Analyzing Software Changes and Versions

Wei Le

October 24, 2023

Motivation: Software Assurance in Continous Integration

- Agile development and continuous integration: small changes, fast delivery, good versions need to be generated frequently
- It is hard to get software changes correct (studies show that patches were buggy)
- Quality assurance techniques need to be flexible and provide fast feedback

Motivation: Example Problems

- Regression testing
  - test selection: which test cases should I run to test the new code?
  - test prioritization: which test to run first
  - test minimization: what is the smallest test set to I should run
- Patch verification
  - does this patch correctly fix the bugs?
  - does this change break the existing code?
  - how many versions this vulnerability affects? can the same patch fix all the vulnerable versions
- Multiple versions of software
  - does this static warning align with the previous version or it is a new warning for the new version of software?
  - can the two versions of programs merge correctly?
  - can the library update break the current applications?
  - in which version, this vulnerability is introduced?

Motivation: Example Problems

- Debugging: which line causes this regression bug?
- How to generate new test inputs for exercising changes?
- How to specify the correct changed behavior

Outline

- Patch verification via MVICFG
- Change impact analysis and regression testing
- Debugging changes
- Extend KLEE to generate tests for exercising changes (optional)
- Software history analysis

MVICFG for Patch Verification and Multiversion Analysis

See Wei Le's ICSE slides

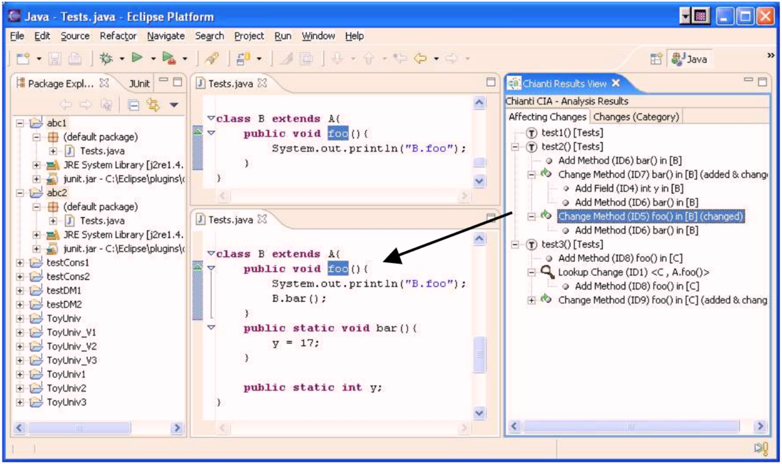
Change Impact Analysis

Change impact analysis: identifies the effects of a software change (2000-2015)

Regression testing: testing for changed software – select and prioritize test inputs that likely exercise the changes (ongoing problem)

Change Impact Analysis

- Change impact for C code: forward slicing
- Change impact for object-oriented code
  - 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
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

Chianti: A Tool for Change Impact Analysis of Java Programs

Which tests are affected by the change(s)?

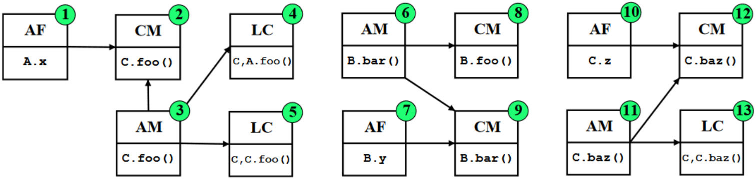
(**AC**)	Add an empty class
(**DC**)	Delete an empty class
(**AM**)	Add an empty method
(**DM**)	Delete an empty method
(**CM**)	Change body of method
(**LC**)	Change virtual method lookup
(**AF**)	Add a field
(**DF**)	Delete a field

Table 1: Categories of atomic changes.

