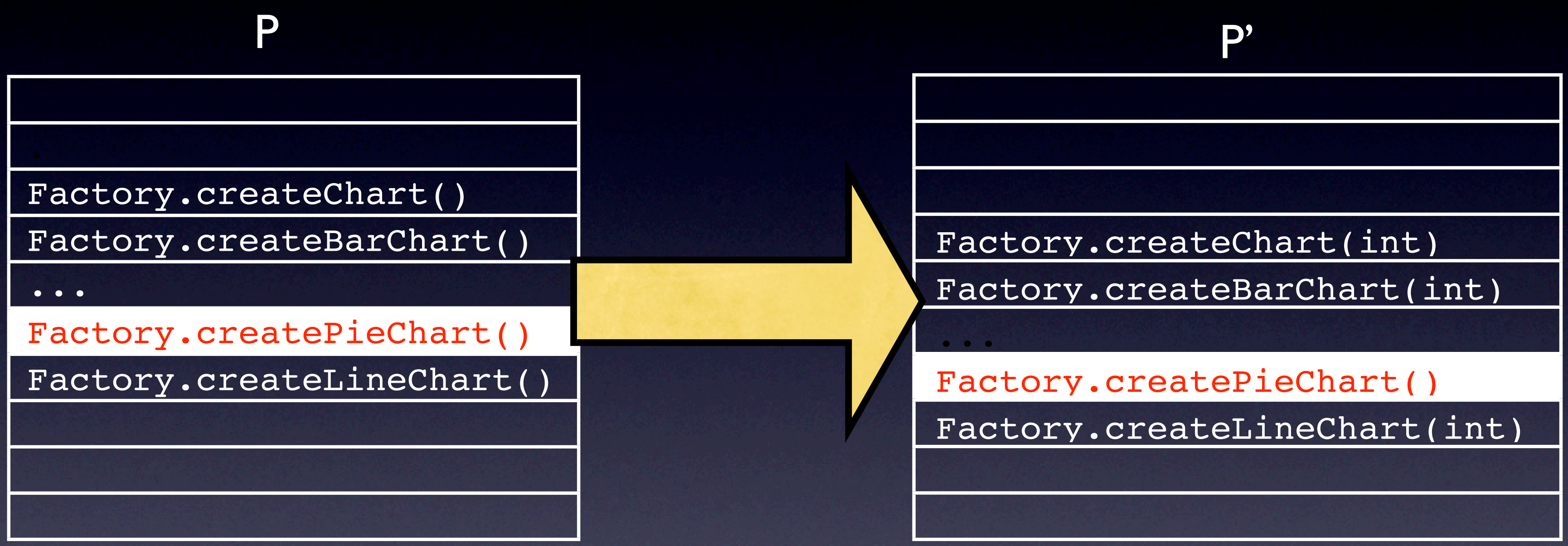
```
Factory.createChart()
Factory.createBarChart()
...
Factory.createPieChart()
Factory.createLineChart()
```

P’

```
Factory.createChart(int)
Factory.createBarChart(int)
...
Factory.createPieChart()
Factory.createLineChart(int)
```

```
FOR ALL x:method-header IN
  Factory.create*Chart(*)
    argAppend(x, [int])
```

Example Change-Rule



P

Factory.createChart()

Factory.createBarChart()

...

Factory.createPieChart()

Factory.createLineChart()

P'

Factory.createChart(int)

Factory.createBarChart(int)

...

Factory.createPieChart()

Factory.createLineChart(int)

```
FOR ALL x:method-header IN
  Factory.create*Chart(*)
    argAppend(x, [int])
except {Factory.createPieChart()}
```

