Automated Program Repair

CS454 AI-Based Software Engineering

Shin Yoo

Automated Program Repair

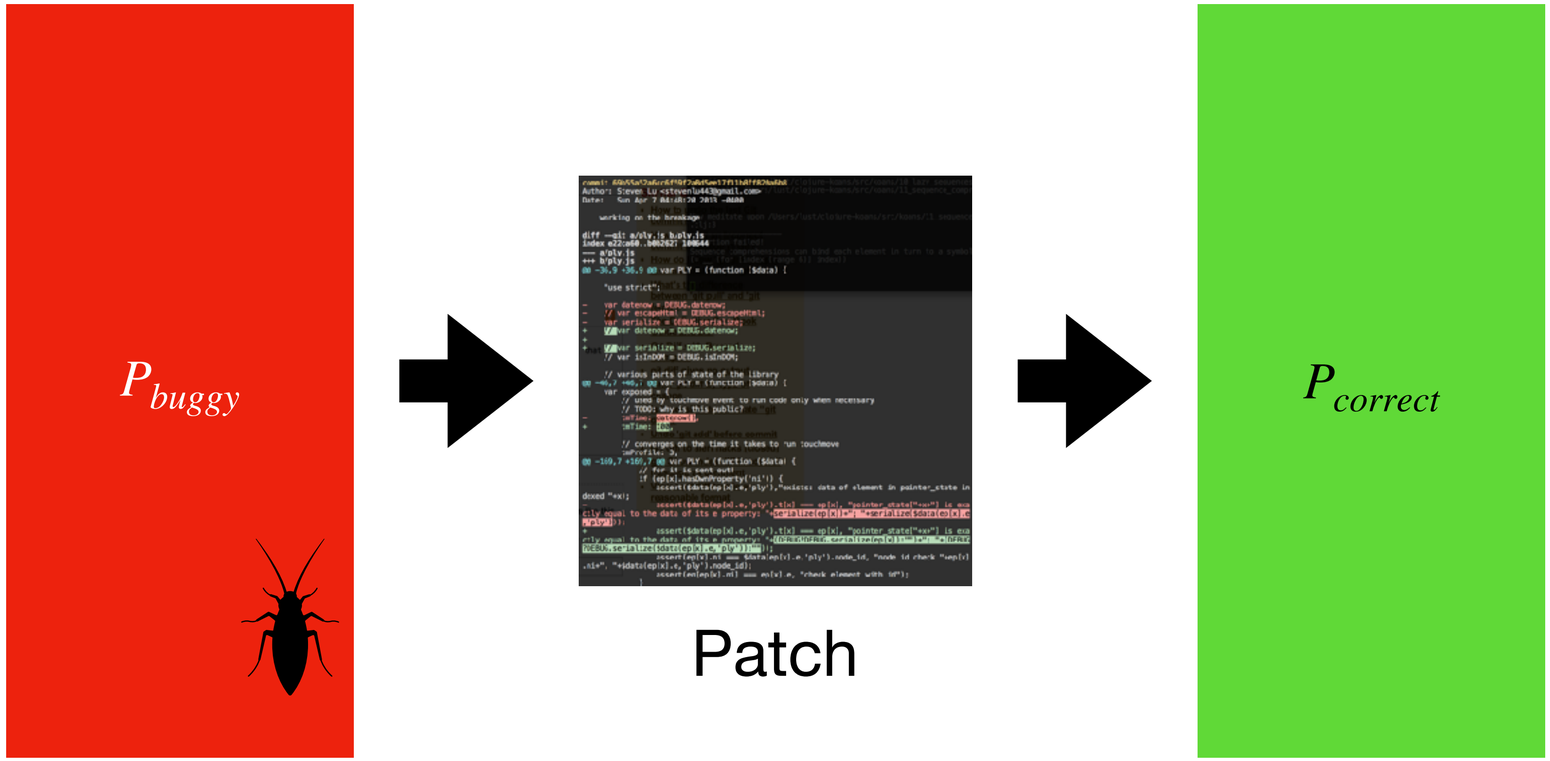
- Along with Program Synthesis, one of the holy grails in automation of software development!
  - Program Synthesis: given specifications, automatically write the program that satisfies them from the scratch (if we consider Input/Output pairs as low-level specification, GP is one way to do program synthesis)
  - Automated Program Repair: given specification and a faulty implementation, automatically fix the buggy program to be correct

Competent Programmer Hypothesis (CPH)

- Originally proposed to justify mutation testing:
  - Programmers write code that is *almost correct*.
  - Mutation Testing aims to evaluate how good your test cases are: it emulates real faults by injecting small syntactic faults (such as changing + to - in the source code) - CPH justifies the “small” part, i.e., real faults are also small.
  - APR is possible directly because CPH. That is, APR aims for small edits that deals with small mistakes or a minor corner cases, not complete rewrites or redesign (for now).