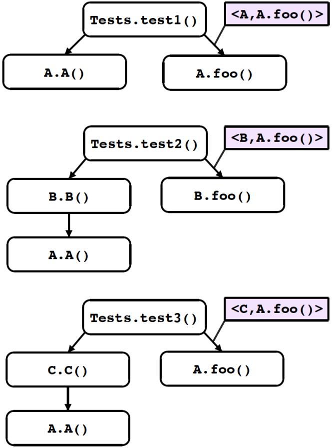
```
class A {
  public A(){ }
  public void foo(){ }
  public int x;
}
class B extends A {
  public B(){ }
  public void foo(){ B.bar(); }
  public static void bar(){ y = 17; }
  public static int y;
}
class C extends A {
  public C(){ }
  public void foo(){ x = 18; }
  public void baz(){ z = 19; }
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

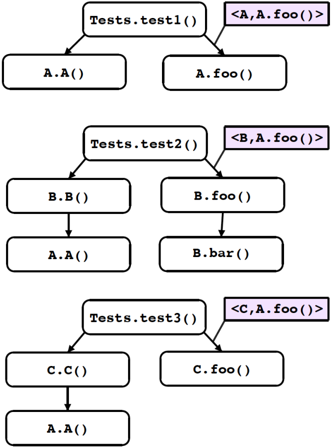
```
class Tests {
  public static void test1(){
    A a = new A();
    a.foo();
  }
  public static void test2(){
    A a = new B();
    a.foo();
  }
  public static void test3(){
    A a = new C();
    a.foo();
  }
}
```

(a)



(b)

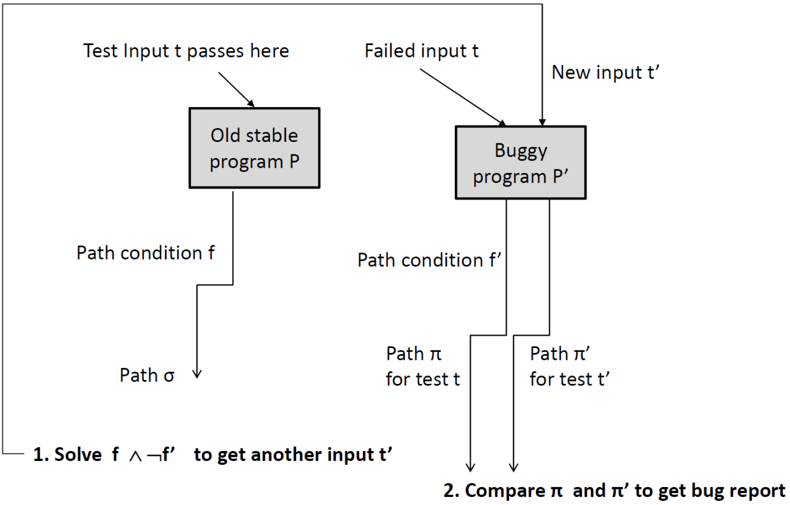

Tests.test1()
<A,A.foo()>
A.A()
A.foo()
Tests.test2()
<B,A.foo()>
B.B()
B.foo()
A.A()
Tests.test3()
<C,A.foo()>
C.C()
A.foo()
A.A()


Tests.test1()
<A,A.foo()>
A.A()
A.foo()
Tests.test2()
<B,A.foo()>
B.B()
B.foo()
A.A()
B.bar()
Tests.test3()
<C,A.foo()>
C.C()
C.foo()
A.A()

Debugging changes: DARWIN

- *Goal*: debugging - find causes of failures in changes
- *Problem*:
  - **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

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'

Analysis:

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

Some notes:

- 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: another example

```c
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

```c
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, what is the root cause?

DARWIN: another 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);
```

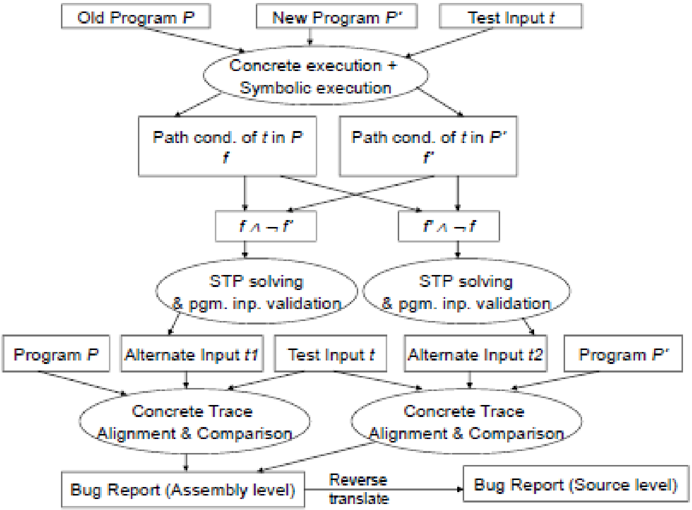
Program P'

f $\wedge\neg$ f': inp $> 9 \wedge \neg$ (inp$\geq$ 1) (no solution)
f' $\wedge\neg$ f: inp $\geq 1 \wedge \neg$ (inp$>$ 9)
Run: inp $==$ 100 and any input inp $==$ x, where $x \in [1, 9)$ on P, the difference between the traces leads to the branch inp$>$9, compared to P', we find that we miss this code in P'

