Put on Your Tester Hat: Improving programs for Automated Program Generation

Shin Hwei Tan

Concordia University

From the Past to Current

Dagstuhl Seminar 18052

Genetic Improvement of Software

(Jan 28 – Feb 02, 2018)

Dagstuhl Seminar 24431

Automated Programming and Program Repair

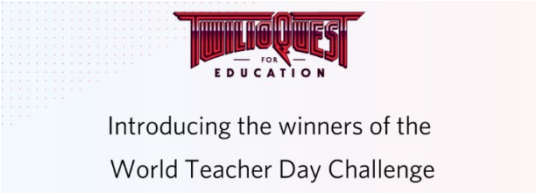
(Oct 20 – Oct 25, 2024)

Testing Program Analyzers and Verifiers: June 9, 2025

The Past: My Prior Work on APR

Applications of Program Repair

- **Education:**
  - Feedback Generation for Programming Assignments [FSE17,ISSTA23]
  - GitHub-OSS Fixit [ICSE21]
    - Taught students to fix bugs in SE class
    - Our lesson plan won World Teacher Day Challenge



- **Others:**
  - First Repair System for Android Apps [ICSE18]
  - Test Repair [ICSE21 Tool, Huawei Grant]
  - GPU programs [ASE19]
  - CrossFix: *Resolution of GitHub issues* via Similar Bugs Recommendation [JSME12]

Benchmarks for APR

- *Codeflaws [ICSE17 Poster]* CODEFORCES β Sponsored by Telegram
  - Programming Competition
  - Diverse types of defects
- *Droixbench [ICSE18]*
  - Reproducible crashes in Android apps
- *LLMDefects [ICSE23]*
  - Defects in auto-generated programs by Codex

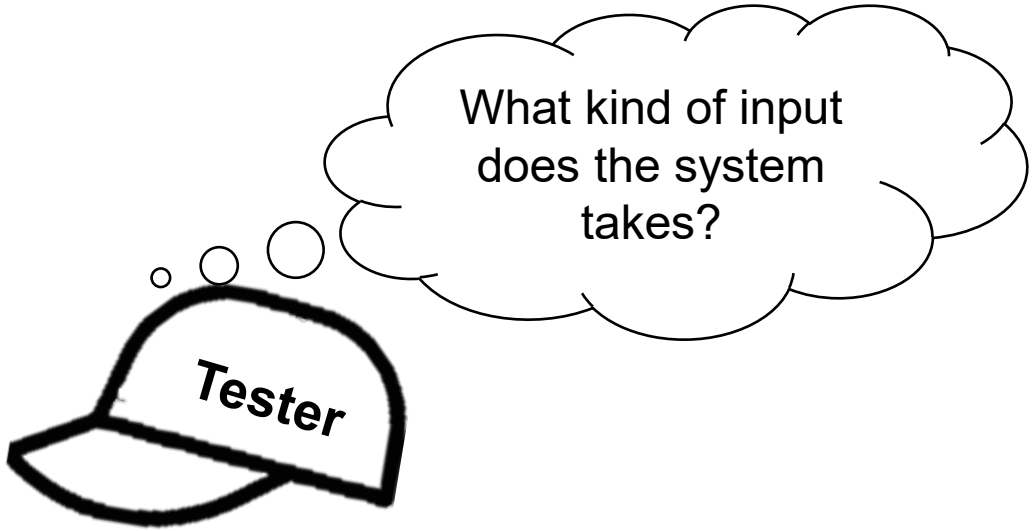
Workshops for APR

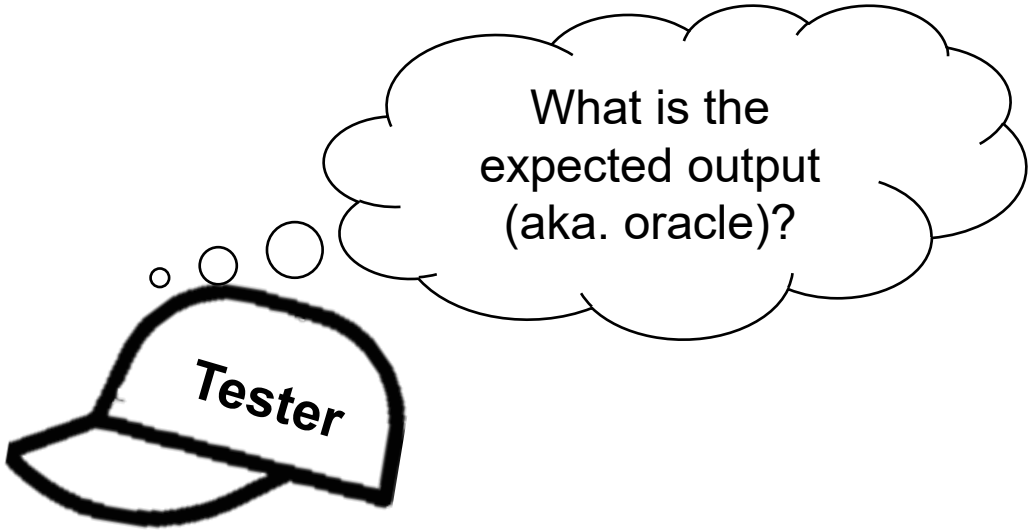
- Co-organized Genetic Improvement Workshop (GI @ ICSE 2019)
- Founded and co-organized 5 editions of International Workshop on Automated Program Repair *(APR20, APR21, APR22, APR23, APR24)*

Automated Program Generation

Automated Program Generator

Input →  → Output





Automated Program Generation

Instruction in natural language

```
Write a program ...
```

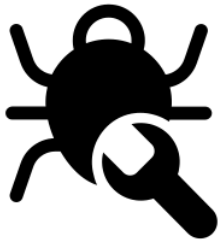
Input Program

```
public String
makeFancyString(String s)
{
  …
}
```

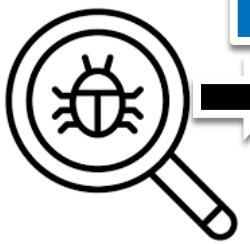
Automated Program Generator

```
public String
makeFancyString(String s) {
 StringBuilder sb = …
  for (int i = 2; i < …
      }
    }
}
```

Fix bugs

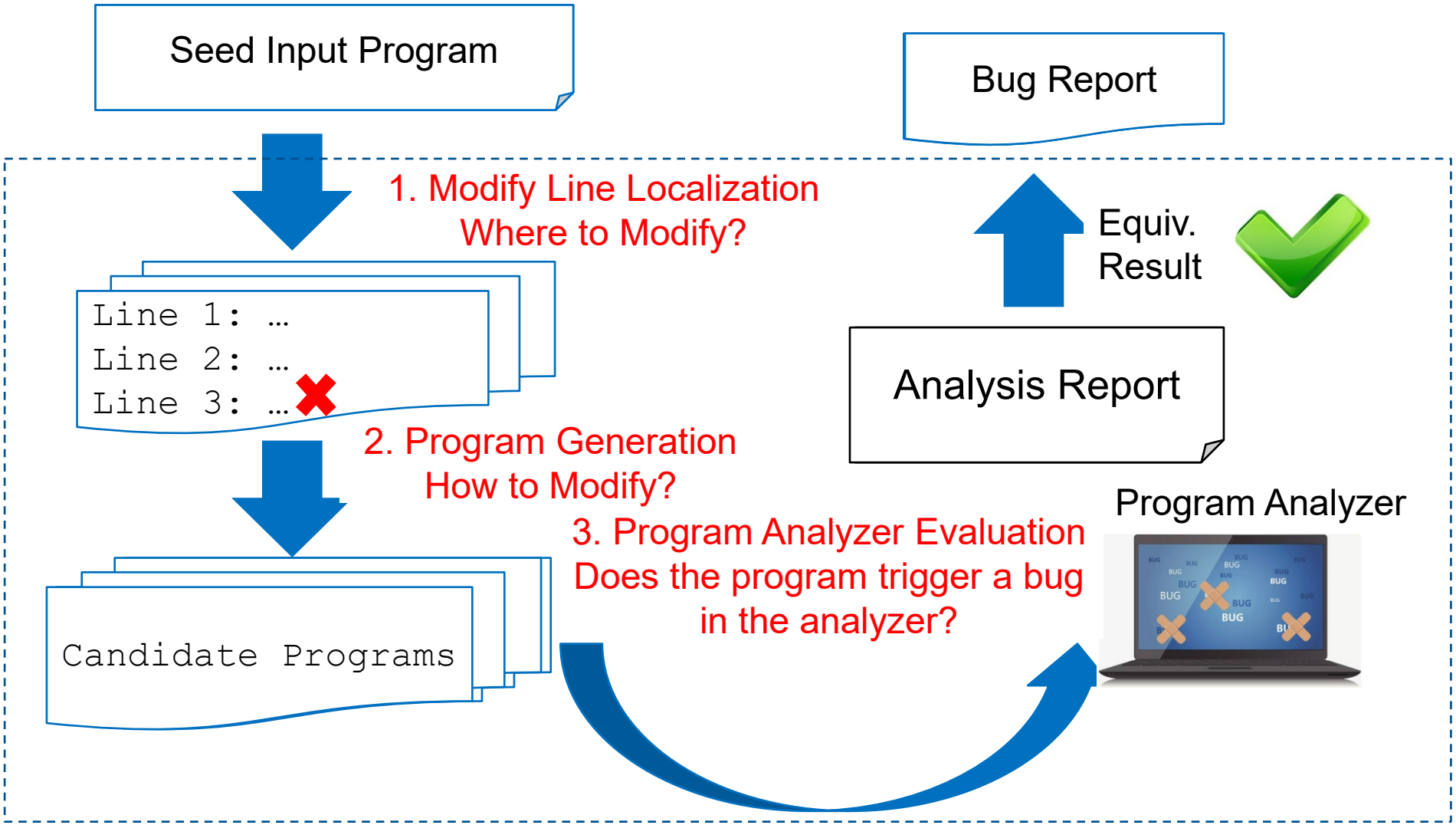


Find bugs



Program Generator

GI versus Testing Program Analyzers



Automated Test Generation for Program Analyzer

Statfier: Automated Testing of Static Analyzers via Semantics-Preserving Program Transformations (FSE'23)

Understanding & Detecting Annotation-Induced Faults of Static Analyzers (FSE'24)

Characterizing & Detecting Program Representation Faults of Static Analysis Frameworks (ISSTA'24)

Program Analyzer

Statfier: Automated Testing of Static Analyzers via Semantics-Preserving Program Transformations

Huaien Zhang, Yu Pei, Junjie Chen, Shin Hwei Tan

THE HONG KONG POLYTECHNIC UNIVERSITY 香港理工大學

SUSTech Southern University of Science and Technology

TIANJIN UNIVERSITY (PEIYANG UNIVERSITY) 1895 天津大学

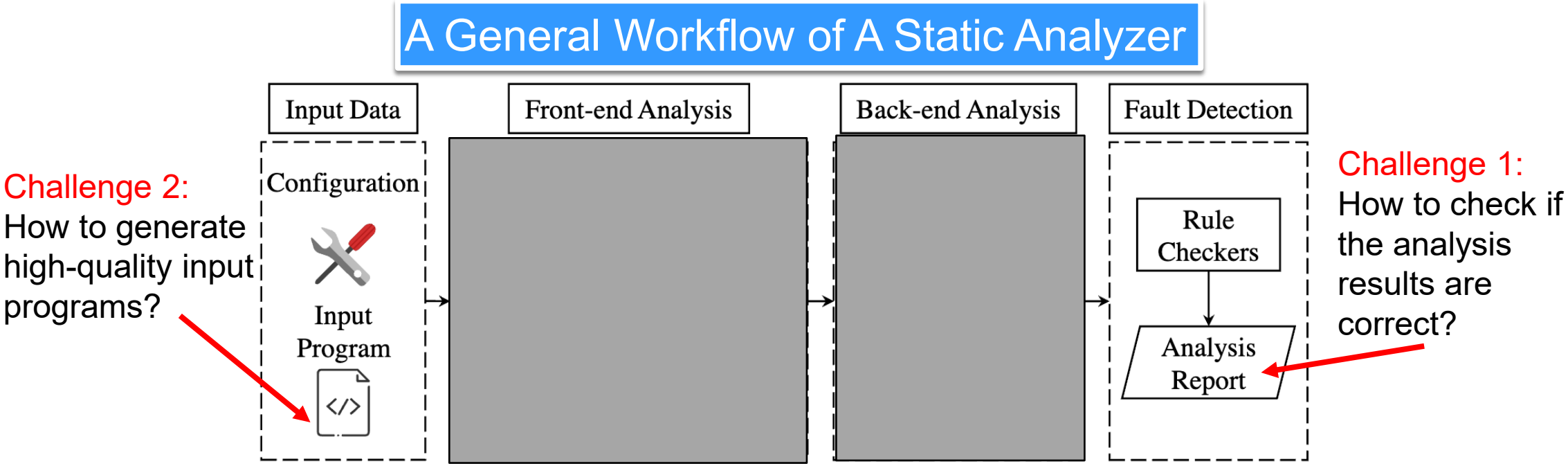
UNIVERSITÉ Concordia UNIVERSITY

*Accepted and Presented in FSE'23

Background: Static Analyzer

- Widely used to detect common issues without running programs.
- Inaccurate or incomplete analysis reports due to unrevealed bugs

