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

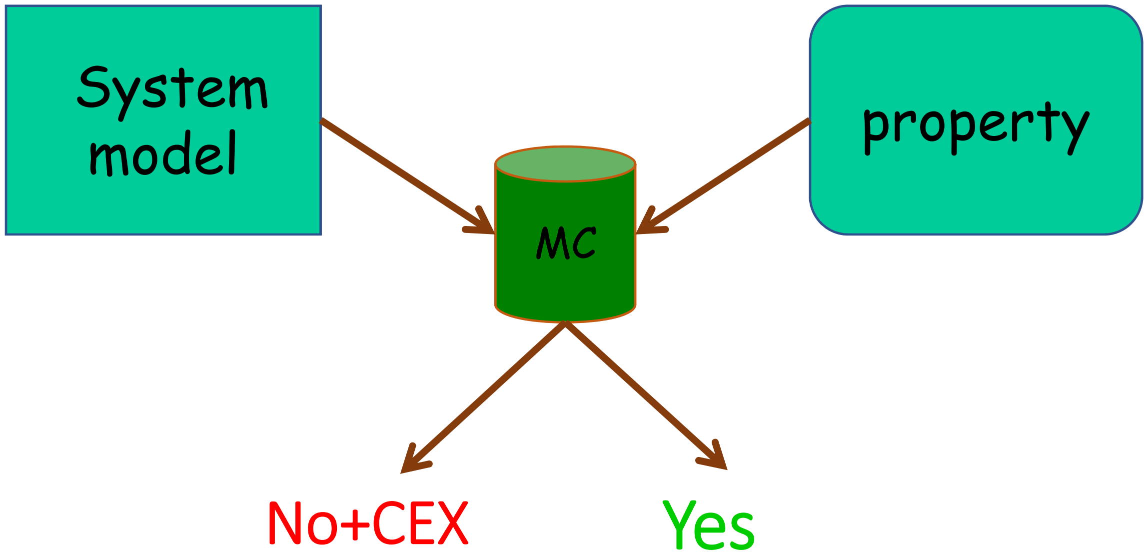
- Bugs found in later stages of the development are expensive

- 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

```
int f(int x, int y){                      x = 5, y = 2
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;                            z = 9
    }
    if (z ≥ 9) z = z − 1;                 z = 8
    assert(z > 8);
    return z;
}
```

Example

$int\ f(int\ x, int\ y)\{$

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

$\}$

At this point $z \geq 9$

Mutation list:

- Replace + with –
- Replace – with +
- Replace $>$ with $\geq$
- Replace $\geq$ with $>$

Note: Repairs are **minimal**

Repair list:

$option\ 1:$

$line\ 7:\ replace\ \ \geq with >$

$option\ 2:$

$line\ 7: replace\ -\ \ with +$

Example

```
int f(int x, int y){
    int z;
    if (x + y > 9) {
        z = x + y;
    } else {
        z = 10;
    }
    if (z ≥ 9) z = z − 1;
    assert(z > 8);
    return z;
}
```

At this point $z \geq 10$

Mutation list:

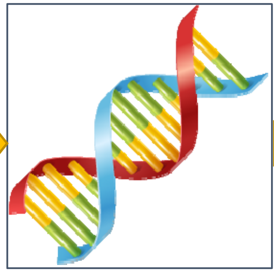
- Replace + with –
- Replace – with +
- Replace $>$ with $\geq$
- Replace $\geq$ with $>$
- Increase constants by 1

Overview of our approach

```
   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 ≥ 9)  z = z − 1;
8.     assert(z > 8);
9.     return z;
   }
```

Input:
a buggy
program

Translation



Mutation

Repair

line 7: replace operator $\geq$ *with* $>$
line 7: replace operator $-$ *with* $+$
…

Output:
All **minimal** repairs,
sorted by size

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

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

```
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;
}
```

$$
\begin{aligned}
\{\ & \boldsymbol{g_1 = x_1 + y_1 > 8}, \\
& z_2 = x_1 + y_1, \\
& z_3 = 9, \\
& z_4 = g_1 ?\, z_2 : z_3, \\
& b_1 = z_4 \geq 9\,, \\
& z_5 = z_4 - 1, \\
& z_6 = b_1 ?\, z_5 : z_4, \\
& z_6 \leq 8 \\
\}\ &
\end{aligned}
$$

First step - Translation

$int\ f(int\ x, int\ y)\{$

$\quad int\ z;$

$\quad if\ \ (x + y > 8)\ \{$

$\quad\quad \boldsymbol{z = x + y};$

$\quad \}\ \ else\ \{$

$\quad\quad z = 9;$

$\quad \}$

$\quad if\ (z \geq 9)\ \ \ z = z - 1;$

$\quad assert(z > 8);$

$\quad return\ z;$

$\}$

$\{\ \ g_1 = x_1 + y_1 > 8,$

$\boldsymbol{z_2 = x_1 + y_1},$

$z_3 = 9,$

$z_4 = g_1 ?\, z_2 : z_3,$

$b_1 = z_4 \geq 9\ ,$

$z_5 = z_4 - 1,$

$z_6 = b_1 ?\, z_5 : z_4,$

$z_6 \leq 8$

$\}$

First step - Translation

```
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;
}
```

$$
\{\ g_1 = x_1 + y_1 > 8,\\
z_2 = x_1 + y_1,\\
z_3 = 9,\\
z_4 = g_1 ?\, z_2 : z_3,\\
b_1 = z_4 \geq 9\,,\\
z_5 = z_4 - 1,\\
z_6 = b_1 ?\, z_5 : z_4,\\
z_6 \leq 8\\
\}
$$

First step - Translation

```
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;
}
```

$\{\ g_1 = x_1 + y_1 > 8,$

$z_2 = x_1 + y_1,$

$z_3 = 9,$

$\mathbf{z_4 = g_1 ? z_2 : z_3},$

$b_1 = z_4 \geq 9,$

$z_5 = z_4 - 1,$

$z_6 = b_1 ? z_5 : z_4,$

$z_6 \leq 8$

$\}$

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

Mutation list:
- Replace + with –
- Replace – with +
- Replace $>$ with $\geq$
- Replace $\geq$ with $>$

```
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;
}
```

$g_1 = x_1 + y_1 > 8$

$\{ z_2 = x_1 + y_1, z_2 = x_1 - y_1 \}$

$\{ z_3 = 9 \}$

$z_4 = g_1 ? z_2 : z_3$

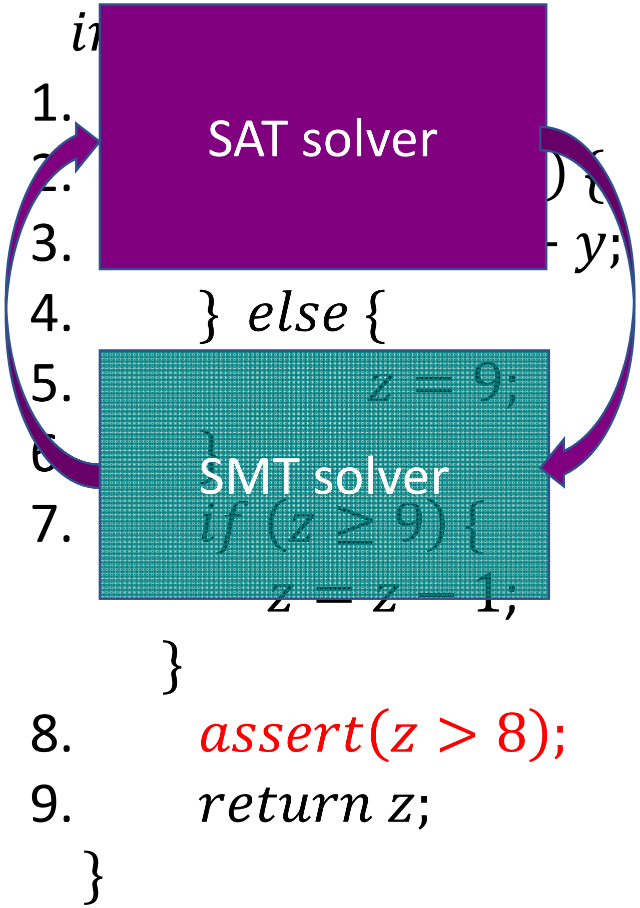
$\{ b_1 = z_4 \geq 9, b_1 = z_4 > 9 \}$

$\{ z_5 = z_4 - 1, z_5 = z_4 + 1 \}$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

