Automated Program Repair
Using Formal Verification Techniques

Orna Grumberg
Technion, Israel

Online Worldwide Seminar on Logic and Semantics (OWLS)
August 26, 2020
Model Checking

- Given a system and a specification, does the system satisfy the specification
Formal automated program repair

• Model checking finds bugs in the program
  • Bug: A program run that violates the specification

• Repair tool automatically suggests repair(s)
  • Repair: Changes to the program code, resulting in a correct program
We present two approaches

• To exploit formal verification techniques for program repair

  • Must Fault Localization for Program Repair

  • Assume, Guarantee or Repair
Must Fault Localization for Program Repair

Joint work with Bat-Chen Rothenberg

CAV 2020
Automated Program Repair

Buggy Program

Automated Program Repair

Patched Program That is Correct
Fault Localization

A buggy program with violating run
Fault Localization

Fault location set
Fault Localization

Repair
Fault localization

- Fault localization should focus the programmer’s attention on locations that are relevant for the bug.

- **Bad fault localization:**
  - Too restrictive might miss a potential repair
  - Too permissive will cause an extra search work
Fault localization

• Often - fault localization algorithms return a set of locations that may be relevant
  • No guarantee that all returned locations are relevant
  • Nor that every relevant location is returned

• We suggest a novel notion of must fault localization
Repair scheme

An important notion:

- **Repair scheme:**
  Identifies the changes to program statements, allowed by repair
Repair scheme example

- Repair scheme $S_{\text{mut}}$
  - Allows syntactic replacement of operators on the right-hand-side of assignments and in conditions

- For example,
  - $+$ → $-$
  - $>$ → $<$
  - $c$ → $c+1$
Must fault localization

- **Must fault localization algorithm:**
  returns a must location set
  - for every buggy program and every bug

- **Must location set:**
  Contains at least one location from any successful repair for the bug
  \[ \Rightarrow \text{It is impossible to fix the bug using only locations outside this set} \]
  \[ \Rightarrow \text{Any repair for the bug must use at least one location from this set} \]
Must and Repair scheme

• Must notions depend on the chosen repair scheme

• A location set might be must for one repair scheme and non-must for another
In this work

• We develop a fault localization algorithm
• Prove that it is must with respect to $S_{\text{mut}}$
• Implement it within the repair tool AllRepair
• Show significant speedups
Our setting:
Formal Automated Program Repair

- **Formal specification**: program should meet the specification for all inputs
- pass *(bounded)* formal verification
Our setting: 
Search-Based Program Repair

Generate and Validate
Algorithm for must fault localization

• By example
Example: Buggy program

proc. foo(x, w)
1. t:= 0
2. y:= x-3
3. z:= x+3
4. if (w>3) then
5. t:= z+w
6. assert (t<x)
7. y:= y+10
8. assert (y>z)
Example: buggy program with buggy run

proc. foo(x, w)
1. t:= 0
2. y:= x-3
3. z:= x+3
4. if (w>3) then
5. t:= z+w
6. assert (t<x)
7. y:= y+10
8. assert (y>z)

I = x←0, w←0

l = x←0, w←0

t ← 0

y ← -3

z ← 3

¬(0 > 3)

¬(-3 > 3) assertion violation for l
Example: Program formula (SSA)

proc. foo(x, w)
1. t := 0
2. y := x - 3
3. z := x + 3
4. if (w > 3) then
5. t := z + w
6. assert (t < x)
7. y := y + 10
8. assert (y > z)

ϕ_{foo} = \{
  t_0 = 0
  y_0 = x_0 - 3
  z_0 = x_0 + 3
  g_0 = w_0 > 3
  t_1 = z_0 + w_0
  y_1 = y_0 + 10
  t_2 = (g_0 ? t_1 : t_0 )
  y_2 = (g_0 ? y_1 : y_0 )
  \neg (y_2 > z_0 ) \lor \neg (g_0 \rightarrow t_1 < x_0 )
\}
Example: Program formula (SSA) with satisfying assignment

\[ \varphi_{foo} = \{ \]
\[ t_0 = 0 \]
\[ y_0 = x_0 - 3 \]
\[ z_0 = x_0 + 3 \]
\[ g_0 = w_0 > 3 \]
\[ t_1 = z_0 + w_0 \]
\[ y_1 = y_0 + 10 \]
\[ t_2 = (g_0? t_1 : t_0) \]
\[ y_2 = (g_0? y_1 : y_0) \]
\[ \neg (y_2 > z_0) \lor \neg (g_0 \rightarrow t_1 < x_0) \]
\} \]

