272: Software Engineering
Fall 2018

Instructor: Tevfik Bultan

Lecture 17: Automated Patch Generation

Motivation

- Fixing bugs is a difficult and time consuming process

- Some reports state that software maintenance accounts for 90% of the total cost of a typical software project

- One Mozilla developer in 2005 stated: “everyday, almost 300 bugs appear … far too much for only the Mozilla programmers to handle”

- If patches can be automatically generated, cost of fixing bugs, hence, the cost software maintenance can be reduced

Automated Patch Generation

- We will discuss two approaches to automated patch generation
- Both of them take some input that characterizes the expected behavior for the patch
- Then then automatically infer the required patch from this input
- By using automatically generated patches instead of manually writing patches the cost of software maintenance can be reduced

Patch Generation with Genetic Programming

- For this approach the input is
  - A program that has bug
  - A set of negative test cases for which the program fails
  - A set of positive test cases for which the program works correctly

- The goal is to modify the input program (i.e, patch the input program) so that all the test cases are handled correctly

- The basic idea is to use genetic programming to obtain the patch

How to change the program?

- The basic idea is to
  - favor changing program locations visited when executing the negative test cases (failed tests), and
  - avoid changing program locations visited when executing the positive test cases

- The program is changed by inserting, deleting and swapping program statements and control flow
  - The insertions are based on the existing program structure

- The primary repair is the first variant that passes all positive and negative test cases
  - This is followed by a phase that minimizes the differences between the patch and the original input program

Program Representation

- Uses an Abstract Syntax Tree (AST) representation
  - from the CIL (C Intermediate Language) toolkit

- In CIL all statements are mapped to
  - Instr
  - Return
  - If
  - Loop

  which include all assignments, function calls, conditionals, and looping constructs (goto is not included)

Program Representation

- Defines a notion called “weighted path”
- A weighted path is a set of <statement, weight> pairs that guide the search for the patch
- A weighted path is a path visited during a failed test
- Weights are assigned as follows:
  - A statement visited only on negative paths would have a high weight (1.0)
  - If a statement is also visited on a positive path then its weight is low (0.1, 0.0)

Genetic Programming

- Genetic programming is a general purpose *randomized* optimization technique that is inspired by evolution
- First represent a subset of the search space of an optimization problem as a *population*
  – Each member of the population corresponds to a solution
- Define a *fitness function* that maps each member of the population to a number indicating how good the corresponding solution is
  – The higher the fitness value, the better the solution
- Generate new solutions from the existing population by
  – *Mutation:* Pick a member of the existing population and mutate it using a mutation operator
  – *Crossover:* Pick two members of the existing population and create a new member that is the descendant of the two by combining the two members
- *Natural selection:* Determine which members of the population survive based on their fitness value

Population

- In genetic programming for patch generation the population consists of program represented as pairs of

  <AST, weighted path>

- Each member of the population is called a variant

Mutation

- To mutate a variant V = <AST, wp>, choose a statement S from wp biased by the weights, choose one of the following:
  - delete S
  - replace S with S1
    - Choose S1 from the entire AST
  - insert S2 after S
    - Choose S2 from the entire AST

- Inserted statements are chosen among the existing statements in the program
  - Assumption: program contains the seeds of its own repair (e.g., has another null check elsewhere)

Crossover

- Only the statements along the weighted paths are crossed over
- Choose a cutoff point along the paths and swap all the statements after the cutoff point
  - For example on input [P1, P2, P3, P4] and [Q1, Q2, Q3, Q4] with cutoff 2 the child variant is [P1, P2, Q3, Q4]
  - Child combines traits from both parents
- In this approach, crossover operator is always takes place between an individual from the current population and the original parent program
  - This is called *crossing back*

Fitness and Selection

- Compile a variant
  - If it fails to compile: Fitness = 0
- Run the variant on the test cases
  - Fitness = number of test cases passed
  - Use weights so that passing the bug test case is worth more

$$\text{Fitness}(P) = \text{WposT} \times |\{t \in \text{PosT} \mid P \text{ passes } t\}| + \text{WnegT} \times |\{t \in \text{NegT} \mid P \text{ passes } t\}|$$

