Program Representations

Abstract Syntax Tree

- Created by compiler at end of syntax analysis phase
- Tree representation for the abstract syntactic structure of source code
  - Node: construct such as a statement or loop
  - Edge: containment relationship
- Basis for code generation and manipulation features
- Can use AST to parse information or perform insert/delete

Control Flow Graph

- Representation using graph notation of all paths that a program during execution
- Formally: CFG = <V, E, Entry, Exit>
  - V = representing an instruction or basic block (a group of instructions)
  - E = control flow
  - Entry = unique program entry
  - Exit = unique program exit
- Constructing CFGs:
  - Creating Basic Blocks:
    - Identify Leaders – first instruction of a basic block
      - First instruction in program
      - Target of branch
      - Instruction following branch (implicit target)
    - In instruction order, construct a block by appending subsequent instructions up to, but not including, the next leader
  - Creating Edges:
    - For the last instruction x of each block i
    - If x is a branch
      - For each branch target y create edge $i \to y$
    - Otherwise create edge from $i \to i + 1$

Loop unrolling:

- Expanding control flow graph for all iterations of a loop.

Program Differencing

- What changes have been made by developers
- Use AST to represent code then use ChangeDistiller algorithm to compare trees for source code differences

ChangeDistiller

- Four change types:

    - Insert a leaf node
    - Delete a node from parent
    - Move a node to another parent
    - Update the value of the node
- Identify and match unchanged nodes
    - Many ways to perform matching, for example:
        * For leaves: true if the labels match AND similarity of the values beats a threshold
        * For inner nodes: true if labels match AND the number of shared children beats a threshold

Challenges:

- Missed string match: consider verticalDrawAction vs drawVerticalAction
- Magic threshold number: if threshold is set too high it misses matches if its set too low it will make too many unrelated matches
- Multiple candidate matches: multiple nodes can match which one to pick?

N-Grams String Similarity

- divide strings into all n character length substrings
- $sim_{ng}(s_a, s_b) = \frac{2 \times |n-grams(s_a) \cap n-grams(s_b)|}{|n-grams(s_a) \cup n-grams(s_b)|}$

Inner Node Weighting

- Inner node match based on value and descendent leaf node values

Best Match

- Consider leaf node x which has candidate matches $n_1, n_2, ...n_m$
- $bestMatch(x) = \{n_b | sim(x, n_b) \geq sim(x, n_k) \text{for} b \neq l \in [1, m]\}$

Final Algorithm

```
Input: trees T1 , T2
Output: matching set M
Algorithm:
    match leaf nodes of T1 and T2 , save all candidate leaf matches
    sort matches in descending order of similarity values
    for each candidate leaf match
        if both nodes are "unmatched" then
            put them into M,
            mark them as "matched"
    endfor
    for each unmatched node x in T1
        find inner node match in T2
    endfor
```

Evaluation

- Gather ground truth dataset
- Measure error, precision, recall
  - Improve precision: reduce false positive rate (increase confidence threshold)
  - Improve recall: reduce false negative rate (decrease confidence threshold)
- Execution time

Grammar & Parsing

Create a parse tree using defined grammar rules

```
"<start>": ["<phone-number>"],
"<phone-number>": ["(<area>)<exchange>-<line>"],
"<area>": ["<lead-digit><digit><digit>"],
"<exchange>": ["<lead-digit><digit><digit>"],
"<line>": ["<digit><digit><digit><digit>"],
"<lead-digit>": ["2", "3", "4", "5", "6", "7", "8", "9"],
"<digit>": ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
```

- Non-Terminal Nodes: represent grammar rules
- Terminal Nodes: leaf node represents actual value

Code Search

Idea we want to search two versions of codebases to find changes

- User specifies template to determine how changes are reported
- Use AST to generate trees for both versions

Testing & Verification

Testing: Show that the program is correct on a set of inputs

- Show the presence of errors

Verification: Show that the program is correct on all inputs

- Proving the absence of errors

Types of Testing

- Unit testing: complete class, routine or small program
- Component testing: class, package, small program, or other program element

- Integration testing: combined two or more classes, packages, components, or subsystems
- System testing: software in final configuration including integration with other software
- Regression testing: repetition of test cases for finding defects
- Black-box: tests which can't see the inner workings of the item being tested
- White-box: tests is aware of the inner workings of the item being tested
- Grey-box: tests generated by test coverage

Testing Coverage

- Statement coverage: how many statements are exercised by tests?
- Branch coverage: how many possible branch evaluations are exercised?
- Path coverage: How any possible routes are exercised by tests?