Program Repair formulated as search

What is the sequence of edits (patch) that will convert P_{buggy} to $P_{correct}$?



How to find the patch?

- Generate & Validate: create MANY candidate patches, and find one that works
  - Search-based APR: use GP to sample candidate patches
  - Template-based APR: use templates of existing fixes to sample candidate patches
- Semantic APR: use formal specification for program paths, and derive patches from them (typically using SMT solvers and symbolic execution)
- Neural APR: rely on neural models (including LLMs) to generate patches

Search-Based APR

- This is the one that started the whole APR sub-field.
- GenProg (Weimer et al., 2009) proposed two very important observations
  - GP can be done to an existing program (i.e., the buggy version)
  - For some bugs, the fix is already in the same codebase
- To begin with, we can represent the program as the AST (i.e., trees, the language that GP speaks naturally)

Performing GP to Existing Programs

- Problem: you have an existing version (a single entity) but you need diversity (multiple entities)
- Solution: define your mutation operator, and your initial population is exact copies of the original, existing program, but once mutated!
  - Initial population = 1-hop neighbours of the original program w.r.t. the mutation operator

“The fix comes from the program itself”

- Initially, GenProg authors gave intuitive answers here: things like missing null-checks can be clearly be fixed with code from the same program.
- Also, sometimes fix is deleting unnecessary stuff from the code.
- Hence, the original GenProg uses the following three low-level genetic operators:
  - Swap two statements
  - Insert a randomly chosen statement1 after statement2
  - Delete a statement
- Crossover is crossing these low level genetic operators along an execution path
- Mutation is applying one of these low level genetic operators randomly

So, how did it work?

- The initial version in 2009 was tested against 10 buggy programs, ranging from 22 to 21,553 Lines of Code, and produced 10 patches.
- The improved version in 2012 (Le Goues et al., ICSE 2012) fixed 55 out of 105 bugs in 8 programs ranging from 77K to 2.8M LoC: running on AWS, it cost GenProg $8 per bug.
  - The scale-up was possible because the program representation was now just patches, not entire ASTs. Since GenProg starts from the same buggy program, individuals can be patches, i.e., the diff between the individual and the original program!
  - When inserting code, choose from those that include in-scope variables only -> fewer compilation errors -> better search!

What is the fitness?

Quantitative measure of program correctness…?

- No such thing is real, so anything we do is a compromise.
- Test case based scoring: this is based on the fact that a correct patch would pass all tests. However, the inverse is not true: a patch that passes all tests may still not be correct!
  - This is known as the plausible patch problem: a patch is plausible if it does not break any tests - but does this mean that the patch is also correct?
  - Evaluating patch correctness requires human investigation…. consequently, it is hard and expensive.
  - The fitness landscape is discrete and entirely dependent on the quality of tests.

The Plastic Surgery Hypothesis

Where do the fix ingredients come? From the same codebase?

- If the GenProg assumption about the fix ingredients is true… then human written bug patches should also be based on the same codebase!
- Plastic Surgery Hypothesis: “Changes to a codebase contain snippets that *already exist in the codebase at the time* of the change, and these snippets can be *efficiently found* and exploited.” - Barr et al., FSE 2014
  - An analysis of 15,723 commits made to open source projects shows that 43% are graftable from the same version being changed!
- Later work tried to narrow down the fix space by looking at various similarity measures between the fault location and other parts.

Graftable

- …as in the graft, i.e., a twig inserted into a slit on other branch of a living plant, from which it receives sap

Search-based APR

- Theoretically speaking, given enough time, GP can find the patch (w.r.t. Plastic Surgery Hypothesis)
- But there is no guidance towards the fix location (similarities, scope analysis, and type checks can help filtering out, but perhaps no direct guidance)
- Without guidance on fix location, search-based APR becomes EXPENSIVE

What if we start from the ingredients?

But how do we know which ingredients we need?

Template Based APR

- Certain types of faults occur repetitively throughout the lifetime of the same project, as well as across different projects.
- Can we extract “templates” of these recurring faults? These are our ingredients!
  - Extraction is data mining of repositories (e.g., all commits that say “bug fix”) combined with abstraction (e.g., variable names do not have to match)
  - We first localize the buggy code; then apply all the fix templates that match!
  - We get multiple patches - now we need to validate them to find the one.