Third step - Repair



$\{\, z_2 = x_1 + y_1,\, z_2 = x_1 - y_1 \}$

$\{\, z_3 = 9 \}$

$z_4 = g_1 ?\, z_2 : z_3$

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

$\{\, z_5 = z_4 - 1, z_5 = z_4 + 1 \}$

$z_6 = b_1 ?\, z_5 : z_4$

$z_6 \leq 8$

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

c_1 c_2

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

c_3

c_4 c_5

$\{ z_2 = x_1 + y_1, z_2 = x_1 - y_1\}$

c_6

$\{ z_3 = 9\}$

c_7 c_8

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

c_9 c_{10}

$\{ z_5 = z_4 - 1, z_5 = z_4 + 1\}$

SAT solver

Choose candidate program of $\textit{size} = 1$

SMT solver

$z_4 = g_1 ? z_2 : z_3$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

Repair

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

c_3

c_4 c_5

$\{z_2 = x_1 + y_1, z_2 = x_1 - y_1\}$

c_6

$\{z_3 = 9\}$

c_7 c_8

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

c_9 c_{10}

$\{z_5 = z_4 - 1, z_5 = z_4 + 1\}$

SAT solver

Choose candidate program of ***size*** $= 1$

SMT solver

$z_4 = g_1 ? z_2 : z_3$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

SAT

$c_1 = 0$

$c_2 = 1$

$c_3 = 0$

$c_4 = 1$

$c_5 = 0$

$c_6 = 1$

$c_7 = 1$

$c_8 = 0$

$c_9 = 1$

$c_{10} = 0$

Repair

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

c_3

c_4 c_5

$\{z_2 = x_1 + y_1, z_2 = x_1 - y_1\}$

c_6

$\{z_3 = 9\}$

c_7 c_8

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

c_9 c_{10}

$\{z_5 = z_4 - 1, z_5 = z_4 + 1\}$

Choose candidate program of ***size = 1***

SMT solver

$z_4 = g_1 ? z_2 : z_3$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

$g_1 = x_1 - y_1 > 8$

$z_2 = x_1 + y_1$

$z_3 = 9$

$b_1 = z_4 \geq 9$

$z_5 = z_4 - 1$

SAT

$c_1 = 0$

$c_2 = 1$

$c_3 = 0$

$c_4 = 1$

$c_5 = 0$

$c_6 = 1$

$c_7 = 1$

$c_8 = 0$

$c_9 = 1$

$c_{10} = 0$

Repair

$\overset{c_1}{\{g_1 = x_1 + y_1 > 8}, \overset{c_2}{g_1 = x_1 - y_1 > 8},$
$\underset{c_3}{g_1 = x_1 + y_1 \geq 8\}}$

$\{\overset{c_4}{z_2 = x_1 + y_1}, \overset{c_5}{z_2 = x_1 - y_1}\}$

$\{\overset{c_6}{z_3 = 9}\}$

$\{\overset{c_7}{b_1 = z_4 \geq 9}, \overset{c_8}{b_1 = z_4 > 9}\}$

$\{\overset{c_9}{z_5 = z_4 - 1}, \overset{c_{10}}{z_5 = z_4 + 1}\}$

SAT (not a repair)

Choose candidate program of ***size*** $= 1$

SMT solver

$z_4 = g_1 ? z_2 : z_3$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

$g_1 = x_1 - y_1 > 8$

$z_2 = x_1 + y_1$

$z_3 = 9$

$b_1 = z_4 \geq 9$

$z_5 = z_4 - 1$

SAT

$c_1 = 0$

$c_2 = 1$

$c_3 = 0$

$c_4 = 1$

$c_5 = 0$

$c_6 = 1$

$c_7 = 1$

$c_8 = 0$

$c_9 = 1$

$c_{10} = 0$

Repair

c_1 c_2

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

c_3

c_4 c_5

$\{ z_2 = x_1 + y_1,\ z_2 = x_1 - y_1 \}$

c_6

$\{ z_3 = 9 \}$

c_7 c_8

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

c_9 c_{10}

$\{ z_5 = z_4 - 1,\ z_5 = z_4 + 1 \}$

SAT solver

Choose candidate program of ***size*** $= \mathbf{1}$

Blocking clause for similar assignments

SAT (not a repair)

SMT solver

$z_4 = g_1 ?\, z_2 : z_3$

$z_6 = b_1 ?\, z_5 : z_4$

$z_6 \leq 8$

$g_1 = x_1 - y_1 > 8$

$z_2 = x_1 + y_1$

$z_3 = 9$

$b_1 = z_4 \geq 9$

$z_5 = z_4 - 1$

Repair

c_1 c_2

SAT solver

Blocking clause for "similar" assignments

- ***Assignments causing a similar bug***

c_9 c_{10}

$\{z_5 = z_4 - 1, z_5 = z_4 + 1\}$

$z_6 \leq 8$

$b_1 = z_4 \geq 9$

$z_5 = z_4 - 1$

Repair

c_1 c_2

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

c_3

c_4 c_5

$\{z_2 = x_1 + y_1, z_2 = x_1 - y_1\}$

c_6

$\{z_3 = 9\}$

c_7 c_8

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

c_9 c_{10}

$\{z_5 = z_4 - 1, z_5 = z_4 + 1\}$

SAT solver

Choose candidate program of ***size*** $= \mathbf{1}$

Blocking clause for similar assignments

SMT solver

$z_4 = g_1 ? z_2 : z_3$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

$g_1 = x_1 + y_1 > 8$

$z_2 = x_1 + y_1$

$z_3 = 9$

$b_1 = z_4 > 9$

$z_5 = z_4 - 1$

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

c_1 c_2

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

c_3

c_4 c_5

$\{z_2 = x_1 + y_1, z_2 = x_1 - y_1\}$

c_6

$\{z_3 = 9\}$

c_7 c_8

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

c_9 c_{10}

$\{z_5 = z_4 - 1, z_5 = z_4 + 1\}$

SAT solver

Choose candidate program of ***size*** $= \mathbf{1}$

Blocking clause for similar assignments

SAT

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

UNSAT (repair found!)

SMT solver

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

$\boldsymbol{g_1 = x_1 + y_1 > 8}$
$\boldsymbol{z_2 = x_1 + y_1}$
$\boldsymbol{z_3 = 9}$
$\boldsymbol{b_1 = z_4 > 9}$
$\boldsymbol{z_5 = z_4 - 1}$

Repair

c_2

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

c_3

c_4 c_5

$\{z_2 = x_1 + y_1, z_2 = x_1 - y_1\}$

c_6

$\{z_3 = 9\}$

c_7 c_8

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

c_9 c_{10}

$\{z_5 = z_4 - 1, z_5 = z_4 + 1\}$

Choose candidate program of ***size*** $= 1$

Blocking clause for similar assignments

Blocking clause for this assignment

And ***all other supersets of changes***

UNSAT (repair found!)

SMT solver

$z_4 = g_1 ? z_2 : z_3$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

$g_1 = x_1 + y_1 > 8$

$z_2 = x_1 + y_1$

$z_3 = 9$

$b_1 = z_4 > 9$

$z_5 = z_4 - 1$

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

c_1 c_2 $SAT\ solver$

$Blocking\ clause\ for\ this\ assignment$

and all other supersets of changes

- ***Repairs that are not minimal***

Repair

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

c_3

c_4 c_5

$\{z_2 = x_1 + y_1, z_2 = x_1 - y_1\}$

c_6

$\{z_3 = 9\}$

c_7 c_8

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

c_9 c_{10}

$\{z_5 = z_4 - 1, z_5 = z_4 + 1\}$

Choose candidate program of ***size*** $= \mathbf{1}$

Blocking clause for similar assignments

Blocking clause for this assignment

And ***all other supersets of changes***

UNSAT (repair found!)

SMT solver

SAT

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

UNSAT

$z_4 = g_1 ? z_2 : z_3$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

$\boldsymbol{g_1 = x_1 + y_1 > 8}$

$\boldsymbol{z_2 = x_1 + y_1}$

$\boldsymbol{z_3 = 9}$

$b_1 = z_4 > 9$

$z_5 = z_4 - 1$

Repair

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

c_3

c_4 c_5

$\{z_2 = x_1 + y_1, z_2 = x_1 - y_1\}$

c_6

$\{z_3 = 9\}$

c_7 c_8

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

c_9 c_{10}

$\{z_5 = z_4 - 1, z_5 = z_4 + 1\}$

Choose candidate program of ***size*** $= 1$

Blocking clause for similar assignments

Blocking clause for this assignment

And ***all other supersets of changes***

UNSAT (repair found!)

