Trusted Software Repair for System Resiliency

Westley Weimer, Stephanie Forrest, Miryung Kim, Claire Le Goues, Patrick Hurley
For The Next 17 Minutes

- Program Repair: Resilient but Untrusted
  - Can we assess post-repair systems to gain trust?

- Assessment: Dynamic Execution Signals

- Assessment: Targeted Differential Testing

- Assessment: Invariants and Proofs
In This Talk

- **Dependability** measures how consistently a system successfully completes its mission.
- **Trust** refers to the human belief that the system is dependable.
  - Understanding is important than correctness when deciding what software to use (NASA)
- A **resilient** system can safely recover from or avoid errors, attacks or environmental challenges.
  - Possibly completing a **variant** of the mission.
Automated Program Repair

- Any of a family of techniques that **generate and validate** or **solve constraints to synthesize** program patches or run-time changes
  - Typical Input: program (source or binary), notion of correctness (passing and failing tests)

- Program repair provides **resiliency**
  - Powerful enough to repair serious issues like Heartbleed, format string, buffer overruns, etc.

- Efficient (dollars per fix via cloud computing)
Program Repair Quality

- GenProg '09

Automatically Finding Patches Using Genetic Programming

Westley Weimer  
University of Virginia  
weimer@virginia.edu

ThanhVu Nguyen  
University of New Mexico  
tnguyen@cs.unm.edu

Claire Le Goues  
University of Virginia  
legoues@virginia.edu

Stephanie Forrest  
University of New Mexico  
forrest@cs.unm.edu

Abstract

Automatic program repair has been a longstanding goal in software engineering, yet debugging remains a largely manual process. We introduce a fully automated method for locating and repairing bugs in software. The approach works on off-the-shelf legacy applications and does not require formal specifications, program annotations or special coding practices. Once a program fault is discovered, an extended form of genetic programming is used to evolve program variants until one is found that both retains required functionality and also avoids the defect in question. Standard test cases are used to exercise the fault and to encode program requirements. After a successful repair has been discovered, it is minimized using structural differencing algorithms and delta debugging. We describe the proposed method and report experimental results demonstrating that it can successfully repair ten different C programs totaling 65,000 lines in under 200 seconds, on average.

To alleviate this burden, we propose an automatic technique for repairing program defects. Our approach does not require difficult formal specifications, program annotations or special coding practices. Instead, it works on off-the-shelf legacy applications and readily-available test cases. We use genetic programming to evolve program variants until one is found that both retains required functionality and also avoids the defect in question. Our technique takes as input a program, a set of successful positive test cases that encode required program behavior, and a failing negative test case that demonstrates a defect.

Genetic programming (GP) is a computational method inspired by biological evolution, which discovers computer programs tailored to a particular task [19]. GP maintains a population of individual programs. Computational analogies of biological mutation and crossover produce program variants. Each variant’s suitability is evaluated using a user-defined fitness function, and successful variants are selected for continued evolution. GP has solved an impressive range of problems (e.g., the 4098-bit RSA challenge), but to our knowledge, it has not...
Program Repair Quality

- GenProg '09 - minimize
- Remove spurious insertions
Program Repair Quality

- GenProg '09 - minimize
- PAR '13 - human changes
  - Mutation operations based on historical human edits
Program Repair Quality

- GenProg '09 - minimize
- PAR '13 - human changes
- Monperrus '14 - PAR is wrong
  - Experimental methodology has several issues
  - Patch prettiness is not patch quality

Automatically Finding Patches Using Genetic Programming

Westley Weimer  
University of Virginia  
weimer@virginia.edu

ThanhVu Nguyen  
University of New Mexico  
tnguyen@cs.unm.edu

Claire Le Goues  
University of Virginia  
legoues@virginia.edu

Stephanie Forrest  
University of New Mexico  
forrest@cs.unm.edu

Automatic Patch Generation Learned from Human-Written Patches

Dongsun Kim, Jaecheong Nam, Jaewoo Song, and Sunghun Kim  
The Hong Kong University of Science and Technology, China  
darkuwj@cs.nchu.edu.tw, jonsong@hku.hk, sunkim@hku.hk


Martin Monperrus  
University of Lille & INRIA, France  
martin.monperrus@univ-lille1.fr

ABSTRACT

At ICSE'2013, there was the first session ever dedicated to automatic program repair. In this session, Kim et al. presented PAR, a novel template-based approach for fixing Java bugs. We strongly disagree with key points of this paper. Our critical review has two goals. First, we aim at explaining why we disagree with Kim and colleagues and why the reasons behind this disagreement are important for research on automatic software repair in general. Second, we aim at contributing to the field with a clarification of the essential ideas behind automatic software repair. In particular, we discuss the main evaluation criteria of automatic software repair: understandability, correctness, and completeness. We show that depending on how one sets up the repair scenario, the evaluation goals may be contradictory. Eventually, we discuss the nature of fix acceptability and its relation to the notion of software correctness.

