Analyze Software Changes and Versions

Wei Le

April 1, 2019

Why we should care?

- Agile development and continuous integration (small changes, fast delivery)
- It is hard to get software changes correct (some of the patches are buggy)
- Quality assurance techniques need to be flexible and provide fast feedback

Topics

- Git log analysis
- MVICFG (multiversion control version graphs) and patch verification
- Impact analysis, regression testing
- History analysis (history slicing, the origin of the bug)
- Debugging changes
- Patch testing
- Multiversion analysis
- Compare and difference programs
- Differential assertions, change contracts

Git 101 for analyzing changes in history

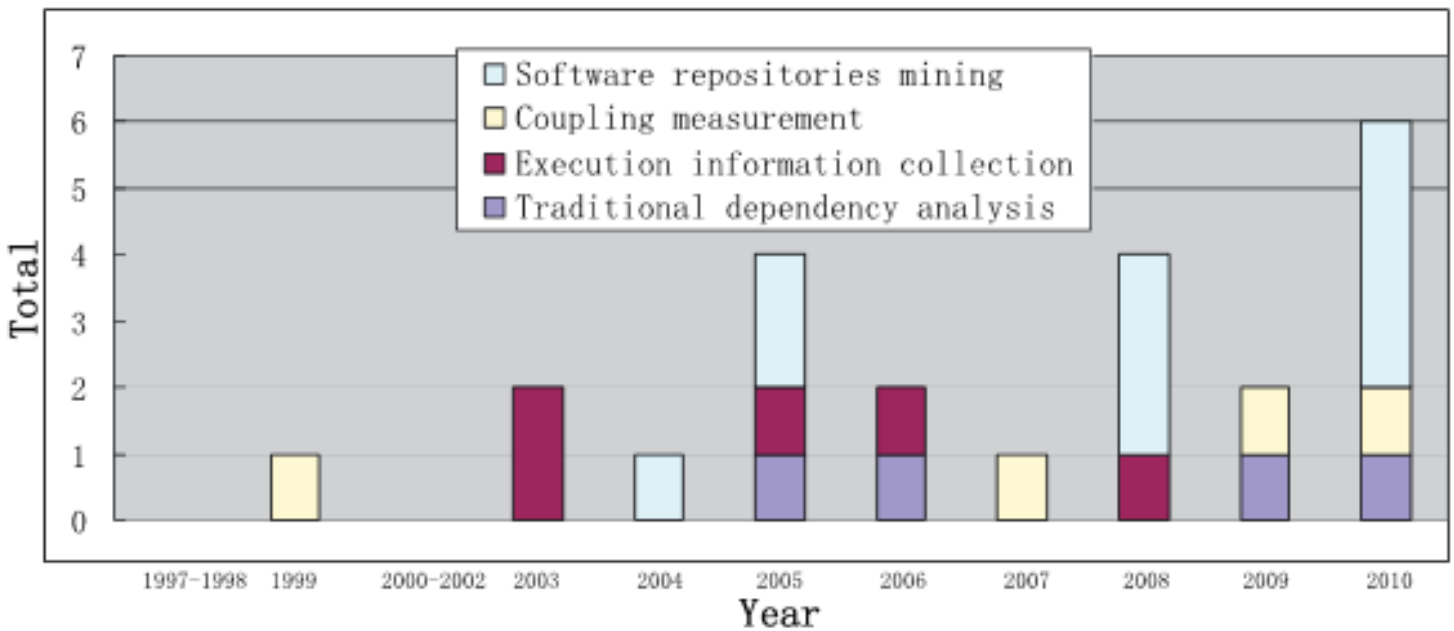
- software repository: code + test cases+ doc
- *code churn*: textual diffs
- git log -p //show the patch
- git log –pretty=oneline
- git blame -L 10,10 hello.c //trace the history of a file or a line

MVICFG

See Wei Le's ICSE slides

Change Impact Analysis

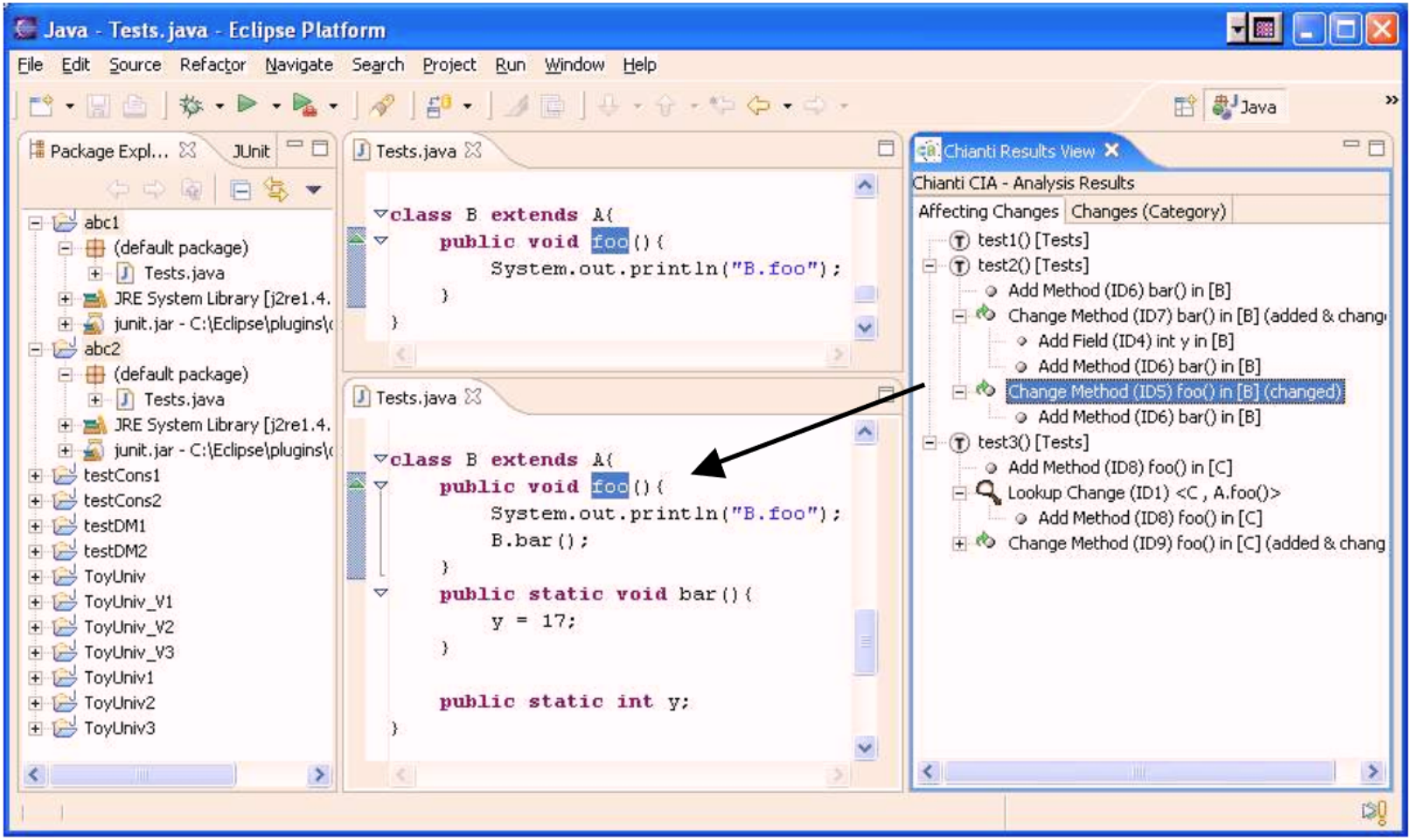
change impact analysis: software impact analysis identifies the effects of a software change request.
regression testing: testing for changed software – select and prioritize test inputs that likely exercise the changes



Distribution of the change impact analysis techniques from four perspectives.

