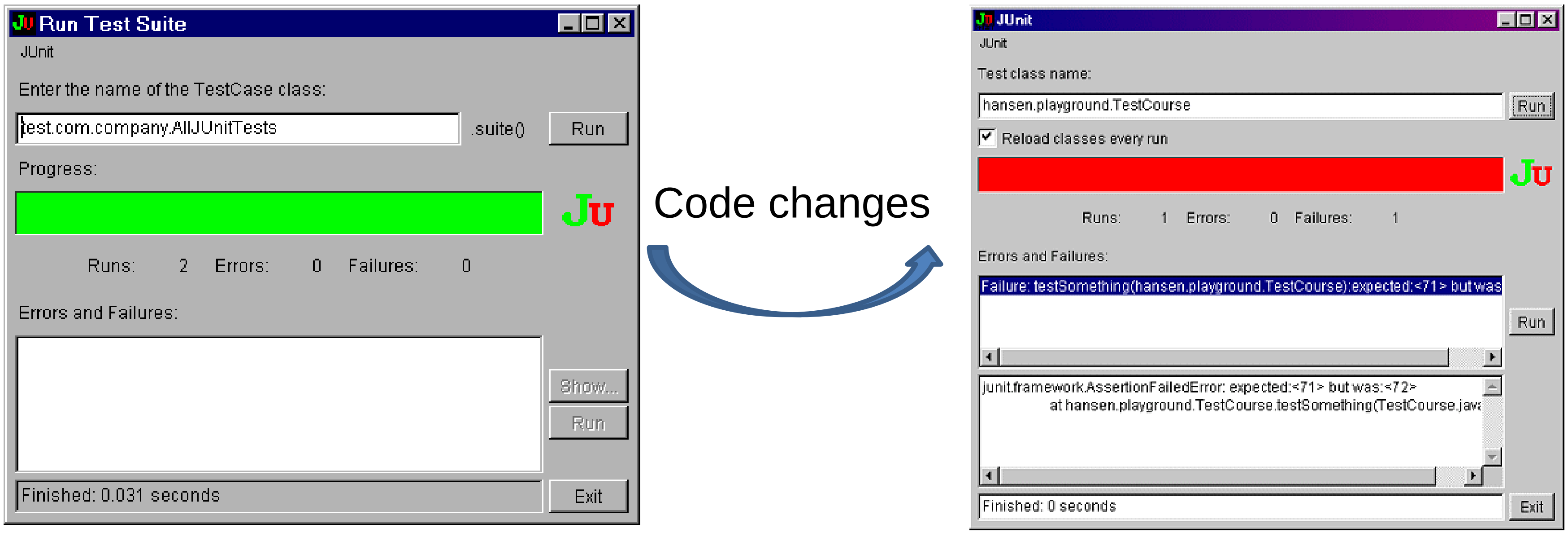
Effective Identification of Failure-Inducing Changes: A Hybrid Approach

Sai Zhang, Yu Lin, Zhongxian Gu, Jianjun Zhao

PASTE 2008

My program fails, why?



- **Which part of code change is responsible for the regression test failure?**
  - Examine each code edits manually might be tedious and laborious
  - Failures may result from a combination of several changes

Identify failure-inducing changes

- **Delta debugging [**Zeller ESEC/FSE'99**]**
  - **A promising approach to isolate faulty changes**
  - **It constructs intermediate program versions repeatedly to narrow down the change set**

- **Can we develop more effective techniques?**
  - Integrate the strength of both *static* analyses and *dynamic* testing, to fast narrow down the change set
  - Goal: A complementary general approach to original debugging algorithm (not restricted to one specific programming language)

Outline

- **Background**
  - Delta debugging
  - Improvement room
- **Our hybrid approach**
  - Prune out irrelevant changes
  - Rank suspicious change
  - Construct valid intermediate version
  - Explore changes hierarchically
- **Experiment evaluation**
- **Related work**
- **Conclusion**