DARWIN: implementation

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

DARWIN: implementation



DARWIN: case studies

1. source diff: 28 files and 1589 code churns
2. program slicing: the slice is too big, covering the entire client + libpng library code
3. statistical bug isolation methods: instrument predicates and correlate failed executions with predicate outcomes – which predicate to instrument? (predicate that has return values and scalar variables)
4. trace comparison: need to have good trace and bad trace, but they can be quite different
5. DARWIN: good trace and bad trace have min differences

DARWIN: case studies

```
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`

1. libpng: 1.0.7 (buggy) and 1.2.21 (fixed)
2. solve the constraints: $f_{fixed} \wedge \neg f_{buggy}$, $f_{buggy} \wedge \neg f_{fixed}$
3. generate 9 inputs (images), one of them is successful
4. compare the successful input and failure inducing input, they list the first branch in the bug report:
   length > (png_uint_32)png_ptr->num_palette

KATCH: Patch Testing (optional)

- Goal: automatically generate tests to exercise patches (GNU 6 years patches of diffutils, binutils, findutils)
- Approach: symbolic execution + heuristics (select test cases have shortest distances to patched code and perform symbolic execution from there)

KATCH: Patch Testing

Steps:

- patch pre-processing: each block consisting of a set of lines is a target, if the test suite already hits the target, we remove the target
- run existing tests, computing the distance of each test to the patches, select the closest one to start with
- three heuristics to reach the patches in symbolic execution: 1) greedy exploration, 2) informed path regeneration, 3) definition switch

KATCH: computing distance to target

- number of branch statements that need to flip between two basic blocks on the control flow graph (function calls are treated equally independent of their context)
- Is 50 better than 150 in this case?

	C−flow	WP
`if (input < 100)`	2	4
`  f(0);`	1	4
`if (input > 100)`	3	3
`  if (input > 200)`	2	2
`    f(input)`	1	1
`void f(int x) {`		
`  if (x == 999)`	1	1
`    // target`	0	0
`}`		

KATCH: reaching the patches

- execute the concrete input
- greedy exploration: at the branch where the unexplored side reaches the target, explore this side (the branch condition conjuncts of the current path conditions)
- informed path regeneration: if the side is not feasible, we traverse back to the branch that makes it infeasible and take the other side of branch
- definition switch: find definitions of relevant variables (push the original target on the stack)

KATCH: example

```
1   void log(char input) {
2     int file = open("access.log", O_WRONLY|O_APPEND);
3     if (input >= '␣' && input <= '~') {
4       write(file, &input, 1);
5     } else {
6       char escinput = escape(input);
7       write(file, &escinput, 1);
8     }
9     close(file);
10  }
```

Figure 4: Example based on lighttpd patch 2660 used to illustrate the greedy exploration step. Lines 3, 5–8 represent the patch.

```
if (0 == strcmp(requestVerb, "GET")) { ... }
...
for (char* p = requestVerb; *p; p++) {
  log(*p);
```

src/io.c

```
217 enum DIFF_wh_sp ig_white_space = ignore_white_space;

...

230 switch (ig_white_space)
231   {
232   case IGNORE_ALL_SPACE:
233     while ((c = *p++) != '\n')
234       if (! isspace (c))
235         h = HASH (h, ig_case ? tolower (c) : c);
236     break;
```

src/diff.c

```
291 while ((c = getopt_long (argc, argv,
                shortopts, longopts, NULL)) != -1)
292 {
293   switch (c)
294   {
...

319   case 'b':
320     if (ignore_white_space < IGNORE_SPACE_CHANGE)
321       ignore_white_space = IGNORE_SPACE_CHANGE;
322     break;
323
324   case 'Z':
325     if (ignore_white_space < IGNORE_SPACE_CHANGE)
326       ignore_white_space |= IGNORE_TRAILING_SPACE;
...

389   case 'E':
390     if (ignore_white_space < IGNORE_SPACE_CHANGE)
391       ignore_white_space |= IGNORE_TAB_EXPANSION;
392     break;
...