Change Impact Analysis

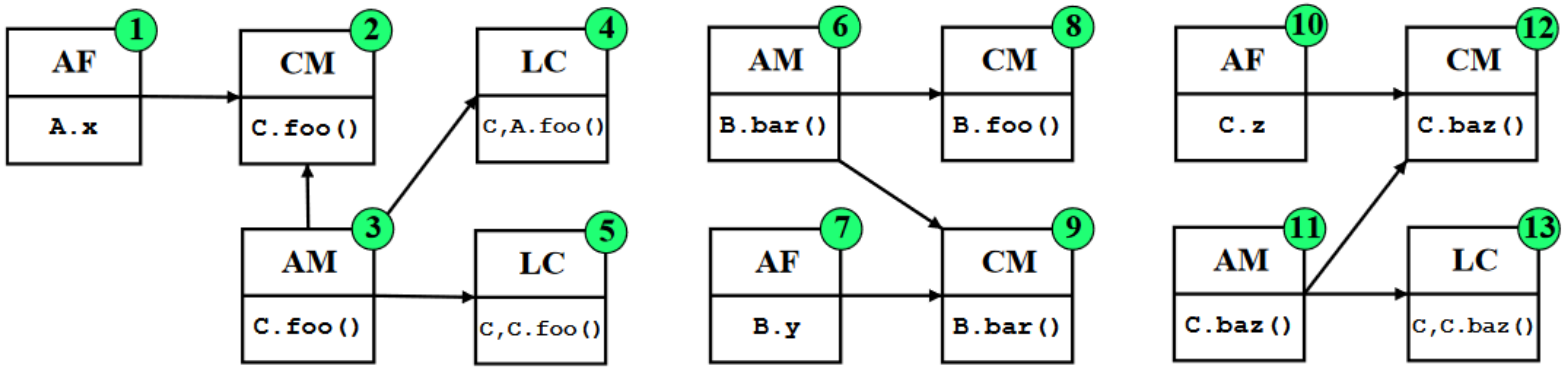
- Impact for single programs: forward slicing
- Impact for changes:
  - Chianti is a change impact analysis tool for Java that is implemented within eclipse
  - Analyse two versions of a Java program
  - Decompose their difference into a set of atomic changes
  - Calculate a partial order of inter-dependencies of these changes
  - Report change impact in terms of affected (regression or unit) tests whose execution behavior may have been modified by the applied changes.
  - For each affected test, determine a set of affecting changes that were responsible for the test's modified behavior.


Java - Tests.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Package Expl... JUnit
abc1
(default package)
Tests.java
JRE System Library [j2re1.4.
junit.jar - C:\Eclipse\plugins\
abc2
(default package)
Tests.java
JRE System Library [j2re1.4.
junit.jar - C:\Eclipse\plugins\
testCons1
testCons2
testDM1
testDM2
ToyUniv
ToyUniv_V1
ToyUniv_V2
ToyUniv_V3
ToyUniv1
ToyUniv2
ToyUniv3
Tests.java
class B extends A{
public void foo(){
System.out.println("B.foo");
}
}
Tests.java
class B extends A{
public void foo(){
System.out.println("B.foo");
B.bar();
}
public static void bar(){
y = 17;
}
public static int y;
}
Chianti Results View
Chianti CIA - Analysis Results
Affecting Changes
Changes (Category)
test1() [Tests]
test2() [Tests]
Add Method (ID6) bar() in [B]
Change Method (ID7) bar() in [B] (added & chang
Add Field (ID4) int y in [B]
Add Method (ID6) bar() in [B]
Change Method (ID5) foo() in [B] (changed)
Add Method (ID6) bar() in [B]
test3() [Tests]
Add Method (ID8) foo() in [C]
Lookup Change (ID1) <C , A.foo()>
Add Method (ID8) foo() in [C]
Change Method (ID9) foo() in [C] (added & chang

```
class A {
  public A(){ }
  public void foo(){ }
  public int x;
}
class B extends A {
  public B(){ }
  public void foo(){ B.bar(); }
  public static void bar(){ y = 17; }
  public static int y;
}
class C extends A {
  public C(){ }
  public void foo(){ x = 18; }
  public void baz(){ z = 19; }
  public int z;
}
```

```
class Tests {
  public static void test1(){
    A a = new A();
    a.foo();
  }
  public static void test2(){
    A a = new B();
    a.foo();
  }
  public static void test3(){
    A a = new C();
    a.foo();
  }
}
```

(a)



(b)

History slicing: assisting code-evolution tasks (2012)

High level:

- assist developers' tasks for software maintenance
- questions about history: like when, how, by whom, and why somecode was changed or inserted.
- visualization of the entire evolution for the code of interest, efficient inspection of a sequence ofchanges for an arbitrary block of code.
- *history slice* for a set of lines of code of interest (i.e., slicing criterion) contains all their corresponding lines of code in all past revisions of the software project in which they were modified.

History slicing: assisting code-evolution tasks

Motivating Examples:

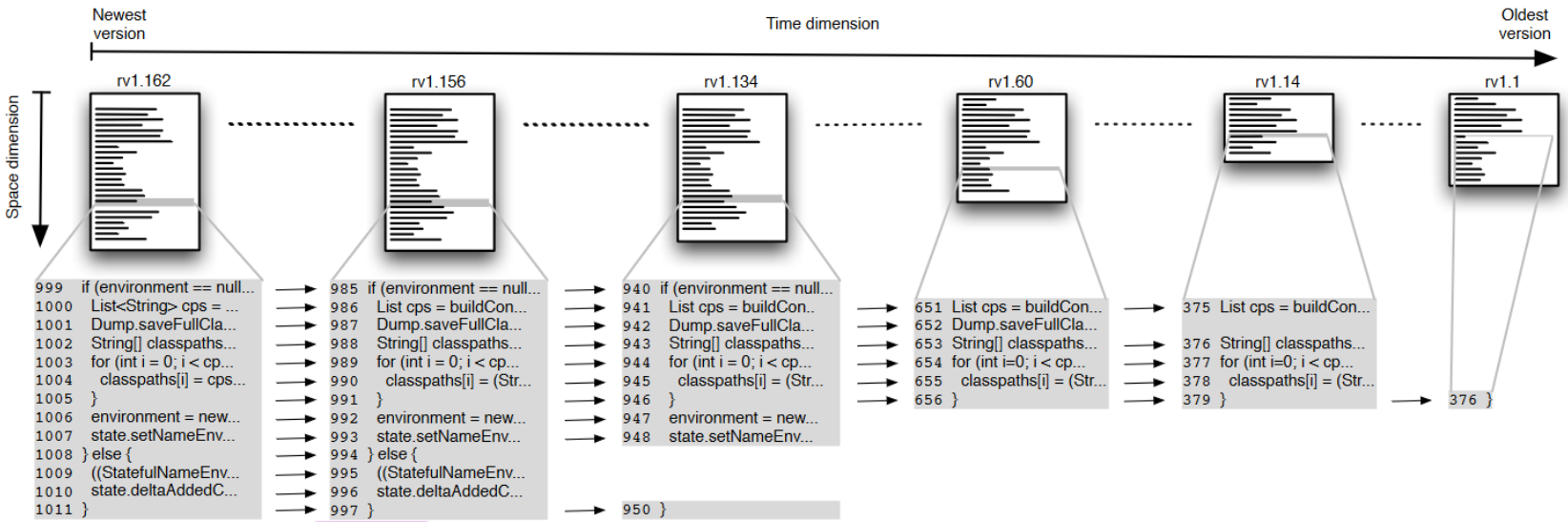
- find a better implementation of the loop in the history
- who modified this section of code
- developers may want to explore the parallel history of multiple segments of source code in order to find out whether and when they were modified together. (evolution coupling)