  ➢ Improving reliability of static analyzers is important

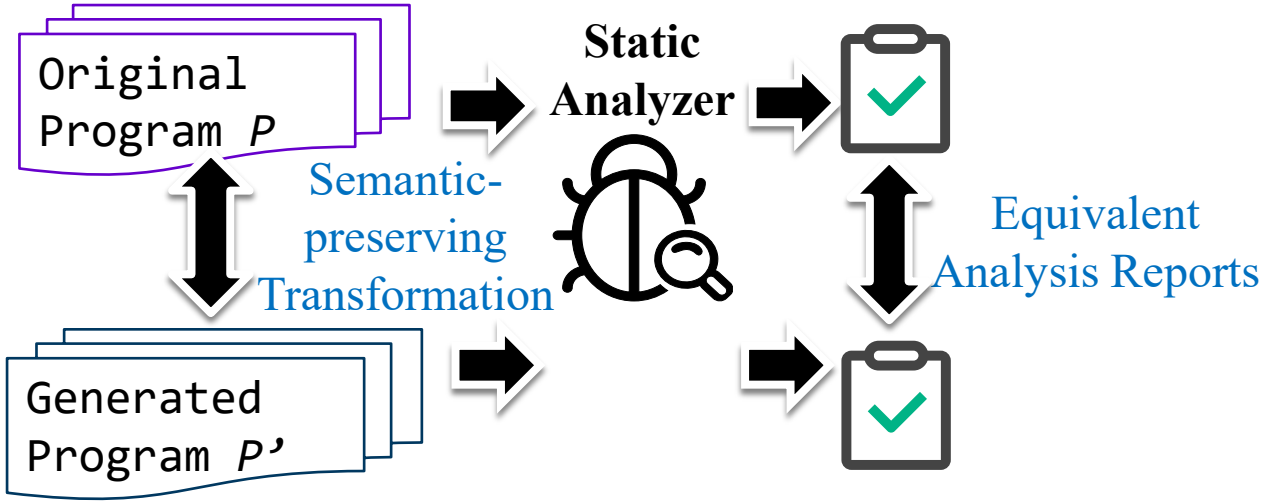


Challenges of Testing Static Analyzer & Our Solution

Challenge 1: Lack of automated test oracle

Metamorphic testing

- **Metamorphic relation:** Original program P and generated program P' from semantics-preserving transformations should have equivalent analysis reports

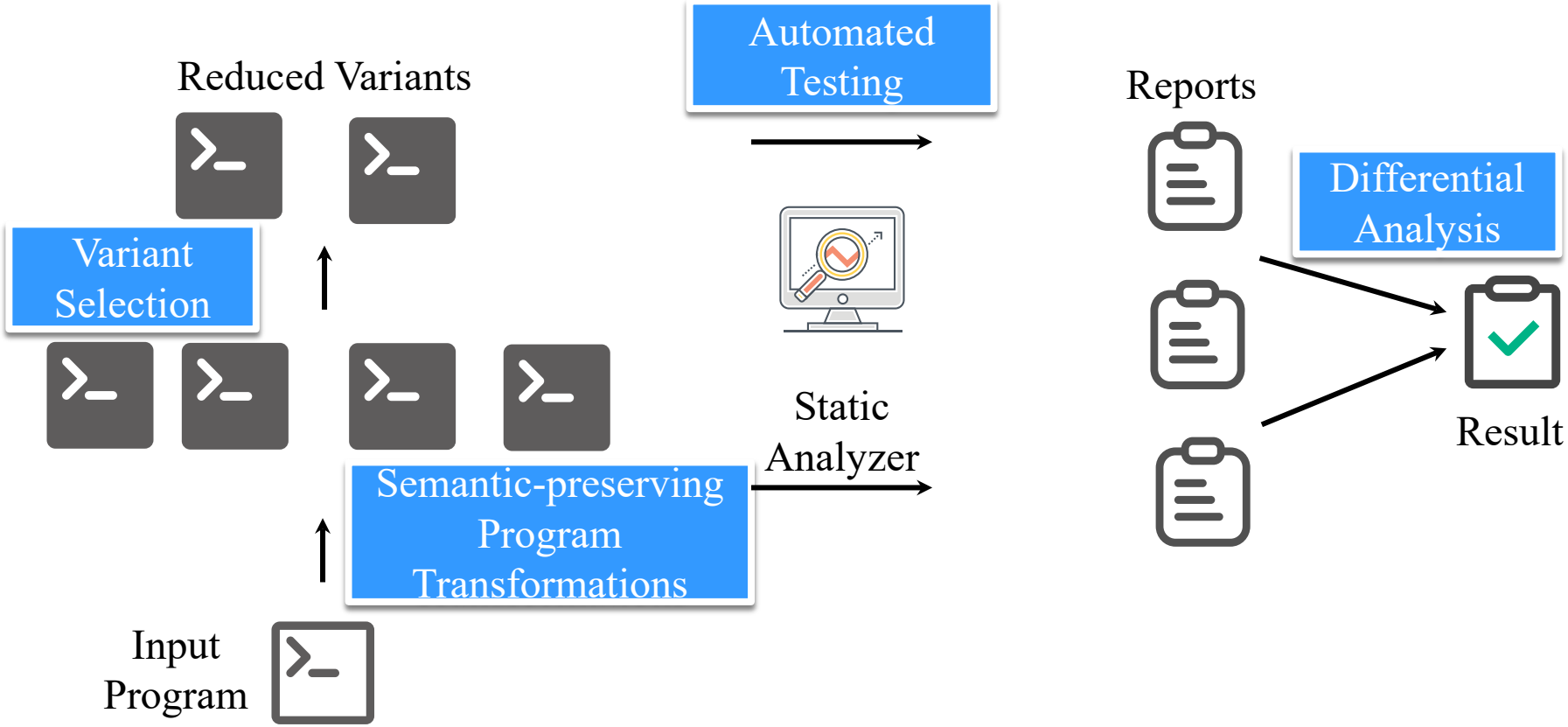


Challenge 2: Automated generation of high-quality input programs

Reusing official test suites & documentation

- **Official test suites** contain test programs with oracles
- **Documentation** includes example programs to explain the rule checkers

But there are too many programs
How do Statfier select Input Programs?



Where to modify?

Analysis report guided location (AL)

- Use locations in analysis report
- Control/data dependency related to these locations

Candidate Selection

Heuristic 2: Structurally diverse variant selection (SS)

- Avoid selecting variant where the context and the selected type of transformation is the same

How to Modify?

Semantically-equivalent Program Transformations

5 levels

Level	Transformation	Example	Source

Variable-level	Extract local variable	`- invoke_method("String Literal");`
		`+ String str = "String Literal";`
		`+ invoke_method(str);`
			[11, 51]

	Move assignment	`- int var = 10;`
		`+ int var;`
		`+ var = 10;`
			[11]

Expression-level	Equivalent boolean expression: Add $\|\|false$ or $\&\&true$ expression Swap symmetrical elements, e.g., $a == b \rightarrow b == a$	`- boolean tag = true;` `+ boolean tag = true || false;`	[27, 31]
	Equivalent arithmetic expression: Add +0, -0, or +1-1 expression	`- int value = 10;` `+ int value = 10 + 0;`	[2, 27]
	Add parenthesis to expression	`- int value = 10;` `+ int value = (10);`	[11, 23]

Statement-level	Equivalent statement conversion:
	Convert to equivalent for/while/do-while/lambda
		`- for(i = 0; i < 1; i++) {}`
		`+ i = 0;`
		`+ while(i++ < 1) {}`
			[11]

	Statement wrapping: Wrap statements with if/while/for/do-while	`- target_statement;` `+ if(true) { target_statement; }`	[27, 65]
	Dead code injection: Insert dead if/while/for statement	`target_statement;` `+ for(int i = 0; i < 0; ) { target_statement; }`	[49, 51]

Method-level	Encapsulate field	`- SecretKeySpec("Hardcode");`
		`+ String getHardcode() { return "Hardcode"; }`
		`+ SecretKeySpec(getHardcode());`
			[11, 23]

Class-level	Nested class wrapping	`- original_program;` `+ class NestClass { original_program; }`	[21]
	Anonymous class wrapping	`- original_program;` `+ Object c = new Object() { original_program; };`	[22]
	Enum wrapping	`- original_program;` `+ enum enumClass { original_program; }`	[56]

Experimental Results

- 5 Static Analyzers (PMD, SpotBugs, CheckStyle, SonarQube, and Infer)

RQ1: How many unique bugs can Statfier find?

✓ Find 79 bugs in 5 analyzers, of which 46 have been confirmed

RQ2: Are proposed heuristics effective?

✓ Two heuristics in Statfier selects less variants (40.2%–41.3%) but still find more unique bugs than other baselines

RQ3: How many bugs can each transformation find?

✓ Each program transformation can find at least one bug in the evaluated analyzers

Statfier: Automated Testing of Static Analyzers via Semantics-Preserving Program Transformations

Huaien Zhang, Yu Pei, Junjie Chen, Shin Hwei Tan

- **Proposed Statfier, an automated testing approach to detect bugs in static analyzers based on semantic-preserving transformations and metamorphic testing**
- **2 heuristics: (1) Analysis report guided location and (2) Structurally diverse variant selection**
- **Find 79 bugs in 5 analyzers, of which 46 have been confirmed**
- **Checkout our website at https://sa-research.github.io/**

Automated Test Generation for Program Analyzer

Statfier: Automated Testing of Static Analyzers via Semantics-Preserving Program Transformations (FSE'23)

Understanding & Detecting Annotation-Induced Faults of Static Analyzers (FSE'24)

Characterizing & Detecting Program Representation Faults of Static Analysis Frameworks (ISSTA'24)

Understanding and Detecting Annotation-Induced Faults of Static Analyzers

Huaien Zhang, Yu Pei, Shuyun Liang, Shin Hwei Tan

*Accepted and Presented in FSE'24

Java Annotation & Challenges of Handling Annotation for Static Analyzers

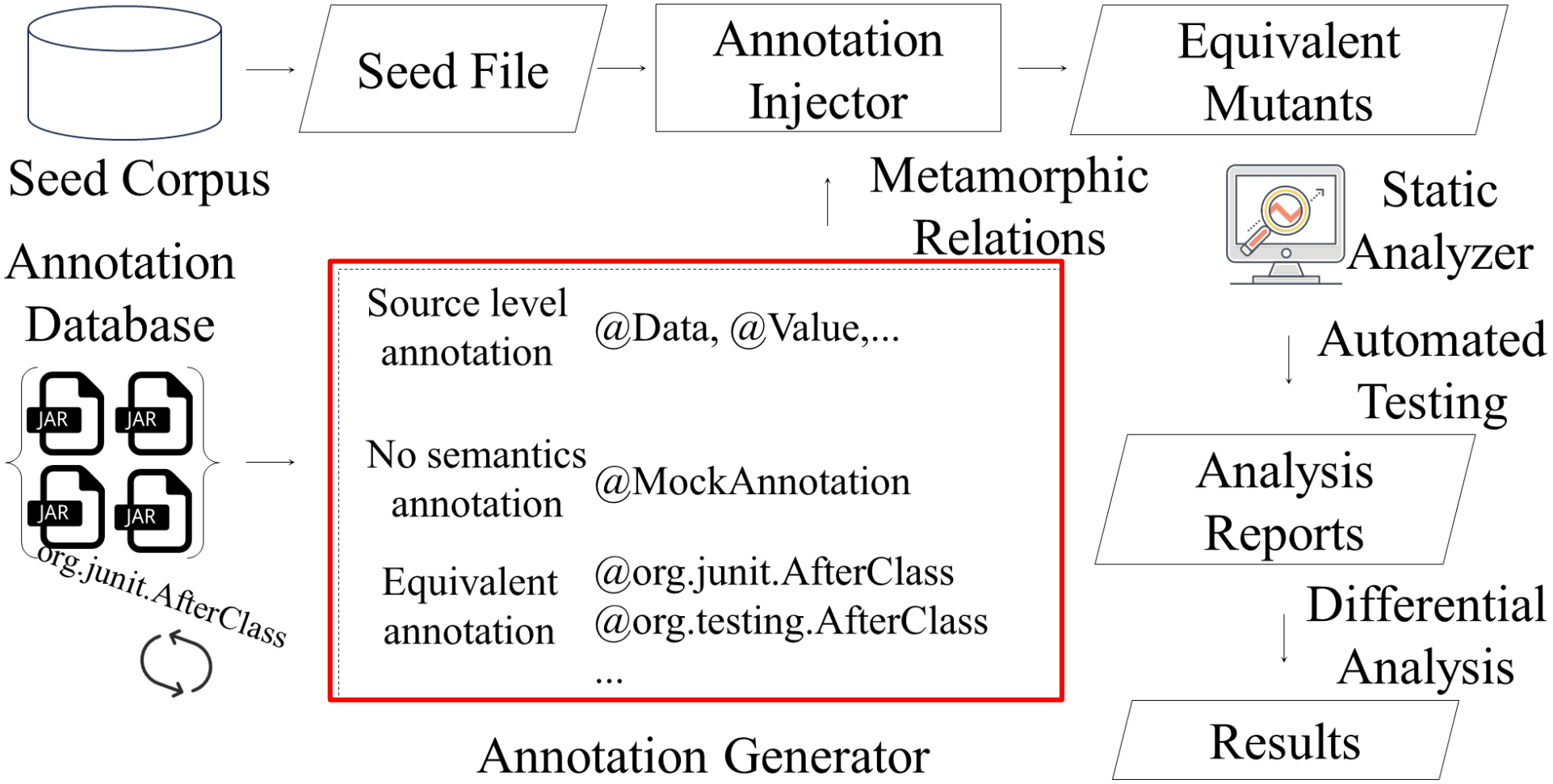
- A form of metadata
- Attach information to program elements

```
public class Machine {
    private List versions;

    @SuppressWarnings("unchecked")
    public void addVersion(String version) {
        versions.add(version);
    }
}
```

- Challenges:
  - Annotations introduce extra tokens
    - Static analyzers may overlook or mishandle the tokens, leading to incorrect analysis results or even crash.
  - Annotations introduce changes to the structure or behavior of the programs at compile or execution time.