494   case 'w':
495     ignore_white_space = IGNORE_ALL_SPACE;
496     break;
```

Figure 6: Example from diffutils revision 8739d45f showcasing the need for definition switching. The patch is on line 235 and is guarded by a condition that is control dependent on the input.

KATCH: experimental setup

- klee + katch: 15 min timeout
- all the patches for findutils (125 patches, 2010/11-2013/1), diffutils (175 patches, 2009/11-2012/5) and binutils (181 patches, 2011/4, 2012/8)

KATCH: results

Table 1: Number of targets covered by the manual test suite, and the manual test suite plus KATCH.

Program Suite	Targets	Covered	
		Test	Test + KATCH
findutils	344	215 (63%)	300 (87%)
diffutils	166	58 (35%)	121 (73%)
binutils	852	150 (18%)	285 (33%)
Total	**1,362**	**423 (31%)**	**706 (52%)**

History slicing (2012): assisting code-evolution tasks

"Slicing" across software versions:

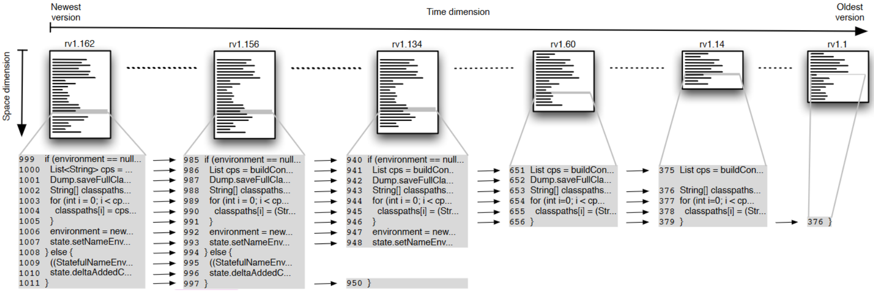
- assist developers' tasks for software maintenance
- questions about history: like when, how, by whom, and why some code was changed or inserted.
- visualization of the entire evolution for the code of interest, efficient inspection of a sequence of changes for an arbitrary block of code.
- *history 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 (2012): assisting code-evolution tasks

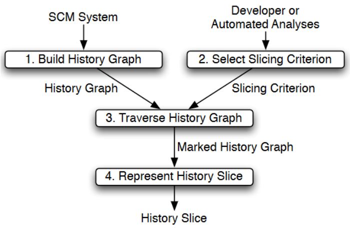
Application Examples:

- 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



History slicing: assisting code-evolution tasks



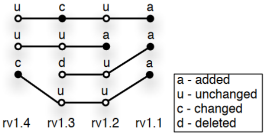


Figure 3: History Graph.

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.

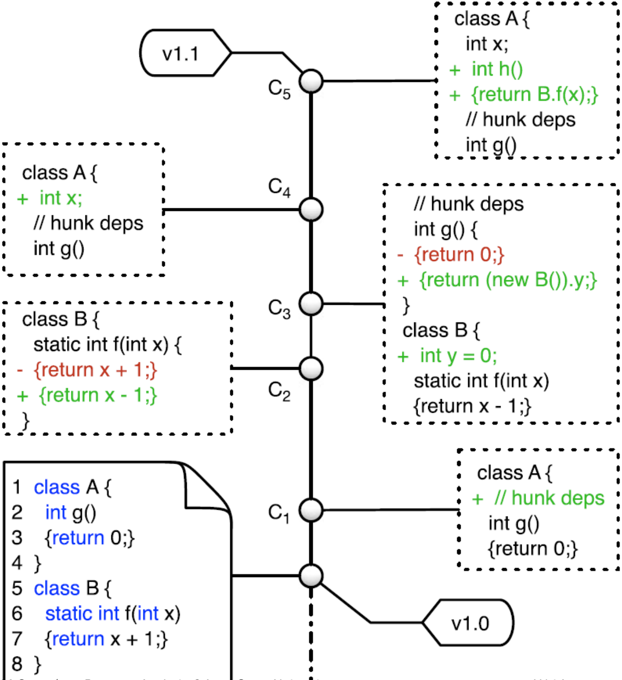
Semantic Slicing of Software Version Histories (2017)

- 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.

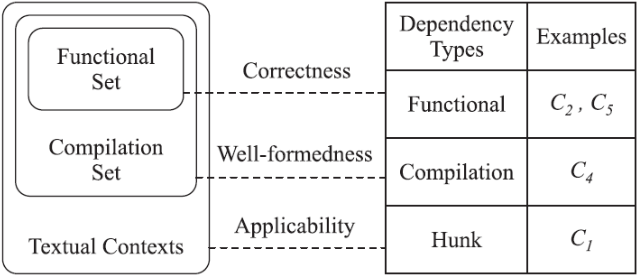
Semantic Slicing of Software Version Histories (2017)

Motivation:

- 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()
// hunk deps
int g() {
- {return 0;}
+ {return (new B()).y;}
}
class B {
+ int y = 0;
static int f(int x)
{return x - 1;}
C3
class B {
static int f(int x) {
- {return x + 1;}
+ {return x - 1;}
}
C2
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



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

- 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)

Semantic Slicing of Software Version Histories

10x to 100 x faster than delta debugging

F	Reduction(%)			Time(s) for Long	
	Short(H)	Medium(H)	Long(H)	Slice	Hunk
C1	94(62)	89(34)	91(28)	<1	1.1
C2	36(14)	39(10)	42(12)	<1	34.8
C3	82(20)	72(13)	71(14)	<1	129.7
E1	72(72)	79(78)	81(79)	2677.5	11.6
E2	94(90)	96(92)	95(91)	2086.4	12.8
E3	94(94)	95(94)	96(94)	2041.0	12.4
H1	52(24)	61(25)	53(21)	852.0	60.5
H2	50(44)	56(50)	67(59)	766.1	53.2
H3	88(84)	87(75)	90(71)	734.4	23.3
M1	94(90)	97(64)	97(59)	11.1	38.3
M2	96(96)	97(80)	98(79)	8.4	11.5
M3	94(94)	93(89)	95(92)	7.1	1.0
Avg.	79(65)	80(59)	81(58)	765.6	32.5

Buginnings (2010): Identifying the Origins of a Bug (optional)

Problem:

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

Buginnings: text diff does not work

