Extending Prep: Program Repair Enhancement Via Preprocessing

Lillian Seebold, Computer Systems Engineering
Mentor: Dr. Forrest, Center Director and Regents Professor, Biodesign Center for Biocomputing, Security, and Society
School of Computing and Augmented Intelligence

Background and Motivation

- **Automated Program Repair (APR):** Automatically repair software vulnerabilities by modifying source code and validating against a known test suite (Codeflaws Dataset).
- **Program Repair Enhancement Via Reprocessing (PREP):** modify source code through static transformations that expose program features relevant to APR tools.

Search based-APR tools can fail to identify viable fixes due to mismatches between tool expectations and real code. This project replicates prior evaluations of the APR tool **Prophet** on the **Codeflaws dataset** with and without PREP and analyzes how prior and new transformations affect **repair quantity** and **alignment with human-written fixes.**

Research Questions

- *Do other PREP-like transformations exist that can improve Prophet's ability to repair bugs?*
- *If so, how do these new transformation(s) affect Prophet's ability to fix bugs and the quality of bug repair?*

Prior Work

Prep Transformations

Type	Example	
T1	`type name = value ->`	`type name` `name = value`
T2	`if(func(args)) ->`	`type tmp_var = func(args)` `if (tmp_var)`

T3	`func(arg1, arg2) ->`	`type tmp1 = arg1`
		`type tmp2 = arg2`
		`func(tmp1,tmp2)`

T4	`{type tmp1; tmp1 = (type) expr2; }`	

Figure 1: PREP Transformation Types and Examples

*Previous Results enabled **23** new, unique, and correct repairs for Prophet*

Methodology

1. Replicate Prior PREP experiments using **Prophet** and the **Codeflaws Dataset**
2. Evaluate how existing transformations influence Prophet's repair success and quality
3. Analyze Prophet's repair behavior by examining:
   - Code structure
   - Defect Localization
   - Schema Generation

*Bug Types of class **DCCR** (replace constant with variable/constant), **HIMS** (insert multiple non-branch statements) and **ORRN** (replace relational operator) are prevalent in the Codeflaws Dataset.*

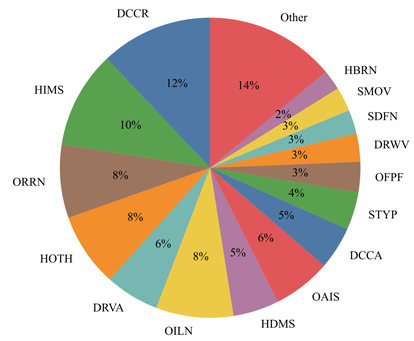

Figure 2: Codeflaws Composition by Defect Type

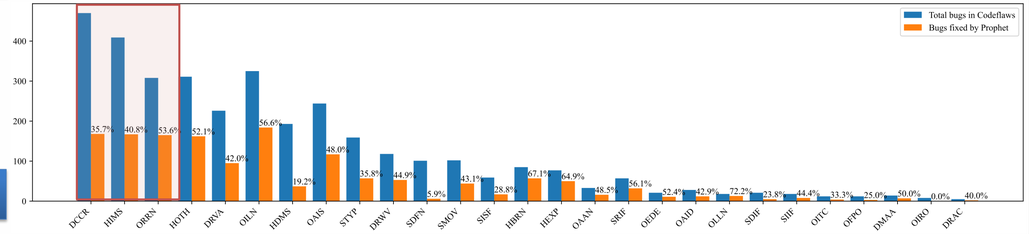

Figure 3: Number of Bugs in Codeflaws vs Repairs Generated by Prophet by Defect Class

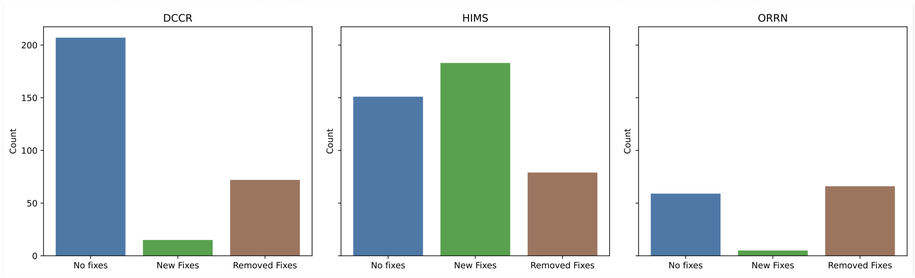

Figure 4: Prophet Results per Class Defect

*Bug Types of class **DCCR** present the most opportunity for Prophet to improve.*

Proposed Transformation for DCCR Defects

Prophet's Limitation: Only use available variables and constants from original program.

Arithmetic Expansion: Provide a larger search space to Prophet

Transformation Step	Definition / Explanation
Arithmetic Expansion	Given an expression e containing a number of constants c from the program constant set $\mathcal{C}$ and threshold values q, generate a set of candidate expressions $\mathcal{X}(e)$ where: $\mathcal{X}(e) = \{e,\ e \oplus c \mid \oplus \in \{+,-,\times,\div\},\ c \in \mathcal{C}\} \cup \{e-q,\ q-c\}$ This introduces new variables $x_i = f_i(e,q)$ representing arithmetic variations of e.
Candidate Enumeration	Replace each generated candidate with an assigned variable: $e \mapsto x_i, \quad x \in \mathcal{X}(e)$ Explicitly enumerate generated variables. For variables p, q in the original expression e.

```c
#include<stdio.h>
int main(int argc, char *argv[])
{
        int count=0,i,n,p,q;

        scanf("%d",&n);

        for(i=0;i<n;i++)
        {
            scanf("%d%d",&p,&q);
            if((p+2)>=q)
                count++;
        }

        printf("%d",count);

        return 0;
}
```

```c
#include<stdio.h>
int main(int argc, char *argv[])
{
        int count=0,i,n,p,q;

        scanf("%d",&n);

        for(i=0;i<n;i++)
        {
            scanf("%d%d",&p,&q);
            int  x = p+2
            int x1 = p - 2
            int x2 = p * 2
            int x3 = p / 2
            int x4 = p - q
            int x5 = p + q
            ...
            if((x)>=q)
```

*The human generated repair changes **(p + 2) ≥ q to (p-q) ≥ 2**. The sub statement **x1 = p - 2** in the transformed code allows Prophet to construct a logically equivalent solution: **(p-2) >= q***

Acknowledgements

Extending PREP project team: Dr. Pemma Reiter, Dr. Stephanie Forrest Sameera Shah, Krishna Priya Bheemavarapu
Programs: Fulton Undergraduate Research Initiative (FURI), Biocomputing Scholars Program (BSS), Barrett Honors Thesis

FURI

Center for Biocomputing, Security and Society

Ira A. Fulton Schools of Engineering
Arizona State University