SMT solver

$z_4 = g_1 ? z_2 : z_3$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

$g_1 = x_1 + y_1 > 8$

$z_2 = x_1 + y_1$

$z_3 = 9$

$b_1 = z_4 > 9$

$z_5 = z_4 - 1$

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

c_1 c_2

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

c_3

c_4 c_5

$\{z_2 = x_1 + y_1, z_2 = x_1 - y_1\}$

c_6

$\{z_3 = 9\}$

c_7 c_8

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

c_9 c_{10}

$\{z_5 = z_4 - 1, z_5 = z_4 + 1\}$

Choose candidate program of ***size = 2***

Blocking clause for similar assignments

Blocking clause for this assignment
And ***all other supersets of changes***

UNSAT (repair found!)

SMT solver

SAT

UNSAT

$z_4 = g_1 ? z_2 : z_3$

$z_6 = b_1 ? z_5 : z_4$

$z_6 \leq 8$

$g_1 = x_1 + y_1 > 8$

$z_2 = x_1 + y_1$

$z_3 = 9$

$b_1 = z_4 > 9$

$z_5 = z_4 - 1$

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

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

```
int f(int x, int y){
  int z;
  z = x
  if (x >= 0) {
      x = x + 1; y = x + 2;
  } else {
      z = 9;
  }
  assert(z > 0);
  return z;
}
```

Fault localization: example

		erroneous run:
	$int\ f(int\ x, int\ y)\{$	
1.	$int\ z; int\ t;$	$x=0, y=0$
2.	$z = x$	$z=0$
3.	$if\ \ (x >= \mathbf{0})\ \{$	
4.	$\quad x = x + 1; y = x + 2;$	$x=1, y=2$
5.	$\}\ else\ \{$	
6.	$\quad z = 9;$	
	$\}$	
7.	$assert(z > 0);$	$z=0$
8.	$return\ z;$	
	$\}$	

Repair: line 3 should change to (x > 0)

Fault localization by slicing

	Code	execution slice	dynamic slice	our slice
	$int\ f(int\ x, int\ y)\{$			
1.	$int\ z;\ int\ t;$			
2.	$z = x$	•	•	•
3.	$if\ \ (x >= \mathbf{0})\ \{$	•		•
4.	$\boldsymbol{x = x + 1; y = 0;}$	•		
5.	$\}\ else\ \{$			
6.	$z = 9;$			
	$\}$			
7.	$assert(z > 0);$	•	•	•
8.	$return\ z;$			
	$\}$			

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

Ver.	Method of [11]		Method of [12]		Our method: Mutation level 1		Our method: Mutation level 2	
	Fixed?	Time[s]	Fixed?	Time[s]	Fixed?	Time[s]	Fixed?	Time[s]
28	+	34	+	35			+	93.678
31					+	1.246	+	4.661
32					+	1.902	+	85.349
35	+	41	+	46			+	92.866
36	+	8	+	6			+	94.599
39	+	82	+	101	+	2.558	+	16.393
	16 (39%)	38	15 (36.6%)	38	11 (26.83%)	2.278	18 (43.9%)	48.151

		Level 1	Level 2
Op. replacement	Arithmetic	$\{+, -\}, \{*, /, \%\}$	$\{+, -, *, /, \%\}$
	Relational	$\{>, >=\}, \{<, <=\}$	$\{>, >=, <, <=\}, \{==, !=\}$
	Logical	$\{\|\|, \&\&\}$	
	Bit-wise	$\{>>, <<\}, \{\&, \|, \hat{}\}$	
Constant manipulation			C→C+1,C→C−1, C→ −C,C→0

FoREnSiC.

Adding fault localization

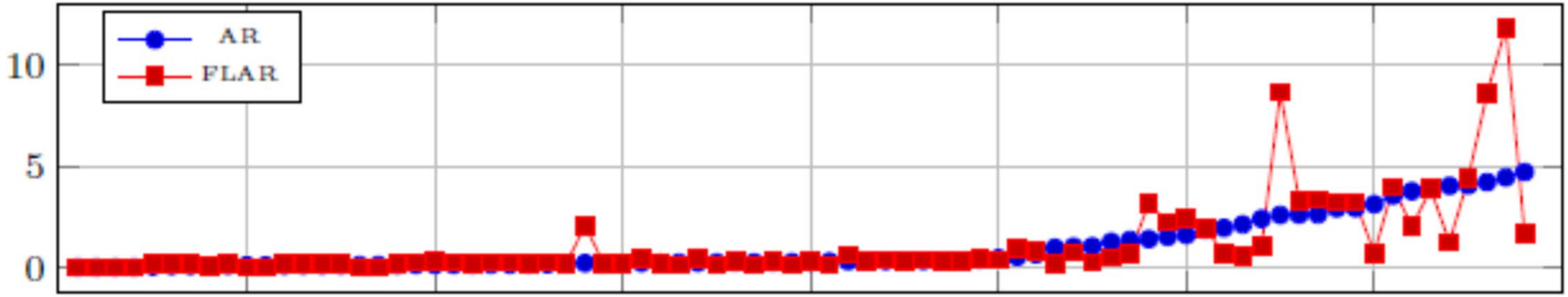
Every generate-validate iteration with fault localization is more expensive

- But we expect to have less iterations

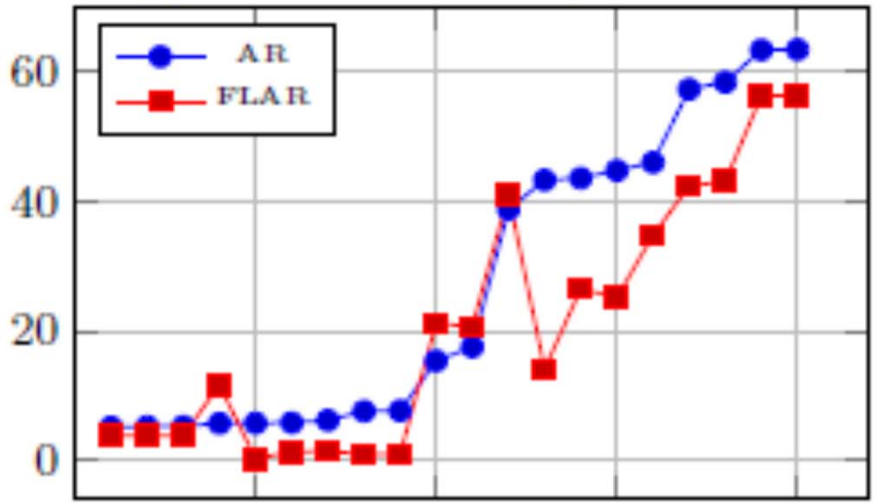
Both AllRepair and FL-AllRepair are complete

- return the same set of repaired programs
- Not necessarily in the same order

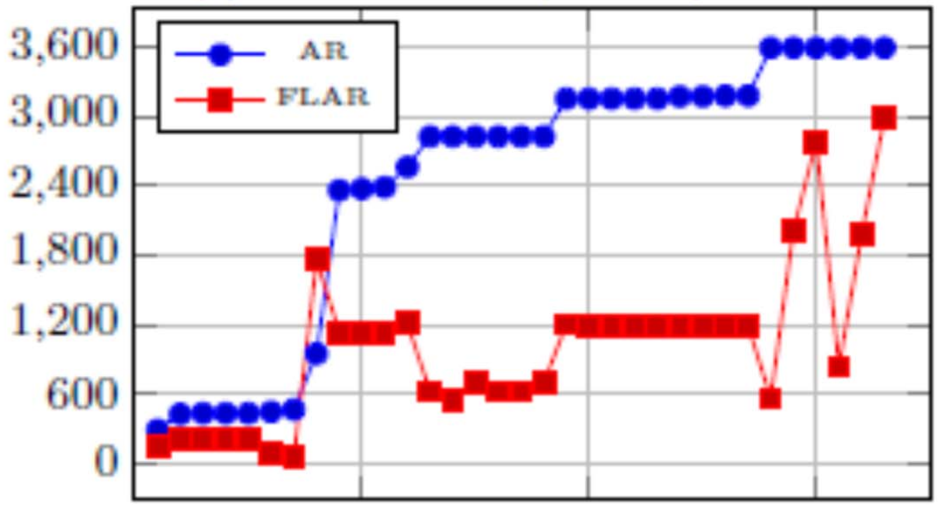
(a) Fast repairs (< 5s)



