Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results

Westley Weimer, UVA
Zachary P. Fry, UVA
Stephanie Forrest, UNM

Automated Program Repair

- “Given a program, a notion of correct behavior, and evidence of a defect, produce a patch that fixes the bug and retains behavior.”
- Rapidly growing subfield (~30 projects now)
  - AutoFix, ClearView, GenProg, FINCH, PACHIKA, PAR, SemFix, …
- Dominant cost: testing candidate repairs
- Reducing that cost:
  - Help fix easy bugs faster
  - Help fix hard bugs at all

State of the Art Woes

- GenProg uses test case results for guidance
  - But ~99% of candidates have identical test results
- Sampling tests improves GenProg performance
  - But GenProg cost models do not account for it
- Not all tests are equally important
  - But we could not learn a better weighting

Desired Solution

- Informative Cost Model
  - Captures observed behavior
- Efficient Algorithm
  - Exploits redundancy
- Theoretical Relationships
  - Explain potential successes

This Talk

- Informative Cost Model
  - Highlights “two searches”, “redundancy”
- Efficient Algorithm
  - Exploits cost model, “adaptive equality”
- Theoretical Relationships
  - Duality with mutation testing

Cost Model

- GenProg at a high level:
  - "Pick a fault-y spot in the program, insert a fix-y statement there."
  - Dominating factor: cost of running tests.
- Search space of repairs = |Fault| x |Fix|
  - |Fix| can depend on |Fault|
    - Can only insert "x=1" if "x" is in scope, etc.
- Each repair must be validated, however
  - Run against |Suite| test cases
    - |Suite| can depend on repair (impact analysis, etc.)

Cost Model Insights

- Suppose there are five candidate repairs.
  - Can stop when a valid repair is found.
  - Suppose three are invalid and two are valid:

$CR_1 \quad CR_2 \quad CR_3 \quad CR_4 \quad CR_5$

- The **order** of repair consideration matters.
  - Worst case: |Fault| x |Fix| x |Suite| x (4/5)
  - Best case: |Fault| x |Fix| x |Suite| x (1/5)
- Let |R-Order| represent this cost factor

Cost Model Insights (2)

- Suppose we have a candidate repair.
  - If it is valid, we must run all |Suite| tests.
  - If it is invalid, it fails at least one test.
  - Suppose there are four tests and it fails one:

$$T_1 \quad T_2 \quad T_3 \quad T_4$$

- The **order** of test consideration matters:
  - Best case: |Fault| x |Fix| x |Suite| x (1/4)
  - Worst case: |Fault| x |Fix| x |Suite| x (4/4)
- Let |T-Order| represent this cost factor.

Cost Model

|Fault| x |Fix| x |Suite| x |R-Order| x |T-Order|

- Fault localization
- Fix localization
- Size of validating test Suite
- Order (Strategy) for considering Repairs
- Order (Strategy) for considering Tests
  - Each factor depends on all previous factors.

Induced Algorithm

- The cost model induces a direct nested search algorithm:

For every repair, in order

  For every test, in order

    Run the repair on the test

    Stop inner loop early if a test fails

  Stop outer loop early if a repair validates

Induced Algorithm

- The cost model induces a direct nested search algorithm:

For every repair, in order

For every test, in order

Run the repair on the test

Stop inner loop early if a test fails

Stop outer loop early if a repair validates

Order can vary *adaptively* based on observations.

Algorithm: Can We Avoid Testing?

- If P1 and P2 are semantically equivalent they must have the same test case behavior.

Algorithm: Can We Avoid Testing?

- If P1 and P2 are semantically equivalent they must have the same test case behavior.
- Consider this insertion:

```
C=99;
```

Algorithm: Can We Avoid Testing?

- If P1 and P2 are semantically equivalent they must have the same test case behavior.
- Consider this insertion:

```
          A=1;
          B=2;
C=99;
          C=3;
          D=4;
          print A,B,C,D
```

Algorithm: Can We Avoid Testing?