Outline

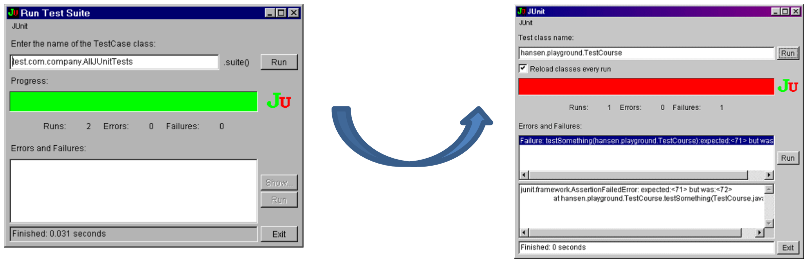
- **Background**
  - Delta debugging
  - Improvement room
- **Our hybrid approach**
  - Prune out irrelevant changes
  - Rank suspicious change
  - Construct valid intermediate version
  - Explore changes hierarchically
- **Experiment evaluation**
- **Related work**
- **Conclusion**

Background

- **Delta debugging**
  - Originally proposed by Zeller in ESEC/FSE'99
  - Aim to isolate failure-inducing changes and simplify failed test input

- **Basic idea**
  - Divide source changes into a set of configurations
  - Apply each subset of configurations to the original program
  - Correlate the testing result to find out the minimum faulty change set

Delta debugging: an example



Suppose there are eight changes: C_1, C_2, C_3, …. C_8

; and C_7 is the only failure-inducing change

Delta debugging works as follows:

Step	Configurations	Result
1	C_1, C_2, C_3, C_4	PASS
2	C_5, C_6, C_7, C_8	FAIL
3	C_5, C_6	PASS
4	C_7, C_8	FAIL
5	C_8	PASS
6	C_7	FAIL
Result	C_7 is the only faulty change!	FOUND!!!

A more complex example

Suppose there are eight changes: C_1, C_2, C_3, …. C_8

; and a combination C_3 and C_6 changes is the failure cause

Step	Configurations	Result
1	C_1, C_2, C_3, C_4	PASS
2	C_5, C_6, C_7, C_8	PASS
3	C_1, C_2, C_5, C_6, C_7, C_8	PASS
4	C_3, C_4, C_5, C_6, C_7, C_8	FAIL
5	C_3, C_5, C_6, C_7, C_8	FAIL
6	C_1, C_2, C_3, C_4, C_7, C_8	PASS
7	C_1, C_2, C_3, C_4, C_5, C_6	FAIL
8	C_1, C_2, C_3, C_4, C_5,	PASS
Result	C_3 and C_6 are the faulty changes	**FOUND!!!**

⟸ **C_3 is found!**

⟸ **C_6 is found!**

Original Delta debugging can also handle configuration inconsistent problem.

Can we make it faster?

- **Key insights:**
  - Searching space
    - Delta debugging (DD) searches the **whole configuration set**.
    - *Is it necessary?*
  - Configuration selection
    - DD selects configurations in **an arbitrary order**.
    - *Can we improve the selection strategy?*
  - Intermediate version construction
    - DD constructs intermediate program version by **syntax difference**, which might result in inconsistence.
    - *Can we introduce semantic dependence information?*
  - Configuration exploration strategy
    - DD treats all changes as **a flat list**.
    - *Can we explore changes hierarchically, and prune out irrelevant ones earlier?*

Outline

- Background
  - Delta debugging
  - Improvement room
- **Our hybrid approach**
  - Prune out irrelevant changes
  - Rank suspicious change
  - Construct valid intermediate version
  - Explore changes hierarchically
- Experiment evaluation
- Related work
- Conclusion

Our hybrid approach: an overview

- **Reduce searching space**
  - Use static change impact analysis
  - Then, focus on the relevant (suspicious) changes
- **Rank suspicious changes**
  - Utilize dynamic testing result of both passing and failing tests
  - Apply changes with higher likelihood first
- **Construct valid intermediate version**
  - Use atomic change representation
  - Guarantee the intermediate version constructed is compliable.
- **Explore changes hierarchically**
  - From method-level to statement-level
  - Prune a large number of changes earlier

Step 1: reduce searching space

- **Generally, when regression test fails, only a portion of changes are responsible**