AnnaTester: Testing annotation-induced Fault



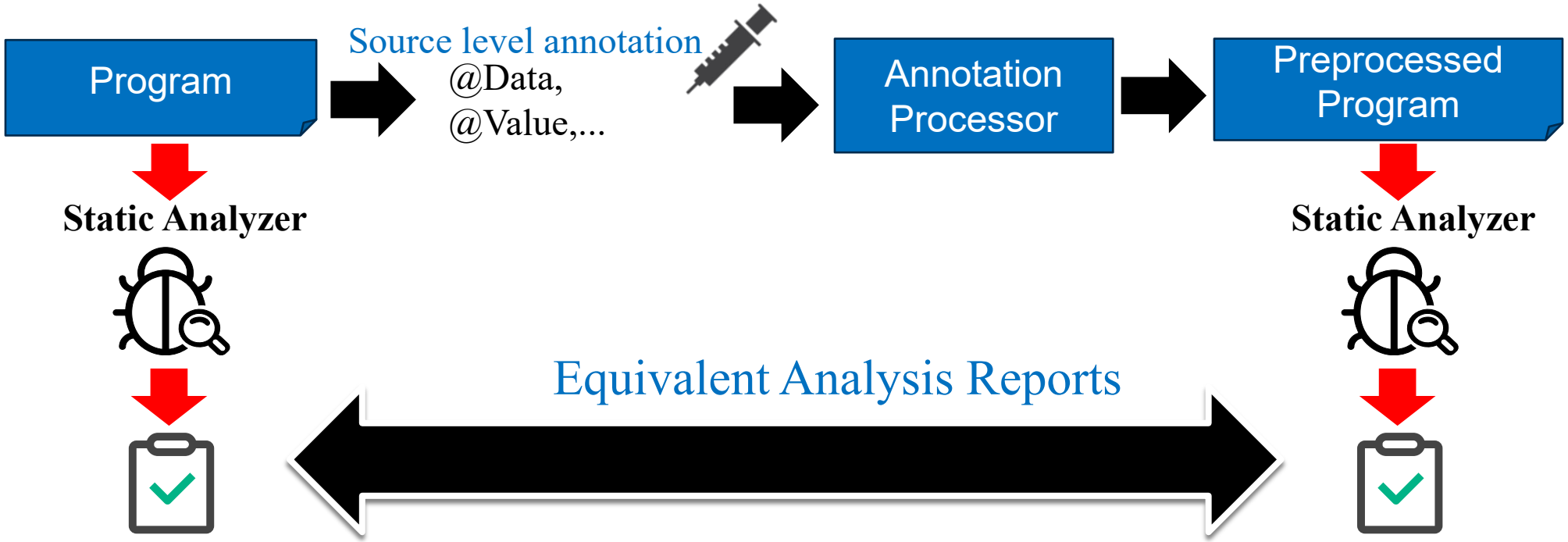
Checkers Design in *AnnaTester*

How to Modify?

- Source level annotation injection

Evaluation: Incomplete Semantics Checker

- Program P should be analysis equivalent to the program produced by processing the annotations in P.



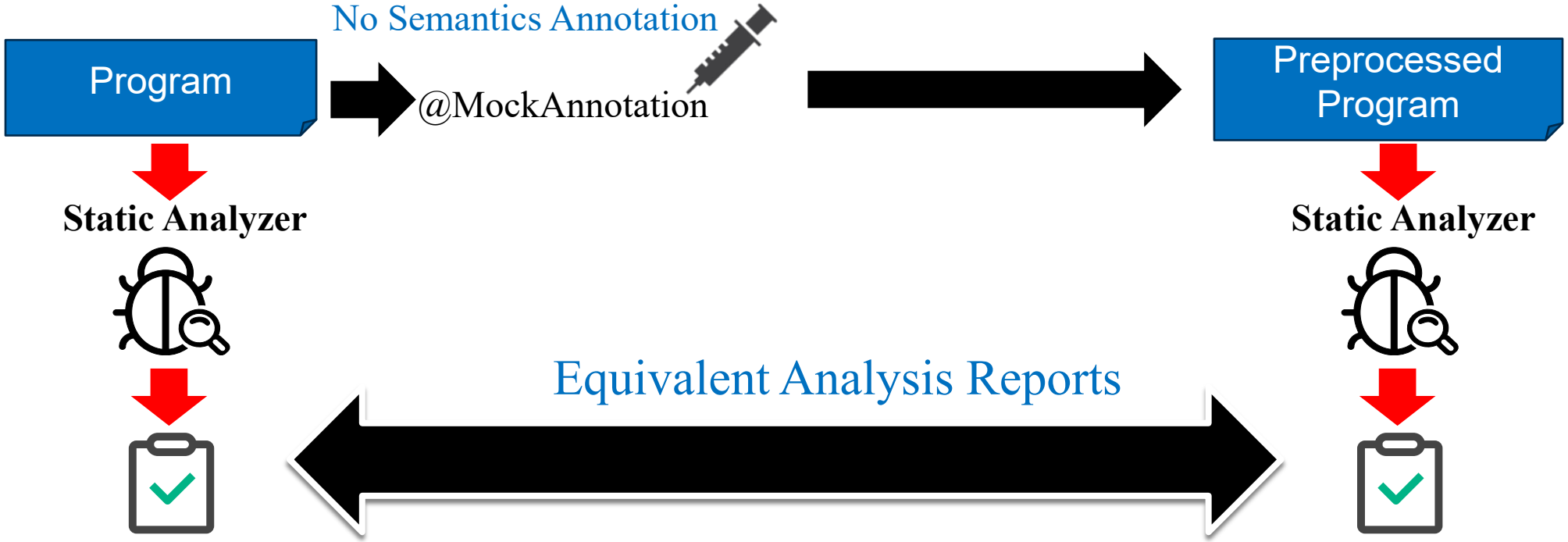
Checkers Design in *AnnaTester*

How to Modify?

- No semantic annotation injection

Evaluation: Annotation Syntax Checker

- Program P and P injected by no semantics annotation should be analysis equivalent.



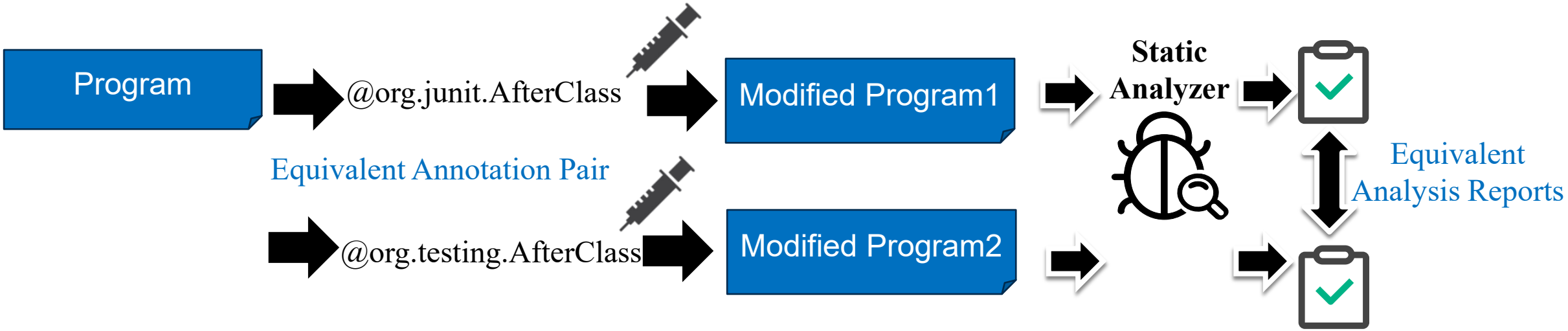
Checkers Design in *AnnaTester*

How to Modify?

- Equivalent Annotation Pair Injection

Evaluation: Equivalent Annotation Checker

- Given a program P annotated with an annotation a_1 and another annotation a_2 that is equivalent to a_1, P should be analysis equivalent with $P_{a_1|a_2}$



Effectiveness of AnnaTester

- 6 Static Analyzers
  - SonarQube, Infer, PMD, CheckStyle, SpotBugs, Soot

Checker	#Violations	#UniqFaults	#FP	#Fixed	Time (min, max) (hour)
ISC	258	19	8	11	(4,62)
ASC	52	8	0	4	(2,24)
EAC	123	16	0	5	(6,87)
Overall	433	43	8	20	(6,87)

- 43 new bugs found in static analyzers, 20 have been confirmed and fixed.

Understanding and Detecting Annotation-Induced Faults of Static Analyzers

Huaien Zhang, Yu Pei, Shuyun Liang, Shin Hwei Tan

- Conducted the first empirical study on annotation-induced faults in static analyzers, and analyzed their root causes, symptoms, fix strategies, and types of AIF annotations, deriving ten findings.
- Proposed an automated testing framework ***AnnaTester*** that uses metamorphic testing to detect three types of annotation-induced faults in static analyzers.
- Evaluated ***AnnaTester*** on six static analyzers, which revealed 43 new bugs in these static analyzers, 20 of them have been confirmed and fixed.

Automated Test Generation for Program Analyzer

Statfier: Automated Testing of Static Analyzers via Semantics-Preserving Program Transformations (FSE'23)

Understanding & Detecting Annotation-Induced Faults of Static Analyzers (FSE'24)

Characterizing & Detecting Program Representation Faults of Static Analysis Frameworks (ISSTA'24)

Characterizing and Detecting Program Representation Faults of Static Analysis Frameworks

Huaien Zhang, Yu Pei, Shuyun Liang, Zezhong Xing, Shin Hwei Tan

THE HONG KONG POLYTECHNIC UNIVERSITY
香港理工大學

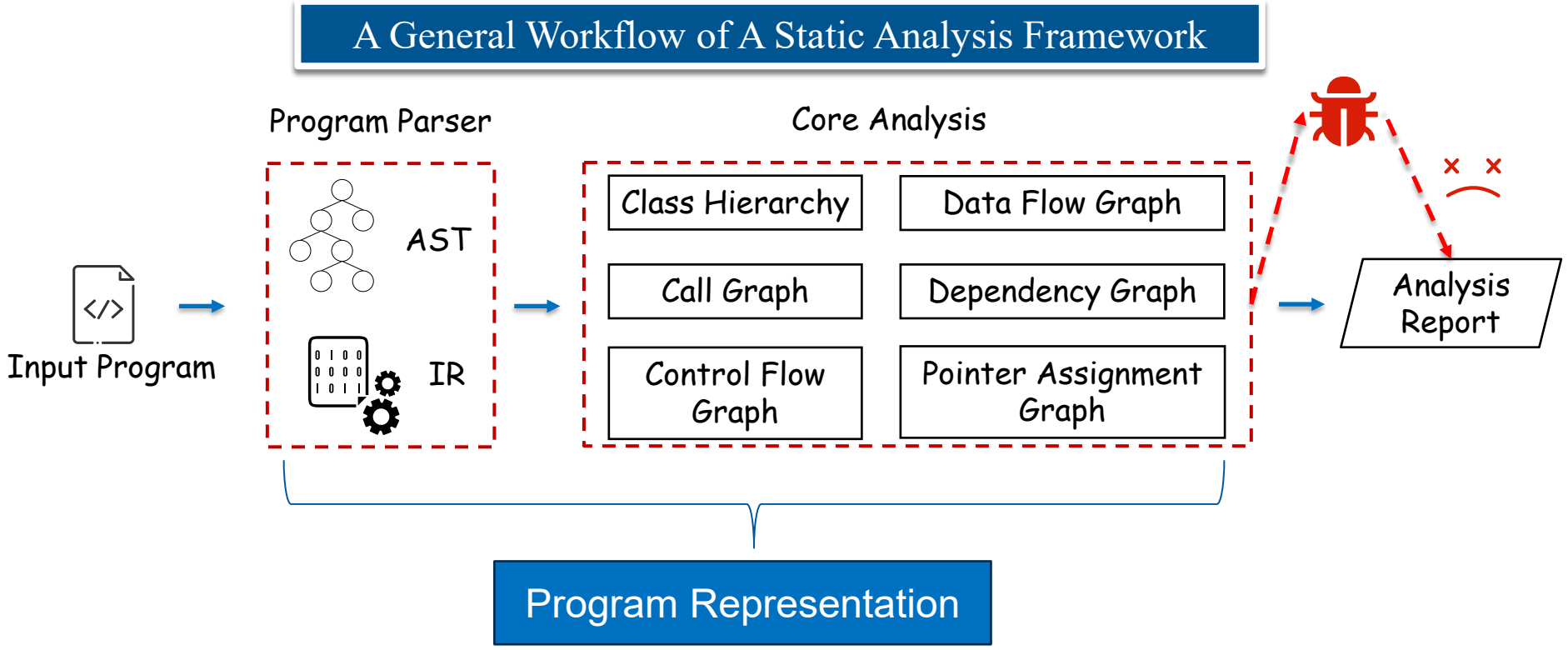
SUSTech
Southern University of Science and Technology