(b) Medium repairs ($5 - 240s$)



(c) Slow repairs ($> 240s$)



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]

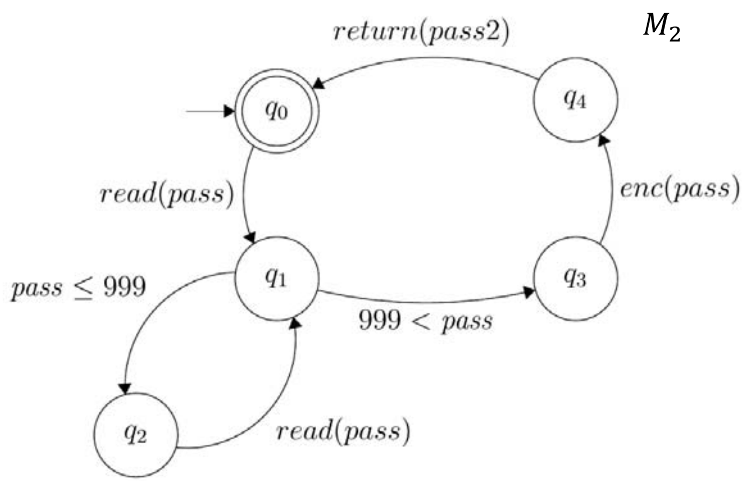
Motivation

- 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

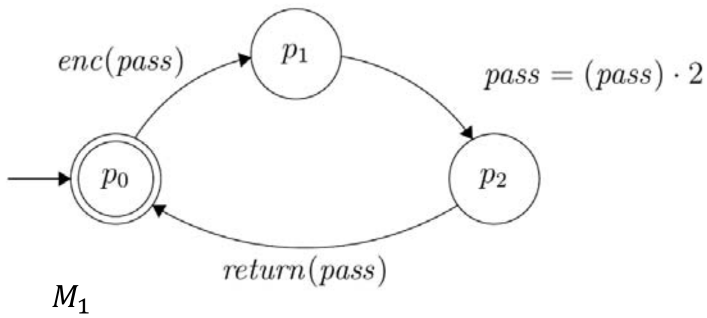
- C-like programs
- Described as a control-flow graph (automaton)
- Use automata learning algorithms

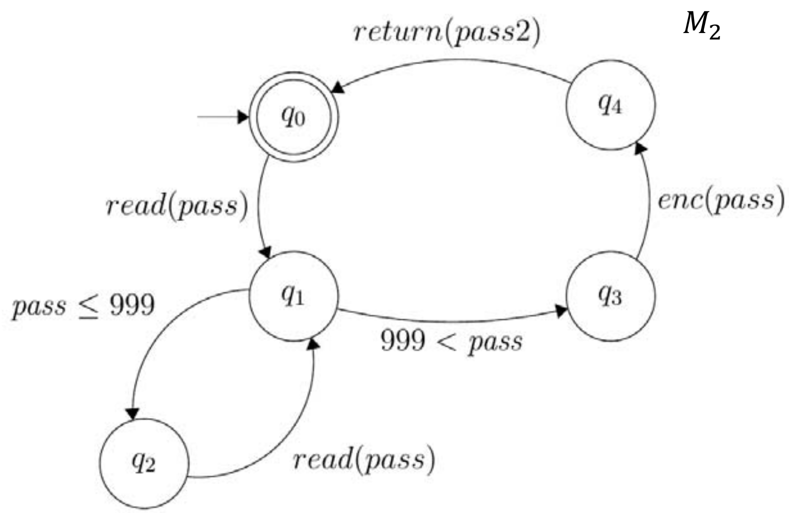
```
while (true)
    pass = readInput;
    while (pass ≤ 999)
     pass = readInput;
    pass2 = encrypt(pass);
    return pass2;
```



Example

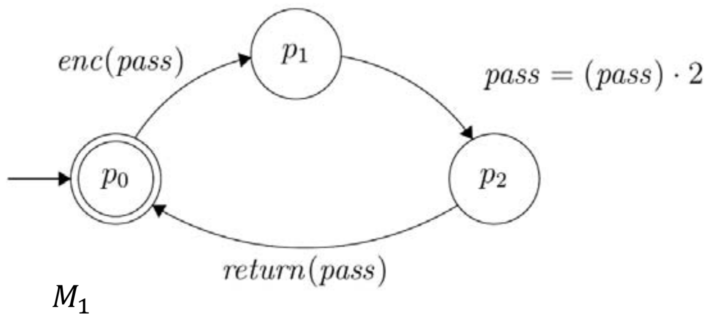
- Components synchronize over common channels

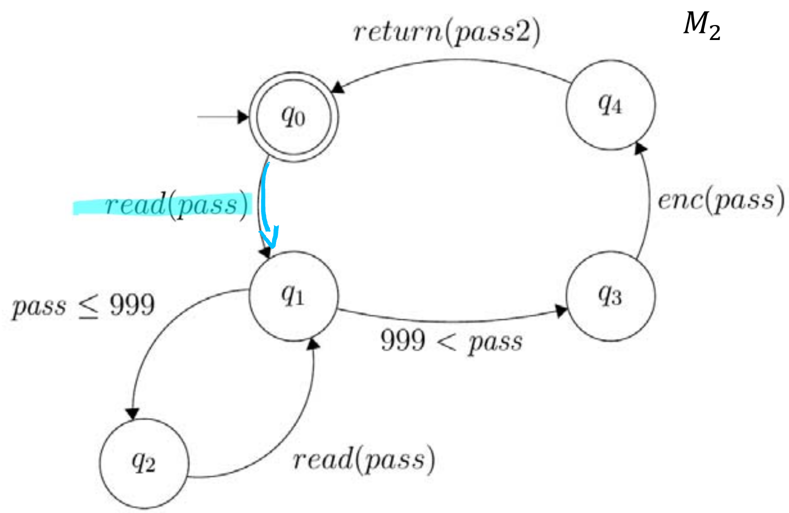




Example

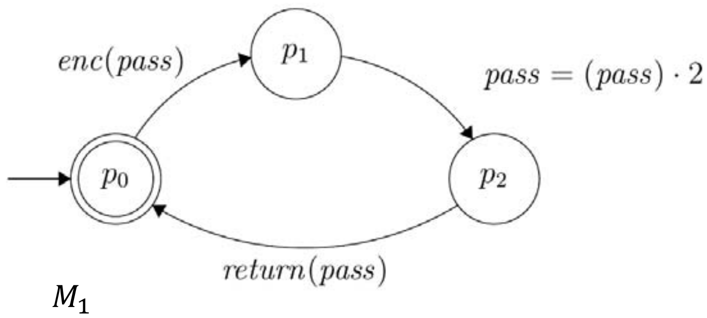
- Components synchronize over common channels

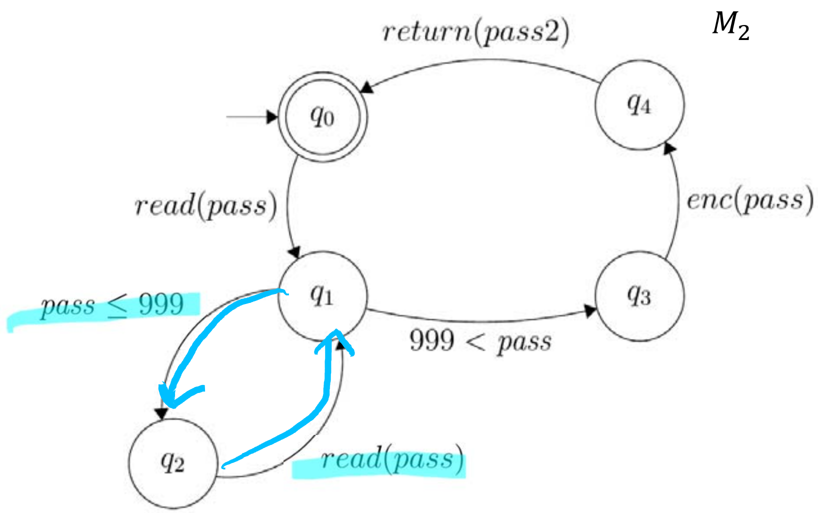




Example

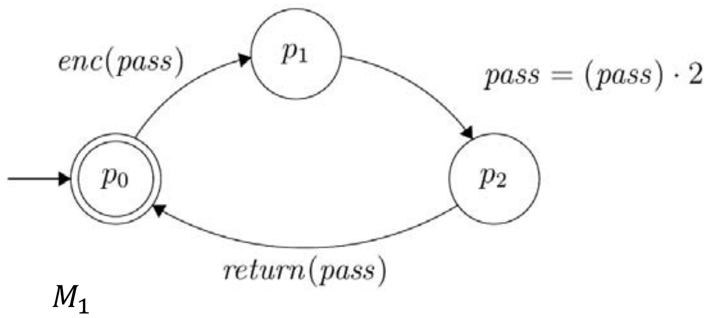
- Components synchronize over common channels