Infeasible Paths

Infeasible path: path that cannot be executed

Measuring the number of paths in a program: assuming branches have two pathss

- With b branches with no infeasible paths: 2^b
- With loops that execute branches a total of $(n * b)$ times: $2^{(n*b)}$

Symbolic Execution

- Assign new input variables
- Unroll loops
- Create logical constraints
- Consider a different branch evaluation outcome as a disjunction
- For each branch, we propagate the constraints for both true and false evaluation of the branch (Basically, we are conservatively estimate the effect of taking either path).

Symbolic execution tree: control flow graph expanded as a tree

```
x = 0
if (x > 1) {
    x++
}
else {
    x--
}
```

path condition: $x > 1$, effect: $x' == x + 1$

path condition: $x \leq 1$, effect: $x' == x - 1$

Constraint: $x = 0$

Symbolic representation: $(x > 1 \land x' == x + 1) \lor (x \leq 1 \land x' == x - 1)$

The left-hand side is false and creates a contradiction.

Example

```
func(x) {
    value = 0
    y = x ^ 2

    if (x > 3) {
        x++
    }
    else {
        x = abs(x)
    }

    if (2 * x - 1 > y) {
        y = 2 * y
    }
    else {
        y = y / 2
    }
}
```

Path 1: $(value = 0) \land (y = x^2) \land (x > 3) \land (x' = x+1) \land (2*x' - 1 > y) \land (y' = 2*y)$

Since $x > 3$ then $2*(x+1) - 1 > x^2$ is not possible, then this path is infeasible.

Path 2: $(value = 0) \land (y = x^2) \land (x \leq 3) \land (x' = abs(x)) \land (2*x' - 1 > y) \land (y' = 2*y)$

Since $x \leq 3$ then $2*|x| - 1 > x^2$ is not possible, then the path is infeasible

path 3: $(value = 0) \land (y = x^2) \land (x > 3) \land (x' = x+1) \land (2*x' - 1 \leq y) \land (y' = \frac{y}{2})$

Since $x > 3$ then $2*(x+1) - 1 \leq x^2$ is possible for example $x = 4$

Path 4: $(value = 0) \land (y = x^2) \land (x \leq 3) \land (x' = abs(x)) \land (2*x' - 1 \leq y) \land (y' = \frac{y}{2})$

Since $x \leq 3$ then $2*|x| - 1 \leq x^2$ is possible for example $x = 3$

Regression Testing

Idea: make sure that whatever changes have been mmade don't cause the software to regress (become worse)

Terinology

- P: old version
- P': new version
- T: test suite for P
- Assume that all tests in T ran on P → Generate coverage matrix C
- Given the delta between P and P' and the coverage matrix C, identify a subset of T that can identify all regression faults. **(Safe RTS)**

Harrold & Rothermel's Regression Test Selection

- Safe efficient regression test selection technique
- Regression test selection based on traversal of control flow graphs for the old and new version.
- Select tests that will eplore dangerous edges
  - Dangerous edges: edges that are different in the new CFG compared to the old CFG.
- Running test cases on dangerous edges are as effective as running all test cases.
- However dynamic binding/dispatching in OOP changes the control flow graph during runtime

Extending to OOP Languages

```
class B etends A {}
class C etends B {}

void foo (A p) { ... }
```

The type of p cannot be determined at compile time.