UNIVERSITÉ Concordia UNIVERSITY

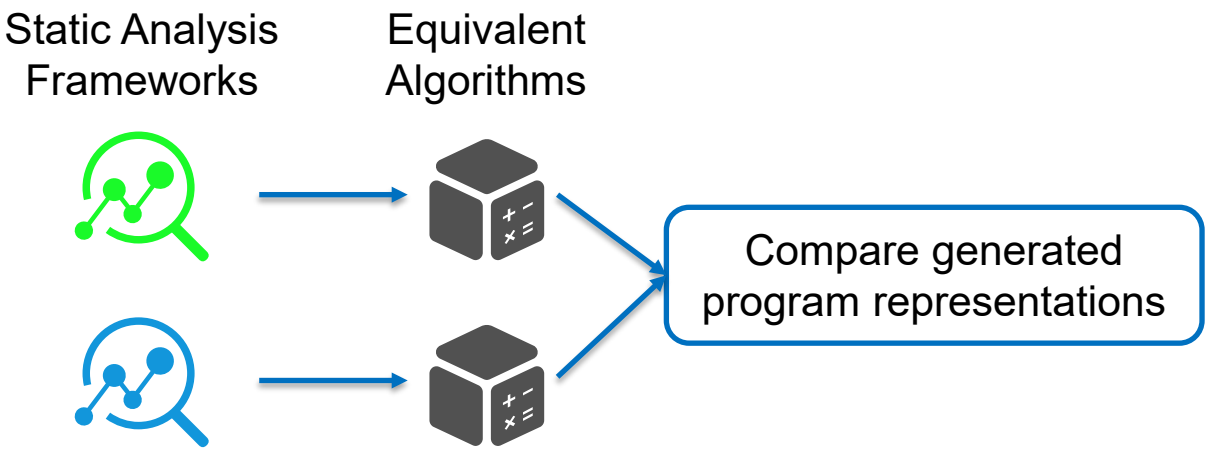
*Accepted and Presented in ISSTA'24

Program Representation Faults

- Construct various program representations to encode the properties and behaviors of the given programs for further analysis

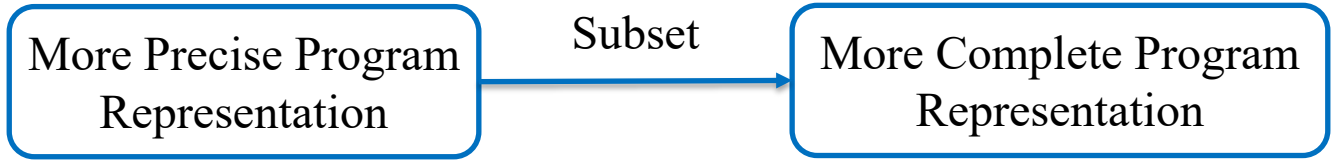


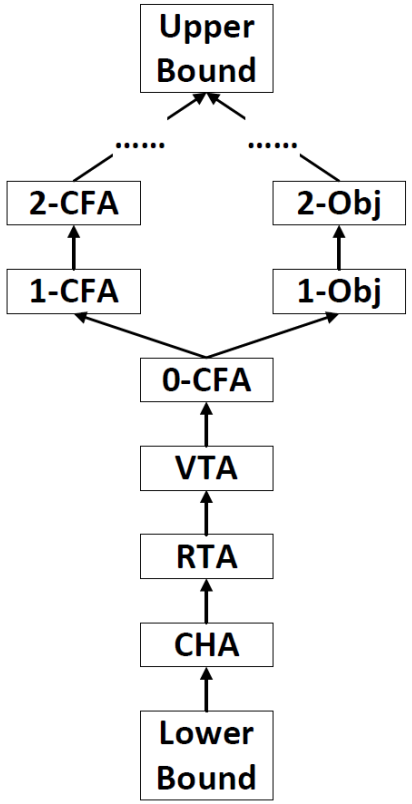
Key insight 1 of *SAScope*: Differential Testing



Two program representation $\phi 1$ and $\phi 2$ are equivalent if and only if (1) G1 = G2 or L1 = L2; (2) $\phi 1$ and $\phi 2$ are generated by the same algorithm (e.g., call graph).

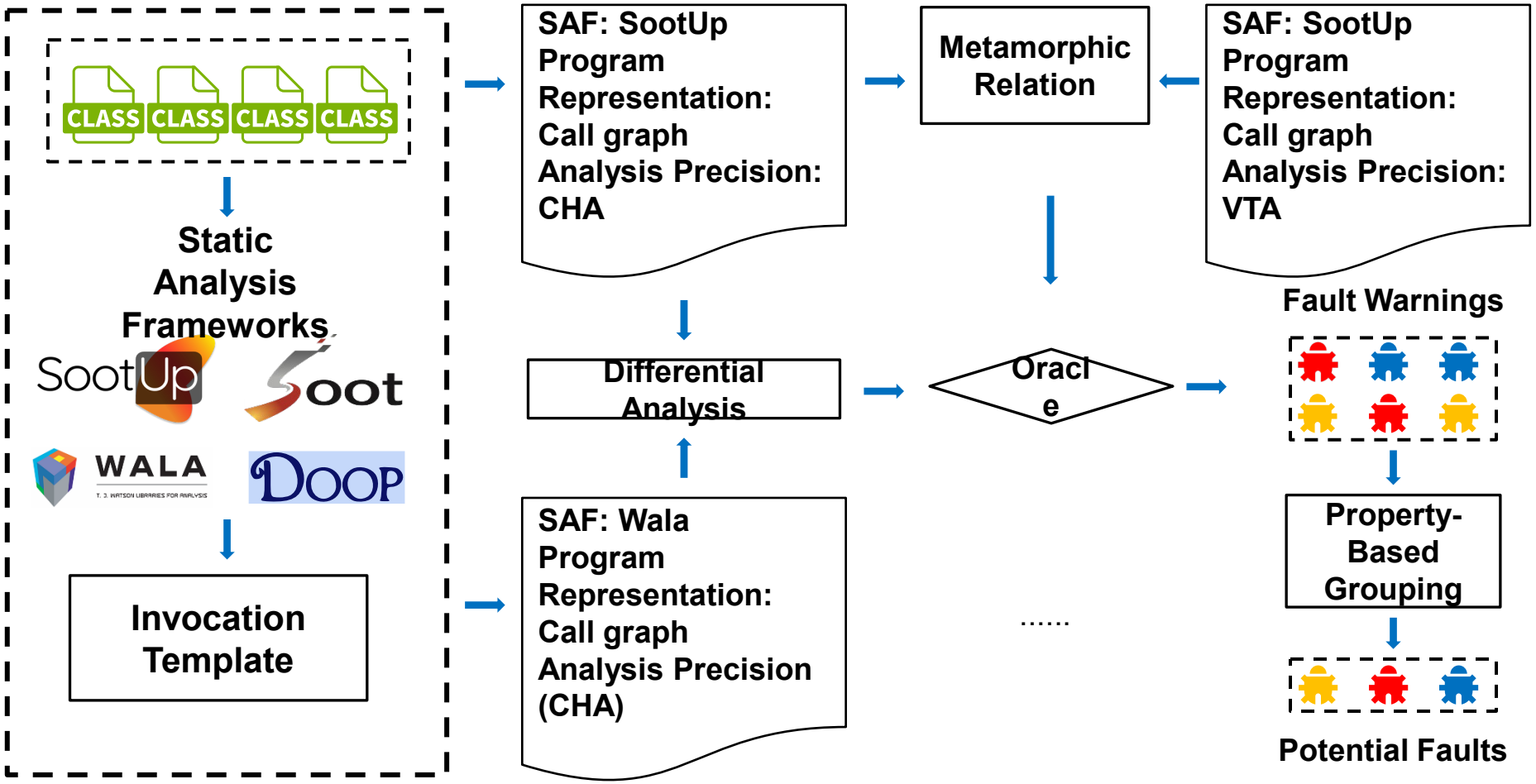
Key insight 2 of *SAScope*: Metamorphic Testing





Given the program representation $\phi 1$ and $\phi 2$ generated by $\delta 1$ and $\delta 2$ under the same input program, they should possess the property $\phi 1 \supseteq \phi 2$ if $\delta 1 \preccurlyeq \delta 2$ ($\delta 1$ less precise than $\delta 2$).

Workflow of *SAScope*



Effectiveness of *SAScope*

- Four Static Analysis Frameworks
  - SootUp, Wala, Soot, Doop
- Dataset
  - Top 200 popular Maven libraries

Number of unique faults detected by *SAScope*

SAFs	# Warnings	# Groups	# Unique Faults	# Fixed
SootUp	26951	10	8	1
Wala	31734	11	7	4
Soot	21051	6	3	0
Doop	12896	4	1	0
Overall	92632	31	19	5

Characterizing and Detecting Program Representation Faults of Static Analysis Frameworks

Huaien Zhang, Yu Pei, Shuyun Liang, Zezhong Xing, Shin Hwei Tan

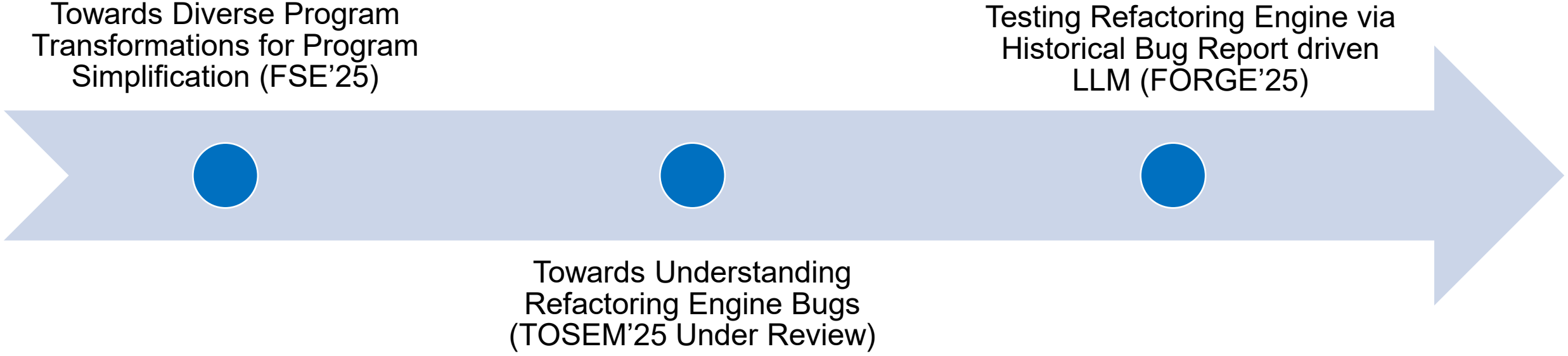
- First empirical study on program representation faults in static analysis frameworks.
- Inspired by study findings, we implemented an automated testing framework SAScope to detect PRFs based on metamorphic and differential testing.
- We evaluated SAScope on four studied static analysis frameworks and found 19 new faults, five of which have been fixed by developers.

GI versus Testing Program Analyzers

1. Profiling/Localization
2. Program Generation
   - Mutations
3. Program Evaluation

1. Modified Line Localization
   - Statfier: Analysis report guided location (AL)
2. Patch Generation
   - Semantically-equivalent Transformation
   - Annotation Injection
3. Program Analyzer Evaluation
   - Design of Metamorphic Relation
     - *Statfier* validates semantically-equivalent programs
     - *AnnaTester* validates annotated programs
     - *SAScope* validates program representations

Understanding and Testing Semantically Equivalent Transformation



Towards Diverse Program Transformations for Program Simplification

Haibo Wang, Zezhong Xing, Chengnian Sun, Zheng Wang,
Shin Hwei Tan

*Accepted and Will Present in FSE'25

Program Simplification

- The simpler the better!
- Why developer simplify program?
  - ✓ Cleanup code
  - ✓ Improve readability
  - ✓ Reduce complexity
  - ✓ Improve Reusability

Transformations in Program Simplification

- Simplification Goal: Produce smaller programs (less lines of code)

Given an input program, what kind of program transformation would you use to produce a simplified program with less lines of code?

Existing Work on Program Simplification

- Syntactic Simplification:
  - **Rule-based** transformation
    - Refactoring:
    - Genetic Programming:
      - "Using **Numerical Simplification** to Control Bloat in Genetic Programming" [SEAL'08]
      - **"Algebraic simplification** of GP programs during evolution" [GECCO'06]
- Semantic Simplification:
  - Use **test executions** to check for behavioral equivalence
  - Deletion-based
    - Delta Debugging
    - Program Reduction
    - Program Debloating
    - Program Slicing
  - Genetic Programing
    - Mutation and crossover operator in "Genetic Programming for Shader Simplification" [TOG'11]

Most technique are deletion-based!
Does these correspond to transformations used by developers?

