# Introduction

Software TEST

- Primary method to test correctness of software
- FAULT BASED TEST
  - Test Data Generation
    - Automated (Divide&Conquer)
    - By Hand



# Background

Approach to study Test Case

- Reliability - Program correct for all input
  - Problem: no effective procedure for generating Test Set of finite size
- Adequacy – Test set causes all possible fault
  - Problem: Test Case can't distinguish correct program from all possible incorrect

# Background (cont'd)

- Relative Adequacy – Test set cause finite number of possible fault
  - Important: Programmers don't produce necessary programs that are correct – (Bugs)
- Generator
  - produces adequate test set
- Acceptor
  - return TRUE if Test Set is adequate



# Test Data Generator

Mothra

- Use mutant operators
  - Test Data that cause fault in mutant is said "Kill Test Data)

Test Data Generator

- Godzilla
  - Constraint Based Technique (CBT)
  - Mutant dies as simple agebraic constraint

# Mutation

## Mutant

- Simple sintactic changes to test program
- Often this change represent common programmer's mistake

```
      FUNCTION MAX (M,N)
1     MAX = M
Δ     MAX = N
Δ     MAX = ABS (M)
2     IF (N .GT. M) MAX = N
Δ     IF (N .LT. M) MAX = N
Δ     IF (N .GE. M) MAX = N
3     RETURN
```



# Mutation (cont'd)

Equivalent Mutants produce same output as the original and they can never be killed

## Equivalent Mutar

- Human
- Heuristic

## Goal

Test case differentiates all mutant by causing the mutant to produce different output

## Adequacy-Based Test case Constraint

Necessity Condition

- The state of statement S is different from state of statement M after esecution
- Incorrect state may be not guarantee to be sufficient to kill mutant (i.e. equal output)
  - If the states of two programs are the same, mutant will not be killed
  - If the states are different **weak** mutation can't guarantee either that mutant will live or die



## Adequacy-Based Test case Constraint (cont'd)

Discovering error

- Global extent (error output) – More desirable
- Local extent (local error) – More practical

# Adequacy-Based Test case Constraint (cont'd)

Path-Analysis Based Approach

- Potential Fault is a discrepancy between the program being tested and a hypotetically correct version of the program
  - (like as Mutants in Mutation Analysis)
- Potential Fault originates if smallest expression, containing that, evaluates incorrectly.



# Adequacy-Based Test case Constraint (cont'd)

- Potential Fault transfers to "*super expression*" which references the erroneus expression if the value of "*super expression*" is also incorrect
- For the Potential Fault to be detected, the incorrect evaluation must transfer through to cause incorrect output from the program

## Adequacy-Based Test case Constraint (cont'd)

### Conclusion

Path-Analysis Based Approach seems similar in intent to CBT, but differs in method.

CBT is besed explicitly on mutations and the method for generating test cases focuses on killing mutants



## Satisfying Sufficient Condition

### Necessity Costraint

- Test case ensures different state of a program

Many of mutation types used by MOTHRA cause one program symbol to be replaced by another one.

In this way the value of original symbol differs from the value of the replaced program symbol

# Satisfying Sufficient Condition (cont'd)

TABLE I
CONSTRAINT TEMPLATES

| Type | Description | Constraint |
|---|---|---|
| aar | array for array replacement | $A(e_1) \neq B(e_2)$ |
| abs | absolute value insertion | $e_1 < 0, e_1 > 0, e_1 = 0$ |
| acr | array constant replacement | $C \neq A(e_1)$ |
| aor | arithmetic operator replacement | $e_1 \rho\ e_2 \neq e_1 \circ e_2$<br>$e_1 \rho\ e_2 \neq e_1$<br>$e_1 \rho\ e_2 \neq e_2$<br>$e_1 \rho\ e_2 \neq Mod(e_1, e_2)$ |
| asr | array for variable replacement | $X \neq A(e_1)$ |
| car | constant for array replacement | $A(e_1) \neq C$ |
| cnr | comparable array replacement | $A(e_1) \neq B(e_2)$ |
| csr | constant for scalar replacement | $X \neq C$ |
| der | DO statement end replacement | $e_2 - e_1 \geq 2$<br>$e_2 \leq e_1$ |
| lcr | logical connector replacement | $e_1 \rho\ e_2 \neq e_1 \circ e_2$ |
| ror | relational operator replacement | $e_1 \rho\ e_2 \neq e_1 \circ e_2$ |
| sar | scalar for array replacement | $A(e_1) \neq X$ |
| scr | scalar for constant replacement | $C \neq X$ |
| svr | scalar variable replacement | $X \neq Y$ |



