Automated Program Repair

Orna Grumberg
Technion, Israel

CS Research Week, National University of Singapore (NUS)
January 7, 2020
Why (formal) verification?

• safety-critical applications: Bugs are unacceptable!
  • Air-traffic controllers
  • Medical equipment
  • Cars

• Bugs found in later stages of the development are expensive

• Hardware and software systems grow in size and complexity: Subtle errors are hard to find by testing

Automated tools for formal verification are needed
Model Checking

- **Given a system and a specification, does the system satisfy the specification.**
Challenges in model checking

Model checking is successfully used for automated software and hardware verification, but more is needed:

• Scalability
• New types of systems
• New specifications (e.g. security)
• Applications in new areas
Technologies to help

Developed or adapted by the MC community

• SAT and SMT solvers
• Static analysis
• Abstraction - refinement
• Compositional verification
• Machine learning, automata learning

And many more...
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
In this talk

- Exploit Model Checking technologies for program repair
  - Mutation-Based Program Repair
  - Assume, Guarantee or Repair
Sound and Complete Mutation-Based Program Repair

[Rothenberg, Grumberg]
Mutation-Based Program Repair

Sequential program

Assertions in code

Given set of mutations

Can we use these mutations to make all assertions hold?

Assignments, conditionals, loops and function calls

Assertion violation

operator replacement (+ \rightarrow -),
constant manipulation (c \rightarrow c + 1)

Return all possible repairs
Example

```c
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z >= 9) z = z - 1;
    assert(z > 8);
    return z;
}
```

\[ x = 5, y = 2 \]
\[ z = 9 \]
Example

```c
int f(int x, int y){
1. int z;
2. if (x + y > 8) {
3.     z = x + y;
4. } else {
5.     z = 9;
6. }
7. if (z \geq 9) z = z + 1;
8. assert(z \geq 8);
9. return z;
}
```

Mutation list:
- Replace + with –
- Replace – with +
- Replace \(\geq\) with \(>\)

Repair list:
- option 1:
  - line 7: replace \(\geq\) with 
- option 2:
  - line 7: replace – with +

Note: Repairs are minimal
Example

```c
int f(int x, int y){
    1. int z;
    2.  if (x + y > 9) {
        3.      z = x + y;
        4.  } else {
        5.      z = 10;
        6.  }
    7.  if (z ≥ 9) z = z - 1;
    8.    assert(z > 8);
    9.    return z;
}
```

Mutation list:
- Replace + with –
- Replace – with +
- Replace > with ≥
- Replace ≥ with >
- Increase constants by 1

At this point \(z \geq 10\)
Overview of our approach

Finding all correct programs from a finite set of programs
Finding all unsatisfiable constraint sets from a finite set of constraint sets

Input:
a buggy program

Output:
All minimal repairs, sorted by size

```
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z >= 9) z = z - 1;
    assert(z > 8);
    return z;
}
```
First step - Translation

Goal: Translate the program into a set of constraints which is satisfiable iff the program has a bug (i.e. there exists an input for which an assertion fails)

Work by Clarke, Kroening, Lerda (TACAS 2004) (CBMC)
- Simplification
- Unwinding of loops
  - a bounded number of unwinding
- Conversion to SSA

Correctness is bounded
First step - Translation

```c
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z >= 9) z = z - 1;
    assert(z > 8);
    return z;
}
```
First step - Translation

```c
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z ≥ 9) z = z - 1;
    assert(z > 8);
    return z;
}
{  
g_{1} = x_{1} + y_{1} > 8,
   z_{2} = x_{1} + y_{1},
   z_{3} = 9,
   z_{4} = g_{1}?z_{2}:z_{3},
   b_{1} = z_{4} ≥ 9,
   z_{5} = z_{4} - 1,
   z_{6} = b_{1}?z_{5}:z_{4},
   z_{6} ≤ 8
}
```
First step - Translation

```c
int f(int x, int y)
{
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z ≥ 9) z = z - 1;
    assert(z > 8);
    return z;
}
```

```c
{  g₁ = x₁ + y₁ > 8,
   z₂ = x₁ + y₁,
   z₃ = 9,
   z₄ = g₁?z₂:z₃,
   b₁ = z₄ ≥ 9,
   z₅ = z₄ - 1,
   z₆ = b₁?z₅:z₄,
   z₆ ≤ 8
}
```
int f(int x, int y) {
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z ≥ 9) z = z - 1;
    assert(z > 8);
    return z;
}
Translation