- **Approach**
  - **We divide code edits into a consistent set of *atomic change* representations [**Ren et al' OOPSLA 04, Zhang et al ICSM'08**].**
  - **Then we construct the *static call graph* for the failed test**
  - **Isolate a subset of responsible changes based on the atomic change and static call graph information**
    - **A safe approximation**

Example

```
class  A {
 int num = 10;
 public int  getNum() {
   return num;
  }
}
```

Figure 1, original program

```
class  A{
 int num = 10;
 int tax = 5;
 public int  getNum() {
    if(tax > 5)
      tax = 5;
    num = num  + tax;
    return num;
  }
 public void setNum(int num) {
    this.num = num;
 }
}
```

Figure 2, program after editing

```
pubic void testGetNum(){
  A a = new A();
  assertTrue(
      a.getNum() == 10);
}
```

Figure 3, a Junit test

Example (cont)

```java
class  A {
 int num = 10;
 public int  getNum() {
   return num;
  }
}
```

Figure 1, original program

```java
class  A{
 int num = 10;
 int tax = 5;
 public int  getNum() {
   if(tax > 5)
     tax = 5;
   num = num  + tax;
   return num;
 }
 public void setNum(int num) {
   this.num = num;
 }
}
```

Figure 2, program after editing

AC	Add an empty class
DC	Delete an empty class
AM	Add an empty method
DM	Delete an empty method
CM	Change body of a method
LC	Change virtual method lookup
AF	Add a field
DF	Delete a field
CFI	Change definition of a instance field initializer
CSFI	Change definition of a static field initializer
AI	Add an empty instance initializer
DI	Delete an empty instance initializer
CI	Change definition of an instance initializer
ASI	Add an empty static initializer
DSI	Delete an empty static initializer
CSI	Change definition of an static initializer

Table 1, A catalog of atomic changes for Java (from Ren et al OOPSLA'04 paper)

Generate atomic changes

AF (`tax`), FI(`tax`), CM(`getNum()`), AM(`setNum(int)`), CM(`setNum(int)`)

Add dependence relations

FI (`tax`) $\preceq$ AF(`tax`), CM(`getNum()`) $\preceq$ AF(`tax`)

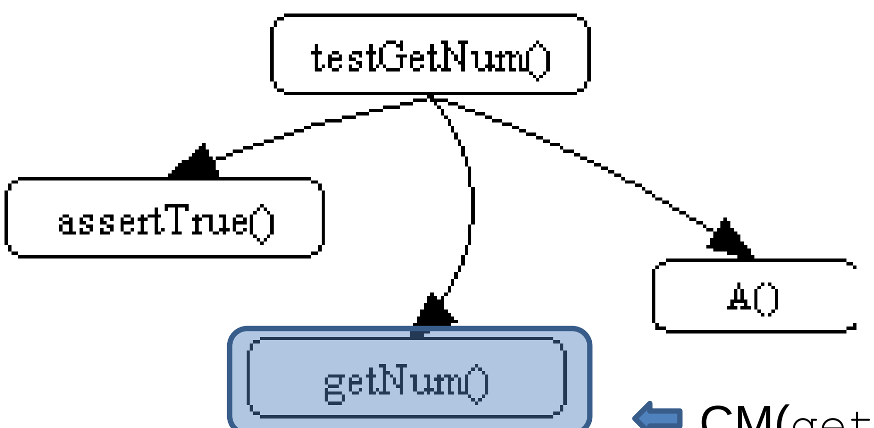
CM(`setNum(int)`) $\preceq$ AM(`setNum(int)`)

Example (cont)

```
pubic void testGetNum(){
  A a = new A();
  assertTrue(
     a.getNum() == 10);
}
```

FI (`tax`) $\preceq$ AF(`tax`),
CM(`getNum()`) $\preceq$ AF(`tax`)
CM(`setNum(int)`) $\preceq$ AM(`setNum(int)`)

Construct static call graph,
and identify responsible changes



CM(`getNum()`)

Call graph of the failed test

The responsible change set is:
① Changes appearing on the call graph either as a node or an edge
② All dependent changes of changes in ①