- If P1 and P2 are semantically equivalent they must have the same test case behavior.
- Consider this insertion:

C=99;

```
A=1;
B=2;
C=3;
D=4;
print A,B,C,D
```

Algorithm: Can We Avoid Testing?

- If P1 and P2 are semantically equivalent they must have the same test case behavior.
- Consider this insertion:

```
C=99;
        A=1;
        B=2;
        C=3;
        D=4;
        print A,B,C,D
```

Formal Equality Idea

- Quotient the space of possible patches with respect to a conservative approximation of program equivalence
  - Conservative: P ≈ Q implies P is equivalent to Q
  - "Quotient" means "make equivalence classes"
- Only test one representative of each class
- Wins if computing P ≈ Q is cheaper than tests
  - Use known-cheap approximations
  - String equality, dead code, instruction scheduling

Adaptive Equality Algorithm

For every repair, ordered by observations

Adaptive Equality Algorithm

- **For every repair, ordered by observations**
  - **Skip repair if equivalent to older repair**

Adaptive Equality Algorithm

For every repair, ordered by observations

Skip repair if equivalent to older repair

For every test, ordered by observations

Adaptive Equality Algorithm

For every repair, ordered by observations
  Skip repair if equivalent to older repair

For every test, ordered by observations
  Run the repair on the test, update obs.

Adaptive Equality Algorithm

For every repair, ordered by observations

- Skip repair if equivalent to older repair

For every test, ordered by observations

- Run the repair on the test, update obs.
- Stop inner loop early if a test fails

Adaptive Equality Algorithm

For every repair, ordered by observations

  Skip repair if equivalent to older repair

  For every test, ordered by observations

    Run the repair on the test, update obs.

    Stop inner loop early if a test fails

  Stop outer loop early if a repair validates

Theoretical Relationship

- The generate-and-validate program repair problem is a dual of mutation testing
  - This suggests avenues for cross-fertilization and helps explain some of the successes and failures of program repair. (See paper for formal details.)

- Very informally:
  - PR Exists M in Mut. Forall T in Tests. M(T)
  - MT Forall M in Mut. Exists T in Tests. Not M(T)

Idealized Formulation

Ideally, mutation testing takes a program that **passes** its test suite and requires that **all** mutants based on human **mistakes** from the **entire** program that are not equivalent **fail** at least one test.

By contrast, program repair takes a program that **fails** its test suite and requires that **one** mutant based on human **repairs** from the fault **localization** only be found that **passes** all tests.

Idealized Formulation

Ideally, mutation testing takes a program that **passes** its test suite and requires that **all** mutants based on human **mistakes** from the **entire** program that are not equivalent **fail** at least one test.

By ... ...gram ... pas...

For mutation testing, the Equivalent Mutant Problem is an issue of *correctness* (or the adequacy score is not meaningful).

For program repair, it is purely an issue of *performance*.

Results and Conclusions

- Evaluated on 105 defects in 5 MLOC guarded by over 10,000 tests
- Adaptive Equality reduces GenProg's test case evaluations by **10x** and monetary cost by **3x**
  - Adaptive T-Order is within 6% of optimal
  - "GenProg – GP ≥ GenProg" ?
- Cost Model (expressive)
- Efficient Algorithm (adaptive equality)
- Theoretical Relationships (mutation testing)

More Duality with Mutation Testing

- Coupling Effect Hypothesis
  - MT: Tests that detect simple faults will detect complex faults
  - PR: Mutations that repair simple faults will repair complex faults
- Confidence
  - MT confidence increases with # of mutants
  - PR confidence increases with # of tests
- Small set of repair ops vs. Selective mutation
- Higher-order repairs vs. Higher-order mutation
- Multiple repairs per executable vs. Super-mutant / Schemata

Equivalent Mutant Problem

- Our proposal to use dataflow heuristics to find equivalent repairs is the dual of Baldwin & Sayward use of them for equivalent mutants
- Offutt and Craft found that six such compiler optimizations could find about 50% of equivalent mutants
  - We use a different set and find different efficiencies: dead code is critical (cf. 6%).
- Used in MT but not yet in PR: constraint solving, slicing, etc.