History slicing: assisting code-evolution tasks

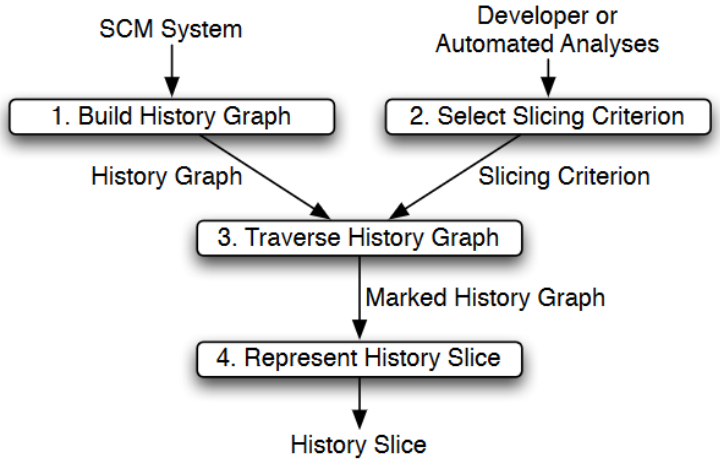
Key idea:

- define *s*napshot as the set of lines of code in a particular version that correlate, either directly or transitively,to the original lines of interest; i.e., a snapshot represents a previous state of the lines of interest.
- how to find the snapshot:
  - Retrieve the previous revision r of a file
  - Find inside revision r which lines correspond to the lines of interest
  - Check the contents of those lines and identify whether they were modified
  - If they were modified, save them
  - Return to Step 1until all history is explored.

History slicing: assisting code-evolution tasks



History slicing: assisting code-evolution tasks



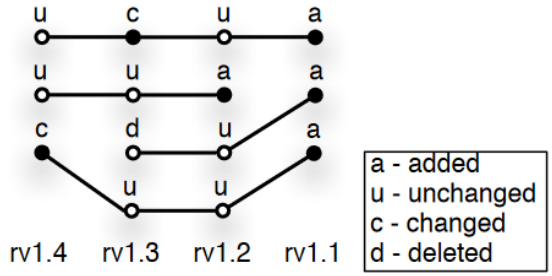
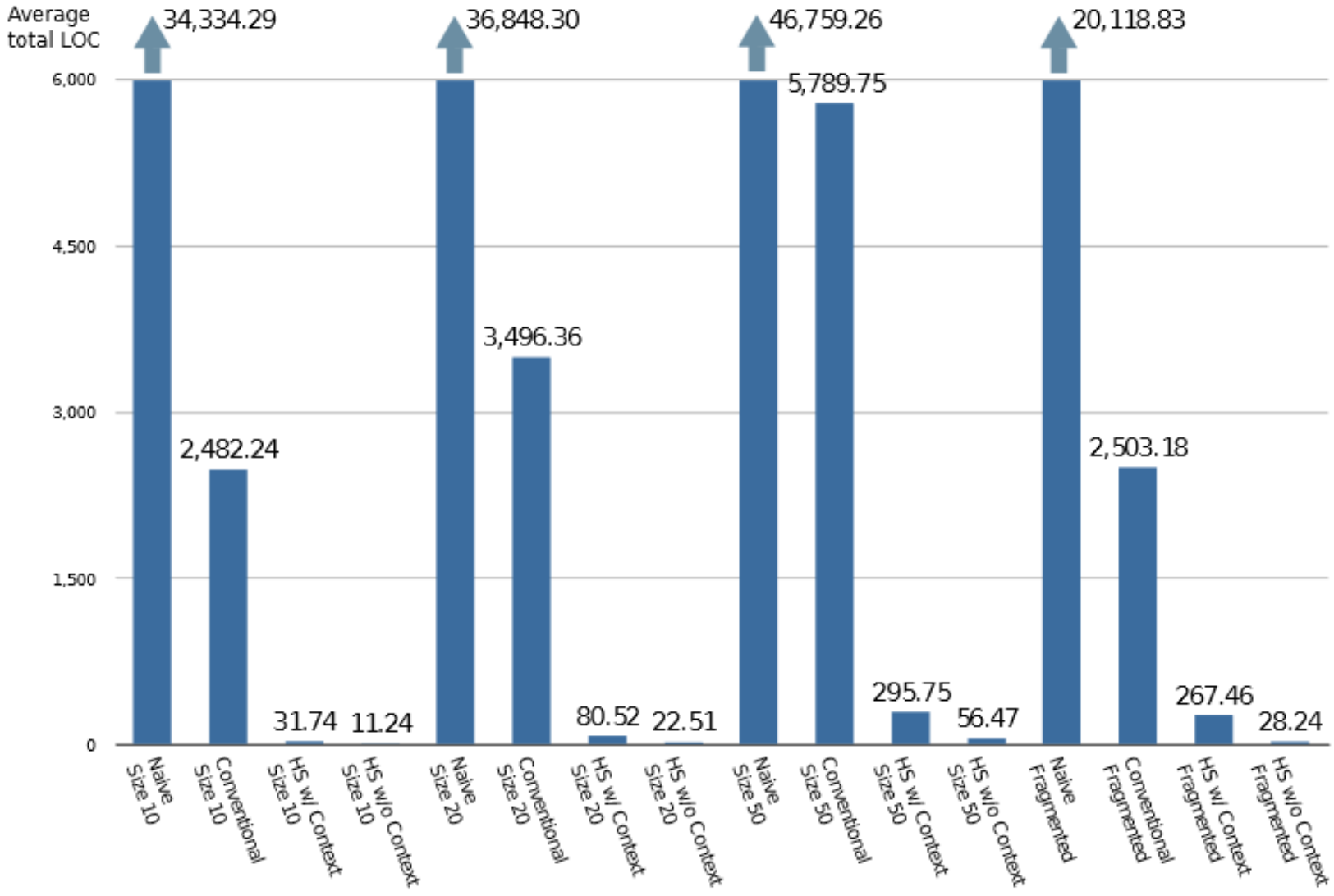


Figure 3: History Graph.