All responsible changes:
CM(`getNum()`)
AF(`tax`), FI(`tax`)

Step 2: rank suspicious changes

- **Ideally speaking, changes which are most likely to contribute to the failure should be ranked highest and tried first.**
- **The heuristic we used for ranking is similar to the Tarantula approach [**Jones et al ICSE'02**]**
- **We compute a value for each atomic change** c

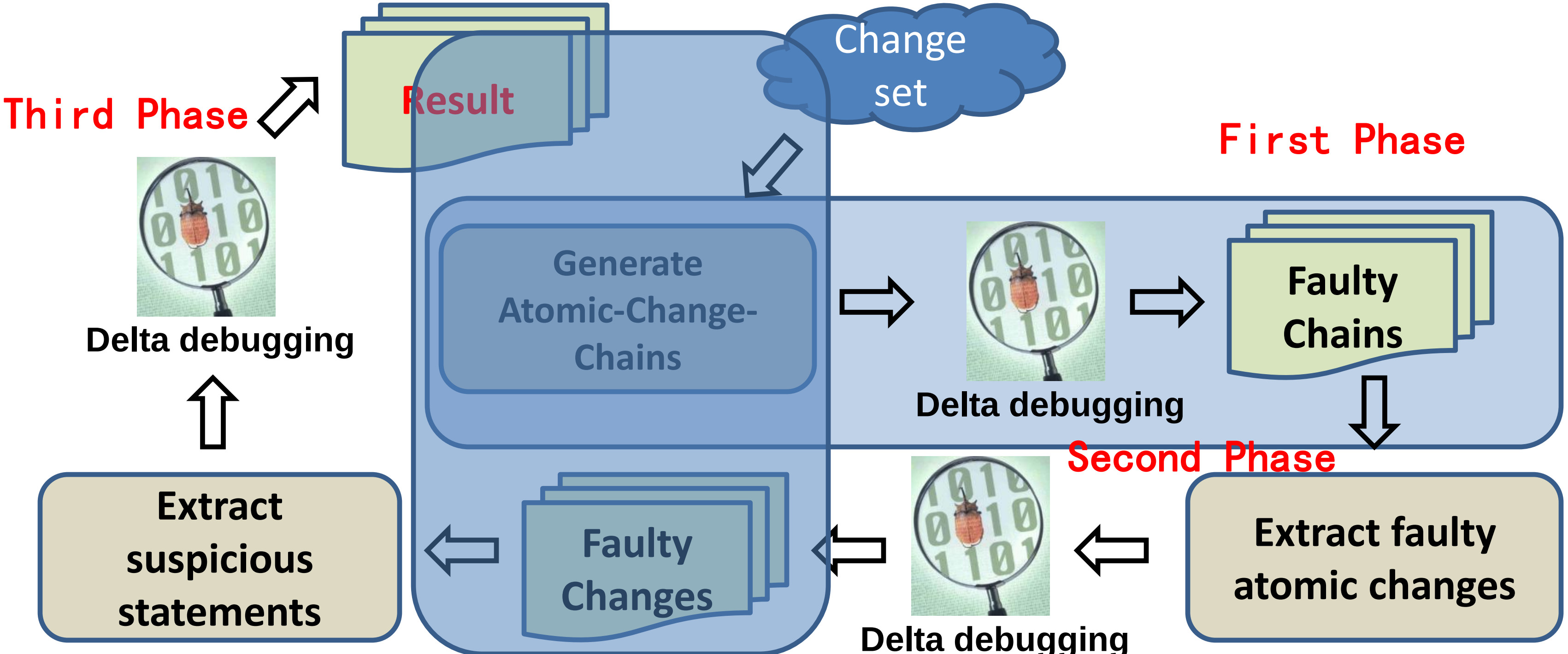
$$score(c) = \frac{\%failed(c)}{\%passed(c) + \%failed(c)}$$

`%failed(c)` returns, as percentage, the ratio of the number of failed tests that cover c as a responsible change to the total failed test number.