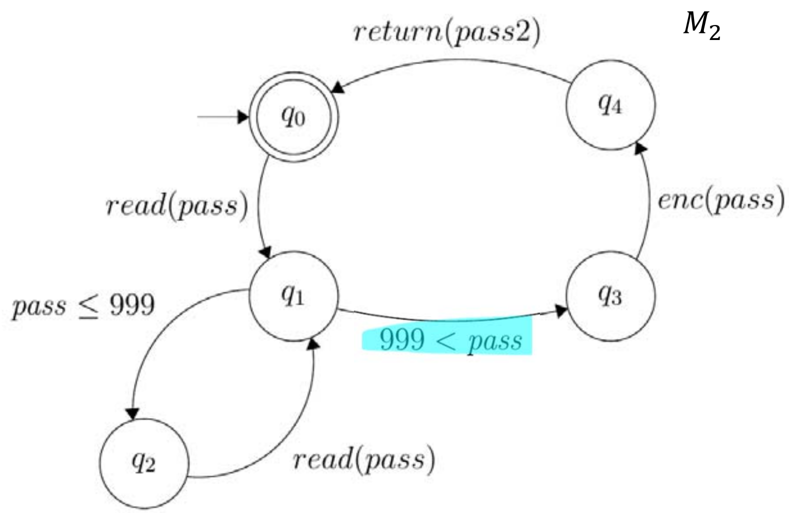




Example

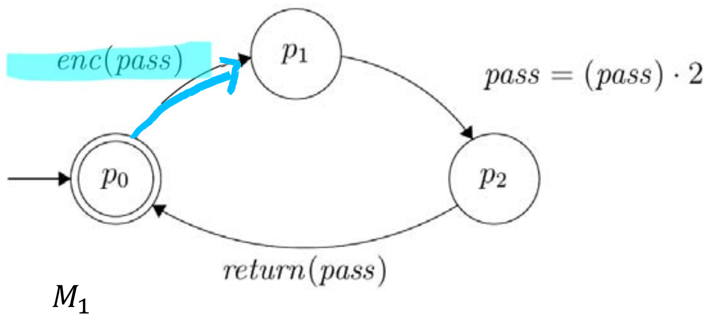
- Components synchronize over common channels

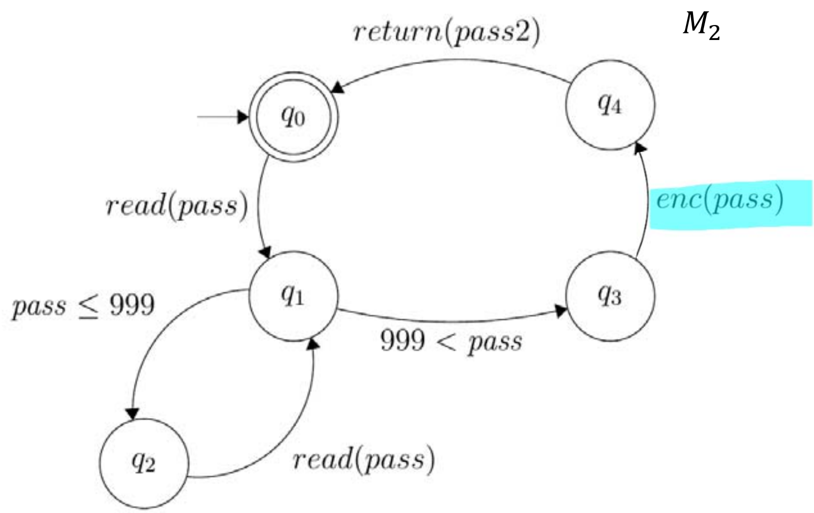




Example

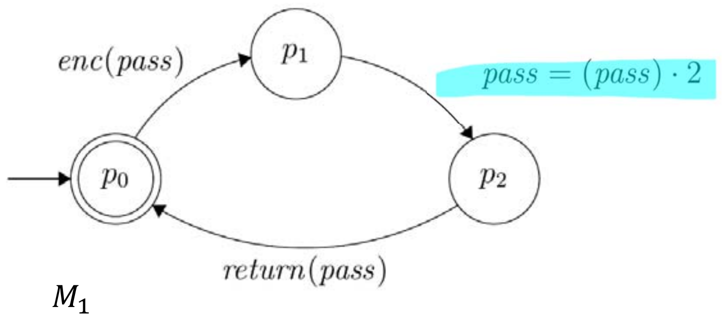
- Components synchronize over common channels

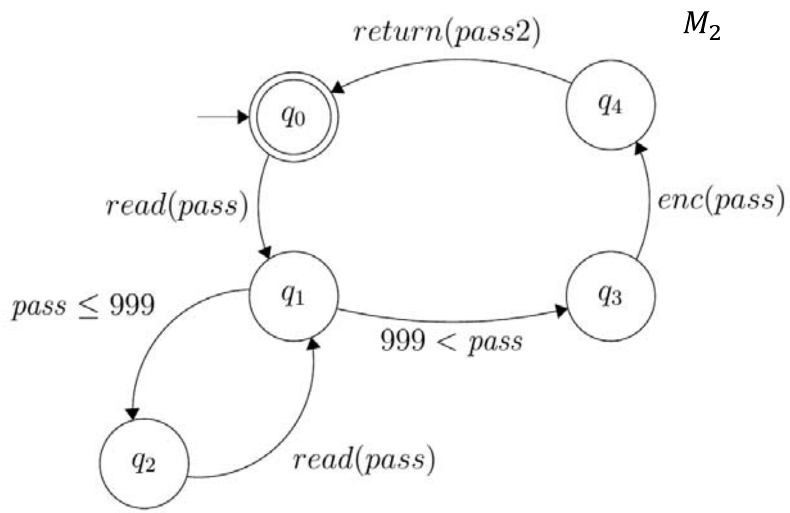




Example

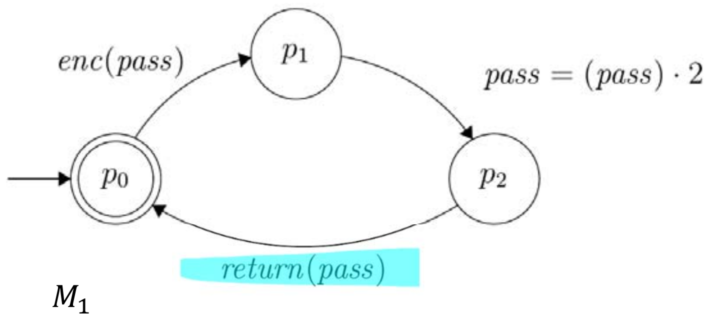
- Components synchronize over common channels