Dynamic dispatch: runtime object changes the target of method call because of polymorphism

External libraries: external library calls can invoke internal method calls if it overrides

Java Interclass Graph: extends CFG to handle java features

- variable and object type information
- internal and external methods
- interprocedural interactions through calls to internal methods or external methods from internal methods
- interprocedural interactions through calls to internal methods from external methods
- exception handling

Java RTS Algorithm:

1. Start from main() node, ECN node, or static method entries

2. Traverse JIGs and add the dangerous edges to E
3. Mark N as visited
4. If the target along the same edge is different between two graphs, then it becomes a dangerous edge.
5. Use lexicographical equivalence to compare two nodes

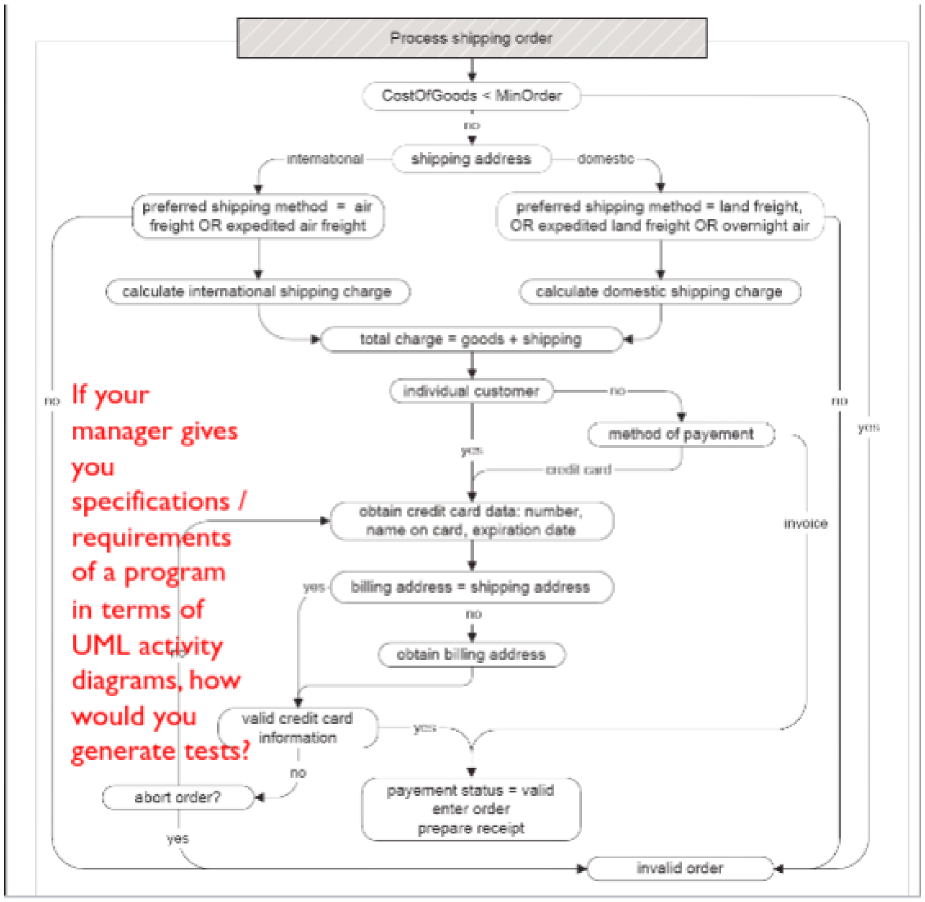
Model Based Testing

Problem: combinatorial explosion in using CFGs can lead to poor scalability

Idea: use a model (abstraction of system's behavior) of the system to decide how to generate tests

UML Activity Diagrams

Represents normal and erroneous behavior, focuses on interaction with system. Similar to CFG but more high level.


Process shipping order
CostOfGoods < MinOrder
no
international
shipping address
domestic
preferred shipping method = air freight OR expedited air freight
preferred shipping method = land freight, OR expedited land freight OR overnight air
calculate international shipping charge
calculate domestic shipping charge
total charge = goods + shipping
individual customer
no
method of payement
yes
credit card
invoice
obtain credit card data: number, name on card, expiration date
yes
billing address = shipping address
no
obtain billing address
valid credit card information
yes
no
abort order?
yes
payement status = valid enter order prepare receipt
invalid order
no
no
yes
If your manager gives you specifications / requirements of a program in terms of UML activity diagrams, how would you generate tests?