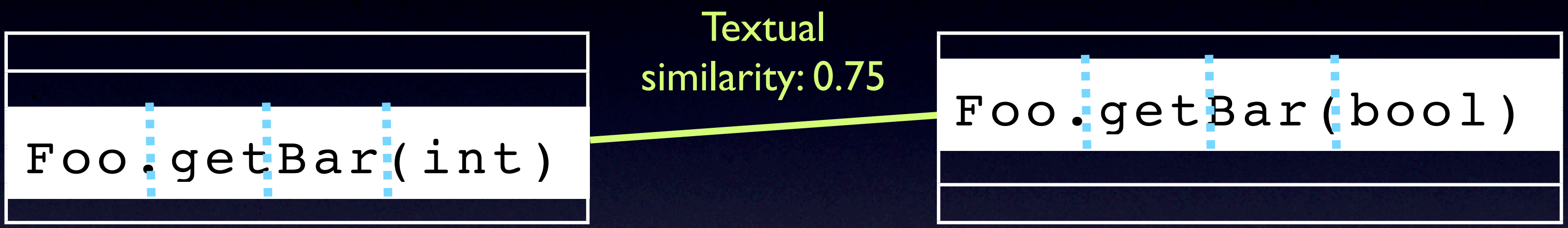
Algorithm Overview

Input: two versions of a program

Output: a set of change-rules

1. Generate seed matches
2. Generate candidate rules by generalizing seed matches
3. Evaluate and select candidate rules

Step 1: Generate Seed Matches



- Seed matches provide ***hints*** about likely changes.
- We generate seeds based on textual similarity between two method headers.
- Seed matches need not be all correct matches.

Step 2: Generate Candidate Rules

For each seed [x, y]

- Compare *x* and *y* and reverse engineer a set of transformations, T.
- Based on *x*, guess a set of scopes, S.
- Generate candidate rules for each pair in S × PowerSet(T).

```
Given a seed match,
[Foo.getBar(int), Boo.getBar(long)]

Transformations = {
replaceArg(x, int, long)
replaceClass(x, Foo, Boo)}

Scopes = {*.*(*), Foo.*(*), ...,
 *.get*(*), *.*Bar(*), ... ,
 Foo.get*(int),... }

Candidate Rules = {
 FOR ALL x IN *.*(*)
  replaceArg(x, int, long),
 FOR ALL x IN Foo.*(*)
  replaceClass(x, Foo, Boo), ...,
 FOR ALL x IN *.*(*)
  replaceArg(x, int, long) AND
  replaceClass(x, Foo, Boo)
... }
```

Step 3: Evaluate and Select Rules

- Greedily select a small subset of candidate rules that explain a large number of matches.
- In each iteration
  - evaluate all candidate rules
  - select a ***valid*** rule with the most number of matches
  - exclude the matched methods from the set of remaining unmatched methods
- Repeat until no rule can find any additional matches.

Optimizations

- We ***create*** and ***evaluate*** rules ***on demand***

1. Candidate rules have subsumption structure
   e.g., *.*.*(*Axis) ⊂ *.*.*(*)

2. The nature of greedy algorithm

- Running time: a few seconds (usual check-ins), average 7 minutes (releases)

Comparative Evaluation

- 3 other tools [Xing and Stroulia 05] [Weißgerber and Diehl 06] [S. Kim, Pan, and Whitehead 05]
- Evaluation data set (E)
- Precision

  $(|M \cap E| / |M|)$
- Recall

  $(|M \cap E| / |E|)$
- Conciseness

Comparison: Recall & Precision

	programs	Other's Recall	Our Recall	Other's Prec.	Our Prec.
[Xing & Stroulia 05]	jfreechart 18 releases	92%	98%	99%	97%
[Weissgerber & Diehl 06]	jEdit 2715 check-ins	72%	96%	93%	98%
	Tomcat 5096 check-ins	82%	89%	89%	93%
[Kim, Pan & Whitehead 05]	jEdit 1189 check-ins	70%	96%	98%	96%
	ArgoUML 4683 check-ins	82%	95%	98%	94%

Comparison: Recall & Precision

	programs	Other's Recall	Our Recall	Other's Prec.	Our Prec.
[Xing & Stroulia 05]	jfreechart 18 releases	92%	98%	99%	97%
[Weissgerber & Diehl 06]					98%
					93%
[Kim, Pan & Whitehead 05]					96%
	ArgoUML 4683 check-ins	82%	95%	98%	94%

Precision: 93-98%
Recall: 89-98%
6-26% higher recall with roughly the same precision

Comparison: Conciseness

	programs	Other's Results	Our Results	Our Improvement
[Xing & Stroulia 05]	jfreechart 18 releases	4004 refactorings	939 rules	77% decrease in size
[Weissgerber & Diehl 06]	jEdit 2715 check-ins	1218 refactorings	906 rules	26% decrease in size
	Tomcat 5096 check-ins	2700 refactorings	1033 rules	62% decrease in size
[Kim, Pan & Whitehead 05]	jEdit 1189 check-ins	1430 matches	1119 rules	22% decrease in size
	ArgoUML 4683 check-ins	3819 matches	2127 rules	44% decrease in size