- Higher fitness variants are retained and combined into the next generation

Some observations about genetic programming

- It is a randomized search
  - Some decisions are made by flipping a coin
    - Which members of the population to choose for mutation or cross over
    - What mutation operation to apply
- So each execution of the algorithm can produce a different results
  - One can run the algorithms multiple times (trials)

- There are bunch of parameters that have to be determined and that will affect the performance of the algorithm, such as
  - Population size
  - Weights for the fitness function (WposT, WnegT)
  - Probability of choosing a statement for mutation (Wpath)
  - Probability of applying a mutation (Wmut)

Repeat until a patch is found, then minimize

- Continue the variant generation using genetic programming until a variant that passes all the tests is found
  - This is the primary repair
- Due to the randomized search (i.e., the randomness in the Mutation and crossover operations), the primary repair may contain irrelevant changes
- We want to find a patch that repairs the defect by making minimal modifications to the original program

Minimize the patch via delta debugging

- Generate a patch from the primary repair by taking the difference between the original program and the primary repair

- Use delta-debugging the discard every part of the patch that can be eliminated while still passing all test cases

- Delta-debugging finds 1-minimal subset of the initial patch in $O(n^2)$ time
  - 1-minimal: removing any single line in the patch causes a test to fail

A different maintenance task

- What we talked about so far was the following scenario:
  - We have a program for which a test case fails (i.e., there is a bug) and we want to find a patch that fixes the bug
- Some maintenance tasks do not fit this scenario
  - There might be a change in a library API and we might want to change all the calls to that library to fit the new API
  - We might already have some examples of these changes
    - A part of the code is updated based on this new API
- Question: Is it possible to automate the modifications required for the given change, so that all the code based can be updated to be consistent with this require change
  - We want to automatically infer a generic patch
  - When we apply this generic patch to the whole code base, all the required changes should be done
    - And nothing more than what is required should be changed

Generic Patch Inference

Basic Problem:

- Input: A set of concrete changes to code
- Output: A generic patch that generalizes the given concrete changes

When the generic patch is applied to a program all the changes that are implied by the given concrete changes will be done

- A generic patch characterizes a change as a pattern
  - When the pattern matches, the change is applied

- The patterns should be general (they should not just enumerate all possible concrete changes)
- The pattern should not change things that are not related to the given concrete changes

Example

```
static int ax25_rx_fragment(ax25_cb *ax25,
struct sk_buff *skb)
{
  struct sk_buff *skbn, *skbo;
  if (ax25->fragno != 0) {
  ...
  /* Copy data from the fragments */
while ((skbo = skb_dequeue(&ax25->frag_queue)) != NULL) {
- memcpy(skb_put(skbn, skbo->len), skbo->data,
-        skbo->len);
+ skb_copy_from_linear_data(skbo,
+       skb_put(skbn, skbo->len), skbo->len);
  kfree_skb(skbo);
}
...
}
static int ax25_rcv(struct sk_buff *skb, ...)
{
  ...
  if (dp.ndigi == 0) { kfree(ax25->digipeat); ax25->digipeat = NULL;
} else {
  /* Reverse the source SABM's path */
  ! memcpy(ax25->digipeat, &reverse_dp,
  !        sizeof(ax25_digi));
}
...
}
```

Example

```
static struct sk_buff *dnrmg_build_message(
              struct sk_buff *rt_skb,
              int *errp)
{
  struct sk_buff *skb = NULL;
  ...
  if (!skb)
    goto nlmsg_failure;
  ...
  - memcpy(ptr, rt_skb->data, rt_skb->len);
  + skb_copy_from_linear_data(rt_skb, ptr, rt_skb->len);
  ...
nlmsg_failure:
  if (skb)
    kfree_skb(skb);
...
}
```

Example Changes

```
- memcpy(skb_put(skbn, skbo->len), skbo->data,
-        skbo->len);
+ skb_copy_from_linear_data(
+      skbo,
+      skb_put(skbn, skbo->len),
+      skbo->len);

- memcpy(ptr, rt_skb->data, rt_skb->len);
+ skb_copy_from_linear_data(rt_skb, ptr, rt_skb->len);
```

Changes

- These changes can be characterized with the following rules:
  1. All calls to memcpy where the second argument is a reference to the field data, are changed into calls to skb_copy_from_linear_data.
  2. The first argument becomes the second
  3. The field reference to data in the second argument is dropped. The resulting expressions, which has type struct sk_bugg *, becomes the first argument of the new function call.
  4. The third argument of memcpy is copied as-is to the third argument of the new function call.