• In the translation, loops are unwound a bounded number of times.

• Important observation: correctness is bounded. That is, repairs found by our method only guarantee that assertions cannot be violated by inputs going through the loop at most \( k \) times.
Second step - Mutation

```c
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x - y;
    } else {
        z = 9;
    }
    if (z >= 9) {
        z = z - 1;
    }
    assert(z > 8);
    return z;
}
```

Mutation list:
- Replace + with –
- Replace – with +
- Replace \(\geq\) with \(>\)

- \(g_1 = x + y_1 > 8\)
- \(z_2 = x + y_1\)
- \(z_2 = x - y_1\)
- \(b_1 = z_4 \geq 9, b_1 = z_4 > 9\)
- \(z_5 = z_4 - 1, z_5 = z_4 + 1\)
- \(b_1 = z_4 \geq 9, b_1 = z_4 > 9\)
- \(z_6 = b_1, z_6 = z_4\)
- \(z_6 \leq 8\)
Third step - Repair

```c
1. int i;
2. if (x + y <= 8) {
3.     z = x + y;
4. } else {
5.     z = 9;
6. } else {
7.     assert(z > 8);
8.     return z;
9. }

\{ z_2 = x_1 + y_1, z_2 = x_1 - y_1 \}
\{ z_3 = 9 \}
\{ b_1 = z_4 \geq 9, b_1 = z_4 > 9 \}
\{ z_5 = z_4 - 1, z_5 = z_4 + 1 \}
\{ z_6 = b_1 ? z_5 : z_4 \}
z_6 \leq 8
```
Third step - Repair

SAT solver:
Checks satisfiability of a propositional formula
• If it is satisfiable - returns a satisfying assignment
Generates mutated programs of increasing size

SMT solver:
Checks satisfiability of a first-order formula over theory (e.g., linear arithmetic)
• If it is satisfiable - returns a satisfying assignment
Checks (bounded) correctness of the mutated programs
**Repair**

\[
\begin{align*}
\text{SAT solver} & \\
\text{Choose candidate program of size } &= 1 \\
\end{align*}
\]

\[
\begin{align*}
c_1 & \quad c_2 \\
\{ g_1 = x_1 + y_1 > 8, g_1 = x_1 - y_1 > 8, \\
& \quad g_1 = x_1 + y_1 \geq 8 \} \\
\end{align*}
\]

\[
\begin{align*}
c_4 & \quad c_5 \\
\{ z_2 = x_1 + y_1, z_2 = x_1 - y_1 \} \\
\end{align*}
\]

\[
\begin{align*}
c_6 & \\
\{ z_3 = 9 \} \\
\end{align*}
\]

\[
\begin{align*}
c_7 & \quad c_8 \\
\{ b_1 = z_4 \geq 9, b_1 = z_4 > 9 \} \\
\end{align*}
\]

\[
\begin{align*}
c_9 & \quad c_{10} \\
\{ z_5 = z_4 - 1, z_5 = z_4 + 1 \} \\
\end{align*}
\]

\[
\begin{align*}
z_4 &= g_1 ? z_2 : z_3 \\
z_6 &= b_1 ? z_5 : z_4 \\
z_6 &\leq 8 \\
\end{align*}
\]

