Static VS Dynamic Analysis

**Static analysis** operates on a model of the SW (without executing it)
- If successful, produces definitive information about the product that holds for all inputs

**Dynamic analysis** operates on dynamic information collected by running the SW
- Results in sampling information about the product that holds for the inputs considered

**Combined static and dynamic analyses**
leverage complementary strengths

Do We Need Dynamic Analysis?

- Some static analyses may be too imprecise (e.g., slicing for impact analysis)
- Need to test for properties of executions, rather than the program (e.g., debugging)
- Need to test assumptions about the environment
- Need to determine performance in practice, for the average case
- Need to test for non-functional properties or qualities, such as usability
- …
Examples of Dynamic Analysis

- Testing
- Profiling
- Coverage analysis
- Dynamic-invariant detection
- Assertions
- Dynamic tainting
- Dynamic slicing
- ...

Issues in Dynamic Analysis

- How do we collect the dynamic data?
  - Instrumentation (i.e., software probes)
  - (Customized) runtime system
- How do we avoid “too much” overhead?
- How do we make sure that the observations don’t change the behavior of the system?
- How do we select inputs for the analysis?
  (i.e., how do we know if the inputs are adequate?)
Execution Tracing and Profiling

Gathering dynamic information about programs
- Execution coverage
- Execution profiling
- Execution tracing

Three main alternatives
- Debugging interfaces
- Customized runtime systems
- Instrumentation
  - Post-processing
  - Online processing
  - Preprocessing

Execution Coverage

Execution Coverage records, as the program executes with a test case, the fact that an event occurs.

For example:
- whether a statement is executed
- whether a branch is taken
- whether a path is followed
- whether a variable is modified
- ...

Applications
- Software testing
- Dynamic analysis (adequacy, information)
Execution Profiling

Execution Profiling records, as the program executes with a test case, the number of times that an event occurs.

For example:
- the number of times a statement is executed
- the number of times a branch is traversed
- the number of times an exception is thrown
- the number of times a variable is modified
- ...

Applications
- Program optimization
- Performance tuning
- Profile-directed compilation

Execution Tracing

Execution Tracing records, as the program executes with a test case (an input to the program), some sequence of events that occur.

For example:
- the sequence of statements executed
- the sequence of memory accesses
- the sequence of program states associated with statements executed
- ...

Applications
- Program optimization
- Dynamic analysis (e.g., impact analysis)
Execution Profiling/Tracing

Procedure AVG
S1     count = 0
S2     fread (fptr, n)
S3     while (not EOF) do
S4        if (n < 0)  
S5           return (error)  
else  
S6           nums[count] = m  
S7           count ++  
endif  
S8        fread (fptr, n)  
endwhile  
S9     avg = mean (nums, count)  
S10   return (avg)

Recording Traces, Profiles

<table>
<thead>
<tr>
<th>Test</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>empty file</td>
<td>0</td>
</tr>
</tbody>
</table>

Procedure AVG
S1     count = 0
S2     fread (fptr, n)
S3     while (not EOF) do
S4        if (n < 0)  
S5           return (error)  
else  
S6           nums[count] = m  
S7           count ++  
endif  
S8        fread (fptr, n)  
endwhile  
S9     avg = mean (nums, count)  
S10   return (avg)
Debugging Interfaces

Debugging Interfaces provide hooks into the runtime system that allow for collecting various dynamic information while the program executes.

Examples:
- Java Platform Debugger Architecture (JPDA)
- JVM Debugging Interface (JVMDI)
- JVM Profiling Interface (JVMPI)
- Java Virtual Machine Tool Interface (JVMTI) [New]
- Valgrind
- DynamoRIO
- Emulators for embedded systems

Customized Runtime Systems

Customized Runtime Systems are runtime systems modified to collect some specific dynamic information.