Step 3: explore faulty changes

- **The core module of our approach, an improved *Three-Phase* delta debugging algorithm**
  - Focus on the responsible change set
  - Increase the change granularity from coarse method level to fine statement level in three steps

Three-Phase delta debugging working flow:



Back to the Example

```
class  A {
 int num = 10;
 public int  getNum() {
   return num;
  }
}
```

```
class  A{
 int num = 10;
 int tax = 5;
 public int  getNum() {
   if(tax > 5)
     tax = 5;
   num = num  + tax;
   return num;
  }
 public void setNum(int num) {
   this.num = num;
  }
}
```

Static change impact analysis

All responsible changes:

CM(`getNum()`)
AF(`tax`), FI(`tax`)

Three-Phase **delta debugging: Phase 1**

Atomic-change-chains:

Only 1 chain, containing changes:
CM(`getNum()`), AF(`tax`), FI(`tax`)

Phase 2

Suspicious atomic changes:
CM(`getNum()`), FI(`tax`)

Delta debugging

Faulty change
CM(`getNum()`)

Phase 3

We prune out all ***def-change*** here.

An **atomic-change-chain** starts from an atomic change without any children, and includes all transitively dependent changes

Extract changed statements:

1. `if(tax > 5) tax = 5;`
2. `num = num + tax;`

Delta debugging

Final output

Faulty statement:
`num = num + tax;`

Other technical issue

- **The correctness of intermediate program version**
  - The dependence between atomic changes guarantee the correctness of intermediate version in phase 1 and 2 [Ren et al OOPSLA'04]
  - However, in phase 3, the configurations could be inconsistent as the original delta debugging

Outline

- Background
  - Delta debugging
  - Improvement room
- Our hybrid approach
  - Prune out irrelevant changes
  - Rank suspicious change
  - Construct valid intermediate version
  - Explore changes hierarchically
- **Experiment evaluation**
- Related work
- Conclusion

Prototype Implementation

- **We implement our prototype called AutoFlow for both Java and AspectJ programs**
  - Build on top of our Celadon [ICSM 08, ICSE'08 demo, ISSTA'08, student poster] framework
  - Modify Java/AspectJ compiler source code