**SMT solver**
Repair

**SAT solver**

Choose candidate program of size = 1

**SAT**

\[ c_1 = 0 \]
\[ c_2 = 1 \]
\[ c_3 = 0 \]
\[ c_4 = 1 \]
\[ c_5 = 0 \]
\[ c_6 = 1 \]
\[ c_7 = 1 \]
\[ c_8 = 0 \]
\[ c_9 = 1 \]
\[ c_{10} = 0 \]

**SMT solver**

\[ z_4 = g_1 ? z_2 : z_3 \]
\[ z_6 = b_1 ? z_5 : z_4 \]
\[ z_6 \leq 8 \]

\[ \{ g_1 = x_1 + y_1 > 8, g_1 = x_1 - y_1 > 8, g_1 = x_1 + y_1 \geq 8 \} \]

\[ \{ z_2 = x_1 + y_1, z_2 = x_1 - y_1 \} \]

\[ \{ z_3 = 9 \} \]

\[ \{ b_1 = z_4 \geq 9, b_1 = z_4 > 9 \} \]

\[ \{ z_5 = z_4 - 1, z_5 = z_4 + 1 \} \]
Repair

Choose candidate program of size = 1

\[
\begin{align*}
g_1 &= x_1 + y_1 > 8, \quad g_1 = x_1 - y_1 > 8, \quad g_1 = x_1 + y_1 \geq 8 \\
c_3 &= \\
c_4 &= \{z_2 = x_1 + y_1, \quad z_2 = x_1 - y_1\} \\
c_5 &= \\
c_6 &= \{z_3 = 9\} \\
c_7 &= \{b_1 = z_4 \geq 9, \quad b_1 = z_4 > 9\} \\
c_8 &= \\
c_9 &= \{z_5 = z_4 - 1, \quad z_5 = z_4 + 1\} \\
c_{10} &= \\
SMT solver

\begin{align*}
z_4 &= g_1 \land z_2 \land z_3 \\
z_6 &= b_1 \land z_5 \land z_4 \\
z_6 &\leq 8 \\
g_1 &= x_1 - y_1 > 8 \\
z_2 &= x_1 + y_1 \\
z_3 &= 9 \\
b_1 &= z_4 \geq 9 \\
z_5 &= z_4 - 1 \\
b_1 &= z_4 > 9 \\
z_5 &= z_4 - 1
\end{align*}

SAT
\[
\begin{align*}
c_1 &= 0 \\
c_2 &= 1 \\
c_3 &= 0 \\
c_4 &= 1 \\
c_5 &= 0 \\
c_6 &= 1 \\
c_7 &= 1 \\
c_8 &= 0 \\
c_9 &= 1 \\
c_{10} &= 0
\end{align*}
\]
Repair

Choose candidate program of size = 1

SAT

\( g_1 = x_1 - y_1 > 8 \), \( g_2 = x_1 + y_1 \geq 8 \)

SAT (not a repair)

\( z_2 = x_1 + y_1 \), \( z_2 = x_1 - y_1 \)

\( b_1 = z_4 \geq 9 \), \( b_1 = z_4 > 9 \)

\( z_5 = z_4 - 1 \)

\( z_5 = z_4 + 1 \)

SMT solver

\( g_1 = x_1 - y_1 > 8 \)

\( z_2 = x_1 + y_1 \)

\( z_3 = 9 \)

\( b_1 = z_4 \geq 9 \)

\( b_1 = z_4 > 9 \)

\( z_6 \leq 8 \)

\( z_5 = z_4 - 1 \)

\( z_6 = b_1 \land z_5 \land z_4 \)

\( c_1 = 0 \)
\( c_2 = 1 \)
\( c_3 = 0 \)
\( c_4 = 1 \)
\( c_5 = 0 \)
\( c_6 = 1 \)
\( c_7 = 1 \)
\( c_8 = 0 \)
\( c_9 = 1 \)
\( c_{10} = 0 \)
**Repair**