RQ1: Frequently used Transformations in Program Simplification

- Refactoring: Extract Method (19%)
- Unsupported:
  - Replace with equivalent API (16.2%)
  - Deletion-based: Remove unnecessary code (12.3%)
  - Simplify boolean and algebraic expression (8.4%)

    false == co.isExpired()→ co.isExpired()
  - Java language feature: Use diamond operator (4.2%)

Set<String> conditionKeys = newHashSet<String>(); →
Set<String> conditionKeys = new HashSet<>();

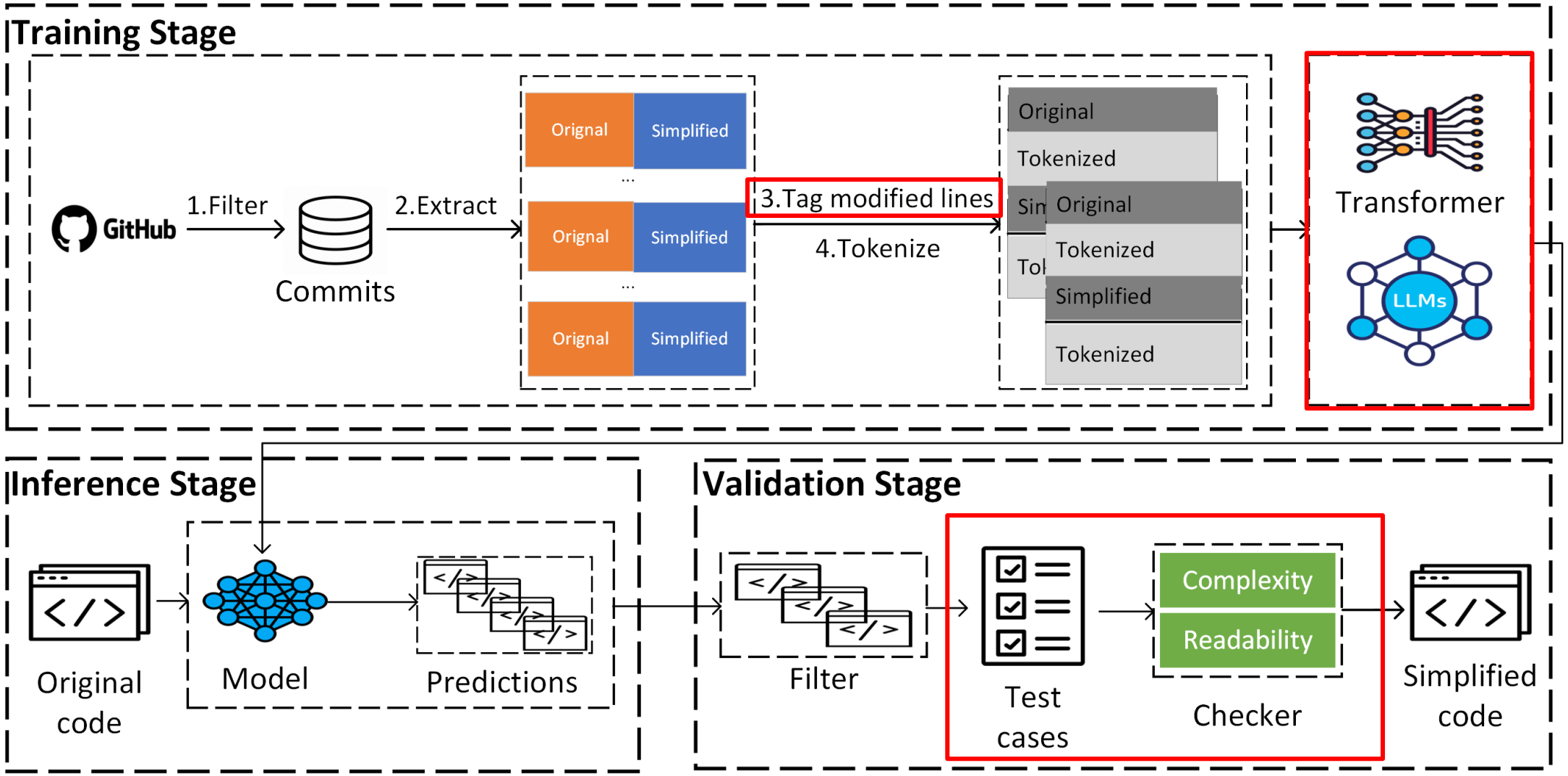
Unsupported transformations: Replace with equivalent API

```
+ public void assertHasAndNotNull(String str) {
+   assert(str != null);
+   assert(!str.equals("")); }
public Builder playlist_id(final String playlist_id) {
-   assert (playlist_id != null);
-   assert (!playlist_id.equals(""));
+   assertHasAndNotNull(playlist_id);
    return setPathParameter("playlist_id", playlist_id);}
```

Replace a few lines of code with equivalent method call

- Challenge in rule-based approaches: Need to check for equivalent code
- Learning-based approaches can be used to support diverse transformations

SimpT5: Program Simplification Framework



GI versus Automated Program Simplification

1. Profiling/Localization
2. Program Generation
   - Mutations
3. Program Evaluation

1. Modified Line Localization
   - Trained Using Tagged modified line
2. Simplified Program Generation
   - CodeT5 tuned using our collected benchmark
3. Program Validation
   - Test-equivalent check
   - Quality Checkers
     - *Complexity*
     - *Readability*

Replace with equivalent API

```
+ public void assertHasAndNotNull(String str) {
+   assert(str != null);
+   assert(!str.equals("")); }
public Builder playlist_id(final String playlist_id) {
-   assert (playlist_id != null);
-   assert (!playlist_id.equals(""));
+   assertHasAndNotNull(playlist_id);
    return setPathParameter("playlist_id", playlist_id);}
```

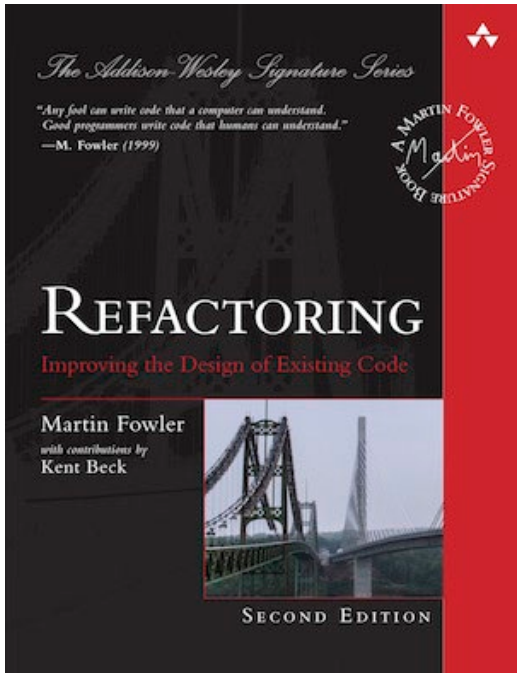
- ✓ SimpT5 successfully generate the correct simplified program!
- ✓ SimpT5 can generate simplified programs via 14 diverse types of transformations

Refactoring

Syntactic Simplification: Rule-based

Refactoring is the process of changing a software system in a way that does not alter the external behavior of the code yet improves its internal structure.

Martin Fowler

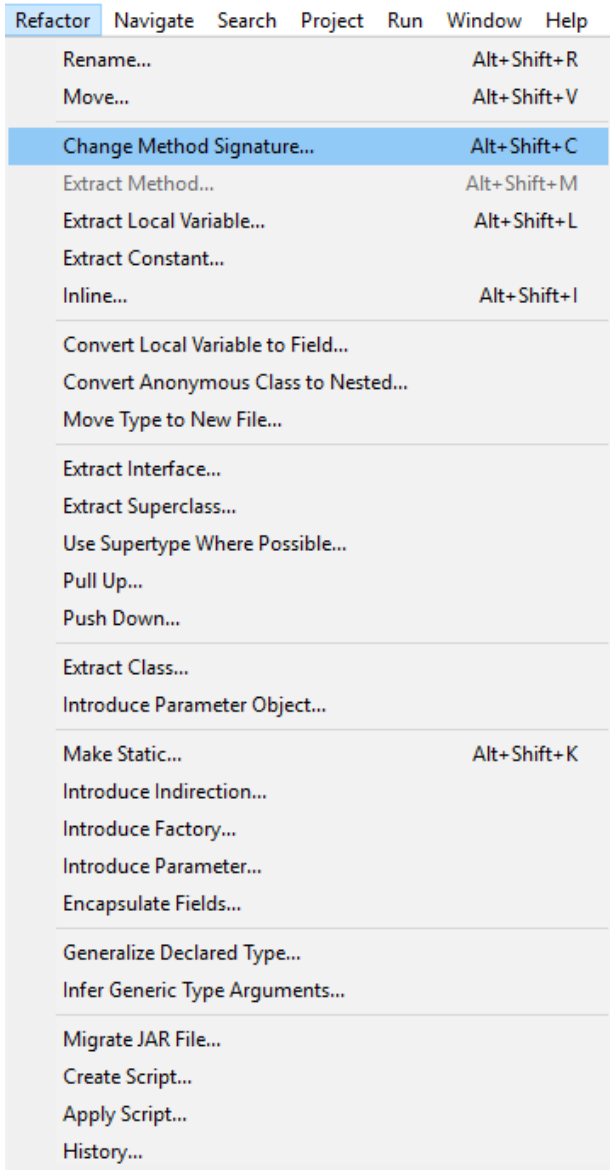


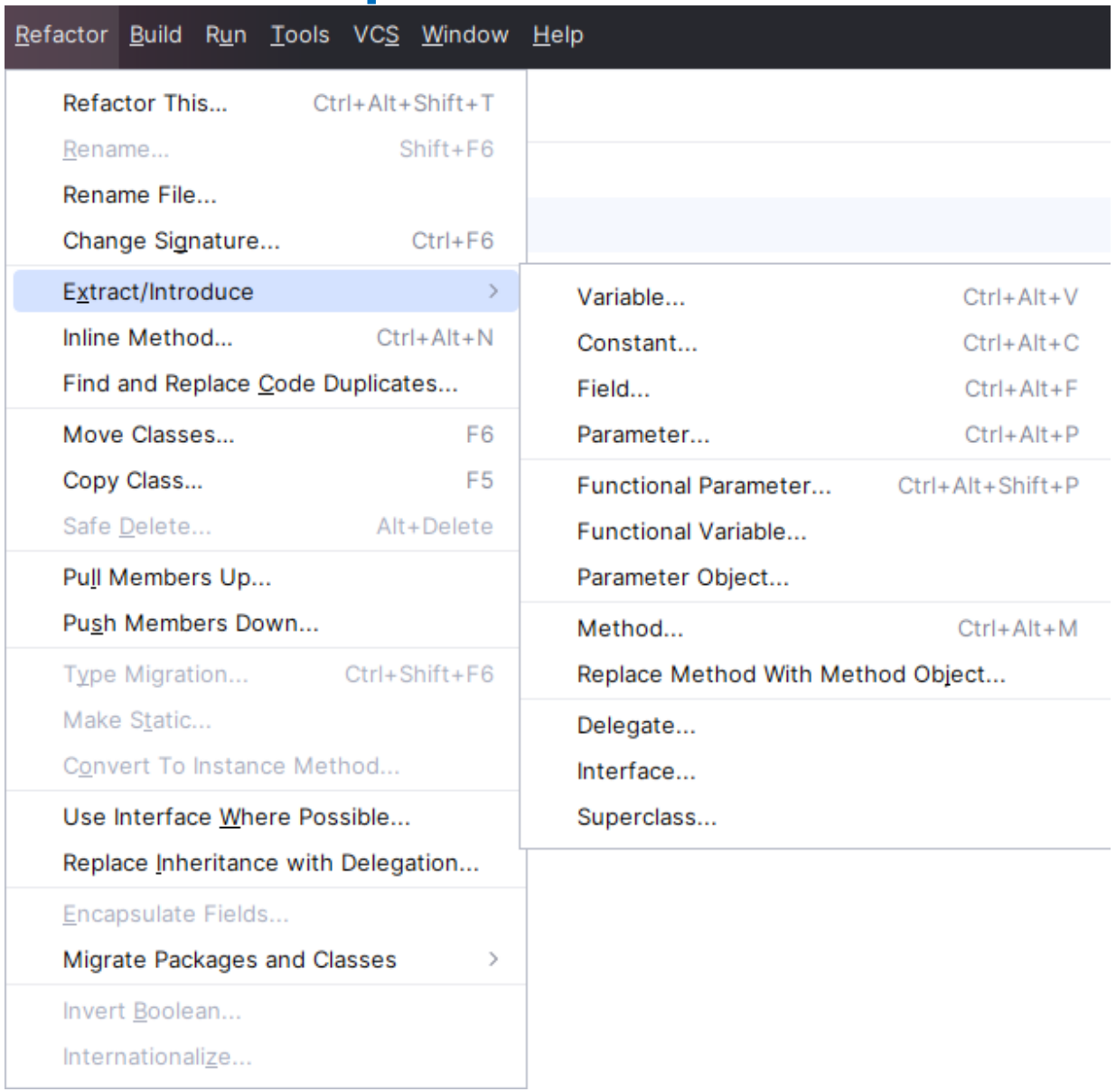
Towards Understanding Refactoring Engine Bugs

Haibo Wang, Zhuolin Xu, Huaien Zhang, Nikolaos Tsantalis,
Shin Hwei Tan

*Under review in TOSEM

Refactoring Engine in Eclipse and IntelliJ





Understanding Refactoring Engine Bugs

RQ1: What kind of refactorings are more likely to trigger refactoring engine bugs?

- Extract
  - Pull Up/Down
  - Extract Method
  - Extract Variable
- Inline
  - Inline Method
  - Inline Variable
- Move
  - Move Method
  - Move Type to New File

Example bugs in Extract Local Variable

Where is the bugs?