History slicing: assisting code-evolution tasks



History slicing: assisting code-evolution tasks

Technique	Task	Avg. time	% Success
Conventional	Task 1	6:04	37.5%
History Slicing	Task 1	3:21	100%
Conventional	Task 2	7:34	37.5%
History Slicing	Task 2	3:15	100%
Conventional	Task 3	9:57	0%
History Slicing	Task 3	5:19	62.5%

max 10 min

- identify the complete set of developers who had ever contributed changes to a segment of code
- identify the original revisions in which a segment of code was originally created.
- identify the revisions in which two segments of code in two different files were changed within a day of each other.

Thoughts and Discussions

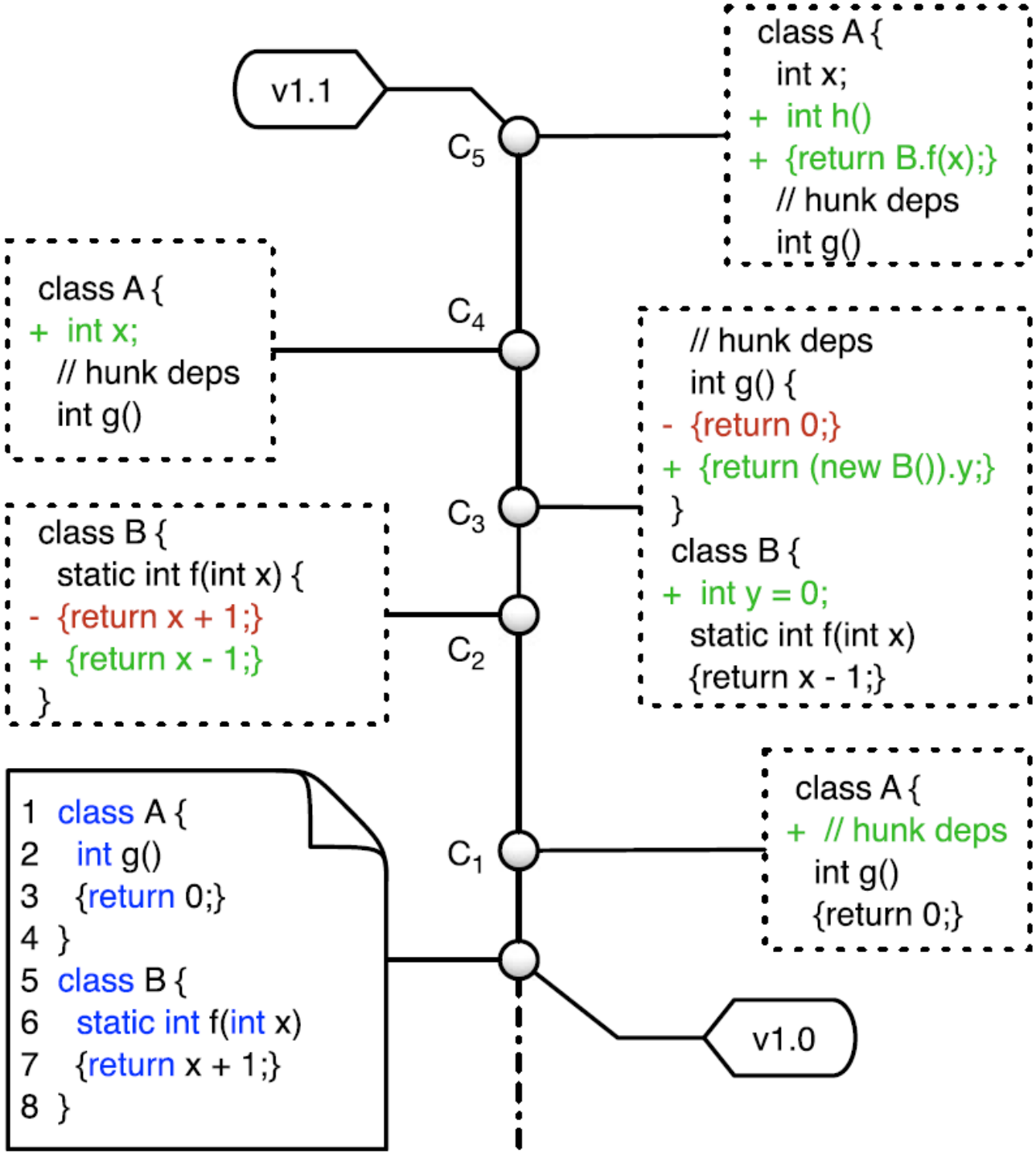
Semantic Slicing of Software Version Histories (2017)

- Problem: identify the exact set of commits that implement the functionality of interest (which is defined by a set of tests) or sequentially port a segment of the change history.
- Approach: identify a set of commits that constitute a slice, and minimize the produced slice.

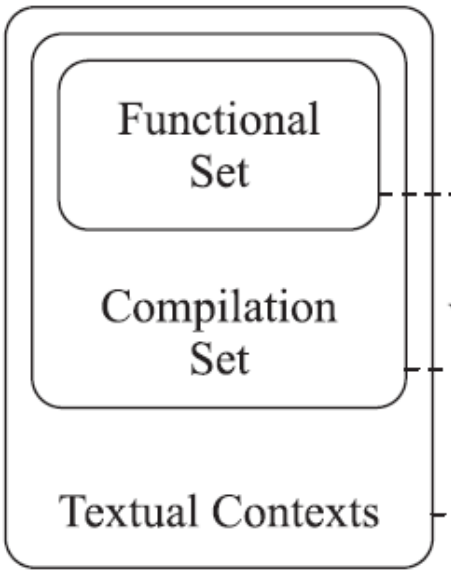
Semantic Slicing of Software Version Histories (2017)

Why we should care?

- Locating and transferring functionality from one branch to another, e.g., for bug fixes
- splitting large chunk commits into multiple functionally independent pull requests
- identifying failure inducing changes


v1.1
C5
class A {
int x;
+ int h()
+ {return B.f(x);}
// hunk deps
int g()
C4
class A {
+ int x;
// hunk deps
int g()
C3
// hunk deps
int g() {
- {return 0;}
+ {return (new B()).y;}
}
class B {
+ int y = 0;
static int f(int x)
{return x - 1;}
C2
class B {
static int f(int x) {
- {return x + 1;}
+ {return x - 1;}
}
C1
class A {
+ // hunk deps
int g()
{return 0;}
1 class A {
2 int g()
3 {return 0;}
4 }
5 class B {
6 static int f(int x)
7 {return x + 1;}
8 }
v1.0

target functionality: A.h() solutions: C1, C2. C4, C5



Correctness

Well-formedness

Applicability

Dependency Types	Examples
Functional	C_2 , C_5
Compilation	C_4
Hunk	C_1

Semantic Slicing of Software Version Histories

p is a syntactically valid program of language P, denoted by $p \in P$, if p follows the syntax rules.

Definition 6 (Semantics-preserving Slice). *Consider a program p_0 and its k subsequent versions $p_1, \ldots, p_k$ such that $p_i \in P$ and p_i is well-typed for all integers $0 \leq i \leq k$. Let H be the change history from p_0 to p_k, i.e., $H_{1..i}(p_0) = p_i$ for all integers $0 \leq i \leq k$. Let T be a set of tests passed by p_k, i.e., $p_k \models T$. A semantics-preserving slice of history H with respect to T is a sub-history $H' \lhd H$ such that the following properties hold:*

1) $H'(p_0) \in P$,
2) $H'(p_0)$ *is well-typed*,
3) $H'(p_0) \models T$.