SAT solver

Choose candidate program of \textit{size} = 1

Blocking clause for similar assignments

\[c_1\]
\[
\{ g_1 = x_1 + y_1 > 8, \quad g_1 = x_1 - y_1 > 8, \quad g_1 = x_1 + y_1 \geq 8 \}
\]

\[c_2\]
\[
\{ g_1 = x_1 - y_1 > 8, \quad g_1 = x_1 + y_1 \geq 8 \}
\]

\[c_3\]
\[
\{ z_2 = x_1 + y_1, \ z_2 = x_1 - y_1 \}
\]

\[c_4\]
\[
\{ z_3 = 9 \}
\]

\[c_5\]
\[
\{ z_6 = b_1 \geq 9, \ b_1 = z_4 > 9 \}
\]

\[c_6\]
\[
\{ b_1 = z_4 \geq 9, \ b_1 = z_4 > 9 \}
\]

\[c_7\]
\[
\{ z_5 = z_4 - 1, \ z_5 = z_4 + 1 \}
\]

\[c_8\]
\[
\{ z_6 = b_1 \geq 9, \ b_1 = z_4 \geq 9 \}
\]

\[c_9\]
\[
\{ g_1 = x_1 - y_1 > 8 \}
\]

\[c_{10}\]
\[
\{ z_3 = 9 \}
\]

\[c_{11}\]
\[
\{ z_6 \leq 8 \}
\]

\[c_{12}\]
\[
\{ b_1 = z_4 \geq 9 \}
\]

\[c_{13}\]
\[
\{ z_5 = z_4 - 1 \}
\]

\[c_{14}\]
\[
\{ z_5 = z_4 + 1 \}
\]
Repair

Blocking clause for "similar" assignments

- Assignments causing a similar bug

\[ \begin{align*}
\{ z_5 &= z_4 - 1 \\ z_5 &= z_4 + 1 \} \\
\end{align*} \]

\[ \begin{align*}
z_6 &\leq 8 \\
b_1 &= z_4 \geq 9 \\
z_5 &= z_4 - 1 \\
\end{align*} \]
Repair

**SAT solver**

Choose candidate program of size = 1

Blocking clause for similar assignments

**SMT solver**

$g_1 = x_1 + y_1 \geq 8$

$g_1 = x_1 - y_1 > 8$

$g_1 = x_1 + y_1 \geq 8$

$z_2 = x_1 + y_1, z_2 = x_1 - y_1$

$z_3 = 9$

$b_1 = z_4 \geq 9, b_1 = z_4 > 9$

$z_5 = z_4 - 1, z_5 = z_4 + 1$

$z_4 = g_1? z_2 : z_3$

$z_6 = b_1? z_5 : z_4$

$z_6 \leq 8$

$g_1 = x_1 + y_1 > 8$

$z_2 = x_1 + y_1$

$z_3 = 9$

$b_1 = z_4 > 9$

$z_5 = z_4 - 1$

$z_6 \leq 8$

$c_1 = 1$

$c_2 = 0$

$c_3 = 0$

$c_4 = 1$

$c_5 = 0$

$c_6 = 1$

$c_7 = 0$

$c_8 = 1$

$c_9 = 1$

$c_{10} = 0$
**Repair**

**SAT solver**

Choose candidate program of \(\text{size} = 1\)

**SMT solver**

Blocking clause for similar assignments

\[
\begin{align*}
 g_1 &= x_1 + y_1 > 8 \\
 c_2 &= g_1 = x_1 - y_1 > 8, \\
 c_3 &= g_1 = x_1 + y_1 \geq 8 \\
 c_4 &= z_2 = x_1 + y_1, z_2 = x_1 - y_1 \\
 c_5 &= z_3 = 9 \\
 c_6 &= z_4 = g_1 \? z_2 : z_3 \\
 c_7 &= b_1 = z_4 \geq 9, b_1 = z_4 > 9 \\
 c_8 &= z_6 = b_1 \? z_5 : z_4 \\
 c_9 &= z_5 = z_4 - 1 \\
 c_{10} &= z_5 = z_4 + 1
\end{align*}
\]

