Canonical Search And Repair

Andrew Hill[1], Kathryn Stolee[1], Corina Pasareanu[2]
[1]North Carolina State University, [2]Ames Research Center and Carnegie Mellon Silicon Valley

Problem

CSAR seeks to automatically repair buggy programs by leveraging code reuse. **Search for repair templates is accomplished via string metrics on canonical-form path conditions from symbolic execution**. For speed, a machine learning algorithm can be used to identify likely incorrect patches without running a test suite. Our learner achieved over 95% precision.

CSAR Algorithm

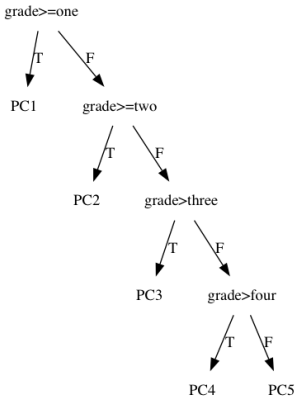
1) Localize fault, expand to buggy region
2) Run symbolic execution on buggy region and get path conditions (see symbolic execution tree)
3) Path conditions to canonical form a la Green
4) Search DB for matches based on string metrics between canonical representations (e.g. edit distance)
5) Use string metrics and a decision tree to remove likely incorrect patches
6) For remaining matches, transform source code to fit code under repair
7) Run unit tests
8) Verify correctness of patched code using heldout tests

Setup

For this work, we used a Java version of the IntroClass dataset, and the Java Symbolic Pathfinder (SPF) symbolic execution engine.

```
1  if (grade >= one){
2      return 'A';
3  }
4  else if (grade >= two){
5      return 'B';
6  }
7  else if (grade > three){
8      return 'C';
9  }
10 else if (grade > four){
11     return 'D';
12 }
13 return 'F';
```

Figure 1: Buggy Code



Symbolic execution of Buggy Code

```
1  if (grade >= one){
2      return 'A';
3  }
4  else if (grade >= two){
5      return 'B';
6  }
7  else if (grade >= three){
8      return 'C';
9  }
10 return 'D';
```

Figure 2: Patch 1 Source (d_{edit} = 29)

```
1  if(score >= aval)
2      return 'A';
3  else if (score >= bval)
4      return 'B';
5  else if (score >= cval)
6      return 'C';
7  else if (score >= dval)
8      return 'D';
9  else  return 'F';
```

Figure 3: Patch 2 Source (d_{edit} = 1)

Table 1: Number of bug patches generated (attempted) for the IntroClass benchmark. If no attempted value, value in Total column is assumed.

Program	SR	AE	GP	TAR	JFix	CSAR	Total
checksum	0	0	8	0	-	-	52
digits	0	17	30	19	-	-	204
grade	5(226)	2	2	2	-	73(75)	228
median	68(168)	58	108	93	-	49(148)	204
smallest	73(155)	71	120	119	16(47)	92(131)	163
syllables	4(109)	11	19	14	-	-	129

SR - SearchRepair, GP - GenProg, TAR - TrpAutoRepair

Table 2: Correct fixes for IntroClass, as a % of total attempts.

Program	SR	AE	GP	TAR	JFix	CSAR
checksum	0	0	.11	0	-	-
digits	0	.06	.11	.07	-	-
grade	.02	.01	.01	.01	-	.97
median	.40	.19	.37	.33	-	.33
smallest	.48	.44	.51	.53	.34	.70
syllables	.04	.06	.11	.08	-	-

Results

Table 1 gives our numbers of correct patches and the number of plausible patches for all other techniques. Table 2 gives the percentage of attempted patches which resulted in a successful patch.

We show an 8% overall improvement in number of correct patches over state of the art, even though we were not able to attempt three of the six programs due to limitations symbolic execution. Additionally, only CSAR can consistently repair **grade** bugs suggesting that at minimum CSAR is a powerful supplement for problem types which are difficult for existing methods.

Conclusion

- 8% improvement in correct patches over state of the art
- 47.6% improvement over previous reuse-based approaches
- Only technique which consistently repairs **grade**
- Use of metrics allows finer-grained control than unit tests

References

Păsăreanu, Corina S., et al. "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis." *Automated Software Engineering* 20.3 (2013): 391-425.

Visser, Willem, Jaco Geldenhuys, and Matthew B. Dwyer. "Green: reducing, reusing and recycling constraints in program analysis." *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012.