Semantic Slicing of Software Version Histories

Workflow:

1. Computing functional set: Executes the test on the latest version of the program. It dynamically collects the program statements traversed by this execution. These include the method bodies of A.h and B.f (the execution traces in the program after slicing remain unchanged, then the test results will be preserved)
2. Computing compilation set: CSLICER statically analyzes all the reference relations based on pk and transitively includes all referenced entities in the compilation set
3. Changeset slicing: iterates backwards from the newest change set Dk to the oldest one D1, collecting changes that are required to preserve the "behavior" of the functional and compilation set elements.

Semantic Slicing of Software Version Histories

Slice minimization problem:

- input: a base version program p0, a semantics-preserving history slice H and the target test suite T
- output: Minimal slice
- approach: static pattern matching
  - remove insignificant changes that may not affect tests, such as refactoring, local refacotring/ rewriting, low impact modifier changes such as removal of the final keyword and update from protected to public, as well as white list statement updates such as modifications to printing and logging method invocations.
  - also consider users' input on which parts of the code may not affect test cases
- approach: dynamic sub-history: cherry pick commits that may affect test results using topological sort

Semantic Slicing of Software Version Histories

Case	Project	#Files	LOC	$\|H\|$	Changed			$\|T\|$
					f	+	−	
1	Hadoop	5,861	1,291 K	267	1,197	111,119	14,064	58
2	Elasticsearch	3,865	616 K	51	75	1,755	304	2
3	Maven	967	81 K	50	16	1,012	250	7
	Collections	525	62 K	39	46	1,678	323	13
	Math	1,410	188 K	33	34	1,531	359	1
	IO	227	29 K	26	59	975	468	13

Each row lists the number of Java files (#Files), lines of code (LOC) of the studied projects, the length of the chosen history fragment ($|H|$), the number of changed files (f), lines added (+), and lines deleted (-) for the chosen range, and the number of test cases ($|T|$) in the target test suites.

Semantic Slicing of Software Version Histories

Evaluation 1: qualitative assessment

- Branch refactoring: Hadoop, input 267 commits, 58 tests =¿ 91 commits, 750 second
- Back porting commits: Elasticsearch, input 51 commits, optimal: 4 commits, CSlicer: 17 commits (test cases will cover code that is not intended)
- Create clean pull requests (untangle commits): miss commits, add more commits

Semantic Slicing of Software Version Histories

Evaluation 2: quantitative encasement
Evaluation 3: delta debugging

- 10x to 100 x faster than delta debugging

Thoughts and Discussions

Buginnings: Identifying the Origins of a Bug (2010)

Problem:

- *Origin of the bug*: code changes that introduced a bug
- Why? defect age, defect residency time, learn patterns of bug introducing changes, why failed to detect such bugs
- Run tests and see incorrect results at V_i but not at V_{i-1}
- Program dependencies: data dependencies, control dependencies, and call relations

Buginnings: Identifying the Origins of a Bug (2010)

Solution:

- Computing bug regions: start at the bug fix version V_n and its previous version V_{n-1}, compute differences between the bug fix version and the previous version to identify the bug fix changes based on program dependency graph
  - for deleted dependencies
  - for added dependencies
  - for just modified statement
- traverse backward in the code revision history to identify the versions in which the affected parts were last touched

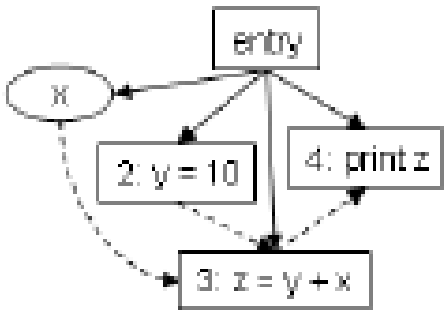
Buginnings: Identifying the Origins of a Bug (2010)

```
1. public void f( int x ) {
2.     int y = 10;
3.     z = y + x;
4.     print z;
   }
```

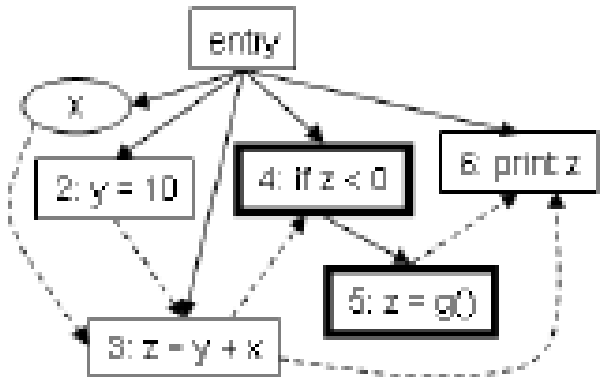
version 1

```
1. public void f( int x ) {
2.     int y = 10;
3.     z = y + x;
4.     if ( z < 0 ) // added
5.         z = f();  // added
6.     print z;
   }
```

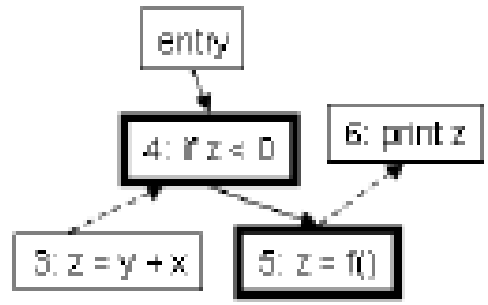
version 2 (bug fix)



(a) PDG for version 1 (G_1)