\[
\begin{align*}
 g_1 &= x_1 + y_1 > 8 \\
 c_2 &= z_2 = x_1 + y_1 \\
 c_3 &= z_3 = 9 \\
 c_4 &= b_1 = z_4 \geq 9, b_1 = z_4 > 9 \\
 c_5 &= z_5 = z_4 - 1 \\
 c_6 &= z_6 = b_1 \? z_5 : z_4 \\
 c_7 &= z_6 \leq 8
\end{align*}
\]
Repair

\[ g_1 = x_1 + y_1 \geq 8 \]
\[ g_1 = x_1 - y_1 > 8, \]
\[ g_1 = x_1 + y_1 \geq 8 \]
\[ z_2 = x_1 + y_1, z_2 = x_1 - y_1 \]
\[ z_3 = 9 \]
\[ b_1 = z_4 \geq 9, b_1 = z_4 > 9 \]
\[ z_5 = z_4 - 1 \]
\[ z_5 = z_4 + 1 \]

Choose candidate program of size = 1
Blocking clause for similar assignments

UNSAT (repair found!)

Blocking clause for this assignment
And all other supersets of changes

SMT solver

\[ g_1 = x_1 + y_1 \geq 8 \]
\[ z_2 = x_1 + y_1 \]
\[ z_3 = 9 \]
\[ b_1 = z_4 \geq 9 \]
\[ b_1 = z_4 > 9 \]
\[ z_5 = z_4 - 1 \]
\[ z_5 = z_4 + 1 \]

SAT
\[ c_1 = 1 \]
\[ c_2 = 0 \]
\[ c_3 = 0 \]
\[ c_4 = 1 \]
\[ c_5 = 0 \]
\[ c_6 = 1 \]
\[ c_7 = 0 \]
\[ c_8 = 1 \]
\[ c_9 = 1 \]
\[ c_{10} = 0 \]
Repair

Blocking clause for this assignment and all other supersets of changes

- Repairs that are not minimal
Repair

Choose candidate program of size = 1

Blocking clause for similar assignments

Blocking clause for this assignment

And all other supersets of changes

SMT solver

UNSAT (repair found!)

SAT

\begin{align*}
\begin{array}{l}
g_1 &= x_1 + y_1 \geq 8, \\
&\quad g_1 = x_1 - y_1 > 8, \\
&\quad g_1 = x_1 + y_1 \geq 8 \\
\end{array} \\
c_4 &\begin{array}{l}
\{ z_2 = x_1 + y_1, z_2 = x_1 - y_1 \} \\
\end{array} \\
c_5 &\begin{array}{l}
\{ z_3 = 9 \} \\
\end{array} \\
c_6 &\begin{array}{l}
\{ b_1 = z_4 \geq 9, b_1 = z_4 > 9 \} \\
\end{array} \\
c_7 &\begin{array}{l}
\{ z_5 = z_4 - 1, z_5 = z_4 + 1 \} \\
\end{array} \\
c_8 &\begin{array}{l}
\{ g_1 = x_1 + y_1 > 8 \\
z_2 = x_1 + y_1 \\
z_3 = 9 \\
z_6 = b_1 ? z_5 : z_4 \\
z_5 = z_4 - 1 \\
\} \\
\end{array}
\end{align*}
Repair

Choose candidate program of \textit{size} = 1

\begin{align*}
\{ & g_1 = x_1 + y_1 > 8, \quad g_1 = x_1 - y_1 > 8, \\
& \quad \quad g_1 = x_1 + y_1 \geq 8 \}\ \\
& \quad \quad c_3
\end{align*}

\begin{align*}
\{ & z_2 = x_1 + y_1, \quad z_2 = x_1 - y_1 \}
\end{align*}

\begin{align*}
& c_4 \quad c_5 \\
& \{ z_3 = 9 \}
\end{align*}

\begin{align*}
\{ & b_1 = z_4 \geq 9, \quad b_1 = z_4 \geq 9 \}
\end{align*}

\begin{align*}
& c_7 \quad c_8 \\
& \{ z_5 = z_4 - 1, \quad z_5 = z_4 + 1 \}
\end{align*}

\begin{align*}
\text{SAT} & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
Repair

