Heuristics for Scalable Dynamic Test Generation
Jacob Burnim, Koushik Sen

OVERVIEW

• Goal: Automatically generate effective test cases for large software systems with concolic execution.
• The Challenge: Path Space Explosion.
  • Even medium-sized programs have $10^{11}$+ paths.
  • Computation tree is too large to exhaustively test each path.
• Revised Goals:
  • Focus on branch coverage.
  • Attain better branch coverage with fewer tests.
• Our Approach: Try to explore only the "important" program paths.
  • Propose three search strategies:
    - Control-flow directed search
    - Uniformly random path search
    - Random-branched search

CONTROL-FLOW DIRECTED SEARCH

• Key Idea: Use static structure of test program to guide search.
  • Dynamic search is not heuristic.
  • Search guided by knowledge of unexplored parts of path space.
• Heuristic: Explore paths statically "close" to uncovered branches.
  • Use distance in the control flow and static call graph (CFCG).
  • Negate conditions with minimal distance to uncovered branches.
• Search Strategy:
  • Given a current execution path:
    1. Negate conditions in topological order.
    2. If there are unexplored paths, take the first.
    3. Otherwise, negate the conditional with the smallest distance to an unexplored branch.
• Dynamic search with a static heuristic.
  • Local search – operates on one current execution path at a time.
  • Recompute estimated distances.
  • Can restart on new, random paths when progress slows.

CONTROL FLOW AND STATIC CALL GRAPH

• Control flow and static call graph (CFCG):
  • Control-flow graph for each procedure.
  • Call edge from each call site to the called function.
  • No return edges back to call sites.
• Static path: a path through the CFCG.
  • Can either skip over a function call or enter the called function and never return.
• Assumptions:
  • All functions return.
  • Paths in skipped functions need not be explored.
• Distance in CFCG captures difficulty of forcing execution down a static path.
  • Edges leading to a conditional have weight one; others weight zero.
  • During search, neglect any conditional requires one iteration.

CONCOSCIC TEST GENERATION

• Concrete + symbolic execution: technique to generate an input exercising each possible path through the program under test.
• Basic Procedure:
  • Given a concrete execution path, do symbolic execution along path.
  • Simplify using concrete values.
  • Yield symbolic path constraint — a formula characterizing which inputs exercise the path.
  • Pick a conditional branch along path to negate.
  • Solve for inputs so branch is not taken.
  • Running on new inputs yields new concrete execution.
• Exhaustively explores every path in computation tree – traditionally with a depth-first search.
  • Given a path, always negate the deepest conditional not yet negated.

BENCHMARK: GNU grep 2.2

• Popular regular expression matching tool.
  • 15K lines of C ~ 4184 branches, an estimated 2854 of which are reachable.
  • Input: Length 20 regexp and 40 characters of context.
  • Popular open-source text editor.

IMPLEMENTATION

• CREST, an open-source test generation tool for C.
  • C++-based for experimenting with concolic search strategies.
  • Uses CIL to instrument programs and extract control flow.
  • Library for symbolic execution along a concrete execution.
  • Uses Yeas SMT solver.
  • Implemented our strategies in CREST.
  • Available at: http://crest.googlecode.com

COMPUTATION TREE OF A PROGRAM

• The computation tree is the tree of all feasible program executions.
• Each node corresponds to an execution of a conditional statement.
• Each edge a sequence of non-conditional statements executed between two successive conditional statements.
• The conditions along a path characterize the equivalence class of inputs that execute the path.

EXAMPLE OF CONTROL-FLOW DIRECTED SEARCH

• Given a path $P$:
  • If $P$ is a control-flow path, negate all unexplored branches.
  • Otherwise, negate the branch $B$ of minimal distance to an unexplored branch.
  • Run program, then negate each branch with probability $1/2$.
  • Generates each path with $L$ feasible branches with probability $2^{-L}$.

UNIFORMLY-RANDOM PATH SEARCH

• Key Idea: Test program on uniformly random paths rather than on random inputs.
  • Much more likely to cover some paths than with random inputs – e.g. only $10^{12}$ inputs reach ABORT.
• Procedure:
  • Run program, then negate each branch with probability $1/2$.
  • Generates each path with $L$ feasible branches with probability $2^{-L}$.
  • Takes $L/2$ expected iterations.

RANDOM-BRANCH SEARCH

• Motivation: Generating paths uniformly at random is too expensive – $L/2$ expected iterations per path with $L$ feasible branches.
• Procedure:
  • Instead, each iteration randomly pick and negate one feasible branch along the current execution path.
  • Samples some random walk through the path space.
• Found to be more effective in practice than generating uniformly random paths.

FIGURE 1 – Illustration of low coverage of traditional concolic testing on a huge path space.

FIGURE 2 – A simple C program, which serves as our running example.

FIGURE 3 – Computation tree of the example program from Figure 2.

FIGURE 4 – Control flow and static call graph for the example program in Figure 2, with distances to the ABORT statement in function $f$.

FIGURE 5 – Computation tree has probabilities for each path under uniformly-random path search.

FIGURE 6 – Computation tree has probabilities for each branch to be negated given a current execution.