Examples:
- Jalapeño JVM (now Jikes)
Instrumentation

Instrumentation is the process of adding code fragments (called probes) to a program such that when the program is executed, it records dynamic information
- Source-level
- Binary-level
- Dynamic

The program with the probes is called an instrumented program

Instrumentation

Types of instrumentation
- Preprocessing
- Online processing
- Post-processing
Instrumentation Process

Instrumentation - statement coverage:

Equation 1

Instrumentation Process

Instrumentation - branch coverage:

Equation 2
Instrumentation Process

Instrumentation - path coverage:

1. read i
2. read j
3. sum = 0
4. while (i > 0) and (i <= 10) do
5.   if (j > 0)
6.     sum = sum + j
7.   endif
8.   i = i + 1
9.   read j
10. endwhile
11. print sum

Def-use pairs for i: (1, (4, 5)), (1, (4, 9)), (1, 7), (7, (4, 5)), (7, (4, 9))
Instrumentation Process

Instrumentation – all-uses coverage

Instrumentation Tools

- Source-level
  - EDG parser (AST)
  - Customized gcc
- Binary/bytecode level
  - Vulcan
  - BCEL
  - SOOT
- Dynamic
  - Dyninst
  - PIN
  - Valgrind
Efficient Path Profiling

Profiling (recap)

Program profiling counts occurrences of an event during a program’s execution
- Basic blocks
- Control-flow edges
- Acyclic path

Application
- Performance tuning
- Profile-directed compilation
- Test coverage
Goal

Goal: efficiently collecting path profiles for acyclic paths

State of the Art

- Edge profiling: 16% overhead
- Estimation of path profiles from edge profiles
  - Not enough... (38% in SPEC benchmarks)

Acyclic Paths

- All paths are intra-procedural
- No cycles (to avoid infinite number of paths)
- Paths from the entry to a back-edge (paths into the first loop iteration)
- Paths from a loop head to the exit (paths out of the last loop iteration)
- Paths within a loop (number of times the loop iterates)
Acyclic-Path Profiling

Subsumes
  • basic block/statement profiling
  • edge/branch profiling

Better approximation of intra
  -procedural path frequencies

Stronger coverage criterion for white
  -box testing

Goals for Instrumentation

Low time and space overhead
  • Minimal number of probes
  • Optimal placement of the probes
  • Paths represented with a simple
    integer value
  • Compact numbering of paths
Algorithm Overview (i)

- Each potential path is represented as a state
- Upon entry all paths are possible
- Each branch taken narrows the set of possible final states
- State reached at the end of the procedure represents the path taken

- Example:
  - P0: 1, 2, 4, 5, 7
  - P1: 1, 2, 4, 6, 7
  - P2: 1, 3, 4, 5, 7
  - P3: 1, 3, 4, 6, 7

![Diagram]

Algorithm Overview (i)

- Each potential path is represented as a state
- Upon entry all paths are possible
- Each branch taken narrows the set of possible final states
- State reached at the end of the procedure represents the path taken

- Example:
  - P0: 1, 2, 4, 5, 7
  - P1: 1, 2, 4, 6, 7
  - P2: 1, 3, 4, 5, 7
  - P3: 1, 3, 4, 6, 7
Algorithm Overview (ii)

- Final “states” (i.e., paths) are represented by integers in [0, n-1] (n == number of paths)
- Instrumentation not at every branch
- Transitions computed by simple arithmetic operations (no tables)
- CFG transformed in acyclic CFGs (DAGs)

- Example:
  - P0: 1, 2, 4, 5, 7
  - P1: 1, 2, 4, 6, 7
  - P2: 1, 3, 4, 5, 7
  - P3: 1, 3, 4, 6, 7

Algorithm Steps

1. Assign integer values to edges such that no two paths compute the same path sum
2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge
3. Select appropriate instrumentation
4. Derive the executed paths from the collected run-time profiles
Algorithm (Step 1 of 4)

Assign to each edge $e$ a value $Val(e)$ such that the sum along a path is

- Unique
- $[0,n-1]$

for each vertex $v$ in rev. top. order {
  if $v$ is a leaf vertex {
    NumPaths($v$) = 1;
  } else {
    NumPaths($v$) = 0;
    for each edge $e = v \rightarrow w$ {
      $Val(e) = NumPaths(v)$;
      NumPaths($v$) += NumPaths($w$);
    }
  }
}

Not necessarily the best placement
Probe placement

Knuth published efficient algorithms for finding the minimum number of edge counters for edge profiling.
Spanning tree $T$ of CFG is computed.
Edges $E$ in CFG minus edges in $T$ are the chords of the spanning tree.
Instrumentation of only the chords is sufficient to deduce execution of remaining edges.

Optimal Placement of Probes

Several spanning trees may be possible on a CFG.
A maximum spanning tree is a spanning tree of maximum weight on its edges.
Weight is defined as the execution frequency of the edge.
Edge Execution Frequency

Program can be profiled to gather edge execution frequency -- generally impractical for purposes of profiling

Edge execution frequency can be approximated statically

Static approximation heuristic defined in [Ball and Larus 94]
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.