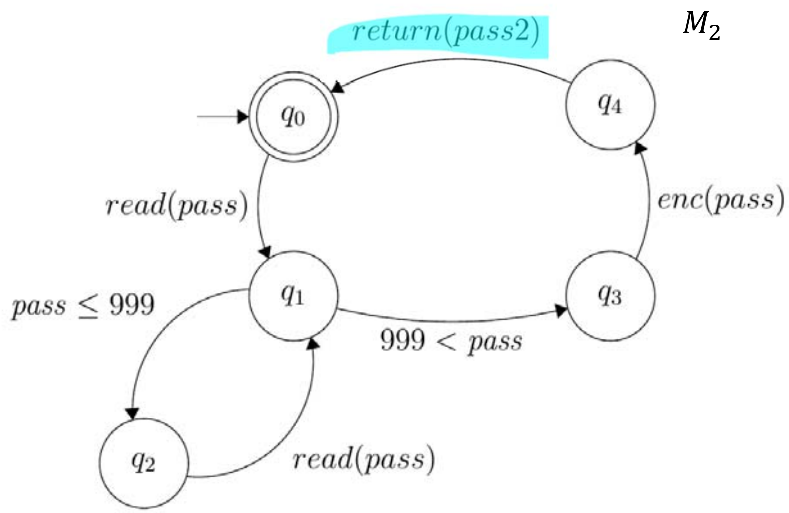




Example

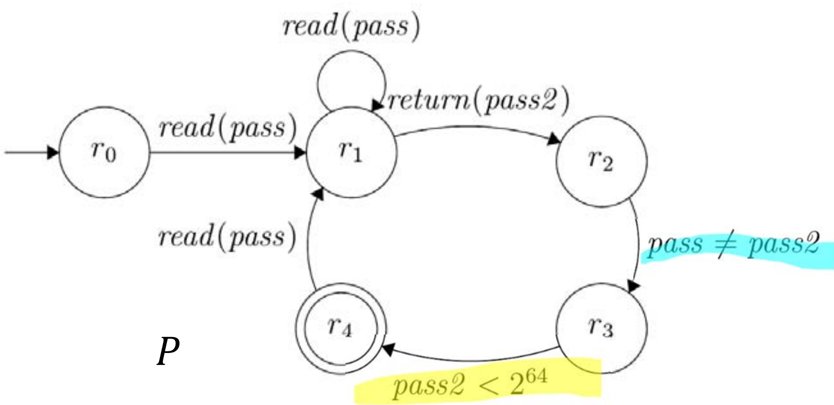
- Components synchronize over common channels





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"



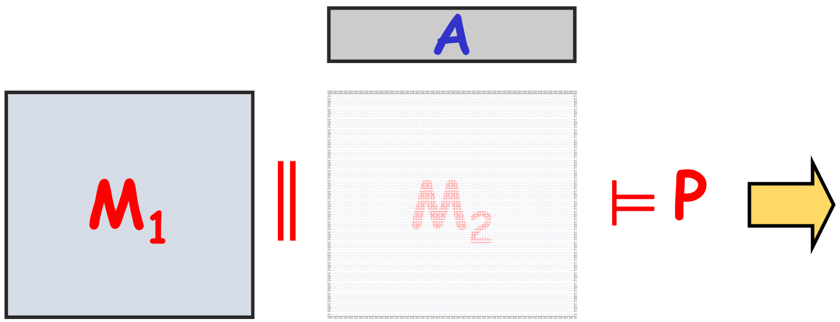
P

Compositional Verification

- **Inputs:**
  - composite system $M_1 \| M_2$
  - property P
- **Goal**: check if $M_1 \| M_2 \models P$

Useful AG Rule

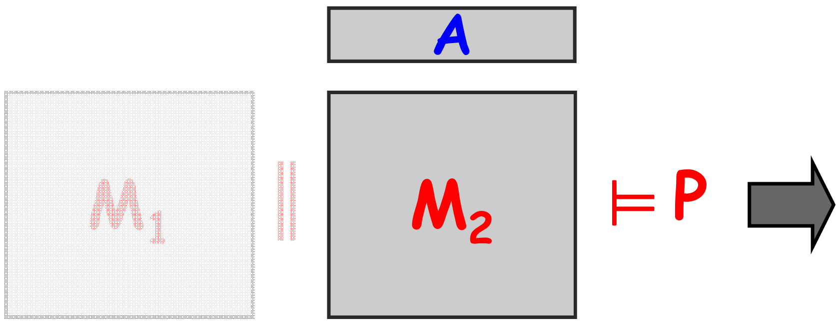
1. check if a component M_1 **guarantees** P when it is a part of a system satisfying **assumption** *A*



$$\frac{A \parallel M_1 \models P}{M_1 \parallel M_2 \models P}$$

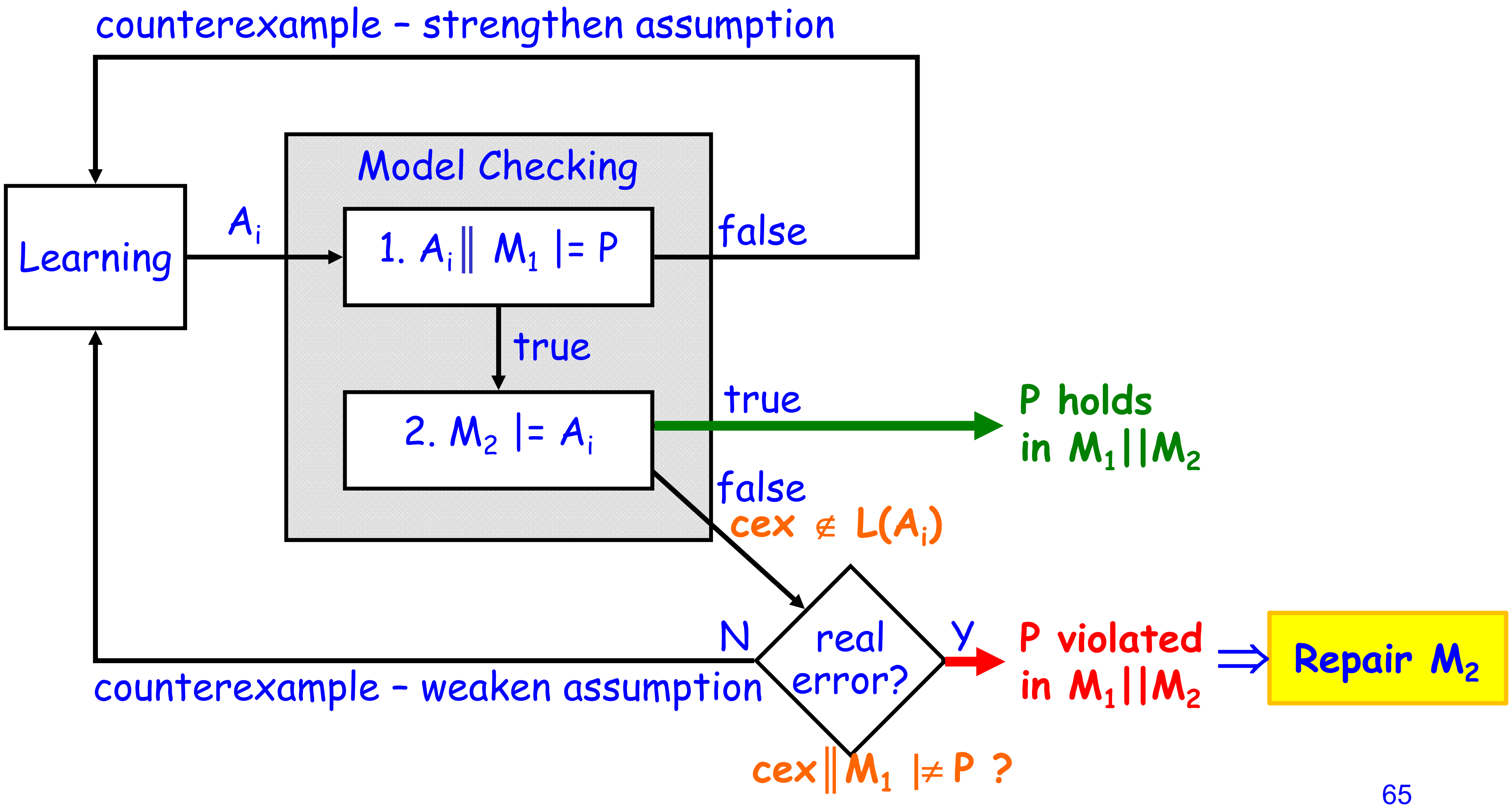
Useful AG Rule for Safety Properties

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.



$$\frac{A \parallel M_1 \models P \qquad M_2 \models A}{M_1 \parallel M_2 \models P}$$

Assume Guarantee or Repair



Semantic repair

- The counterexample contains constraint
- Goal:
  to make the counterexample infeasible by adding another constraint $\mathcal{C}$ to it