Node coverage: ensure tests cover all the nodes in UML

Branch coverage: ensure tests cover all branches in UML

Mutations: introduce mutations where the inputs does not follow the UML

Finite State Machine

Describes interactions in systems with a small number of modes

Single state path coverage: collection of paths that cover the states

Single transition path coverage: collection of paths that cover all transitions

Boundary interior loop coverage: criterion on number of times loops are exercised

Mutation: discover how system responds to unexpected inputs

Use probabilistic automata to represent distributions of inputs for randomized testing

Grammar

Use grammar to define well formed inputs to system. Use to generate sample inputs.

- Every production (every possible path in the grammar construction) at least once
- Boundary conditions on recursive productions - 0, 1, many
- Probabilistic CFGs allow us to prioritize heavily used constructs.
- Probabilistic CFGs can be used to capture and abstract real world data.

Mutation Testing

Problem: how to assess adequacy of tests?

Idea: mutate programs by purposefully injecting errors in the program to see if tests catch the error

Process: given program P and tests T

- Systematically apply mutations to program P to obtain mutants $P_1..P_n$
- Run test suite in each of the mutants, **T kills mutant P_j if it detects an error**
- Mutant killing ratio is the number of killed mutants to number of total mutants

Types of mutants:

- Value mutations: change value of constants or parameters
- Decision mutations: modifying conditions to reflect possible logical errors

- Statement mutations: delete or swap lines of code or order of arithmetic operations

Fuzzing

Idea: systematic and random testing to find crashes, improper error handling, or other ciritical faults

- We can generate randomized inputs based on some information about the system to the program

Questions:

- Which inputs to we retain?
- Which operators and statements to mutate and which should we prioritize?
- When should we stop fuzzing?
- How to assess latent fault inducing capability of inputs/seeds
- What kind of strategies to generate inputs?
- How to evaluate generation metrics?

Grammar Based Fuzzing

- We can use a grammar to generate examples by exapnding the grammar through recursion
- Recursion may have infinite loops due to cycles
- How to control recursion to prevent infinite loops and also produce correct values?
  - Set a recursion depth to prevent infinit loops
  - Eliminate recursion options when the depth is hit to ensure that the output is finalized
    - For each node, if the remaining depth avaliable is less than the distance to a terminal node, eliminate the option

Note: wide fan-out but narrow recursion depth or narrow fan-out and deep recursion depth

Probabilistic Grammar Based Fuzzing

- Some expansions are more likely to occur naturally, so we should use probabilites during expansions
- Learn weights from parsing real examples to generate similar inputs
  - If you invert weights, you can generate rare or unexpected inputs

Hoare Logic

- State predicates: boolean function on the program state
  - ie: $m \leq n \rightarrow \forall j | 0 \leq j < a.length, a[j] \neq NaN$

- Hoare Triples: for predicates P, Q and program S: $\{P\}S\{Q\}$ means that if S started started in state P then it terminates in state Q
    - Strongest Postcondition: if $\{P\}S\{Q\}$ and $\{P\}S\{R\}$ then $\{P\}S\{Q \wedge R\}$, and the most precise Q such that $\{P\}S\{Q\}$ is the strongest postcondition
    - Weakest Precondition: if $\{P\}S\{Q\}$ and $\{Q\}S\{R\}$ then $\{P \vee Q\}S\{R\}$, and the most general P such that $\{P\}S\{Q\}$ is the weakest postcondition
- How to choose post conditions? Specifications/requirements decide what programs should do.

Weakest Precondition Rule $\{P\}S\{Q\} \iff P \to wp(S|Q)$

For each construct, we can define the weakest precondition rules:

- Skip/No-op: $wp(\text{skip}, Q) = Q$
- Assert: $wp(\text{assert P}, Q) = P \wedge Q$
- Assignment: $wp(w := E, Q) \to Q[w := E]$ (replace in Q all occruences of w with E)
- Sequential Statement: $wp(S; T, Q) \to wp(S, wp(T, Q))$
- Conditional Statement: $wp(B?S : T, Q) \to B \wedge wp(S, Q) \vee \neg B \wedge wp(T, Q)$
- Loops: $wp(PwhileBdoSendQ)$
    - Loop Invariant J is true
        1. Invariant is held initially: $P \to J$
        2. Invariant is maintained: $\{J \wedge B\}S\{J\}$
        3. Invariant is held when the loop exits and the post condition is true: $J \wedge \neg B \to Q$
    - Loop terminates if variant function VF is bounded and decreases
        * VF bounded: $J \wedge B \to (0 \leq VF)$
        * VF decreases: $J \wedge B \wedge vf = VFSvf < VF$

Delta Debugging

Idea: we take a program with an error and run a search algorithm to find the error. For each iteration, we divide the program into parts and test each part to see which one fails.