- Add edge EXIT -> ENTRY
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.
   • Add edge EXIT -> ENTRY
   • Compute a maximal spanning tree (find chords)
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.
   - Add edge EXIT -> ENTRY
   - Compute a maximal spanning tree (find chords)

[Ball and Larus 94]
Algorithm (Step 2 of 4)

2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge.
   - Add edge EXIT -> ENTRY
   - Compute a maximal spanning tree (find chords)
   - Assign increments: start from Val(e) and “propagate”
   [Ball and Larus 94]

Algorithm (Step 3 of 4)

3. Select appropriate instrumentation
   - Initialize path register (r=0)
   - Update r in chords (r += inc)
   - Increment path’s counter at EXIT (count[r]++)
Algorithm (Step 3 of 4)

3. Select appropriate instrumentation
   • Initialize path register \((r=0)\)
   • Update \(r\) in chords \((r += inc)\)
   • Increment path’s counter at EXIT \((\text{count}[r]++)\)

   ![Diagram of the algorithm](image_url)

   - Optimization:
     • Initializations (first chord on paths)
     • Path’s counter increment (last chord on paths)
Algorithm (Step 3 of 4)

3. Select appropriate instrumentation

- Initialize path register \((r=0)\)
- Update \(r\) in chords \((r += \text{inc})\)
- Increment path’s counter at EXIT \((\text{count}[r]++)\)
- Optimize
  - Initializations (first chord on paths)
  - Path’s counter increment (last chord on paths)

\[
\begin{align*}
B & \quad r=2 \\
C & \quad r=4 \\
D & \quad r=2 \\
E & \quad \text{count}[r]++ \\
F & \quad \text{count}[r+1]++
\end{align*}
\]

Algorithm (Step 4 of 4)

4. Regenerating a path after collecting a profile

- Start at ENTRY
- Let \(r\) be the path value
- Select which edge to follow by finding the edge with the largest value \(\text{Val}(e) \leq r\)
- Traverse edge \(e\) and \(r = r - \text{Val}(e)\)
Arbitrary Control Flow (loops)

• Loop implies the presence of a back-edge
• Back-edges instrumented to increment path counter and reinitialize path register
  \[ \text{count}[r]++; r=0 \]
• This is not enough; with loops 4 types of paths (v->w and x->y are back-edges)
  • ENTRY to EXIT
  • ENTRY to v (ending with execution of v->w)
  • w to x (after executing v->w and ending with the execution of x->y, v->w and x->y can be the same back-edge)
  • w to EXIT (after executing v->w)
• Need to distinguish them

Arbitrary Control Flow (loops)

• Loop implies the presence of a back-edge
• Back-edges instrumented to increment path counter and reinitialize path register
  \[ \text{count}[r]++; r=0 \]
• This is not enough; with loops 4 types of paths (v->w and x->y are back-edges)
  • ENTRY to EXIT
  • ENTRY to v (ending with execution of v->w)
  • w to x (after executing v->w and ending with the execution of x->y, v->w and x->y can be the same back-edge)
  • w to EXIT (after executing v->w)
• Need to distinguish them
Arbitrary Control Flow (loops)

- Loop implies the presence of a back-edge
- Back-edges instrumented to increment path counter and reinitialize path register (count[r]++; r=0)
- This is not enough; with loops 4 types of paths (v->w and x->y are back-edges)
  - ENTRY to EXIT
  - ENTRY to v (ending with execution of v->w)
  - w to x (after executing v->w and ending with the execution of x->y, v->w and x->y can be the same back-edge)
  - w to EXIT (after executing v->w)
- Need to distinguish them
Arbitrary Control Flow (loops)

- Loop implies the presence of a back-edge
- Back-edges instrumented to increment path counter and reinitialize path register \((\text{count}[r]++; \ r=0)\)
- This is not enough; with loops 4 types of paths \((v->w \text{ and } x->y \text{ are back-edges})\)
  - ENTRY to EXIT
  - ENTRY to \(v\) (ending with execution of \(v->w\))
  - \(w\) to \(x\) (after executing \(v->w\) and ending with the execution of \(x->y\), \(v->w\) and \(x->y\) can be the same back-edge)
  - \(w\) to EXIT (after executing \(v->w\))
- Need to distinguish them

