Shaping Program Repair Space with Existing Patches and Similar Code

Jiajun Jiang[1], Yingfei Xiong[1], Hongyu Zhang[2], Qing Gao[1], Xiangqun Chen[1]
[1] *Peking University, China,* [2] *The University of Newcastle, Australia*

I. Introduction

Automated program repair (APR) has great potential to reduce bug-fixing effort and many approaches have been proposed in recent years. In this paper, we propose a novel automatic program repair approach, called ***SimFix***, that utilizes two kinds of data sources:

(1) **Similar code within faulty program.**
(2) **Exiting patches from other projects.**

where we (1) extract modifications to the faulty code via AST differencing with similar code and (2) further filter those modifications using frequent abstract modifications obtained from existing patches. We evaluate ***SimFix*** on the Defects4J benchmark, which results in:

- **34 bugs** were repaired (most number so far).
- **13 bugs** were never repaired by others.

II. Motivation

- **Pros of similar code:**
  Provide repair guidence with fixing ingredients.
- **Cons of existing patches:**
  Cannot cover all repairs under specific context.

```
// donor (similar) snippet
if(last!=null && last.getType()==Token.STRING){
   propName=last.getString();
   return (propName.equals(methodName));
}
// correct patch based on the similar donor snippet
+ if(target!=null && target.getType()==Token.STRING){
- if(target!=null){
   className=target.getString();
}
```

Figure 1: Correct patch for *Closure-57*.

- **Cons of similar code:**
  Cannot tell the likelihood of modifications.
- **Pros of existing patches:**
  Provide repair modification distribution.

```
// donor (similar) snippet
if(instant >= 0){
   return (int)(instant % DateTimeConst.MILLI_PER_DAY);
}
// incorrect patch with rare bug-fixing modification
if(instant<firstWeekMillis1){
-    return getWeeksInYear(year-1);
+    return (int)(instant%DateTimeConst.MILLI_PER_DAY);
}
```

Figure 2: Plausible patch of *Time-24*.

Our approach combines the merits of them.

III. Approach

Two Stage of *SimFix* :

- **Mining Stage** mines a set of frequent abstract modifications from open-source programs.
- **Repairing Stage** generages concrete modifications to faulty code via AST differencing with similar code, and take intersection with abstract modifications to rule out invalid modifications.

Mining Stage (Offline) :

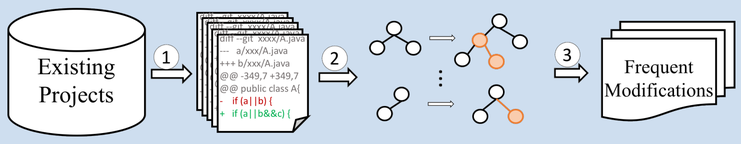


Figure 3: Mining frequent modifications.

- **The process of mining modifications:**

① **Extract repair history** from existing projects.
② **Compare changed files** before and after repairing with abstract syntax tree (AST) matching.
③ **Extract frequent modifications** using AST-diff.

- In total, we identified 16 kinds of frequent abstract repair modifications.
- Achieving a **102.5x reduction** of search space in theory and **2.0x speedup** in experiment.

Repair Stage with Running Example (Online) :

1. Extract Faulty Snippet.
- Locate faulty line of code
- Extract snippet surrounding it.

Faulty snippet
```
if(target!=null){
   className=target.getString();
}
```
⇩

2. Identify Similar Snippets.
- Search similar code by features:
  - How similar of structures
  - How similar of variable names
  - How similar of method names

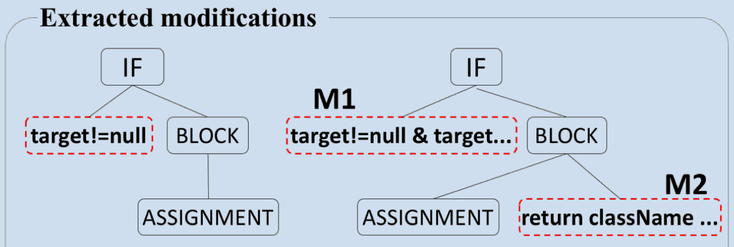
A similar donor snippet
```
if(last!=null && last.getType()==Token.STRING){
   propName=last.getString();
   return (propName.equals(methodName));
}
```
⇩

3. Rename Variables.
- Establish var-mapping by features:
  - Use structure similarity.
  - Type compatibility.
  - Variable name similarity.

Renamed donor snippet
```
if(target!=null && target.getType()==Token.STRING){
   className=target.getString();
   return (className.equals(functionName));
}
```
⇩

4. Extract Modifications.
- Match faulty and similar ASTs.
- Extract modifications based on node differences.
- Filter modifications with frequent abstract modifications.
- Combine different modifications.
- Rank modifications by heuristics.