*

Comparison: Conciseness

	programs	Other's Results	Our Results	Our Improvement
[Xing & Stroulia 05]	jfreechart 18 releases	4004 refactorings	939 rules	77% decrease in size
[Weissgerber & Diehl 06]				% decrease in size
				% decrease in size
[Kim, Pan & Whitehead 05]				% decrease in size
	ArgoUML 4683 check-ins	3819 matches	2127 rules	44% decrease in size

22-77% reduction in the size of matching results

Summary of Code Matching

- Our change-rules ***concisely*** capture ***API-level changes*** and identify anomalies to systematic changes

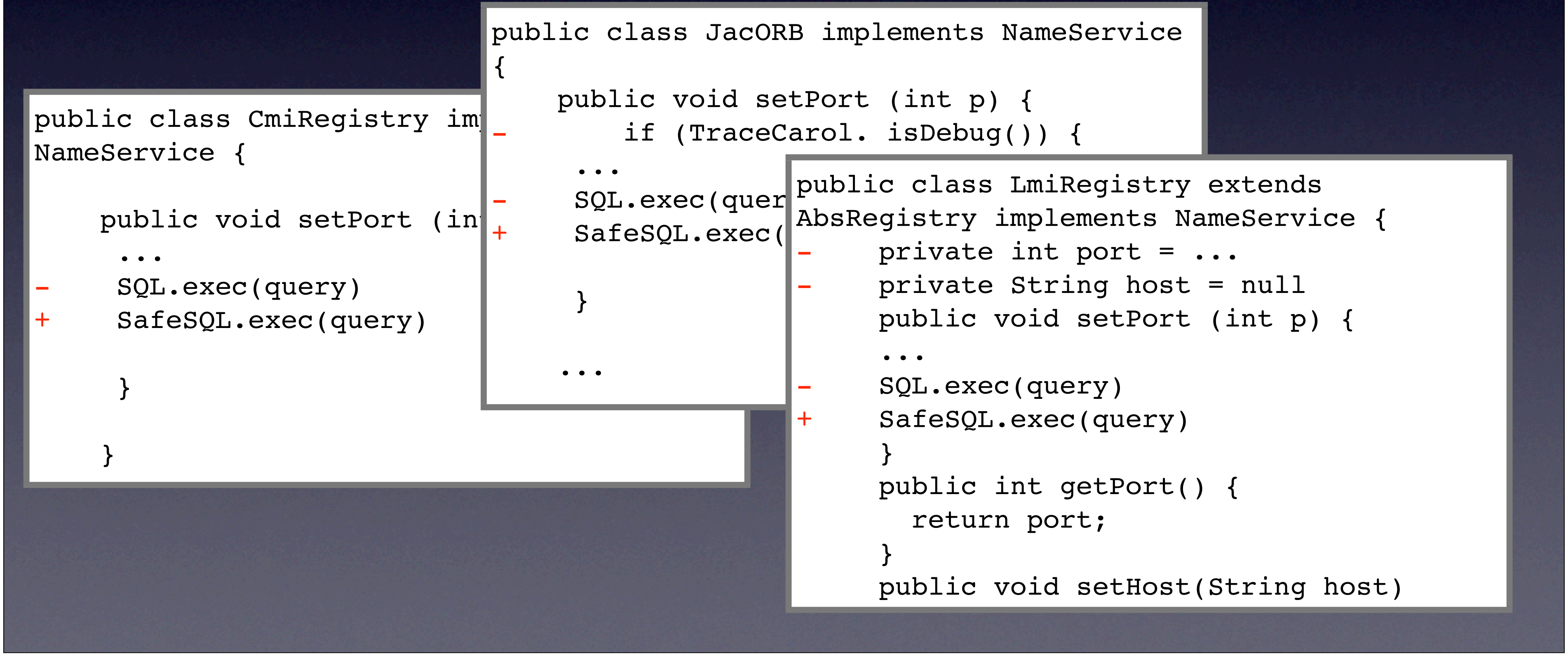
- By inferring such rules, we find method-header level matches with ***high recall and precision***

Outline

- Empirical Analyses of Code Clone Evolution
- Automatic Inference of High-Level Change Descriptions
  - Changes to API name and signature
  - Changes to Code Elements and Structural Dependencies (Logical Structural Diff)
- Future Directions