Fix Patterns of TBar (Liu et al., ISSTA 2019)

35 patterns in 15 categories

FP1. Insert Cast Checker. Inserting an *instanceof* check before one buggy statement if this statement contains at least one unchecked cast expression. **Implemented in:** PAR, Genesis, AVATAR, SOFix†, HDRepair†, SketchFix†, CapGen†, and SimFix†.

```
+ if (exp instanceof T) {
      var = (T) exp; ......
+ }
```

FP2. Insert Null Pointer Checker. Inserting a *null* check before a buggy statement if, in this statement, a field or an expression (of non-primitive data type) is accessed without a null pointer check. **Implemented in:** PAR, ELIXIR, NPEfix, Genesis, FixMiner, AVATAR, HDRepair†, SOFix†, SketchFix†, CapGen†, and SimFix†.

```
FP2.1:  + if (exp != null) {
            ...exp...; ......
        + }
FP2.2:  + if (exp == null) return DEFAULT_VALUE;
          ...exp...;
FP2.3:  + if (exp == null) exp = exp1;
          ...exp...;
FP2.4:  + if (exp == null) continue;
          ...exp...;
FP2.5:  + if (exp == null)
        +     throw new IllegalArgumentException(...);
          ...exp...;
```

FP3. Insert Range Checker. Inserting a range checker for the access of an array or collection if it is unchecked. **Implemented in:** PAR, ELIXIR, Genesis, SketchFix, AVATAR, SOFix† and SimFix†.

```
      + if (index < exp.length) {
            ...exp[index]...; ......
      + }
OR
      + if (index < exp.size()) {
            ...exp.get(index)...; ......
      + }
```

where *exp* is an expression representing an array or collection.

FP4. Insert Missed Statement. Inserting a missing statement before, or after, or surround a buggy statement. The statement is either an expression statement with a method invocation, or a *return/try-catch/if* statement. **Implemented in:** ELIXIR, HDRepair, SOFix, SketchFix, CapGen, FixMiner, and SimFix.

```
FP4.1:  + method(exp);
FP4.2:  + return DEFAULT_VALUE;
FP4.3:  + try {
            statement; ......
        + } catch (Exception e) {   ...   }
FP4.4:  + if (conditional_exp) {
            statement; ......
        + }
```

Template Based APR

- Pros
  - Eliminates the fix ingredient search problem
  - Can generate patches fast
  - Can control the scope of APR effectively
- Cons
  - Can generate a LOT of patches (remember templates still have holes, which results in combinatorial explosions), which increases the validation cost
- In some sense, Template Based APR can be considered as scoped exhaustive search.

Semantic APR

A motivating example from Angelix (Mechtaev et al., ICSE 2016)

```
1  - if (max_range_endpoint < eol_range_start)
2  -   max_range_endpoint = eol_range_start;
3
4  - printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
5  + if (max_range_endpoint)
6  +   printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(a) The developer-provided bug patch for coreutils bug 13627 where multiple locations are repaired

```
1  if (max_range_endpoint < eol_range_start)
2    max_range_endpoint = eol_range_start;
3
4  if (1)
5    printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(b) The buggy version after semantics-preserving transformation (the shaded part is added)

```
1  if (α)
2    max_range_endpoint = β;
3
4  if (γ)
5    printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(c) Suspicious expressions are replaced with symbolic variables

```
1  if (0)
2    max_range_endpoint = eol_range_start;
3
4  if (! (max_range_endpoint == 0))
5    printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(d) A repair generated from our repair algorithm; expressions in the shaded areas are synthesized from our repair tool, Angelix.

Figure 1: Motivating example

Neural Program Repair

- Deep Neural Network can generate or convert (=translate) sequences (=words, or a series of tokens).
- Can we translate buggy code into correct code using NMT (Neural Machine Translation)?
  - Similarly to template based APR, we need to collect actual fix commits: not to extract templates, but to learn what a fix commit typically does

SequenceR

Chen et al., IEEE TSE 2021

- Trained a sequence-to-sequence DNN model against 35,587 pairs of buggy and fixed code.
- Need to deal with OOV (Out-of-Vocabulary) problem, as source code is full of unique identifiers: a particular translation that was correct with one program may not be correct with another.
- Still, can generate interesting one-line patches

Call change Here a call to method **writeUTF** is replaced by a call to method **writeString**.

```
− out.writeUTF( failure );
+ out.writeString( failure );
```

Listing 10: Call change

4.6.4 Case study: change from field access to method call

A good practice of software engineering is to implement encapsulation by calling methods instead of directly accessing fields, this is handled by SEQUENCER as follows (`size` to `size()`)

```
− app.log( "PixmaPackerTest", ( "Number of textures: " + ( atlas.
      getTextures().size ) ) );
+ app.log( "PixmaPackerTest", ( "Number of textures: " + ( atlas.
      getTextures().size() ) ) );
```

Listing 16: change from field access to method call

NMT based Neural APR

- Pros
  - An end-to-end solution: buggy line goes in, fixed line comes out
  - Pipeline itself is language agnostic
- Cons
  - Needs well prepared training data
  - This is just template based APR, with "templates" in the training data learnt in the neural weights?