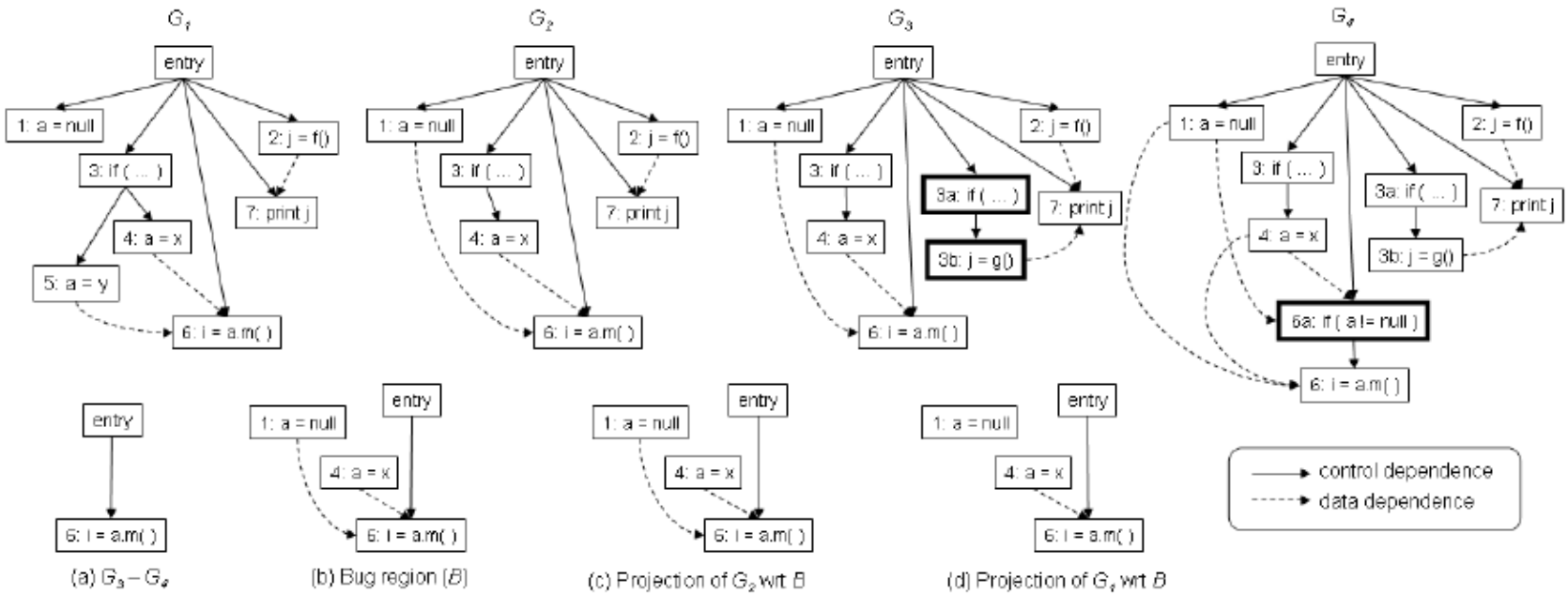


(b) PDG for version 2 (G_2)



(c) Added dependences ($G_2 - G_1$)

Buginnings: Identifying the Origins of a Bug (2010)



Buginnings: Identifying the Origins of a Bug (2010)

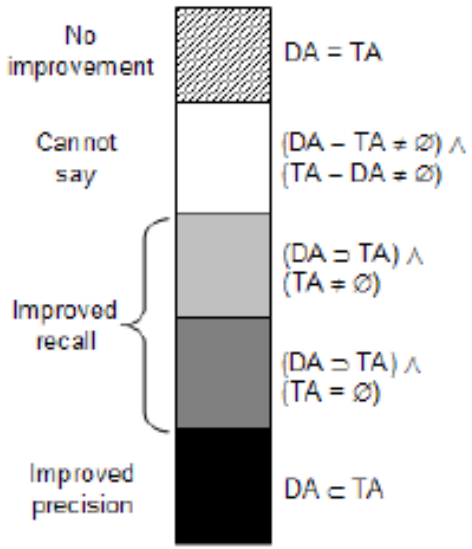
Evaluation:

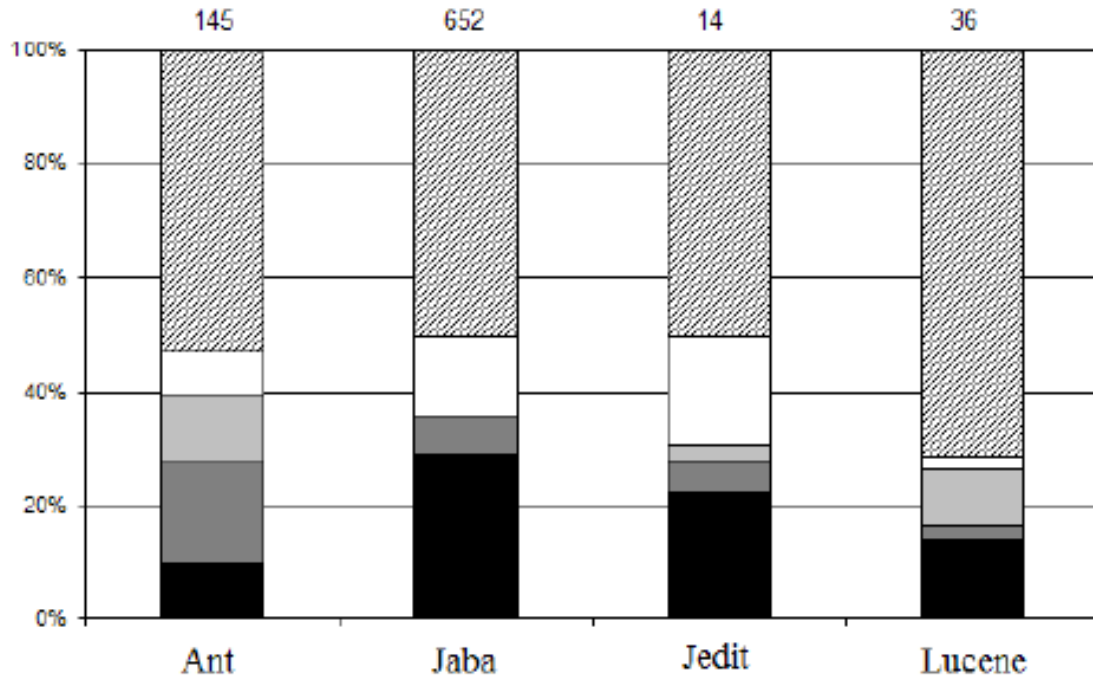
- identify bug fix commits : git log –all –grep = "bugs"
- subjects

Subject	Version history	Lines of code (last version)	Number of trans	Bug-fix trans
Ant	Sep 2003 – Jan 2006	95557	446	59 (13%)
Jaba	Jul 2003 – Oct 2005	40536	113	19 (17%)
Jedit	Jun 2006 – Dec 2006	65148	406	72 (18%)
Lucene	Jan 2004 – Dec 2006	21297	1485	129 (9%)
Average			612	70 (14%)

Buginnings: Identifying the Origins of a Bug (2010)

Results: better precision for 19% of bug fixes, better recall for 15% bug fixes





Comparison of results computed by our approach (DA) and the text approach (TA). T

Buginnings: Identifying the Origins of a Bug (2010)