Research Question

"What is a concise change representation beyond API-level refactorings?"



```
public class CmiRegistry im
NameService {

    public void setPort (in
     ...
-    SQL.exec(query)
+    SafeSQL.exec(query)

     }

    }
```

```
public class JacORB implements NameService
{
    public void setPort (int p) {
-       if (TraceCarol. isDebug()) {
     ...
-    SQL.exec(quer
+    SafeSQL.exec(

     }

    ...
```

```
public class LmiRegistry extends
AbsRegistry implements NameService {
-    private int port = ...
-    private String host = null
     public void setPort (int p) {
     ...
-    SQL.exec(query)
+    SafeSQL.exec(query)
     }
     public int getPort() {
       return port;
     }
     public void setHost(String host)
```

Logical Structural Diff

Abstraction Level	**Code elements** and **structural dependencies** (package, type, method, field, overriding, subtyping, method call, field access, and containment)
Scope	**Conjunctive logic literal**
Transformation	**Structural differences** Account for changes in method-bodies as well as at a field level

Example Rule	`past_method(m,t)^`
	`past_subtype(“Factory”,t)^`
	`past_calls(m,“render()”)`
	`=> added_calls(m, “Util.log()”)`

Logical Structural Diff Algorithm

Output: ***logic rules*** and ***facts*** that describe changes to ***code elements and structural dependencies***

1. Extract a set of facts from a program using JQuery [Jensen & DeVolder 03]
2. Compute fact-level differences
3. Learn Datalog rules using an inductive logic programming algorithm

Logical Structural Diff Output

- “Replace all calls to SQL.exec with SafeSQL.exec”

```
deleted_calls(m,“SQL.exec”)=>
added_calls(m,“SafeSQL.exec”)
```

- “All setHost methods in Service’s subclasses in the old version deleted calls to SQL.exec except the setHost method in the NameSvc class.

```
past_subtype(“Service”, t) ∧ past_method
(m, “setHost”, t)
⇒ deleted calls(m, “SQL.exec”)
except t=“NameSvc”
```