```
public void f( int x ) {
    int y = 10;
    y = y + x;
  }
```

version 1

```
public void f( int x ) {
    int y = 10;
    y = y + x;
    print y; // added
  }
```

version 2

```
public void f( int x ) {
    int y = 10;
    int z = y + x; // modified
    print y;
  }
```

version 3 (bug introducing)

```
public void f( int x ) {
    int y = 10;
    int z = y + x;
    print z; // modified
  }
```

version 4 (bug fix)

patch:

- print y

+ print z

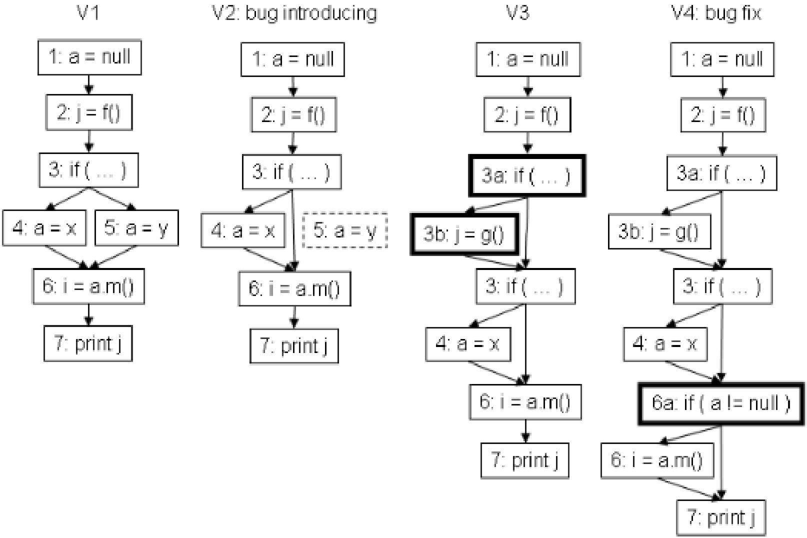
using text approach to trace the bug origin: version 2 – is this correct?

Buginnings: solution

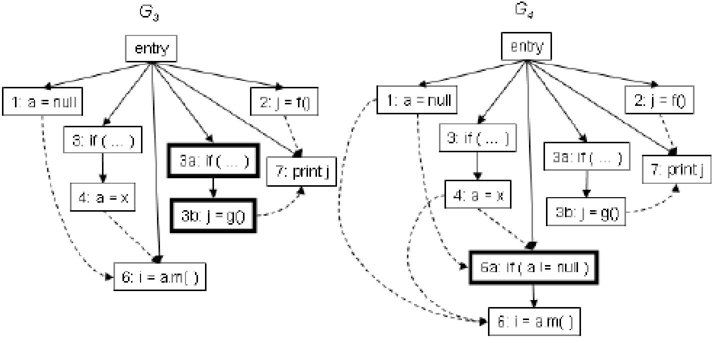
Solution:

- 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

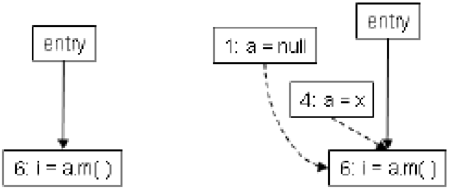
Buginnings: an example on how to compute bug origin



Buginnings: construct dependency graphs



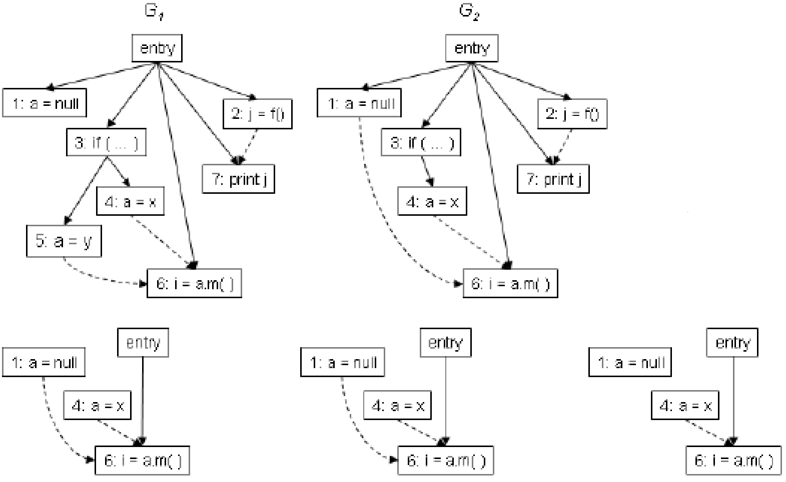
Buginnings: compute bug region



(a) $G_3 - G_4$ (b) Bug region (B)

computing bug region: perform diff for dependency graphs between fixed version and its previous version

Buginnings: projection of bug region to the version history



(b) Bug region (B) (c) Projection of G_2 wrt B (d) Projection of G_1 wrt B

Buginnings: algorithm

algorithm ComputeBugVersion