Eclipse-104293: extract local does not replace all concurrences of expression

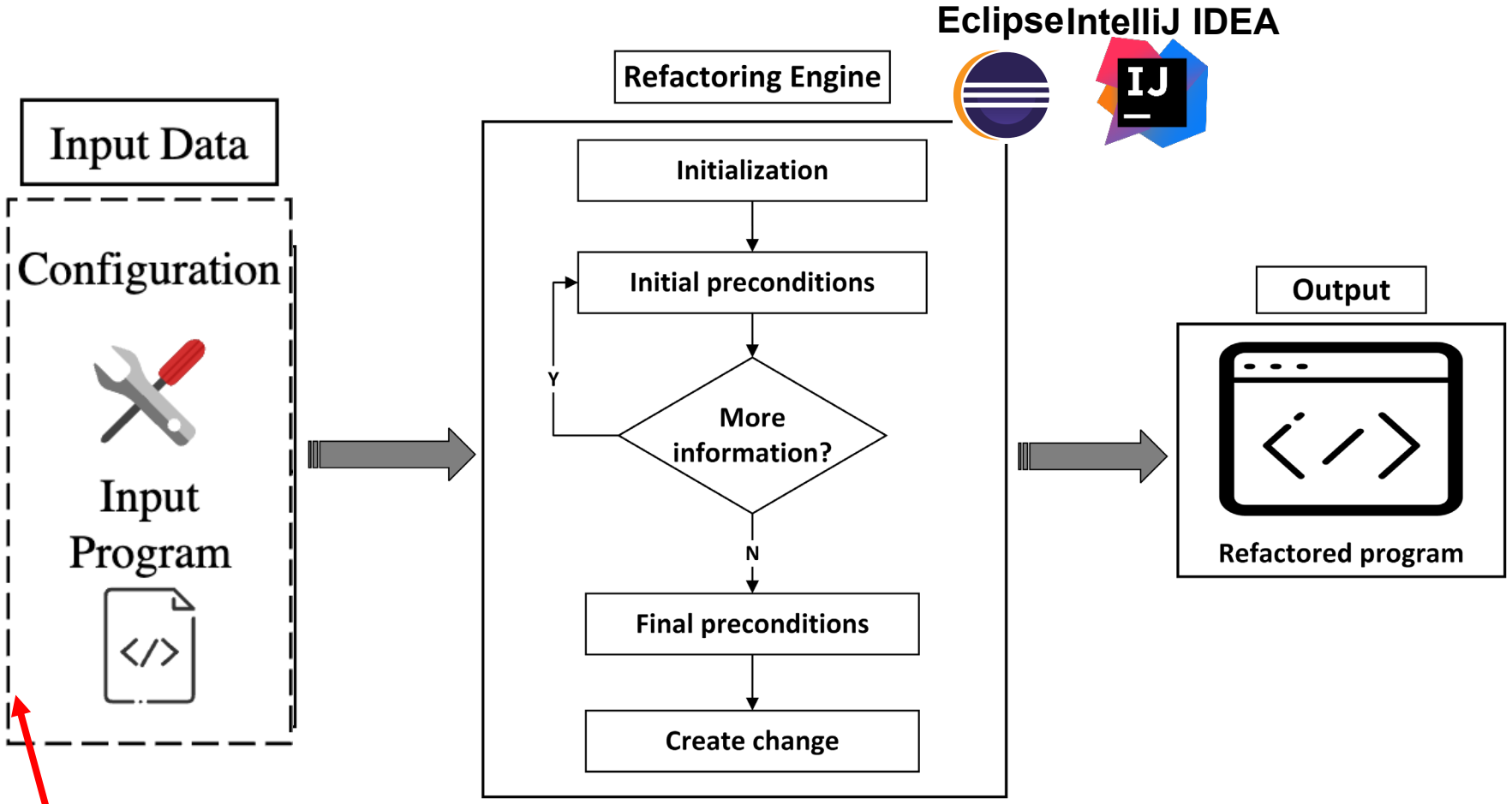
```
import static java.util.Arrays.*;
class Bug {
    {
        String[]side=new String[0];
        if(true){
-           System.out.println(asList(side));
        }
        else{
            System.out.println(asList(side));
        }
    }
}
```

Listing 3. Code before refactoring.

```
+import java.util.List;
import static java.util.Arrays.*;
class Bug {
    {
        String[]side=new String[0];
+        List<String> list = asList(side);
        if(true){
+           System.out.println(list);
        }
        else{
            System.out.println(asList(side));
        }
    }
}
```

Listing 4. Code after refactoring.

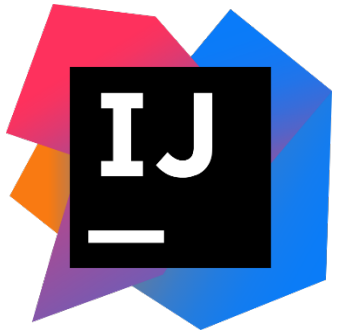
Understanding Refactoring Engine Bugs



RQ2: What are the symptoms of refactoring engine bugs?
RQ3: What are the root causes of refactoring engine bugs
RQ4: What are the input characteristics that trigger bugs in refactoring engine?

Understanding Refactoring Engine Bugs

Eclipse



IntelliJ IDEA

RQ2: Common symptoms

- Compilation Error
- Crash
- Behavior Change

RQ3: Common root causes

- Incorrect transformations
  - Improper handling code comments
  - Incorrect modifier modification
- Incorrect precondition checking
- Incorrect flow analysis

RQ4: Error-prone Input Characteristics

Category	Sub-category
(T1) Language Features	(T1.1) Lambda expression
	(T1.2) Java generics
	(T1.3) Enum
	(T1.4) Record
	(T1.5) Varargs
	(T1.6) instanceof
	(T1.7) Foreach
	(T1.8) Switch case
	(T1.9) Try-with-resources
	(T1.10) Var
	(T1.11) Try-catch-finally
	(T1.12) Joint variable/field declaration
	(T1.13) Multi-dimension Array
	(T1.14) Vector
	(T1.15) Synchronized block
	(T1.16) Java ternary conditional
	(T1.17) Keyword "this"
(T2) Class-related	(T2.1) Inner class
	(T2.2) Anonymous class
	(T2.3) Cyclically dependent class
(T3) Annotations	(T3.1) Annotations
(T4) Code Comment	(T4.1) Comment related
(T5) Method-related	(T5.1) Overloaded method
	(T5.2) Static method
	(T5.3) Method reference
	(T5.4) Recursive method
	(T5.5) Default method
(T6) Static	(T6.1) Static initializer
	(T6.2) Static import
	(T6.3) Static field
(T7) Constructor-related	(T7.1) Super constructor
	(T7.2) Nested constructor
	(T7.3) Implicit constructor
(T8) Others	(T8.1) Special String
	(T8.2) Arithmetic expression
	(T8.3) Time-consuming method call
	(T8.4) Dead code block
	(T8.5) Method chaining

- Lambda expression
- Java generics
- Annotations

Can we use these input characteristics for testing refactoring engine?

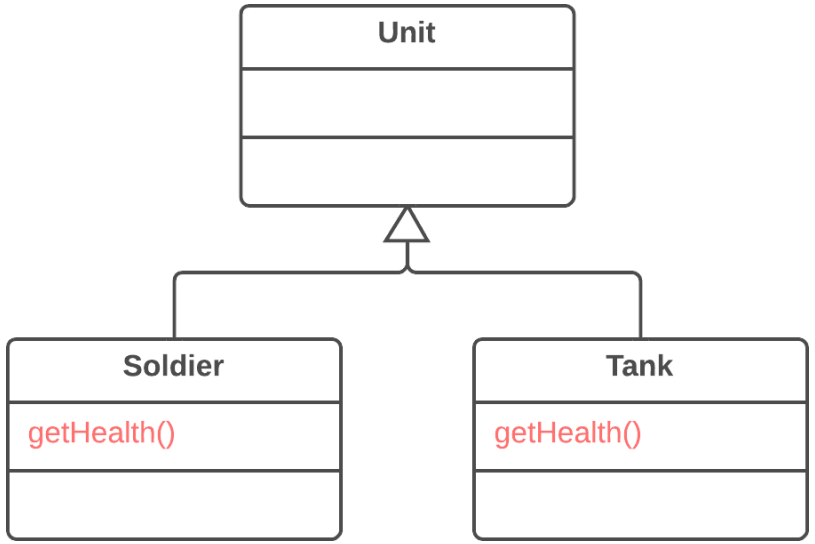
Testing Refactoring Engine via Historical Bug Report driven LLM

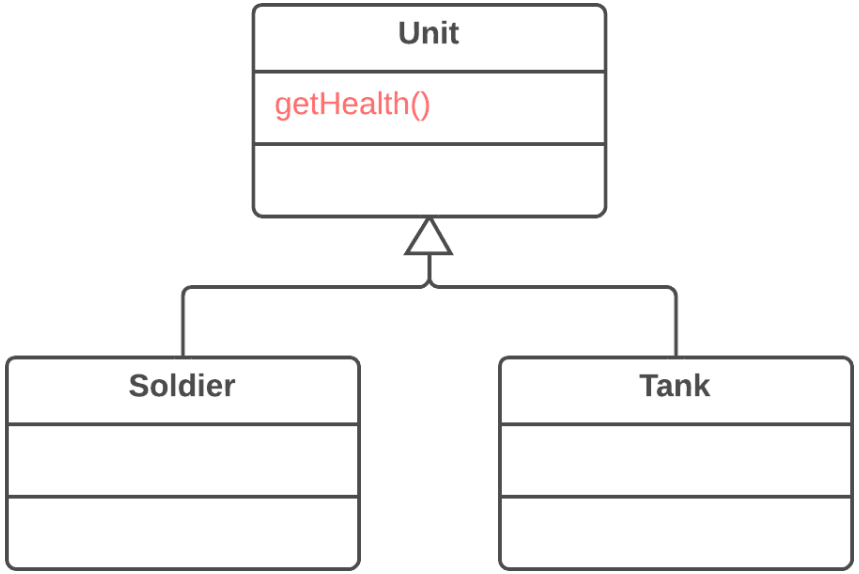
Haibo Wang, Zhuolin Xu, **Shin Hwei Tan**

*Accepted and Will Present in FORGE’25

Example: Pull Up Method

"Pull Up Method" means moving a method from a subclass to its superclass, promoting code reuse and reducing redundancy when multiple subclasses share similar functionality.





Example: Pull Up Method

```java
class Animal {
}

class Dog extends Animal {
    void sleep() {
        System.out.println("Sleeping...");
    }
}

class Cat extends Animal {
    void sleep() {
        System.out.println("Sleeping...");
    }
}
```

Pull up sleep()

```java
class Animal {
    void sleep() {
        System.out.println("Sleeping...");
    }
}

class Dog extends Animal {
}

class Cat extends Animal {
}
```

A more complex input program for pull up method refactoring

```
public class A {
    public class BaseInner {}

    public class Outer {
        public int x = 0;
        public void foo(){};

        public class Inner extends BaseInner {
            void innerMethod() { // Pull this method up to class BaseInner
                System.out.println(Outer.this.x);
                Outer.this.foo();
            }
        }
    }
}
```

Eclipse (V202406) produces a syntax-error program

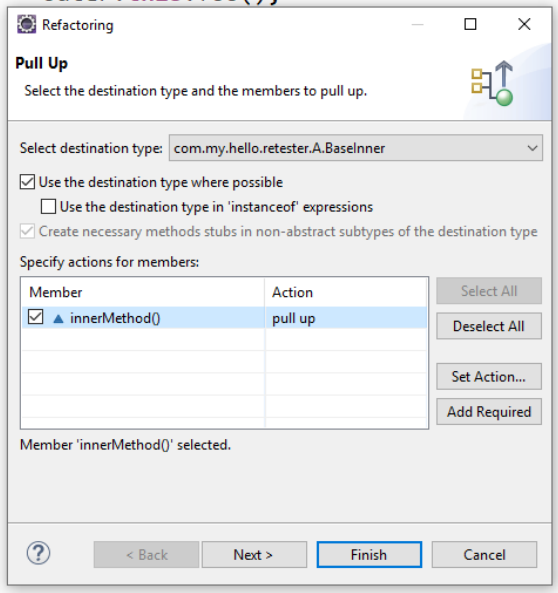
A.java

```java
package com.my.hello.retester;

public class A {
    public class BaseInner {}

    public class Outer {
        public int x = 0;
        public void foo(){};

        public class Inner extends BaseInner {
            void innerMethod() { // Pull this method up
                System.out.println(Outer.this.x);
                Outer.this.foo();
            }
        }
    }
}
```



Pull up innerMethod()

A.java

```java
package com.my.hello.retester;

import com.my.hello.retester.A.Outer.Inner;

public class A {
    public class BaseInner {

        void innerMethod(Inner inner) { // Pull
            System.out.println(inner.x);
            inner.foo();
        }}

    public class Outer {
        public int x = 0;
        public void foo(){};

        public class Inner extends BaseInner {
        }
    }
}
```

https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1533

Historical bug reports

[Bug][Pull Up Refactoring] Pull up method refactoring for method in the inner class fails #1533

Edit | New issue

Closed | RETester66 opened this issue on Jul 22 · 0 comments · Fixed by #1590

RETester66 commented on Jul 22

Steps to reproduce

1. Create a class `A`:

```
public class A {
    public class BaseInner {}

    public class Outer {
        public int x = 0;
        public void foo(){};

        public class Inner extends BaseInner {
            void innerMethod() { // Pull this method up to class BaseInner
                System.out.println(Outer.this.x);
                Outer.this.foo();
            }
        }
    }
}
```

2. Like I commented on the above code, left click method `innerMethod`, then right click -> Refactor -> Pull Up, set the destination as class BaseInner, and use the default configuration as following, click Finish:

A.java

```
1 package com.my.hello.retester;
2
3 public class A {
4     public class BaseInner {}
```

Assignees: jjohnstn

Labels: bug

Projects: None yet

Milestone: No milestone

Development: Successfully merging a pull request may close this issue.