I = x_0 ← 0, w_0 ← 0
\[ t_0 ← 0 \]
\[ y_0 ← -3 \]
\[ z_0 ← 3 \]
\[ g_0 ← (0 > 3) = false \]

\[
... \]

\[ y_2 ← -3 \]
\[ \neg(-3 > 3) \text{ assertion violation for } I \]
Computing fault localization using dependency graphs

\[ \varphi_{\text{foo}} = \{ \\
\quad t_0 = 0 \\
\quad y_0 = x_0 - 3 \\
\quad z_0 = x_0 + 3 \\
\quad g_0 = w_0 > 3 \\
\quad t_1 = z_0 + w_0 \\
\quad y_1 = y_0 + 10 \\
\quad t_2 = (g_0 ? t_1 : t_0) \\
\quad y_2 = (g_0 ? y_1 : y_0) \\
\quad \neg (y_2 > z_0) \lor \\
\quad \neg (g_0 \rightarrow t_1 < x_0) \\
\} \]

Static dependency graph

Dynamic dependency graph

For bug in which \( g_0 \) is false
Must location set, based on dynamic slicing

$\varphi_{\text{foo}} = \{ t_0 = 0 \}
\begin{align*}
y_0 &= x_0 - 3 \\
z_0 &= x_0 + 3 \\
g_0 &= w_0 > 3 \\
t_1 &= z_0 + w_0 \\
y_1 &= y_0 + 10 \\
t_2 &= (g_0 ? t_1 : t_0 ) \\
y_2 &= (g_0 ? y_1 : y_0 ) \\
\neg (y_2 > z_0) \lor \neg (g_0 \rightarrow t_1 < x_0)
\}

\text{slice}_\mu(y_2) \cup \text{slice}_\mu(z_0) = \{ y_2=(g_0? y_1:y0), \ y_0=x_0-3, \ g_0=w_0>3 \} \cup \{ z_0=x_0+3 \}

\textbf{Must fault location set:}
set of statements from the program
\{ y:= x-3, \ z:=x+3, \ g:= w>3 \}
Implementing must fault localization

We implemented our must fault localization algorithm
• within the AllRepair tool

• AllRepair is based on generate - validate
• It returns all minimal repairs from the search space
  • Based on $S_{\text{mut}}$

• Minimal with respect to the number of changes (mutations) applied to the code
Sound and Complete Mutation-Based Program Repair: AllRepair

Search Space

SMT solver
Making AllRepair more efficient

Goal: reducing the search space

1. When a correct mutated program is generated (Validate succeeds)
   • Eliminate non-minimal correct mutated programs

2. When a buggy mutated program is generated (Validate fails)
   • Eliminate “similar” buggy mutated programs
Unsuccessful repair:
Buggy mutated program $P_M$ is generated

Elimination:

- **Find a must location set** $F$ **for the bug in** $P_M$
  - $F$ is a set of statements that **guarantee the bug**, if not changed
- **Eliminate** from the search space **any mutated program**, containing $F$
Adding must fault localization to program repair: FL-AllRepair
**Theorem:** FL-AllRepair is sound and complete

That is, no good repair is eliminated by our pruning of the search space
Experimental results - Benchmark

• **TCAS**
  traffic collision avoidance system for aircrafts

• **Codeflaws**
  solutions submitted by programmers to the programming contest site Codeforces
  Loops were unwound 2, 5, 8, 10 times

**Specification**: Checking equivalence to a correct version
Comparing times of AllRepair and FL-AllRepair

Each $x$ value represents a single repair; $y$ represent the time in seconds
Timeout of 10 minutes; at most 2 mutations
Comparing times of AllRepair and FL-AllRepair

(b) Medium repairs (5 – 240s)

(c) Slow repairs (> 240s)

X values represent a single repair; Y represent the time in seconds
Timeout of 10 minutes; at most 2 mutations
Summary

• A novel must fault localization
  • With respect to a repair scheme

• “must” and not “may”: you must change at least one of the lines returned

• Even though fault localization is must, its computation is relatively cheap
Summary

• Our must fault localization **significantly speeds up** the mutation-based automated program repair tool: **AllRepair**
  • By **pruning** the search space
  • No good potential repair is lost!
Assume, Guarantee or Repair

Joint work with
Hadar Frenkel, Corina Pasareanu, Sarai Sheinvald

TACAS 2020
Goal

• Exploit the partition of the system into components

• Compositional model checking verifies small components and conclude the correctness of the full system

• If a bug is found, repair is applied to one of the components
Communicating systems

- C-like programs
- Each component is described as a control-flow graph (automaton)
- Enable using automata learning algorithms

```plaintext
1: while (true)  
2:    pass = readInput;  
3:    while (pass ≤ 999)  
4:        pass = readInput;  
5:    pass2 = encrypt(pass);
```
Example