The automatic detection of bugs has been a vast research field for decades, with a large spectrum of static and dynamic techniques. Active research on the automatic repair of bugs is more recent. A seminal line of research started in 2009 with the GenProg system [7, 15], and at the 2013 International Conference on Software Engineering, there was the first session ever dedicated to automatic program repair. The PAR system [9] was presented there, it is an approach for automatically fixing bugs of Java code. The repair problem statement is the same as GenProg [10] “given a test suite with at least one failing test, generate a patch that makes all test cases passing”. PAR introduces a new technique to fix bugs, based on templates. Each of PAR's ten repair templates represents a common way to fix a common kind of bug. For instance, a common bug is the access to a null pointer, and a common fix of this bug is to add a null pointer check just before the undesired access: this is template “Null Pointer Checker.”

We strongly disagree with Kim et al.’s paper on PAR. This is our motivation to present this critical review of their
Program Repair Quality

- GenProg '09 - minimize
- PAR '13 - human changes
- Monperrus '14 - PAR is wrong
- SPR '15 - condition synthesis
  - Solve constraints to synthesize expressions for conditionals
  - Not just deletions
Program Repair Quality

- GenProg '09 - minimize
- PAR '13 - human changes
- Monperrus '14 - PAR is wrong
- SPR '15 - condition synthesis
- Angelix '16 - SPR is wrong
  - SPR still deletes
  - Use semantics and synthesis

A recent study revealed that the majority of GenProg repairs avoid bugs simply by deleting functionality. We found that SPR, a state-of-the-art repair tool proposed in 2015, still deletes functionality in their many “plausible” repairs.
Resilient but Untrusted

- Program repair does provide **resiliency**
- But the “quality” of repairs is unclear
  - So they are not trusted
  - Thus far: algorithmic changes (e.g., mutation operators, condition synthesis, etc.)
- We propose a post hoc, repair agnostic approach to increasing operator trust
  - Provide multiple **modalities of evidence**
  - Approximate solutions to the **oracle problem**
Proposed Framework

• Augment repairs with three **assessments** that allow the human operator to trust in the post-repair dependable operation of the system
  • These assessments are aspects of the oracle problem for legacy systems
  • Each features a training or analysis phase in which a **model of correct behavior** (oracle) is constructed
Dynamic Execution Signals

- Insight: a program that produces unintended behavior for a given input often produces other observable inconsistent behavior
  - cf. printf debugging
- Measure binary execution signals
  - Number of instructions, number of branches, etc.
- In supervised learning, our models predict whether new program runs correspond to intended behavior 74-100% of the time (nsh)
Example: Zune Bug

- Microsoft Zune Player
- Infinite loop on last day of leap year (line ~8)
  - Branch counts, instruction counts, cache hits, etc., all differ

```c
void zunebug(int days) {
    int year = 1980;
    while (days > 365) {
        if (isLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
    }
    printf("current year is %d\n", year);
}
```
Targeted Differential Testing

- Code clones (intentional or not) are prevalent
- Repairs are often under-tested
  - They may insert new code, etc.
- Insight: We can adapt tests designed for code clones to become tests targeted at repairs
  - Identify variants, transplant code, propagate data
- Adapted tests in 17/17 Apache examples (nsh)
  - TarFileSet $\rightarrow$ ZipFileSet, ContainsSelector $\rightarrow$ FilenameSelector, etc.
Invariants and Proofs

- Insight: The post-repair system is not equivalent to the pre-repair system, but it may maintain the same invariants (or more).
- Identify invariants, prove them correct
  - No spurious or incorrect invariants remain
- We can infer 60% of the documented invariants necessary to prove functional correctness of AES (nsh)
  - Linear, nonlinear, disjunctive, and array invariants
Example: Zune Bug

- Ex. Invariants in Buggy Program
  - days_top > 365
- Ex. Correct Invariants
  - days_top > 365
  - days_bot < days_top
  - year_bot > year_top

```c
void zunebug(int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear(year)) {
      if (days > 366) {
        days -= 366;
        year += 1;
      }
    } else {
      days -= 365;
      year += 1;
    }
  }
  printf("current year is %d\n", year);
}
```
Evidence and Assessments

• Approximations to the Oracle Problem

• A post-repair system is correct when ...
  • It produces similar binary execution signals to previous known-good runs
  • It passes tests adapted from similar known-good methods
  • It provably maintains non-spurious invariants

• These can be assessed regardless of how the repair is produced
Summary

- Significant interest in trusted resilient systems
- Repair provides resilience but not trust
- We propose three modalities of evidence
  - Models of Execution Signals
  - Targeted Differential Testing
  - Proven Inferred Invariants
- These can provide an expanded assessment of trust in a resilient repaired system