Add outer class checking to Pull Up Refactoring
jjohnstn/eclipse.jdt.ui-1

Notifications | Customize

Unsubscribe

You're receiving notifications because you authored the thread.

2 participants

Source	Initial	Compilable	Mean LOC	Median LOC
ECLIPSE	245	101	11	9
INTELLIJ IDEA	213	66	11	9
Total	458	167	–	–

The seed bug report information

Error-prone input program characteristics

Category	Sub-category
(T1) Language Features	(T1.1) Lambda expression
	(T1.2) Java generics
	(T1.3) Enum
	(T1.4) Record
	(T1.5) Varargs
	(T1.6) instanceof
	(T1.7) Foreach
	(T1.8) Switch case
	(T1.9) Try-with-resources
	(T1.10) Var
	(T1.11) Try-catch-finally
	(T1.12) Joint variable/field declaration
	(T1.13) Multi-dimension Array
	(T1.14) Vector
	(T1.15) Synchronized block
	(T1.16) Java ternary conditional
	(T1.17) Keyword "this"
(T2) Class-related	(T2.1) Inner class
	(T2.2) Anonymous class
	(T2.3) Cyclically dependent class
(T3) Annotations	(T3.1) Annotations
(T4) Code Comment	(T4.1) Comment related
(T5) Method-related	(T5.1) Overloaded method
	(T5.2) Static method
	(T5.3) Method reference
	(T5.4) Recursive method
	(T5.5) Default method
(T6) Static	(T6.1) Static initializer
	(T6.2) Static import
	(T6.3) Static field
(T7) Constructor-related	(T7.1) Super constructor
	(T7.2) Nested constructor
	(T7.3) Implicit constructor
(T8) Others	(T8.1) Special String
	(T8.2) Arithmetic expression
	(T8.3) Time-consuming method call
	(T8.4) Dead code block
	(T8.5) Method chaining

Select

Characteristic	Description
Lambda	Anonymous functions used to implement functional interfaces with a more streamlined syntax
Java generics	Java generics allow to create classes, interfaces, and methods that operate with unspecified types
Anonymous class	Class defined without a name, often used for one-time implementations of interfaces or abstract classes

Select error-prone input program characteristics from our study of refactoring engine bugs

Wang, H., Xu, Z., Zhang, H., Tsantalis, N., & Tan, S. H. (2024). An Empirical Study of Refactoring Engine Bugs. arXiv preprint arXiv:2409.14610.

How to mutate?

```java
public class A {
    public class BaseInner {}

    public class Outer {
        public int x = 0;
        public void foo(){};

        public class Inner extends BaseInner {
            void innerMethod() {
                System.out.println(Outer.this.x);
                Outer.this.foo();
            }
        }
    }
}
```

Seed input program from historical bug report

Characteristic	Description
Lambda	Anonymous functions used to implement functional interfaces with a more streamlined syntax
Java generics	Java generics allow to create classes, interfaces, and methods that operate with unspecified types
Anonymous class	Class defined without a name, often used for one-time implementations of interfaces or abstract classes

Error-prone input program characteristics

Leverage LLM to perform mutation

Now, I will give the definition of the current refactoring, you need to understand it. You need to make sure the original refactoring could still be applied on the variant.
1. **{Refactoring Type}**: **{Definition}**
2. To expose more bugs in the refactoring engines, please generate edge case variant considering the **{Characteristic}** in current refactoring scenario. You need to generate the variant according to the Input Program Structure Template, it is **{Template}**.
3. You should give me the variant, the program elements to be refactored, and the procedures to refactoring.
4. The generated variant should not contain any syntax errors. The Java program you generated should conformance with the JDK **{Version}** standard.
Please generate one edge case variant considering different edge usage scenarios of **{Characteristic}** based on the template. The variant format should be **{Format}**.

The prompt template used to perform mutation

Extract template

```java
public class A {
    public class BaseInner {}

    public class Outer {
        public int x = 0;
        public void foo(){};

        public class Inner extends BaseInner {
            void innerMethod() {
                System.out.println(Outer.this.x);
                Outer.this.foo();
            }
        }
    }
}
```

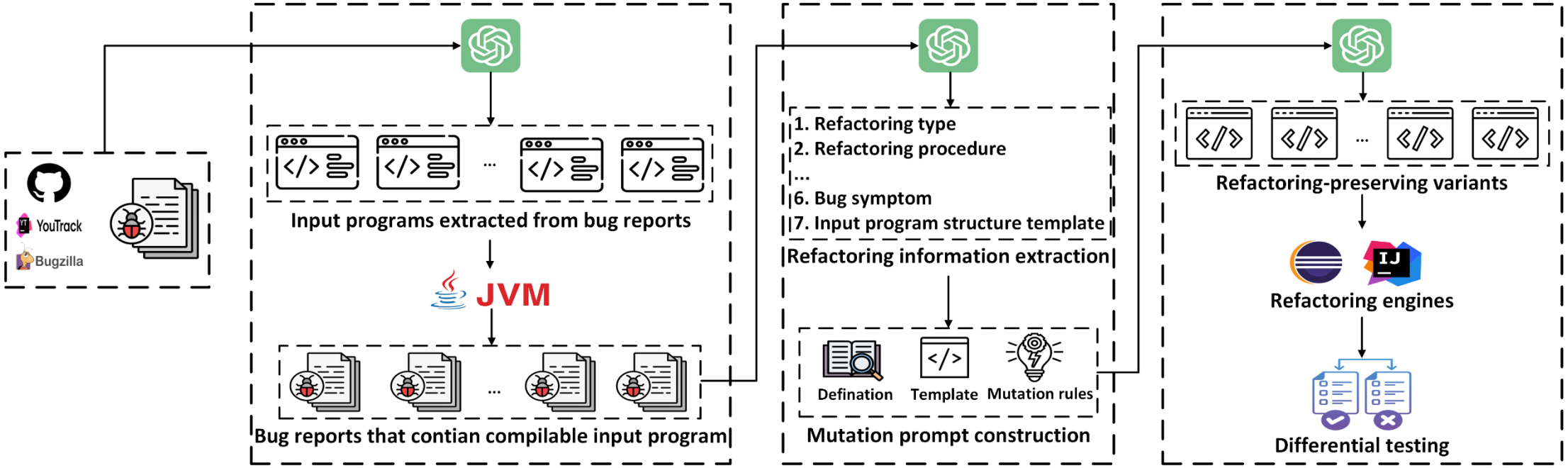
Extract

```java
public class OuterClass {
    public class BaseTargetClass {}
    public class OriginalClass {
        public DataType memberVariable;
        public void memberMethod();
        public class NestedOriginalClass extends BaseTargetClass {
            void methodToBePulledUp() {
                // Method logic that accesses OriginalClass's context
            }
        }
    }
}
```

✓ Intuitive

- Larger mutation space
- Reusable

RETESTER: Automated Refactoring Engine Testing



Overall workflow of RETESTER

- ✓ LLM for mutation
- ✓ Oracle: Different testing of refactoring engine

Experiment setup

ID	Source	Issue No.	Refactoring Type	Symptom
S-1	Eclipse	1533	Pull up	Compile error
S-2	Eclipse	1529	Inline method	Compile error
S-3	IDEA	142361	Extract variable	Compile error
S-4	IDEA	354116	Make static	Behavior change
S-5	IDEA	354122	Extract method	Compile error

Five refactoring types

Characteristic	Description
Lambda	Anonymous functions used to implement functional interfaces with a more streamlined syntax
Java generics	Java generics allow to create classes, interfaces, and methods that operate with unspecified types
Anonymous class	Class defined without a name, often used for one-time implementations of interfaces or abstract classes

Three characteristics

Refactoring	Template	ET (s)	TGV	MT (s)	CV	RPV	Oracles			Bugs	
							UC	WS	Diff.	EC	IDEA
Extract method	Y	6	30	87	27	27	1	0	0	1	0
	N	7	30	131	28	28	0	0	0	0	0
Inline method	Y	8	30	91	26	26	5	0	0	5	0
	N	7	30	81	20	20	3	0	0	3	0
Extract variable	Y	6	30	78	30	30	0	0	0	0	0
	N	6	30	73	25	25	0	0	0	0	0
Pull up	Y	7	30	105	20	20	8	1	0	7	2
	N	8	30	109	20	20	2	0	0	1	1
Make static	Y	10	30	104	25	22	0	0	2	0	0
	N	11	30	179	26	21	1	0	0	1	0
Average	–	7.6	30	103.8	25	24		–		–	
Total	–	76	300	1038	247	239	20	1	2	18 (15)	3

Template = Whether input program template is used during mutation, ET = Time taken in seconds to extract refactoring information, TGV = Total generated variants, MT = Mutation time for TGV in seconds, CV = Compilable variants, RPV = Refactoring-preserving variants; Oracles: UC = Uncompilable Oracle, WS = Warning Status Oracle, Diff. = Differential Oracle; EC = ECLIPSE, IDEA = INTELLIJ IDEA.

Variant generated by Lambda Operator

```java
public class A {
    public class BaseInner {}

    public class Outer {
        public int x = 0;
        public void foo(){};

        public class Inner extends BaseInner {
            void innerMethod() {
                System.out.println(Outer.this.x);
                Outer.this.foo();
            }
        }
    }
}
```

Seed input program

```java
public class OuterClass {
    public class BaseTargetClass {}
    public class OriginalClass {
        public DataType memberVariable;
        public void memberMethod();
        public class NestedOriginalClass extends BaseTargetClass {
            void methodToBePulledUp() {
                // Method logic that accesses OriginalClass's context
            }
        }
    }
}
```

Template

```java
public class A {
    public class BaseInner {
        void innerMethodLambda(Outer outer) {
            Runnable r = () -> {
                System.out.println(outer.x);
                outer.foo();
            };
            r.run();
        }
    }

    public class Outer {
        public int x = 0;
        public void foo(){};

        public class Inner extends BaseInner {
            void innerMethod() { // Pull this method up to class BaseInner
                innerMethodLambda(Outer.this);
            }
        }
    }
}
```

Variant generated by RETESTER

Eclipse-issue-1823 (Fixed)

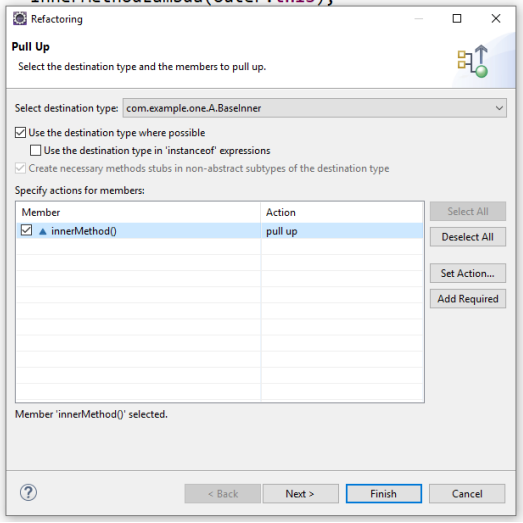
A.java

```java
package com.example.one;

public class A {
    public class BaseInner {
        void innerMethodLambda(Outer outer) {
            Runnable r = () -> {
                System.out.println(outer.x);
                outer.foo();
            };
            r.run();
        }
    }

    public class Outer {
        public int x = 0;
        public void foo(){};

        public class Inner extends BaseInner {
            void innerMethod() { // Pull this method up to class BaseInner
                innerMethodLambda(Outer.this);
            }
        }
    }
}
```



A.java

```java
package com.example.one;

import com.example.one.A.Outer.Inner;

public class A {
    public class BaseInner {
        void innerMethodLambda(Outer outer) {
            Runnable r = () -> {
                System.out.println(outer.x);
                outer.foo();
            };
            r.run();
        }

        void innerMethod(Inner inner) { // Pull this method up to class BaseInner
            innerMethodLambda(Outer.this);
        }
    }

    public class Outer {
        public int x = 0;
        public void foo(){};

        public class Inner extends BaseInner {
        }
    }
}
```

Syntax error.