• Components synchronize over common channels
Example

• Components synchronize over common channels
Example

- Components synchronize over common channels

\[
\begin{align*}
M_1 & \quad \text{In} \ x_{pw} \\
& \quad x_{pw} \leq 999 \\
& \quad x_{pw} > 999 \\
M_2 & \quad \text{getEnc} \ ?x_{pw2} \\
& \quad \text{enc} \ !x_{pw} \\
q_0 & \quad \text{q}_0 \quad \text{q}_1 \quad \text{q}_2 \quad \text{q}_3 \quad \text{q}_4
\end{align*}
\]
Example

- Components synchronize over common channels
Example

- Components synchronize over common channels

\[ M_1 \]

- \( p_0 \):
  - \( \text{enc?}y_{pw} \)
  - \( \text{getEnc!}y_{pw} \)

- \( p_1 \):
  - \( y_{pw} := 2 \cdot y_{pw} \)

- \( p_2 \):
  - \( \text{getEnc!}y_{pw} \)

\[ M_2 \]

- \( q_0 \):
  - \( \text{In?}x_{pw} \)
  - \( x_{pw} \leq 999 \)
  - \( \text{In?}x_{pw} \)

- \( q_1 \):
  - \( \text{getEnc?}x_{pw2} \)
  - \( x_{pw} > 999 \)

- \( q_2 \):
  - \( \text{In?}x_{pw} \)

- \( q_3 \):
  - \( \text{enc!}x_{pw} \)

- \( q_4 \)
Example

• Components synchronize over common channels
Specifications

- Safety requirements - given as an automaton
- Behavior of the program through time
- “the entered password is different from the encrypted password”
- “there is no overflow”
Specifications

- Safety requirements - given as an automaton
- Behavior of the program through time
- “the entered password is different from the encrypted password”
- “there is no overflow”
Assume-Guarantee (AG) Rule

1. check if a component $M_1$ guarantees $P$ when it is a part of a system satisfying assumption $A$
Assume-Guarantee (AG) Rule

1. check if a component \( M_1 \) guarantees \( P \) when it is a part of a system satisfying assumption \( A \)

2. show that the other component \( M_2 \) (the environment) satisfies \( A \)
Assume Guarantee or Repair

1. $A_i \parallel M_1 \models P$

2. $M_2 \models A_i$

- If $P$ holds in $M_1 \parallel M_2$, then $P$ violated in $M_1 \parallel M_2$.
- If $P$ violated in $M_1 \parallel M_2$, then $cex \not\in L(A_i)$.
- If $cex \models M_1 \models P$?

- If real error?, then $N$.
- If real error?, then $Y$.

- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
- If $cex \not\in L(A_i)$, then $YN$.
Semantic repair  
(cex contains constraint)

• AGR returns a counterexample \( t \) (for \( x_{pw} = 2^{63} \)), which contains constraints

• \( \varphi_t \) a formula (in SSA) representing \( t \)
\[
\varphi_t = (x_{pw} > 999) \land (y_{pw} = x_{pw}) \land (y'_{pw} = 2 \cdot y_{pw}) \land (x_{pw2} = y'_{pw}) \land (x_{pw} \neq x_{pw2}) \land (y'_{pw} \geq 2^{64})
\]

• **Goal:**
  to make the counterexample **infeasible** by adding another constraint \( C \) to it
  • \((\varphi_t \land C \rightarrow \text{false})\)

• Using abduction
Semantic repair

- Using abduction to repair $M_2$
- Find $C$ over the variables of $M_2$ only such that $\left( \varphi_t \land C \rightarrow \text{false} \right)$

- $C = \forall y_{pw} \forall y'_{pw} \left( \neg \varphi_t \right)$

- After quantifier elimination and simplification we get:
  - $C = \left( x_{pw} < 2^{63} \right)$
1: while (true)
2:   pass = readInput;
3:   while (pass ≤ 999)
4:     pass = readInput;
5:   pass2 = encrypt(pass);
6:   assume (pass < $2^{63}$)
Syntactic repair
(cex contains no constraints)

• The counterexample \( t \) contains no constraint
  • It consists of communication actions and assignments
• Abduction will not help

3 methods to removing counterexample \( t \):
• Exact: remove exactly \( t \) from \( M_2 \)
• Approximate:
• Aggressive:
Comparing Repair Methods (logarithmic scale)

#15, #16, #18, #19 apply also abduction
Adapting L* for communicating C programs

L* is supposed to learn a regular language, over finite alphabet

Our setting:
• Infinite-state programs with first-order constraints:
  • L* Learns words over alphabet including statements in the code:
    - assignment, communication action, constraints
• We identify a target language for L*, which is regular:
  • The set of words in M2: sequences of statements
Summary

- Learning-based Assume Guarantee algorithm for infinite-state communicating programs
  - Adjustment of L* for handling infinite-state systems
- Incremental use of subsequent L* applications
- AGR often produces small assumptions, much smaller than $M_2$
- Semantic and syntactic repair
Summary

• Two approaches to automatic program repair
  • based on formal method technologies
Thank you