- If both parts succeed, we can instead split into 4 parts and test each compliment (0~1, 1~2, 2~3, 3~4) and then the complements (1~4, 0~1 2~4, 0~2 3~4, 0~3)

Limitations

- non-deterministic programs
- bugs requiring a combination of inputs

Algorithm

Given a list of circumstances $\delta_1 ... \delta_n$. Given a $test(c) \in \{pass, fail, unknown\}$.

A failure inducing-configuration: $test(c_{fail}) = fail$

Relevant configuration: there exists a subset of c_{fail} which are the minimum required circumstances required for failure: - $c'_{fail} \subseteq c_{fail}, \forall \delta \in c'_{fail} : test(c'_{fail} - \delta) \neq fail$

The search stratgy can be:

Split inputs with granualrity n $c_{fail} = c_1 \Cup ... \Cup c_n$ Remove c_i and retest - If the removal still fails, remove c_i from c_{fail} and reduce the granularity to $n-1$ - If no removals fail, increase granularity to $2n$

Assumptions / Properties

Monotone configuration: we assume that if a configuration c succeeds, then we assume that the subsets of c also succeed. This is not true in general, but allows DD to be fast.

Unambigious: if two configurations c_1, c_2 both fail, then the intersection $c_1 \Cap c_2$ should not pass.

Consistent: configuration c is deterministic

Spectra Based Fault Localization

Idea: represent range of values or set of statements in program state during multiple test executions, contrast value profile or set of statements of failing runs against successful runs

We assume that statements that lead to failures (ie bugs) are more likely to be run in failing test cases than successful test cases

Ranking/Finding Statements: - Rank statements by $\frac{fail(s)/totalfail}{fail(s)/totalfail + pass(s)/totalpass}$ - Given statements set $Fail$ and $Pass$ we can find $Fail - Pass$ that are statements unique to the failing tests - Calculate the nearest passing test case neighbor to failing test cases - Leverage delta debugging to isolate failure inducing values and identify statements which involve failure-inducing variables - consider transition points as bug locations - causes: value of a variable that causes a failure at some state in the program - transition: narrow down statements where causes differ, at some point in the execution the causes change, use binary search to find that transition - Generate Program Dependency Graph (similar to control flow graph with addition of data dependency) - Use BFS search to show that a reported bug statement from automated to find real fault statement

Tool Demos

CodeQL

Code analysis to do taint analysis by tracking flow of possibly malicious data through program.

SemGrep

Static analysis tool for detecting bugs using pattern rules.

JaCoCo

Java code coverage tool, measures line and branch coverage during unit tests.

Inserts code to monitor coverage on-the-fly during class loading in JVM.

Hypothesis

Python property based testing which writes specifications for program and asserts property of results. Multiple test cases generated, guarantees property is true under specifications.

Uses invariants, pre/postconditions to find test cases more likely to cause errors. Uses shrinking (minimize smallest couterexample), weighted random sampling, and custom generators.

SPF

Java symbolic execution engine generates symbolic path conditions.

ANTLR

Standard language for writing context free grammars. ANTLR engine can parse these grammars into an abstract syntax tree. Can be used to create DSLs, code analyzers, transpilers, fuzzing with grammarinator.

Grammarinator

ANTLR v4 based fuzzing test generator. Can be used to simulate a wide range of random inputs which can better cover edge cases. Automates test case generation.

SpotBug

Static java analysis tool for identifying bugs in compiled code. Finds known bug patters, poor quality code, common vulnerabilities.

Performs static analysis of bytecode, simulates potential code paths using symbolic execution, and examines branches for unreachable code, constant conditions, or infinite loops.

Static inference: creates CFG to map execution paths. Traverses each path and track variable states. Evaluates conditions that can trigger an error such as null pointer.

Klee

C++ symbolic execution engine that helps generate test cases towards high code coverage. Explores code paths by forking new process for each branch with different symbolic assumptions.

Comby