GPTx & chatGPT

- Original GPTx models are autocompletion engines in the form of (large) language model
- chatGPT has been put through RLHF (Reinforcement Learning from Human Feedback) for conversational finetuning

(Un)reasonable Effectiveness of LLMs

- They are fundamentally trained to be autocompletion engines (i.e., a model that predicts the probability distributions, but...
- Above certain size, they also exhibit interesting emergent behaviour, such as few-shot and zero-shot learning - not via training, but via prompting
  - Few-shot learning: you show the model a few examples of a problem instance, and the model learns to solve the problem
  - Zero-shot learning: you simply explain the problem, and the model learns to solve the problem

The Bleeding Edge

So many recent research outcomes...

- ChatRepair (Xia & Zhang, 2023): Outperforms all existing Neural APR techniques, as well as some template based techniques.
- AutoSD (Kang & Yoo, 2023): Employs a debugging guideline designed for human, and tells the LLM to adopt it - which it does surprisingly well.
- Pros
  - Very little technical barrier: you are essentially asking the model to repair something in natural language - with additional information in the form of text
- Cons
  - We do not really understand the repair logic, it is a huge black box
  - The usual risks of adopting LLMs: hallucination, verification of the results, etc...

Some auxiliary concerns about APR...

Patch Validation

- Increasingly, we are moving towards techniques that will produce multple patch candidates.
- Initial validation means we need to check for the plausiblity - which is costly, as we need to execute tests against patch candidates.
- Further, eventually we need to check for the correctness - which is often only possible with humans.
- If we cannot choose the correct patch automatically, at least we can try to rank them...?

Benchmarking

- How do we evaluate new and upcoming APR techniques? By comparing their performance against the same set of bugs.
- This led to huge popularity of some bug benchmarks, such as Defects4J, a collection of open source Java bugs.
- There is the danger of the community gradually overfitting to only a small number of known benchmarks, instead of being generally effective.
- LLMs make this even more complicated, because their training corpus is extremely large (i.e., they have seen everything)

Long Term Maintenance

- This was one of the very initial concerns back in 2009: if we start accepting machine generated patches, what happens later when someone does not understand why certain changes were made?
  - Very difficult to objectively measure
  - Depends heavily on what type of APR is adopted too: only trivial things (for example, deleting stuff so that the build passes) or serious patches?
  - We only have a few cases of industrial adoption, so this remains to be seen.
  - On the other hand, we have embraced GitHub CoPilot....?

Human Adoption

- The makers of automated technique tend to assume that end users will simply welcome the automation :)
- In reality, they may not trust any automated results, unless they feel they understand the underlying process.

Practical Adoptions

Meta

SapFix, Marginean et al., ICSE 2019

- Adopted APR Framework for six key Android apps, including Facebook, Messenger, Instagram
  - Focuses on NPE, which are in turn automatically found by Sapienz, a search-based automated GUI testing tool
  - The paper reports that, for the studied buggy versions, 48% of patches automatically generated by SapFix were approved to be correct by the developers.

Bloomberg

Kerbas et al., IEEE Software 2021

- Bloomburg, London, collaborated with UK academics to adopt APR for a production environment. Here are their emphasis:
  - Fix Novelty: APR does not really have to technically novel, as long as it brings value to practitioners
  - Fix Simplicity: academia focuses on finding increasingly more difficult bugs - but Bloomberg wanted to focus on trivial yet frequently recurring bugs as they waste more developer time overall
  - Fix Verification: instead of fully automated patch verification, process-based human checks are fine with practitioners

Going beyond “repair”…

Genetic Improvement (GI)

- APR “improves” programs by removing bugs. That is, it is an improvement with respect to the functional property of correctness.
- Can we improve other aspects of a given software?
  - Faster? Use less memory?
  - Faster for a specific class of inputs, i.e., specialization?

GP, but for non-functional properties

- Langdon & Harman, IEEE Transactions on Evolutionary Computation, 2013: optimized Bowtie2 (a widely used genome sequencing tool) to be 70 times faster on average.
  - Genome sequencing results are not binary: there are margins for acceptable error. GP exploits this to discard some computation that only affects a very small number of cases.
- Petke et al., EuroGP 2014: optimizes a SAT solver to be more efficient for a particular class of problems, and achieves 17% improvement, which matches the manual optimization done by human experts
  - Here, GI is performed as selective transplantation of code from two donors: one is the winner of MiniSAT competition in 2009, the other is the participant of the same competition that performed the best for the given problem instance.

Summary

- Automatically fixing bugs is within the reach of reality: we are seeing the first wave of industry adoptions, and LLMs will only accelerate this.
- Many different approaches exist in order to explore the space of possible “variants” of the buggy program; all focus on different trade-off points between accuracy and cost.
- Important take-away message: human written code is not always correct, not always immutable; applying automated changes to code can sometimes result in productive outcome :)