# Satisfying Sufficient Condition (cont'd)

## Predicate Constraint

- Causes differences in program predicate and not only in variables values
- Necessity – often results are the same as the original ones
  - It's necessary to have different values than the corresponding in original

( e ≠ e' )

# Constraint satisfaction

Value generation

- Randomly
- Heuristic (i.e. Godzilla)
  - Simplification: constraint linear, few variables, more constant than variables



# Constraint satisfaction (cont'd)

Constraint Representation in Godzilla

- Algebric Expression
  - From testing program
  - Right side of assignment
- Constraint
  - Pair of Algebric Expressions related by logical operands
- Clause
  - List of constraints connected by AND (conjunctive) and OR (disjunctive)
  - Godzilla uses Disjunctive Normal Form

# Constraint satisfaction (cont'd)

## Domain reduction in Godzilla

Finding variables in a constraint consistent with other (remaining) constraint
- Domain is reduced by all constraints

When no more simplifications are available, heuristic mode sets the value of remaining variables in the constraints (using back sostitution)

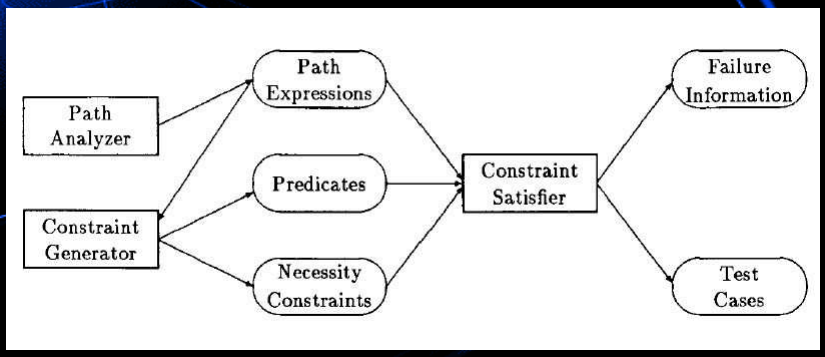
The variable chosen is variable with the smallest current domain. The value is chosen arbitrary
- WARNING: Infeasibility (i.e. [x=4 AND x>8] )



# Godzilla

- Developed by C language
- 15,000 lines of code
- Structured by
  - Path Analyzer
  - Constraint Generator
  - Constraint Satisfier

# Godzilla (cont'd)






# Godzilla (cont'd)

## Path Analyzer

- Creates path expression constraint
- If test statement reaches that statement the Constraint will be TRUE

## Constraint Generator

- Constructs necessity and predicate Constraint

# Godzilla (cont'd)

Constraint Satisfier

- Conjoins constraints (each ones) with appropriate path expression and generates test cases
- If satisfier fails (no test case product), satisfier can supply information about fault to tester



# Conclusion

The development of this technology is still in it's infancy, yet it already seems to provide better test data than other testing techniques.

This is not because it's revolutionary, but because it's evolutionary

Using Constraint to develop test data allows us to combine path-coverage techniques and symbolic execution with mutation analysis

# Conclusion (cont'd)

For example, methods,  to use contstraints to detect equivalent mutants and to analyze software specification, are developing nowaday

The ultimate goal is to completely automate the software testing process, to reduce the human tester's work.