Choose candidate program of size = 2

Blocking clause for this assignment
And all other supersets of changes

SMT solver

SAT

c₁ = 1
c₂ = 0
c₃ = 0
c₄ = 1

c₅ = 1

c₆ = 1

c₇ = 0

c₈ = 1

c₉ = 1

c₁₀ = 0

UNSAT

UNSAT
Making repair more efficient

Repair traverses the search space of all mutated programs
• running iterations of *Generate - Validate*

**Goal: reducing the search space**

1. When a **correct mutated program** is generated (*Validate succeeds*)
   • Eliminating non-minimal correct mutated programs
2. When a **buggy mutated program** is generated (*Validate fails*)
   • Eliminate “similar” buggy mutated programs
Correct mutated program

Successful repair:
A set of mutations $M$ that results in a (bounded) correct program

Eliminate non-minimal repairs:
Any superset of $M$ is not minimal
• Add a blocking clause to the SAT solver that disallows to choose any superset of $M$
Buggy mutated program

Unsuccessful repair:
A set of mutations $M$ that results in a buggy program

Elimination:
- Find a small explanation $S$ for the bug
  - $S$ is a set of statements in the code
- Disallow any mutated program, containing $S$
Fault localization

Fault localization: A (small) explanation $S$ to a bug

In other works:

- **May** explanation
  - Changes to statements from $S$ may result in a repaired program
Fault localization

Fault localization: A (small) explanation $S$ to a bug

In our work:

- **Must** explanation
  - If *none* of the statements in $S$ is changed, then
    - regardless of changes applied to other statement
    - the same bug will remain
  - $\Rightarrow S$ must be changed
Reducing the search space

For a **must** fault localization $S$:

- **Remove** from the **search space** all programs containing $S$
- If $S$ is **small**, more programs will be removed
Fault localization: example

```c
int f(int x, int y){
    1. int z;
    2. z = x
    3. if (x >= 0) {
        4.     x = x + 1; y = x + 2;
    5. } else {
        6.     z = 9;
    7. } assert(z > 0);
    8. return z;
}
```
Fault localization: example

```c
int f(int x, int y){
    1. int z; int t;
    2. z = x
    3. if (x >= 0) {
        4. x = x + 1; y = x + 2;
    5. } else {
        6. z = 9;
    }
    7. assert(z > 0);  
    8. return z;
}

 erroneous run:
  x=0, y=0
  z=0
  x=1, y=2
  z=0
```

Repair: line 3 should change to (x > 0)
Fault localization by slicing

```c
int f(int x, int y){
    int z; int t;
    z = x
    if (x >= 0) {
        x = x + 1; y = 0;
    } else {
        z = 9;
    }
    assert(z > 0);
    return z;
}
```
Theorem:

Our algorithm is sound and complete

That is, for a given bound \(b\):
A program is returned by our algorithm
iff
it is minimal and \(b\)-bounded correct