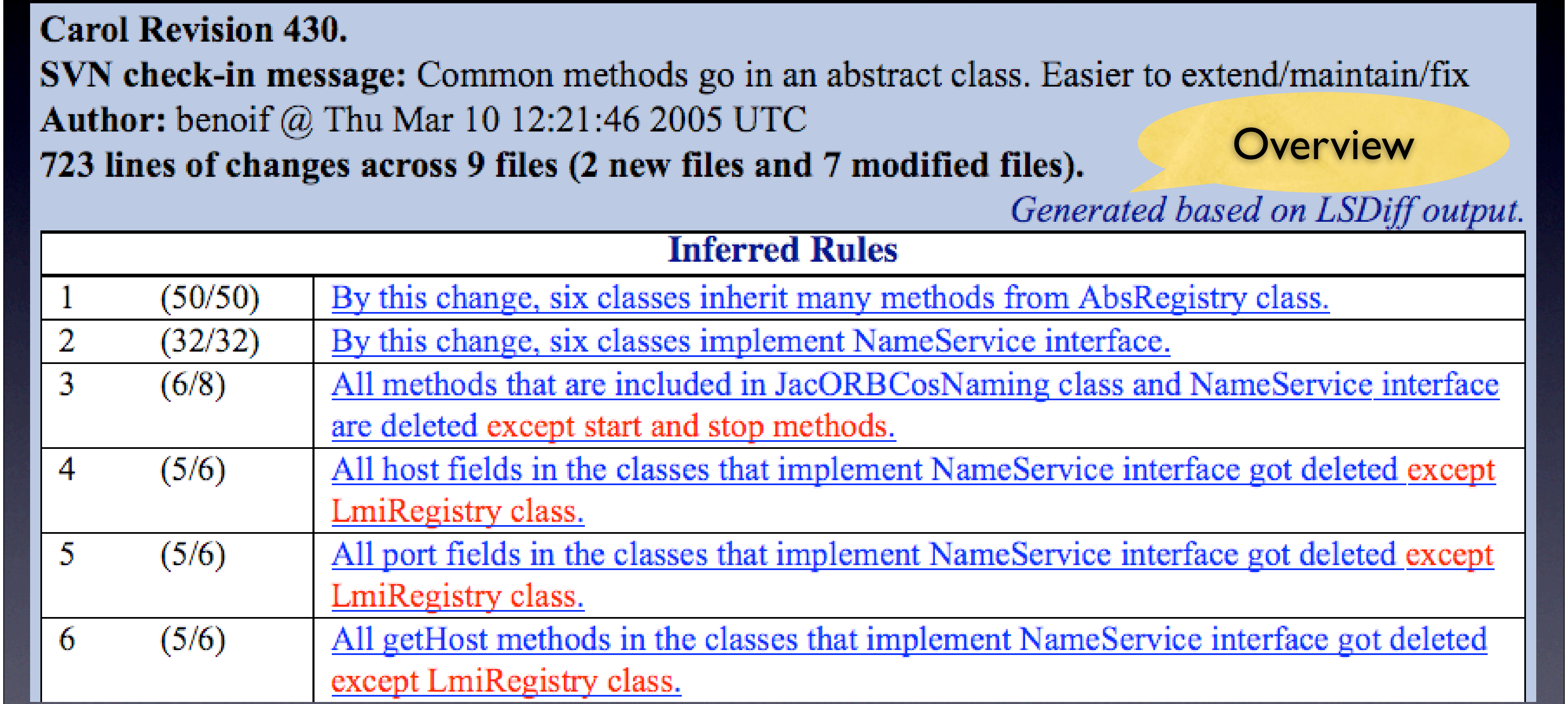
Quantitative Assessment of *LSDiff*

- 75% of fact-level differences are explained by rules.
- vs. fact-level delta: 9.3 times more concise
- vs. fact-level delta: 9.7 additional contextual facts
- vs. *Diff*: on average 7 rules and 27 facts for 997 lines of changes across 16 files

Focus Group Study

- Pre-screener survey
- Participants: five professional software engineers
  - industry experience ranging from 6 to over 30 years
  - use *diff* and *diff*-based version control system daily
  - review code changes daily except one who did weekly
- One hour structured discussion
  - I worked as the moderator. We also had a note-taker transcribe the discussion. Discussion was audio-taped and transcribed.

Focus Group Hands-On Trial



Carol Revision 430.
SVN check-in message: Common methods go in an abstract class. Easier to extend/maintain/fix
Author: benoif @ Thu Mar 10 12:21:46 2005 UTC
723 lines of changes across 9 files (2 new files and 7 modified files).