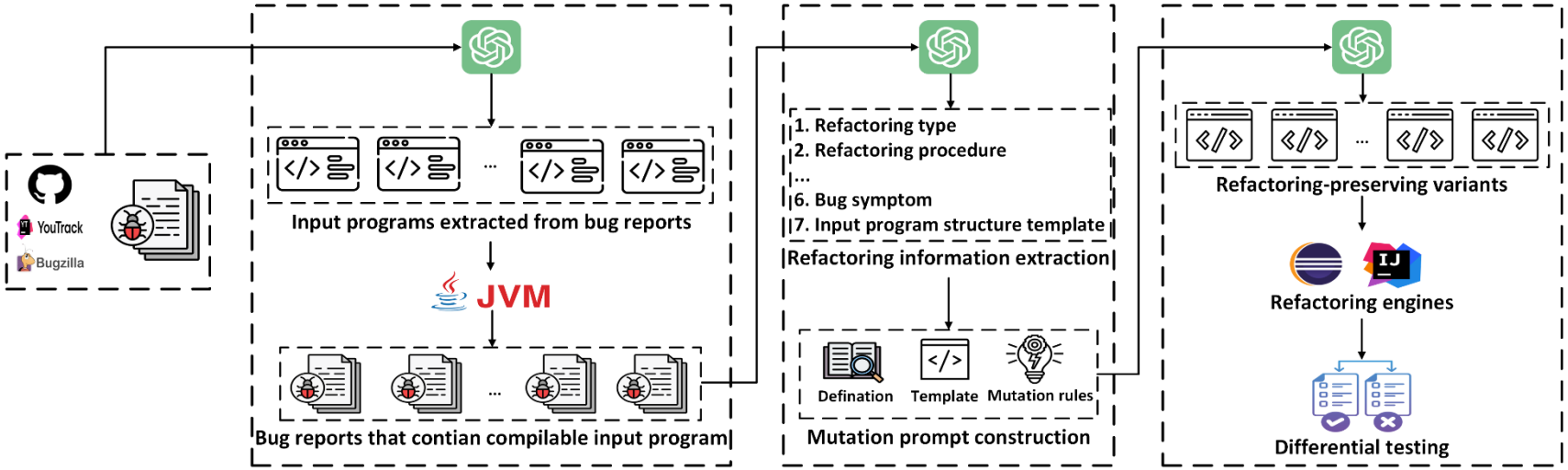
Bugs Detected by RETESTER

ID	IDE	Issue No.	Refactoring Type	Symptom	Status
B-1	Eclipse	1785	Extract Method	Compile error	Submitted
B-2	Eclipse	1824	Make Static	Compile error	**Confirmed**
B-3	Eclipse	1783	Inline Method	Compile error	Submitted
B-4	Eclipse	1781	Inline Method	Compile error	Submitted
B-5	Eclipse	1780	Inline Method	Compile error	**Fixed**
B-6	Eclipse	1779	Inline Method	Compile error	Submitted
B-7	Eclipse	1778	Inline Method	Compile error	Submitted
B-8	Eclipse	1777	Pull Up	Compile error	Submitted
B-9	Eclipse	1776	Pull Up	Compile error	Submitted
B-10	Eclipse	1775	Pull Up	Compile error	Submitted
B-11	Eclipse	1774	Pull Up	Failed refactoring	Submitted
B-12	Eclipse	1773	Pull Up	Compile error	**Fixed**
B-13	Eclipse	1772	Pull Up	Compile error	Submitted
B-14	Eclipse	1766	Pull Up	Compile error	Submitted
B-15	Eclipse	1823	Pull Up	Compile error	**Fixed**
B-16	IDEA	364110	Pull Members Up	Compile error	**Confirmed**
B-17	IDEA	362805	Pull Members Up	Compile error	**Confirmed**
B-18	IDEA	362804	Pull Members Up	Compile error	**Confirmed**

The issues of INTELLIJ IDEA, and ECLIPSE can be found at https://youtrack.jetbrains.com/issue/IDEA-XXX, and https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/XXX, where "XXX" can be replaced with the concrete numbers in **Issue No.**.

18 new bugs, 7 confirmed, 3 fixed.

Summary



Test refactoring engine via historical bug report driven LLM

18 new bugs, 7 confirmed, 3 fixed.

Haibo Wang
haibo.wang@mail.concordia.ca

Open-sourced repository

Automated Test Generation for Program Analyzer

Statfier: Automated Testing of Static Analyzers via Semantics-Preserving Program Transformations (FSE'23)

Characterizing & Detecting Program Representation Faults of Static Analysis Frameworks (ISSTA'24)

Understanding & Detecting Annotation-Induced Faults of Static Analyzers (FSE'24)

Understanding and Testing Semantically Equivalent Transformation

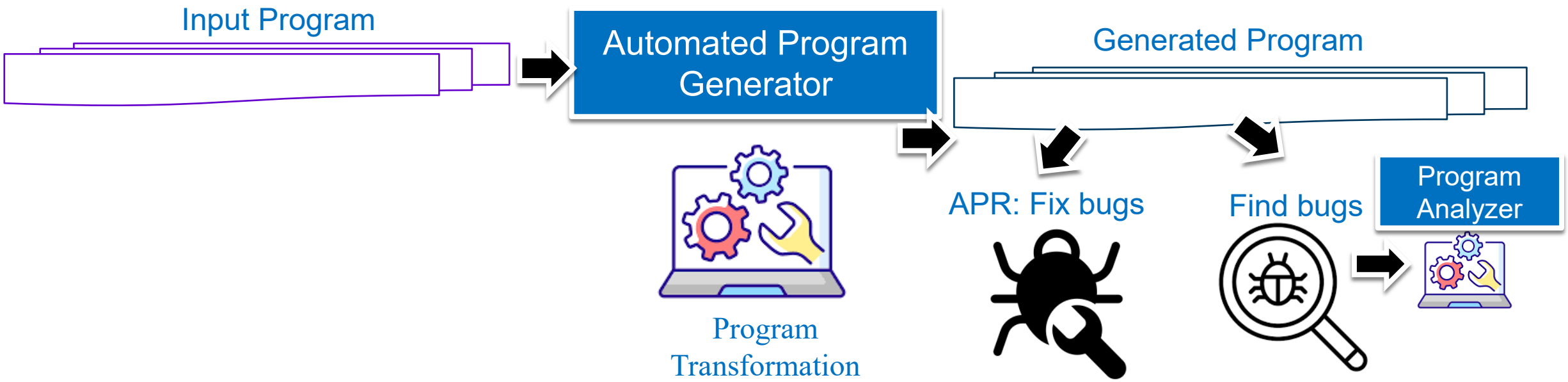
Towards Diverse Program Transformations for Program Simplification (FSE'25)

Testing Refactoring Engine via Historical Bug Report driven LLM (FORGE'25)

Towards Understanding Refactoring Engine Bugs (TOSEM'25 Under Review)

- ✓ Understanding and finding bugs in Program Analyzers
- ✓ Understanding and Automating Diverse Program Simplification
- ✓ Understanding and finding bugs in Syntactic Program Simplification (Refactoring Engine)

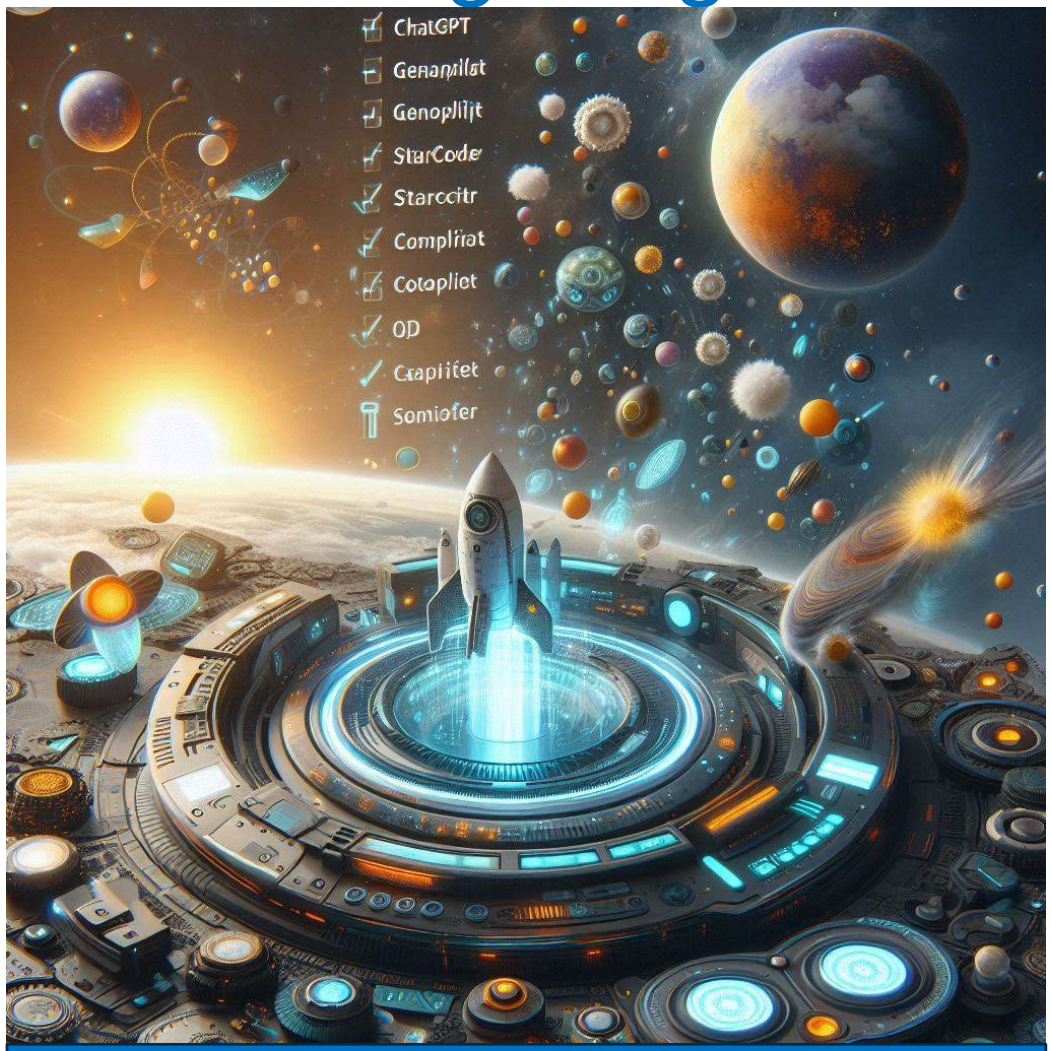
Broader View of Automated Program Generation



Questions to think about:

- What are other software maintenance tasks where you can use automated program generation?
- What kind of automated program generation techniques have/would you used?

Long Term Future Work: Generating Programs to Test Program Generation Tools



Generated by Bing Image Creator

Why restricted to Program Analyzer and Refactoring Engine?

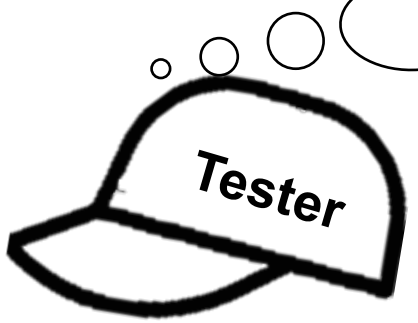
- Many program generation tools
  - Code Generation Models
  - Any program generation tools that take in programs
- Have you developed a new automated programming tool or a new APR?
  - Let me and my group test it!
  - Ensure reliability of program generation tools via test generation

Testing Genetic Improvement Engine

Genetic Improvement Engine

Input Program

What kind of input does Genetic Improvement Engine takes?



- Good News! Genetic Improvement Engine is also an Automated Program Generator!
- ✓ Similar technique can be used to generate semantically equivalent variants as inputs for testing genetic improvement engine!
  - ✓ Check if particularly semantically equivalent variant is easier to be improved?

Testing Genetic Improvement Engine

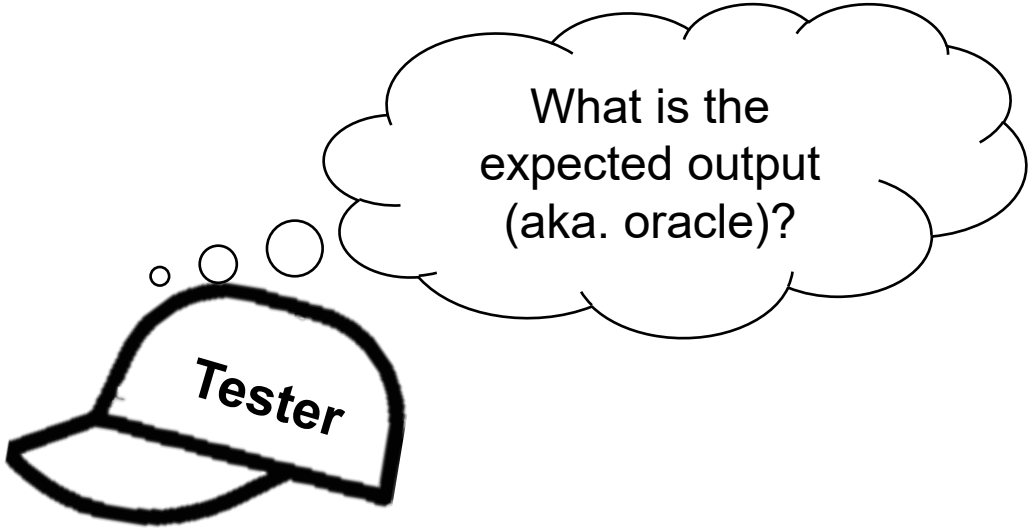
Genetic Improvement Engine

Input Program

Improved Output Program

- Not so Good News :(We may need to design new oracle as we need to check if the improved output program are the same for different semantically equivalent variants
- We may be able to use **differential testing** to compare the outputs of different GI engines?

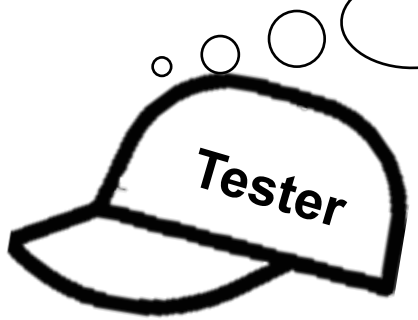
??? Any idea on this?



Tester Perspective: Coverage

Genetic Improvement Engine

What are the criteria to cover?



Software Quality Standard

- ✓ My talk mainly cover “Maintainability” and “Reliabilility”
- ✓ Most GI papers focus on “Performance Efficiency”
- ✓ How about improvement of other quality aspects?