Extracted modifications



M1 : replace `target!=null` with `target!=null&&target.getType()==Token.STRING`

M2 : insert `return (className.equals(functionName));`

M3 : combine **M1** and **M2**

⇩

5. Validate Patch Correctness.
- Run test suite.
- Manually check.

Patched code
```
if(target!=null && target.getType()==Token.STRING){
   className=target.getString();
}
```

IV. Evaluation

▷ Overall effectiveness of ***SimFix*** on Defects4J.

- **34 correct patches** (most number so far) and 22 incorrect patches, leading to a precison of 60.7% (Table 1).
- **13 bugs** repaired by ***SimFix*** were never repaired by other approaches. (Figure 4).

Proj.	SimFix	jGP	jKali	Nopol	ACS	HDR	ssFix	ELIXIR	JAID
Chart	**4**	0	0	1	2	-(2)	3	**4**	2(4)
Closure	**6**	0	0	0	0	-(7)	2	0	5(9)
Math	**14**	5	1	1	12	-(7)	10	12	1(7)
Lang	**9**	0	0	3	3	-(6)	5	8	1(5)
Time	1	0	0	0	1	-(1)	0	**2**	0(0)
Total	**34**	5	1	5	18	13(23)	20	26	9(25)

Table 1: Repair result comparison among approaches.

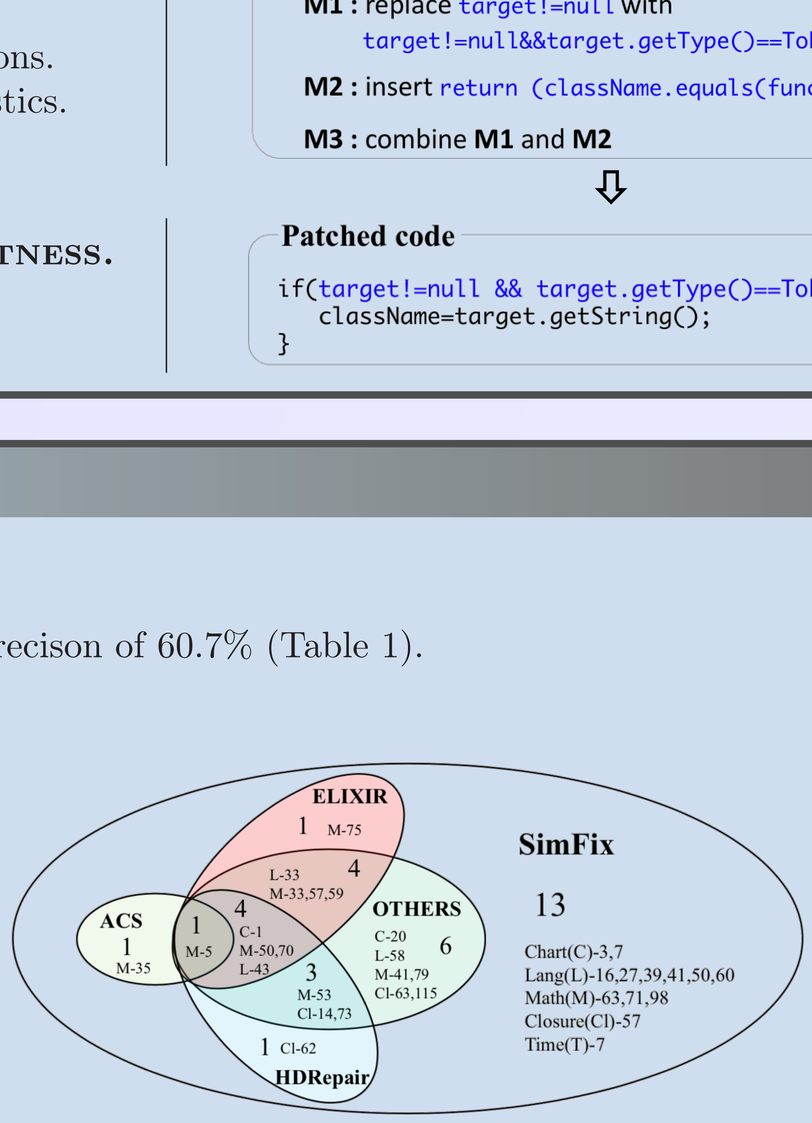


Figure 4: Overlaps of different techniques among the faults fixed by ***SimFix***.

▷ Effectiveness of other conponents.

- **Existing patches :** **12 less correct** and **14 more incorrect** patches without using existing patches.
- **Fine-grained code reuse :** **17 less bugs** would be repaired without fine-grained code reuse (AST node level).