• Minimal number of changes
• Every assertion reachable along a computation of bounded length \(b\) is correct
Experimental results on the 41 faulty versions of the TCAS program from the Siemens suite, which implements a traffic collision avoidance system for aircrafts. Comparison to two earlier methods by Konighofer and Bloem implemented in the tool FoReSiC.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op. replacement</td>
<td>Arithmetic {+, -, {*, /, %}</td>
<td>Arithmetic {+, -, {*, /, %}</td>
</tr>
<tr>
<td>Relational</td>
<td>{{&gt;, &gt;=}} {&lt;, &lt;=}</td>
<td>{{&gt;, &gt;=, &lt;, &lt;=}, {==, !=}</td>
</tr>
<tr>
<td>Logical</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>Bit-wise</td>
<td>{&gt;&gt;, &lt;&lt;}, {&amp;}, ^}</td>
<td></td>
</tr>
<tr>
<td>Constant manipulation</td>
<td>C→C+1, C→C−1, C→−C, C→0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>+</td>
<td>34</td>
<td>+</td>
<td>35</td>
<td>+</td>
<td>93.678</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td>+</td>
<td>1.246</td>
<td>+</td>
<td>4.661</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td>+</td>
<td>1.902</td>
<td>+</td>
<td>85.349</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>+</td>
<td>41</td>
<td>+</td>
<td>46</td>
<td>+</td>
<td>92.866</td>
<td>+</td>
<td>94.599</td>
</tr>
<tr>
<td>36</td>
<td>+</td>
<td>8</td>
<td>+</td>
<td>6</td>
<td>+</td>
<td>2.558</td>
<td>+</td>
<td>16.393</td>
</tr>
<tr>
<td>39</td>
<td>+</td>
<td>82</td>
<td>+</td>
<td>101</td>
<td>+</td>
<td>93.678</td>
<td>+</td>
<td>69.166</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 16 (39%) | 38       | 15 (36.6%) | 38 | 11 (26.83%) | 2.278 | 18 (43.9%) | 48.151 |
Adding fault localization

Every generate-validate iteration with fault localization is more expensive
- But we expect to have less iterations

Both AllRepair and FL-AllRepair are complete
- return the same set of repaired programs
- Not necessarily in the same order
Summary

Mutation-based automated repair can assist a programmer in debugging in initial stages of development

• When bugs are simple, but many

• It also can help beginner programmers
  • Educational tool for students

• Analysis can be used to prioritize the returned repaired programs
Assume, Guarantee or Repair

[Frenkel, Grumberg, Pasareanu, Sheinvald]
Motivation

• Find bugs in a large system

• Model checking of large systems may not scale

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

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

• C-like programs
• Described as a control-flow graph (automaton)
• Use automata learning algorithms

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

- Components synchronize over common channels
Example

- Components synchronize over common channels
Example

• Components synchronize over common channels
Example

• Components synchronize over common channels
Example

- Components synchronize over common channels
Example

- Components synchronize over common channels
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”
Compositional Verification

• Inputs:
  - composite system $M_1 \parallel M_2$
  - property $P$

• Goal: check if $M_1 \parallel M_2 \models P$
Useful AG Rule

1. check if a component $M_1$ guarantees $P$ when it is a part of a system satisfying assumption $A$
Useful AG Rule for Safety Properties

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 $A$. 

\[
\begin{align*}
A & \parallel M_1 \models P \\
M_2 \models A \\
M_1 \parallel M_2 & \models P
\end{align*}
\]
Assume Guarantee or Repair

1. $A_i \parallel M_1 \models P$
2. $M_2 \models A_i$

- True
  - $P$ holds in $M_1 \parallel M_2$
  - Repair $M_2$
- False
  - $P$ violated in $M_1 \parallel M_2$
  - $cex \models M_1 \not\models P$

Learning

Counterexample - strengthen assumption

Counterexample - weaken assumption

Real error?

Semantic repair

• The counterexample contains constraint

• Goal:
  to make the counterexample infeasible by adding another constraint $C$ to it

• Using abduction
Semantic repair

- learn a constraint $C$ such that:
  - $C \land pass > 999 \land pass2 = pass \cdot 2 \rightarrow pass2 < 2^{64}$
  - $C$ is over the input variables of $M_2$: $pass$

\[
C := \forall pass2 \left[ pass > 999 \land pass2 = pass \cdot 2 \rightarrow pass2 < 2^{64} \right]
\]

- After quantifier elimination & simplification: $C = pass < 2^{63}$.