- Using abduction

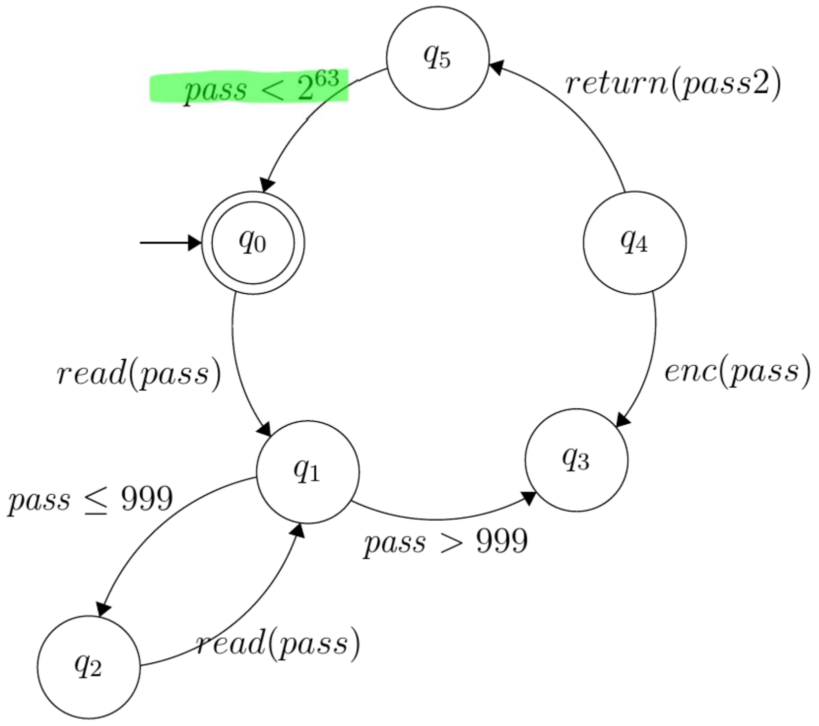
Semantic repair

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

- $\mathcal{C} := \forall pass2\,[\,pass > 999 \wedge pass2 = pass \cdot 2 \rightarrow pass2 < 2^{64}\,]$
- After quantifier elimination & simplification: $\mathcal{C} = \boldsymbol{pass} < \mathbf{2^{63}}$.

Abduction - "Logical Magic"

Semantic Repair




```
while (true)
    pass = readInput;
    while (pass ≤ 999 or pass ≥ 2^63)
     pass = readInput;
    pass2 = encrypt(pass);
    return pass2;
```

Syntactic repair

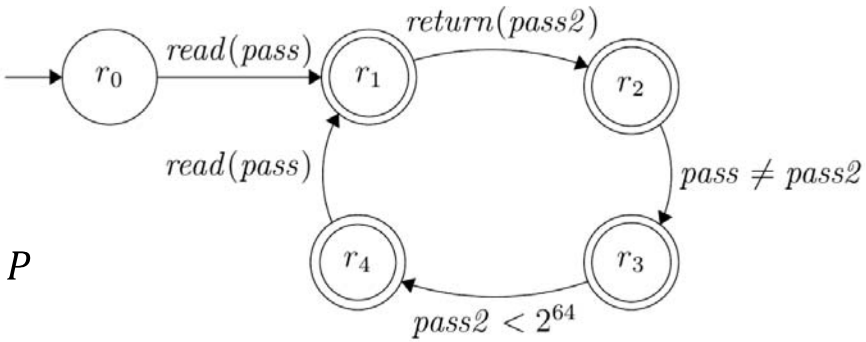
- The counterexample t contains no constraint
  - It consists of communication actions and assignments
- Abduction will not help

3 methods to removing counterexample t:

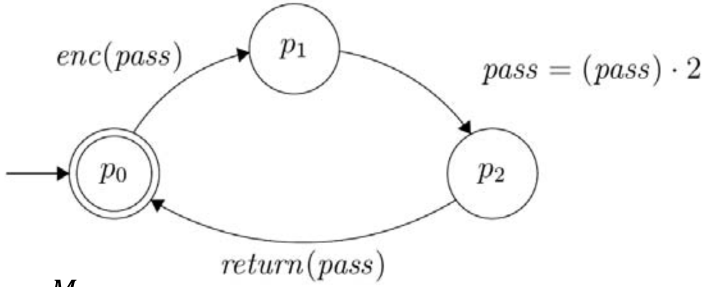
- Exact: remove exactly t from M_2
- Approximate:
- Aggressive:

Example – Syntactic Repair

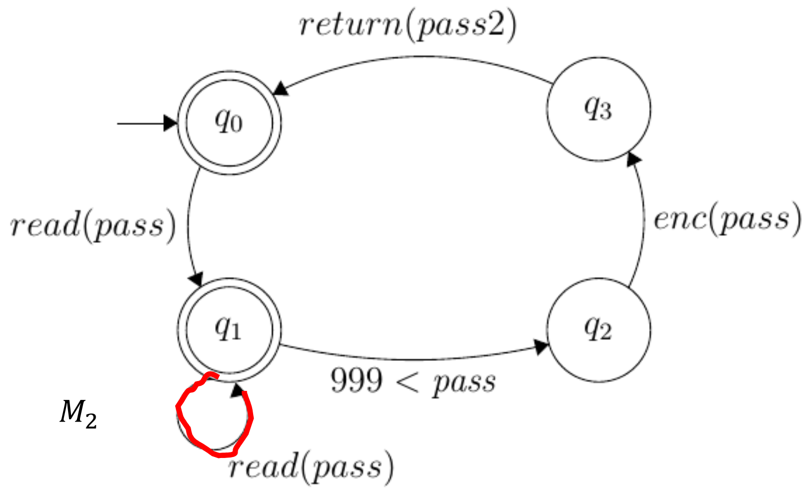
No self loop, cannot *read* more than once each time!