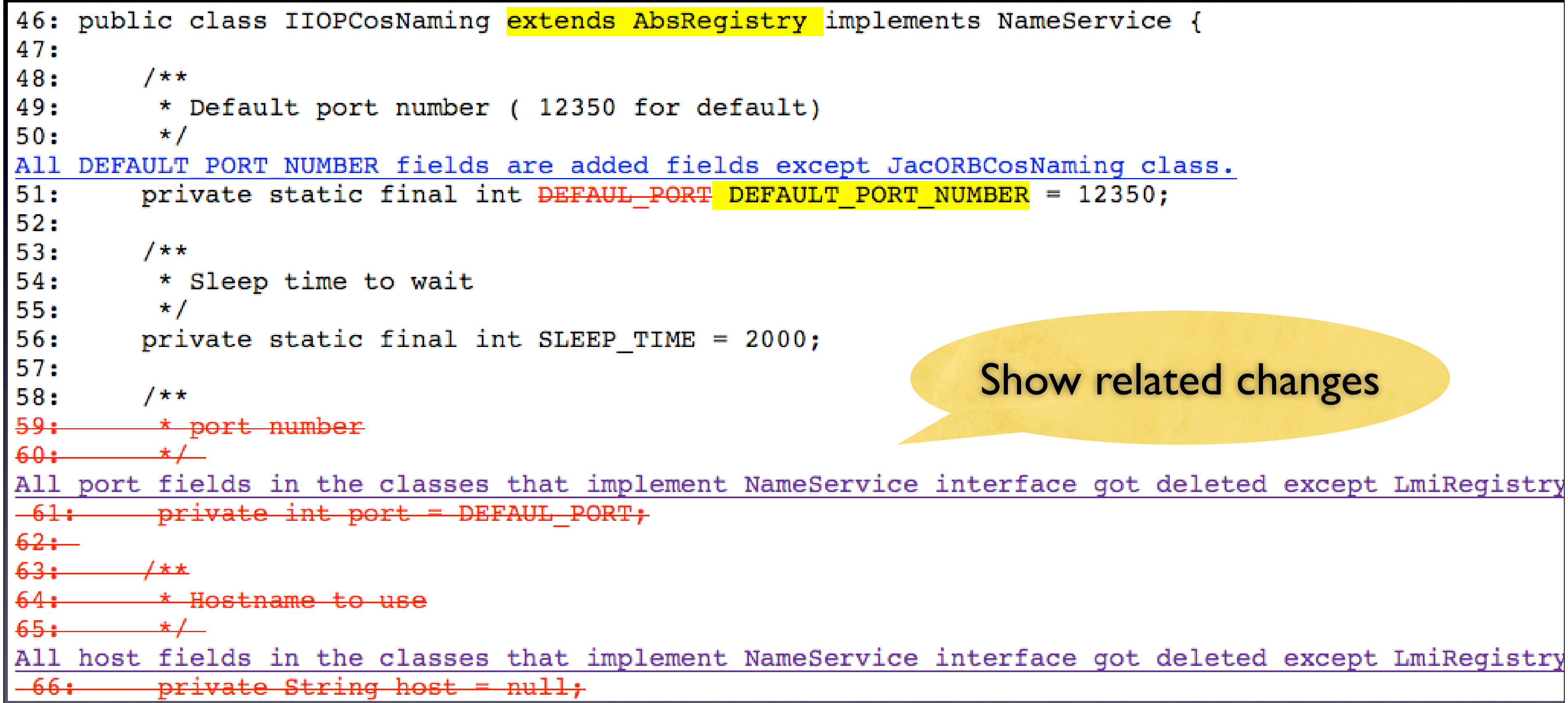
Overview

Generated based on LSDiff output.

Inferred Rules		
1	(50/50)	By this change, six classes inherit many methods from AbsRegistry class.
2	(32/32)	By this change, six classes implement NameService interface.
3	(6/8)	All methods that are included in JacORBCosNaming class and NameService interface are deleted except start and stop methods.
4	(5/6)	All host fields in the classes that implement NameService interface got deleted except LmiRegistry class.
5	(5/6)	All port fields in the classes that implement NameService interface got deleted except LmiRegistry class.
6	(5/6)	All getHost methods in the classes that implement NameService interface got deleted except LmiRegistry class.

http://www.cs.washington.edu/homes/miryung/LSDiff/carol429-430.htm

Focus Group Hands-On Trial



```
public class IIOPCosNaming extends AbsRegistry implements NameService {

    /**
     * Default port number ( 12350 for default)
     */
All DEFAULT_PORT_NUMBER fields are added fields except JacORBCosNaming class.
    private static final int ~~DEFAUL_PORT~~ DEFAULT_PORT_NUMBER = 12350;

    /**
     * Sleep time to wait
     */
    private static final int SLEEP_TIME = 2000;

    /**
~~     * port number~~
~~     */~~
All port fields in the classes that implement NameService interface got deleted except LmiRegistry
~~    private int port = DEFAUL_PORT;~~

~~    /**~~
~~     * Hostname to use~~
~~     */~~
All host fields in the classes that implement NameService interface got deleted except LmiRegistry
~~    private String host = null;~~
```

Show related changes

http://www.cs.washington.edu/homes/miryung/LSDiff/carol429-430.htm

Focus-Group Participants' Comments

"You can't infer the intent of a programmer, but this is pretty close."

"This 'except' thing is great!"

"This is cool. I'd use it if we had one."

"This is a definitely winner tool."

Focus-Group Participants' Comments

"This looks great for big architectural changes, but I wonder what it would give you if you had lots of random changes."

"This wouldn't be used if you were just working with one file."

"This will look for relationships that do not exist."

Summary of Logical Structural Diff

- We extended our rule-based approach to infer systematic changes ***within method bodies***
- LSDiff produces 9.3 times more concise results by identifying 75% of structural differences as systematic changes
- LSDiff complements *diff*
  - by grouping systematic structural differences
  - by detecting potential missed updates.

Outline

- Empirical Analyses of Code Clone Evolution
- Automatic Inference of High-Level Change Descriptions
  - Changes to API name and signature
  - Changes to Structural Dependencies
- Future Directions

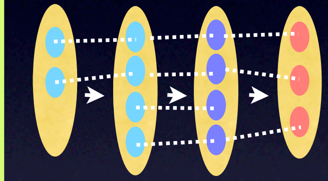
Next Steps

- Develop higher-order representations
- Use change-rules to improve regression testing
- Use change-rules to backport security patches to old versions
- Search program changes of interest in a source code repository by evaluating programmer-provided rules

My long-term vision is to help programmers ***by making software change a first class entity***

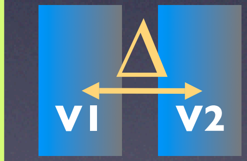
- Changes in models, requirements, and run-time behavior
- Use change history to help programmers make decisions
    - “When and how should I refactor my program?”

Contributions



Analyses of Software Evolution

- Disproving conventional wisdom about clones
- Insights into systematicness of high-level changes



Automatic Inference of High-Level Change Descriptions

- Rule-based change representations
- Rule learning algorithms

Questions?