Abduction - "Logical Magic"
1: while (true)
2:     pass = readInput;
3:     while (pass ≤ 999 or pass ≥ 2^{63})
4:         pass = readInput;
5:     pass2 = encrypt(pass);
6:     return pass2;
Syntactic repair

• 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:
Example – Syntactic Repair

No self loop, cannot read more than once each time!

Multiple reads are allowed
Agressive Repair

• Remove accepting states (can make the language of $M_2$ empty)
Approximate Repair

- Add an intermediate state to eliminate bad traces
Exact Repair

- Remove bad traces one by one
- First bad trace spotted is $\text{read(pass)}, \text{read(pass)}$
### AGR Results on Various Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>$M_1$ Size</th>
<th>$M_2$ Size</th>
<th>$P$ Size</th>
<th>Time (sec.)</th>
<th>$A$ size</th>
<th>Repair Size</th>
<th>Repair Method</th>
<th>#Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>#4</td>
<td>64</td>
<td>64</td>
<td>3</td>
<td>0.106, 0.126, 0.132</td>
<td>5, 6, 8</td>
<td>27</td>
<td>aggress. approx. exact</td>
<td>2</td>
</tr>
<tr>
<td>#6</td>
<td>2</td>
<td>27</td>
<td>2</td>
<td>0.130, 0.138, 0.165</td>
<td>6, 7, 9</td>
<td>28, 28, 81</td>
<td>approx. exact</td>
<td>2</td>
</tr>
<tr>
<td>#7</td>
<td>2</td>
<td>81</td>
<td>2</td>
<td>0.150, 0.170, 0.223</td>
<td>8, 8, 10</td>
<td>243, 244, 729</td>
<td>approx. exact</td>
<td>2</td>
</tr>
<tr>
<td>#8</td>
<td>2</td>
<td>243</td>
<td>2</td>
<td>4.88, 92</td>
<td>4.44, 109</td>
<td>verification</td>
<td>verification</td>
<td>2</td>
</tr>
<tr>
<td>#11</td>
<td>5</td>
<td>256</td>
<td>6</td>
<td>6.63, 5.94, 12.87</td>
<td>113, 113, 155</td>
<td>256, 257, 1280</td>
<td>approx. exact</td>
<td>2</td>
</tr>
<tr>
<td>#14</td>
<td>5</td>
<td>256</td>
<td>6</td>
<td>1.07, 1.12, 1.26</td>
<td>18, 18, 18</td>
<td>18, 18, 18</td>
<td>approx. exact</td>
<td>3</td>
</tr>
<tr>
<td>#15</td>
<td>3</td>
<td>16</td>
<td>5</td>
<td>0.69, 0.28, 4.27</td>
<td>12, 13, 44</td>
<td>16, 18, 864</td>
<td>approx. exact</td>
<td>3</td>
</tr>
<tr>
<td>#16</td>
<td>4</td>
<td>256</td>
<td>8</td>
<td>1.09, 0.21</td>
<td>1, 6</td>
<td>4 (trivial), 8</td>
<td>approx. exact</td>
<td>4</td>
</tr>
<tr>
<td>#19</td>
<td>3</td>
<td>16</td>
<td>5</td>
<td>1.07, 1.12, 1.26</td>
<td>18, 18, 18</td>
<td>18, 18, 18</td>
<td>approx. exact</td>
<td>3</td>
</tr>
<tr>
<td>#22</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>0.09, 0.21</td>
<td>1, 6</td>
<td>4 (trivial), 8</td>
<td>approx. exact</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>timeout</td>
<td></td>
<td>timeout</td>
</tr>
</tbody>
</table>
Comparing Repair Methods (logarithmic scale)
Summary

• Learning-based Assume guarantee algorithm for infinite-state communicating programs

• Incremental automata learning algorithm

• Semantic and syntactic repair

• Experiments provide proof of concept
Thank you