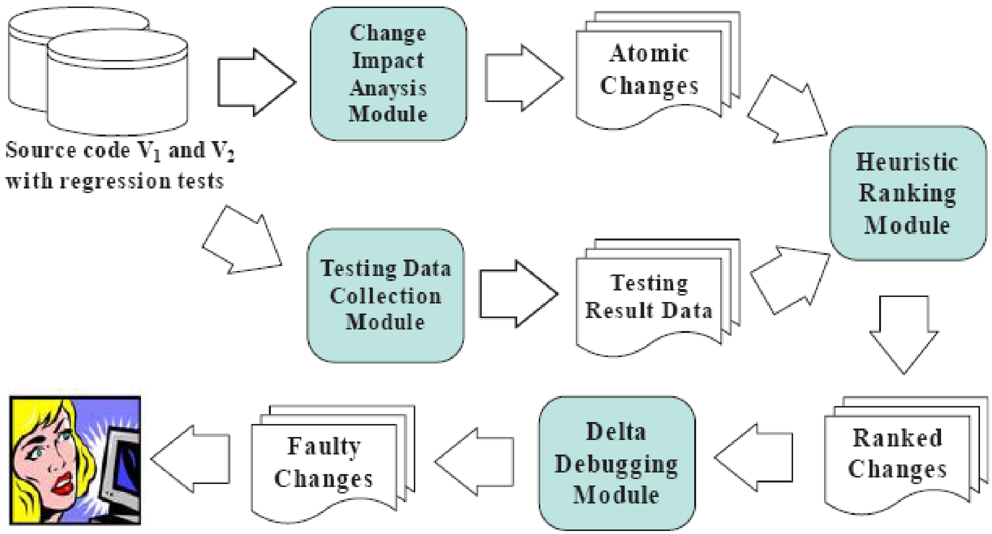


Figure 4, tool architecture

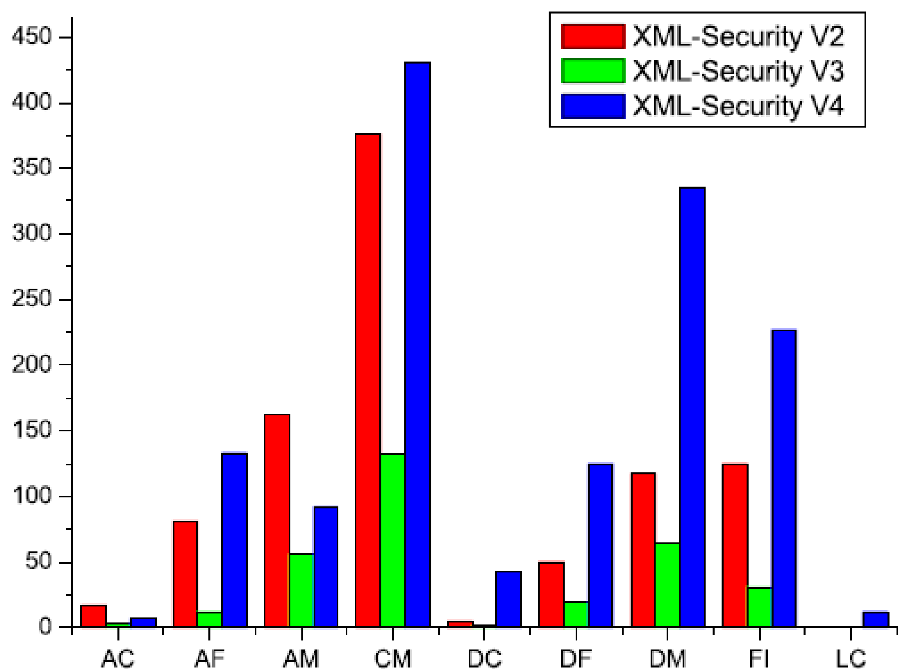
Subject programs

- **Two medium-sized Java/AspectJ programs, from UNL SIR and AspectJ distribution package**

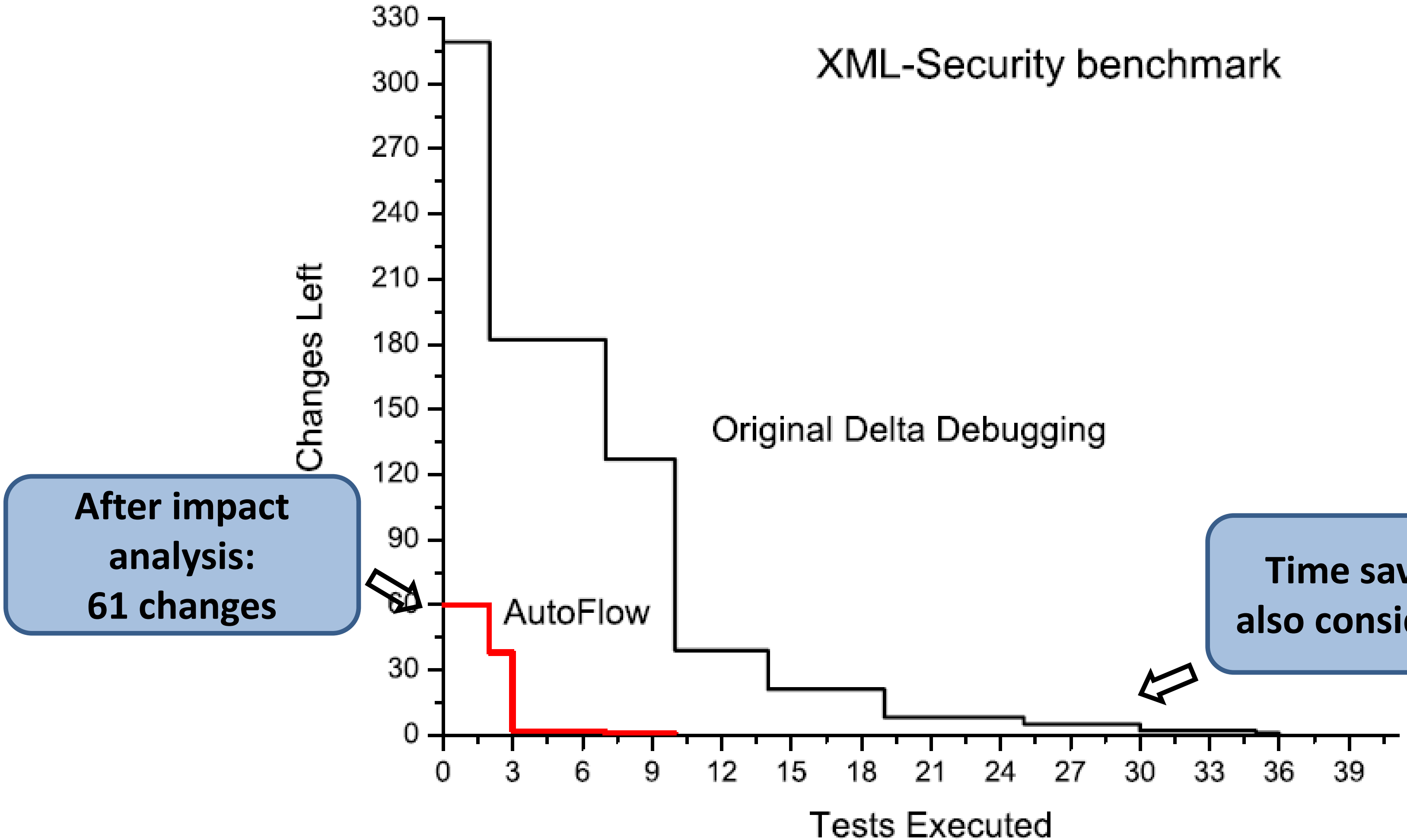
Subject	Type	LOC	#Ver	#Me	#Tests
XML-Security	Java	16800	4	1221	112
Dcm	AspectJ	3423	2	2	157