input versions $\langle V_1, \ldots, V_n \rangle$ of prog $\mathcal{P}$; V_n is bug-fix version
output $V_i (1 \leq i \leq n-1)$, the bug-introducing version
begin
$\mathcal{G}_n$ = SDG for version V_n
$\mathcal{G}_{n-1}$ = SDG for version V_{n-1}
$\mathcal{G}^{diff} = \mathcal{G}_{n-1} - \mathcal{G}_n$
if $\mathcal{G}^{diff} \neq \emptyset$ **then**
    $\mathcal{B}$ = 1-step backward slice in $\mathcal{G}_{n-1}$ from nodes in $\mathcal{G}^{diff}$
else $\mathcal{G}^{diff} = \mathcal{G}_n - \mathcal{G}_{n-1}$
    if $\mathcal{G}^{diff} \neq \emptyset$ **then**
        $\mathcal{B}$ = projection of $\mathcal{G}_{n-1}$ with respect to $\mathcal{G}_{diff}$
    else $\mathcal{B}$ is the set modified statements
foreach i in $n-2$ to 1 **do**
    $\mathcal{G}_i^{proj}$ = projection of $\mathcal{G}_i$ with respect to $\mathcal{B}$
    if $\mathcal{G}_i^{proj} \subset \mathcal{B}$ **then return** V_{i+1}
return V_1
end

Buginnings: Identifying the Origins of a Bug (2010)

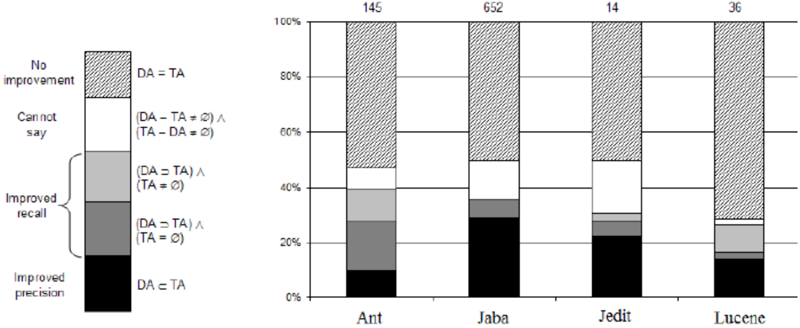
Evaluation:

- 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 compared to text based approaches



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

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