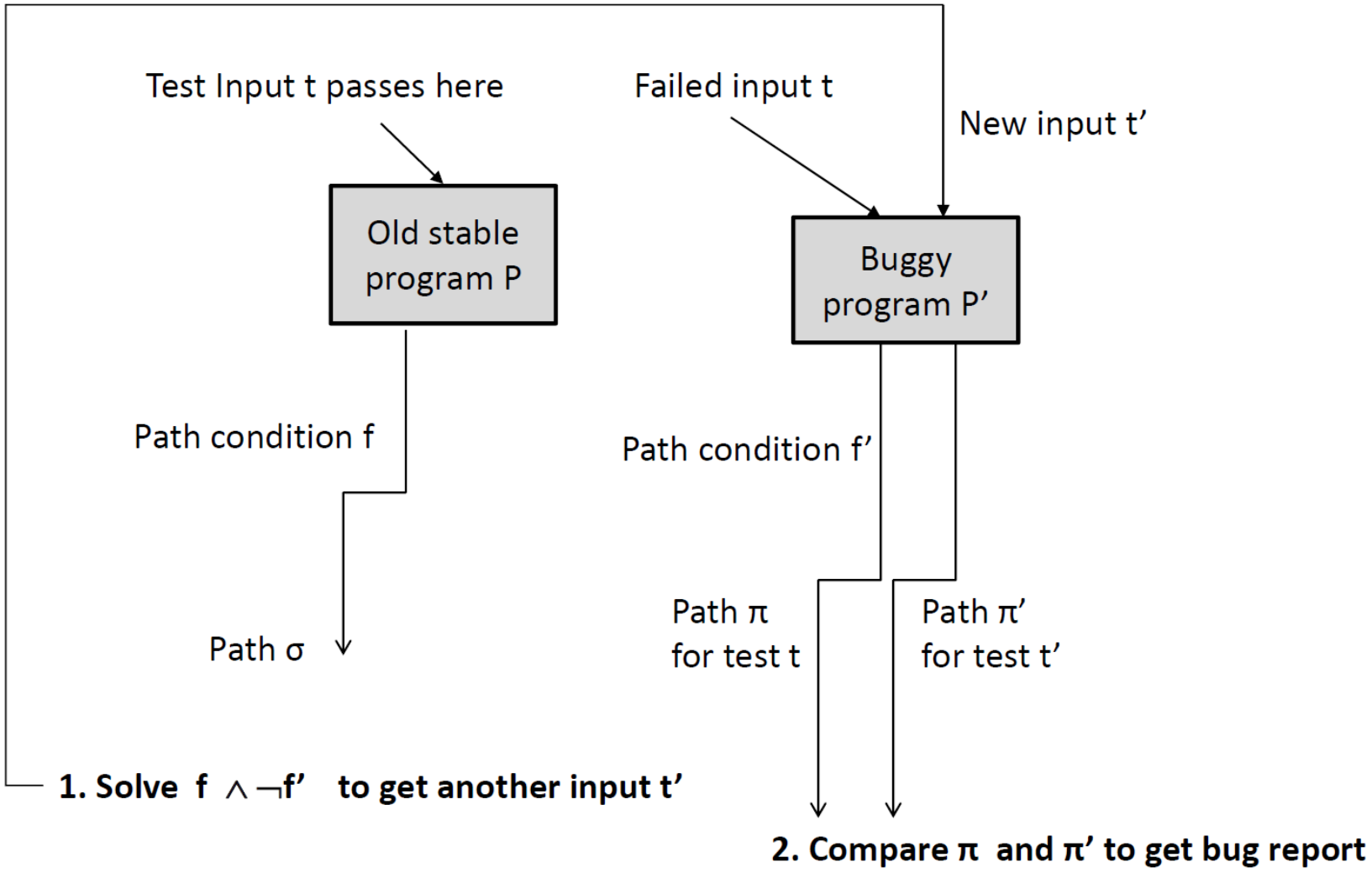
Results: Performance (TA vs DA) 7.2 times more than TA on average
Ant: 28 min vs 5.75 hours
Jaba: 1.8 min vs 58 min

Thoughts and Discussions

Debugging changes: DARWIN

- *Motivation*: debugging - find causes of failures
- *Problem statement*:
  - **input**: a stable program P, a modified program P', input t that passes on the stable program but fails on the modified program (P and P' can be even different implementations as long as they form to the same specification, documented using a set of test suite T)
  - **output**: bug report (branches in P' and or in P that can explain the bug)
  - note: can handle code missing errors by pointing out the relevant code
- *Overall approach*: generate new input t', such that t and t' take the same path i P but in different path in P'. t' pass both P and P'; compare the trace of t and t', we then can identify the likely causes; work for binary code
- *Evaluation*: Libpng, webserver programs like miniweb, savant and apache

DARWIN: overall approach



DARWIN: an example

```
int inp, outp;
scanf("%d", &inp);
if (inp !=1){
outp = g(inp);
} else{
outp = h(inp);
}
printf("%d", outp);
```

Program P


```
int inp, outp;
scanf("%d", &inp);
if (inp !=1 && inp !=2){
outp = g(inp);
} else{
outp = h(inp);
}
printf("%d", outp);
```

Program P'

Problem: When inp == 2, P' fails

input/versions	P	P'
inp = 1	else	else
inp = 2	if	else
inp = 3	if	if

Solution:

- DARWIN generates inp == 3, where inp = 2 and inp =3 lead to the same paths in P, but different paths in P', inp == 3 passes
- the branch inp≠1 && inp≠2 is highlighted as a root cause

DARWIN: the idea

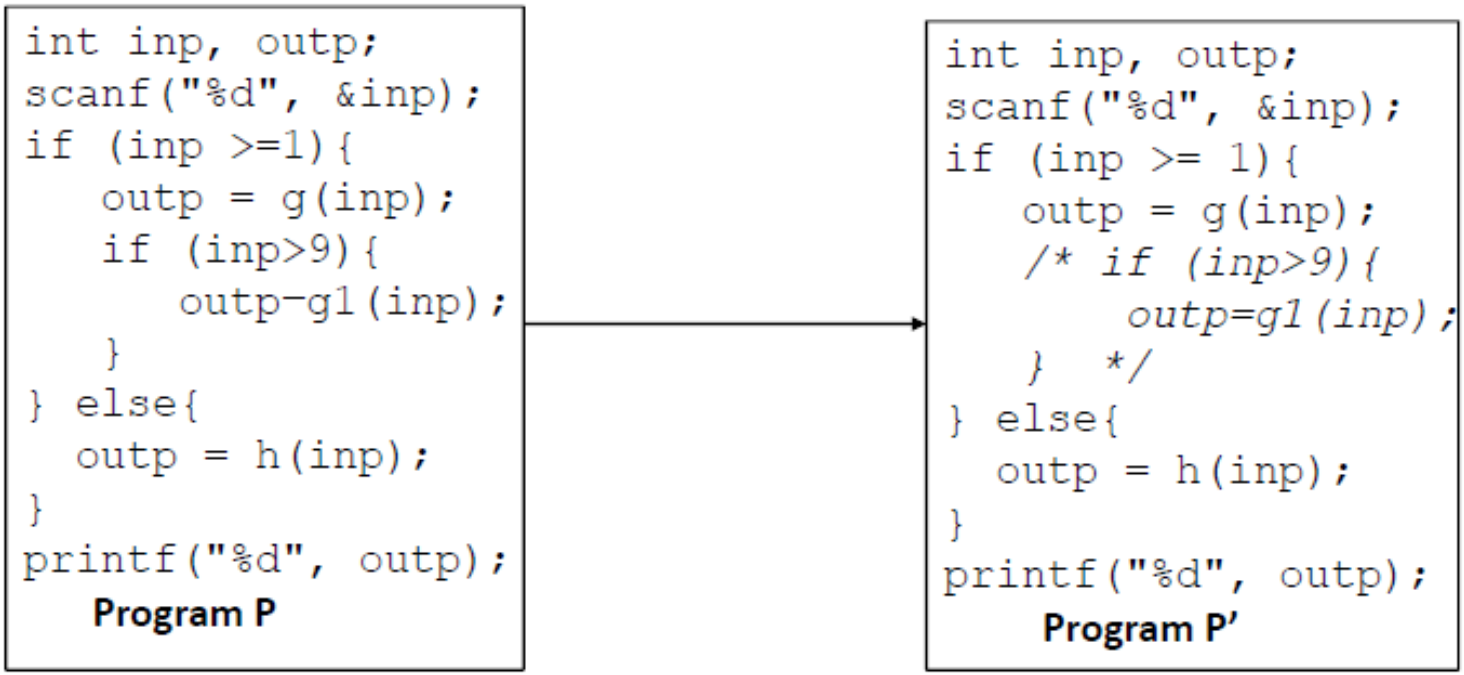
When P changes to P', the mapping of inputs to paths changed, find more than one input that can show differences in P or in P', reduce the problem to fault localization problems for a single version of program