P



M_1

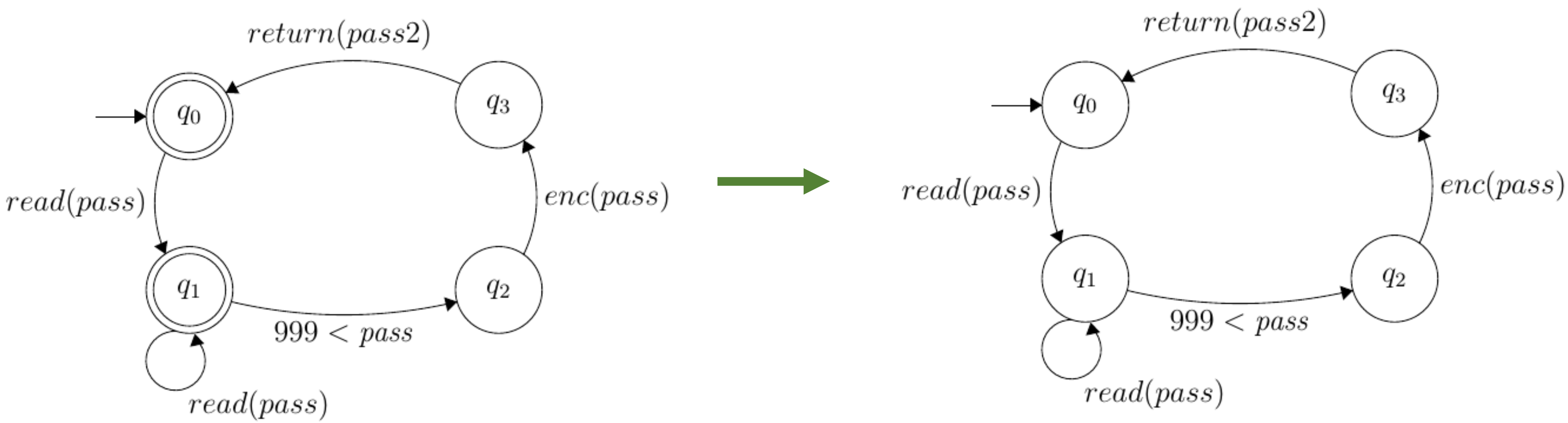


M_2

Multiple reads are allowed

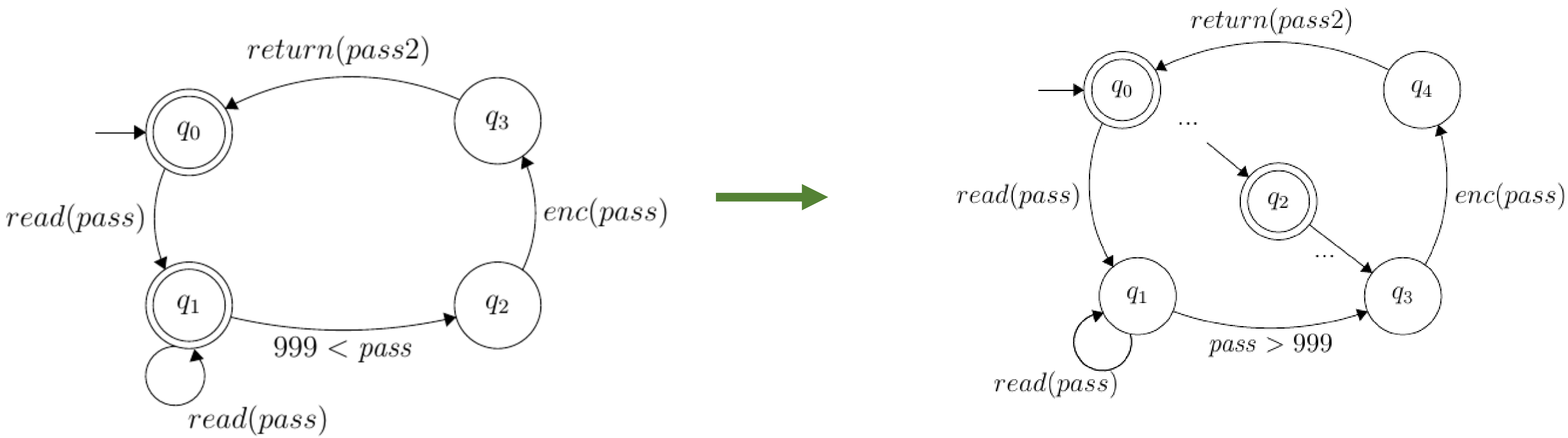
Agressive Repair

- Remove accepting states (can make the language of M_2 empty)



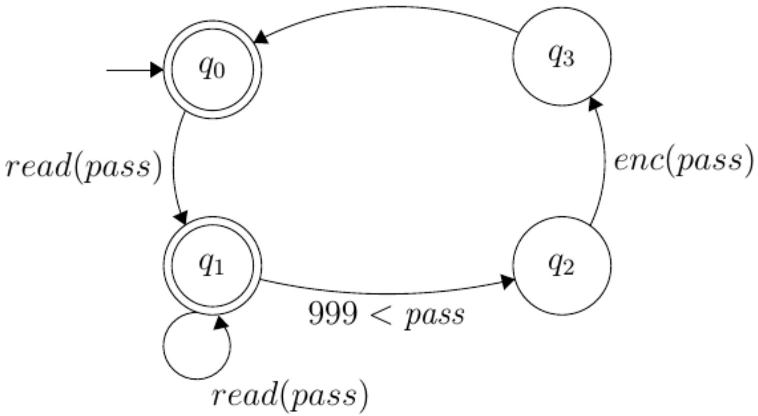
Approximate Repair

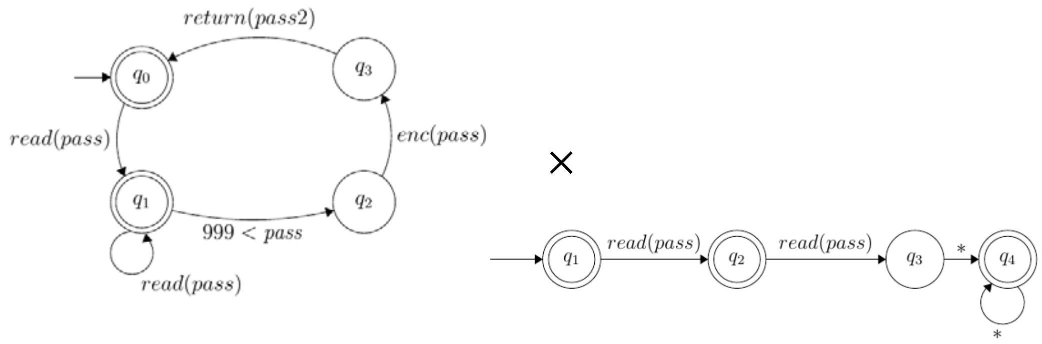
- Add an intermidiate state to eliminate bad traces



Exact Rapair

- Remove bad traces one by one
- First bad trace spotted is $read(pass),\ read(pass)$

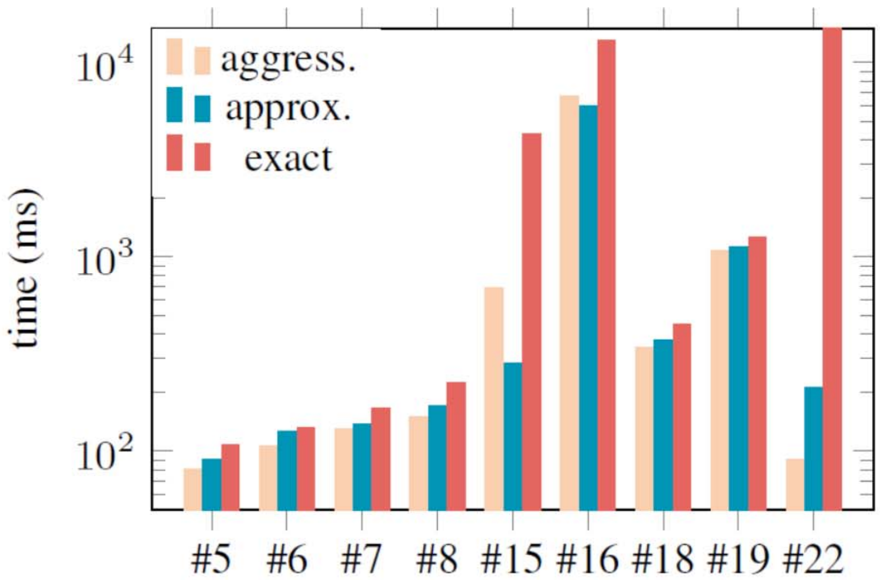


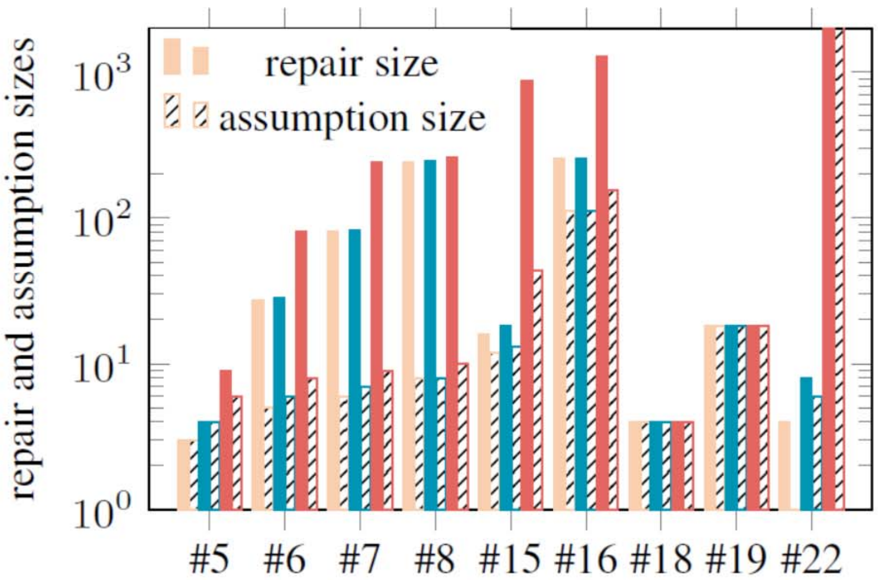


AGR Results on Various Examples

Example	M_1 Size	M_2 Size	P Size	Time (sec.)	A size	Repair Size	Repair Method	#Iterations
#4	64	64	3	95	7	verification		
#6	2	27	2	0.106	5	27	aggress.	2
				0.126	6	28	approx.	2
				0.132	8	81	exact	2
#7	2	81	2	0.13	6	81	aggress.	2
				0.138	7	82	approx.	2
				0.165	9	243	exact	2
#8	2	243	2	0.15	8	243	aggress.	2
				0.17	8	244	approx.	2
				0.223	10	729	exact	2
#11	5	256	6	4.88	92	verification		
#14	5	256	6	4.44	109	verification		
#15	3	16	5	0.69	12	16	aggress.	5
				0.28	13	18	approx.	3
				4.27	44	864	exact	5
#16	4	256	8	6.63	113	256	aggress.	2
				5.94	113	257	approx.	2
				12.87	155	1280	exact	2
#19	3	16	5	1.07	18	18	aggress.	3
				1.12	18	18	approx.	3
				1.26	18	18	exact	3
#22	2	4	2	0.09	1	4 (trivial)	aggress.	4
				0.21	6	8	approx.	5
				timeout			exact	timeout

Comparing Repair Methods (logarithmic scale)





#15, #16, #18, #19 apply also abduction

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