Case Study , XML-Security

- **We found one test** `testSecOctetStreamGetNodeSet1()` **passes in its 2nd version, but fails in its 3rd version**
- **Changes between 2nd and 3rd version (total 312 atomic changes)**



Exploring changes by AutoFlow



Only needs **10 tests** by AutoFlow vs **40 tests** by Original delta debugging

Outline

- Background
  - Delta debugging
  - Improvement room
- Our hybrid approach
  - Prune out irrelevant changes
  - Rank suspicious change
  - Construct valid intermediate version
  - Explore changes hierarchically
- Experiment evaluation
- **Related work**
- Conclusion

Related Work

- **Delta debugging and its applications**
  - Zeller ESEC/FSE’99, FSE’02, ICSE’04, ISSTA’05
  - Misherghi ICSE’06
- **Change impact analysis and its applications**
  - Ryder et al PASTE’01, Ren et al OOPSLA’04
  - Ren et al TSE’06, Chesley et al ICSM’05, Max FSE’06
- **Fault localization techniques (closely related)**
  - Jones ICSE’02, Ren et al ISSTA’07, Jeffery et al ISSTA’08

Outline

- Background
  - Delta debugging
  - Improvement room
- Our hybrid approach
  - Prune out irrelevant changes
  - Rank suspicious change
  - Construct valid intermediate version
  - Explore changes hierarchically
- Experiment evaluation
- Related work
- **Conclusion**

Conclusion

- **We present a hybrid approach to effectively identify failure-inducing changes (requires 4X less tests)**
- **Implement the tool and present two case studies**
- **We recommend our approach to be an integrated part of the delta debugging technique; when a regression test fails:**
  - Remove unrelated changes first
  - Rank suspicious change, and
  - Explore code edits from coarse-grained to fine-grained level

Future Directions

- **Eliminate searching space**
  - Using more precise impact analysis approaches, such as dynamic slicing, Execution-After information
- **Perform more experiment evaluations**
- **Investigate the correlations between change impact analysis and heuristic ranking**

- **Long term plan**
  - **Explore how to incorporate static/statistical analysis techniques into debugging tasks**
  - **Combine testing and verification for effective/scalable fault localization**