- Or, these changes can be represented with the following generic patch:

```
- memcpy(X0, X1->data, X2);
+ skb_copy_from_linear_data(X1, X0, X2);
```

Generic Patches

- Generic patches specify changes as patterns using meta-variables
  - Meta-variables are nonterminal symbols that can be substituted by terms in order to match to a program expression

- So, the task is to infer a generic patch that generalizes a set of concrete changes

- After the pattern for the generic patch is found, it can be automatically applied to the all parts of the code to propagate the change

Required Properties of Generic Patches

- *Compactness*
  - The inferred patch should be a compact representation of the required generic change.
  - It should not enumerate all given concrete changes in the input. This would not generalize to other parts of the code.
  - The goal is to generate a compact representation of the changes that generalizes the change.

- *Safety*
  - Only things that actually changed in the input should be changed by the inferred generic patch.
  - The parts of the code that are not related to the input change should not be affected.

Generic patches

- Generic patches can be either term-replacement patches
  - Replacing a source term (which may contain meta-variables) with target term (which may also contain meta-variables)
- A term-replacement patch is applied when the source term can be matched to a part of the input program by substituting program expressions for the meta-variables
  - When source term is matched, it is replaced with the target term (where meta-variables are replaced with the corresponding the program expressions)
- A generic patch is a sequence of term-replacement patches

Safe Patches

- Assume that the input change is given as a pair of source and target concrete terms

- A patch is safe with respect to a concrete change if the parts of the program that are modified by the patch do not have to be modified again to reach the target term
  - So, a safe patch will not modify the same part of the program more than once

- Given a set of changes as pairs of source and target terms
  - A common safe patch is a patch that is safe with respect to all the changes

Patch Ordering

- Patches can be ordered in a way that corresponds to compactness
  - If a patch p1 contains the transformations specified in another patch p2, then p2 is a sub-patch of p1.
  - This defines an ordering of the patches

- The patches that are bigger based on the above ordering are more compact
  - The intuition is that a patch generalizes the changes defined by all its sub-patches

- Given a set of concrete changes, the Largest Common Sub-patch is a patch
  - that is safe with respect to all the input concrete changes, and
  - it is bigger than any other patch that is safe with respect to all the input concrete changes

Example

```
t1 =
void foo(void) {
  int x;
  f(117);
  x = g(117);
  return x;
}

t1' =
int foo(void) {
  int x;
  f(117,GFP);
  x = h(g(117));
  return x+x;
}
```

```
t2 =
void bar(int y) {
  int a;
  a = f(11)+g(y);
  return a;
}

t2' =
void bar(int y) {
  int a;
  a = f(11,GFP)+g(y);
  return a+a;
}
```

- Concrete changes as source and target pairs: { (t1, t1'), (t2, t2')}

Example

Updates applied to t1:

```
- f(117)
+ f(117, GFP)

- g(117)
+ h(g(117))

- return x;
+ return x+x;
```

Updates applied to t2:

```
- f(11)
+ f(11,GFP)

- return a;
+ return a+a;
```

- Two Largest Common Patches:

  patch 1: f(X) □-> f(X,GFP); return Y □-> return Y+Y,

  patch 2: return Y □-> return Y+Y; f(X) □-> f(X,GFP)

although the sequential ordering is different, these are equivalent patches

SPFIND

- Spfind algorithm finds the largest common sub-patch for a given set of concrete changes

- It follows the following steps:
  - For each pair of terms in the set of concrete changes, it constructs all possible term-replacement patches
  - It computes the intersection of all of the generated sets of term replacement patches for every pair of terms
  - It combines the term-replacement patches in this intersection into larger sequential patches to obtain largest common subpatches for the given set of term pairs

- The authors implemented this algorithm in a tool called spdiff and applied it to LINUX patches

Results

- In application to Linux patches the spdiff find compact patches that characterize patches that were implemented by the developers
- In provides a very compact description of the modifications
- In some cases it finds cases where the required change was not propagated appropriately
  - Some files were missed