DARWIN: concrete steps

- Compute f, the path condition of t in P.
- Compute f', the path condition of t in P'.
- Check whether f $\wedge\neg$ f' is satiable. If yes, it yields a test input t'. Compare the trace of t' in P' with the trace of t in P'. Return bug report.
- If f $\wedge\neg$ f' is unsatisfiable, find a solution to f' $\wedge\neg$ f. This produces a test input t'. Compare the trace of t' in P with the trace of t in P. Return bug report

- generate and run more than one input
- symbolic constraints changed, which path condition changes/symbolic value updates contribute to the different behaviors of the failure inducing input in two versions
- the diff can be manifested by the trace diffs, return the first branch of such valid tests

DARWIN: an example



```
int inp, outp;
scanf("%d", &inp);
if (inp >=1){
   outp = g(inp);
   if (inp>9){
      outp=g1(inp);
   }
} else{
  outp = h(inp);
}
printf("%d", outp);
```

Program P

```
int inp, outp;
scanf("%d", &inp);
if (inp >= 1){
   outp = g(inp);
   /* if (inp>9){
        outp=g1(inp);
   }  */
} else{
  outp = h(inp);
}
printf("%d", outp);
```

Program P'

Problem: When inp == 100, P' fails

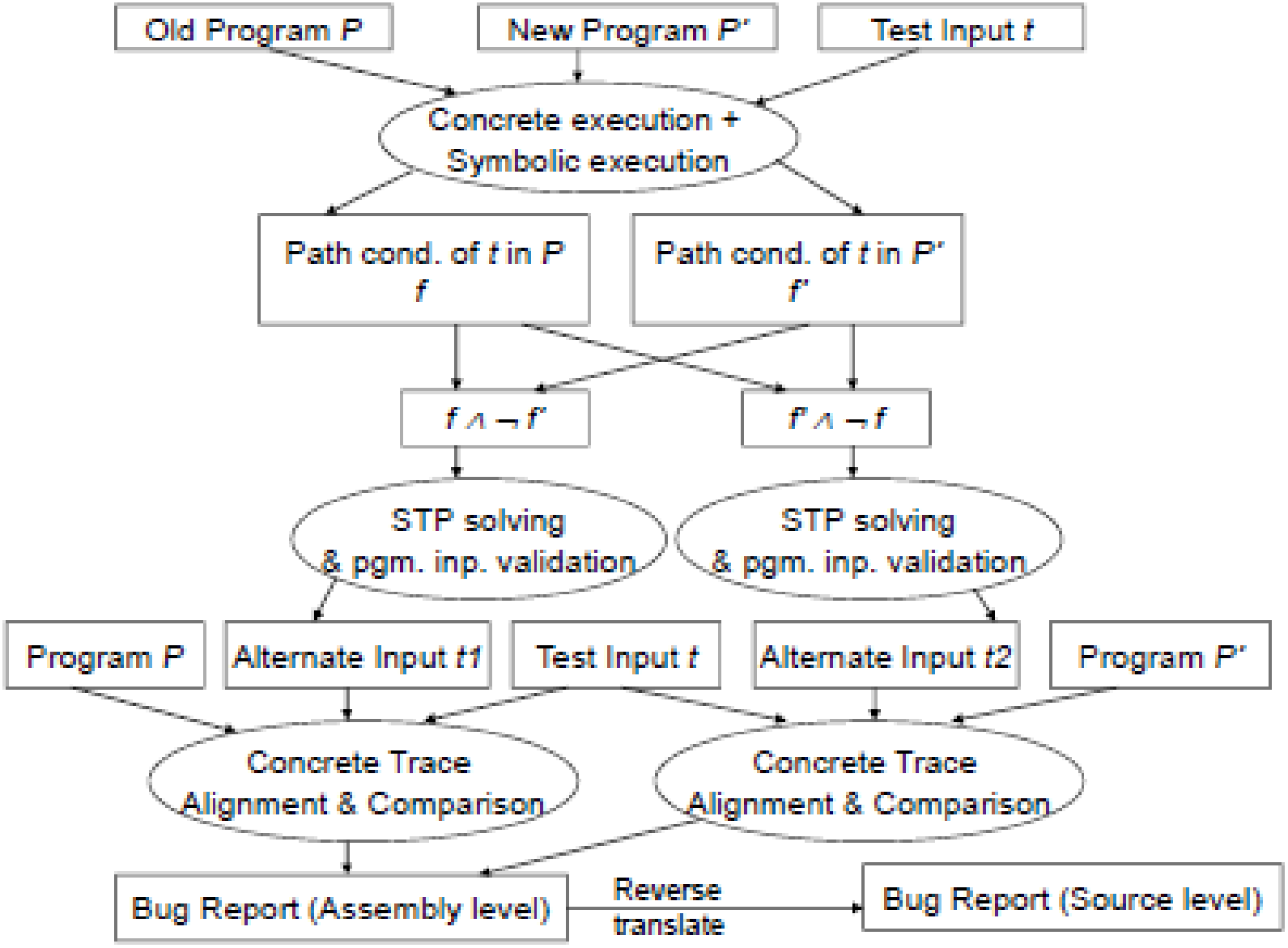
Solution: f $\wedge\neg$ f': inp $> 9 \wedge \neg$ (inp≥ 1) (no solution)

f' $\wedge\neg$ f: inp $\geq 1 \wedge \neg$ (inp> 9)

Run: inp == 100 and any input inpt [1, 9) on P, the diff lead to the branch inp>9 – missing this code in P' (the trace diff in P' can help understand the changes)

DARWIN: implementation

- BitBlaze: binary symbolic execution
- QEMU: concrete execution for both windows and linux


Old Program P
New Program P'
Test Input t
Concrete execution +
Symbolic execution
Path cond. of t in P
f
Path cond. of t in P'
f'
f ∧ ¬ f'
f' ∧ ¬ f
STP solving
& pgm. inp. validation
STP solving
& pgm. inp. validation
Program P
Alternate Input t1
Test Input t
Alternate Input t2
Program P'
Concrete Trace
Alignment & Comparison
Concrete Trace
Alignment & Comparison
Bug Report (Assembly level)
Reverse
translate
Bug Report (Source level)

DARWIN: results

```
if (!(png_ptr->mode & PNG_HAVE_PLTE))
{
   png_warning(png_ptr, "Missing PLTE before tRNS");
}
else if (length > (png_uint_32)png_ptr->num_palette)
{
   png_warning(png_ptr, "Incorrect tRNS chunk length");
   png_crc_finish(png_ptr, length);
   return;
}
```

Figure 7: Buggy code fragment from `libPNG`

DARWIN: results

- miniweb: get x returns index.html instead of error, compared to Apache
- savant: got /index.html, not report errors, compared to Apache

Thoughts and Discussions

Further Reading

- Questions programmers ask during software evolution tasks
- Chianti: A Tool for Change Impact Analysis of Java Programs
- Patch verification via multi-version control flow graphs
- History slicing: assisting code-evolution tasks
- Semantic Slicing of Software Version Histories (TSE)
- Buginnings: Identifying the Origins of a Bug
- DARWIN: An Approach for debugging evolving programs
- KATCH: High-Coverage Testing of Software Patches