Convert Arbitrary CFGs to DAGs

Eliminate back-edges before computation of edge values and chord increments

- Remove a loop back-edge
- Add two edges
  1. ENTRY -> Target of back-edge
  2. Source of back-edge -> EXIT
- The dummy edges create extra paths \(\text{ENTRY-EXIT}\) that the value assignment algorithm takes into account
  - Edge (1) represents reinitializing along the back-edge
  - Edge (2) represents incrementing along the back-edge
Implementation

- Implemented in a tool called PP
- PP instruments SPARC binaries
- Built on top of EEL (binary instrumenter)
- Uses a register to store r
- Replaces array of counters with hash table if number of paths too large
- Plus some other optimizations

Experimental Results

- When hashing is used performance is hurt
- Using no hashing, overhead is comparable with edge profiling
- Edge profiling correctly predicted 37.9% of the acyclic paths
- Paths covered by a small portion of the test suite comprised most of the paths covered by the full test suite
Software Testing

Testing Fundamentals

Failure, Fault, Error

**Failure**

Observable incorrect behavior of a program. Conceptually related to the behavior of the program, rather than its code.

**Fault (bug)**

Related to the code. Necessary (not sufficient!) condition for the occurrence of a failure.

**Error**

Cause of a fault. Usually a human error (conceptual, typo, etc.)
Failure, Fault, Error: Example

1. int double(int param) {
2.     int result;
3.     result = param * param;
4.     return(result);
5. }

- A call to double(3) returns 9
- Result 9 represents a failure
- Such failure is due to the fault at line 3
- The error is a typo (hopefully)

Dependability Qualities

Correctness
Absolute consistency with a specification

Reliability
Likelihood of correct behavior in expected use

Robustness
Ability of software systems to function even in abnormal conditions

Safety
Ability of the software to avoid dangerous behaviors
**Verification and Validation**

**Validation:** Are we building the right product?
To which degree the software fulfills its (informal) requirements?

**Verification:** Are we building the product right?
To which degree the implementation is consistent with its (formal or semi-formal) specification?

---

**Approaches to Verification**

**Testing:** exercising software to try and generate failures

**Static verification:** identify (specific) problems statically, that is, considering all possible executions

**Inspection/review/walkthrough:** systematic group review of program text to detect faults

**Formal proof:** proving that the program text implements the program specification
Comparison

Testing
  • Purpose: reveal failure
  • Limits: small subset of the domain
    (⇒ risk of inadequate test set)

Static verification
  • P: consider all program behaviors (and more)
  • L: false positives, may not terminate

Review
  • P: systematic in detecting defects
  • L: informal

Proof
  • P: prove correctness
  • L: complexity/cost (requires a spec)

Today, QA is Mostly Testing

“50% of my company employees are testers, and
the rest spends 50% of their time testing!”

Bill Gates, 1995
What is Testing?

Testing == To execute a program with a sample of the input data
Dynamic technique: program must be executed
Optimistic approximation:
• The program under test is exercised with a (very small) subset of all the possible input data
• We assume that the behavior with any other input is consistent with the behavior shown for the selected subset of input data
• The opposite of conservative (pessimistic) analysis

Goals of Testing

Improve software quality by finding errors
“A Test is successful if the program fails” (Goodeneogh, Gerhart, “Toward a Theory of Test Data Selection”, IEEE Transactions on Software Engineering, Jan. 85)

Provide confidence in the dependability of the software product
**Correctness**

A program $P$ is a function from a set of data $D$ (domain) to a set of data $R$ (co-domain)

$P(d)$ denotes the execution of $P$ with input $d \in D$

$P$ is correct iff, $\forall \; d \in D, \; P(d) = S(d)$

**Test Suite, Test Set, Test Case**

A test suite or test set $T$ for $P$ is a subset of $D \times R$

An element $t$ of $T$ is called a test case

$T$ is an ideal test for $P$ iff the correctness of $P$ for all $t$ in $T$ implies the correctness of $P$ for the whole $D$

In general, it is impossible to define an ideal test and we try to approximate it by suitably defining test selection criteria
Testing Techniques

There are a number of techniques
- Different processes
- Different artifacts
- Different approaches

There are no perfect techniques
- Testing is a best-effort activity

There is no best technique
- Different contexts
- Complementary strengths and weaknesses
- Trade-offs

Granularity Levels

Unit testing: verification of the single modules
Integration testing: verification of the interactions among the different modules
System testing: testing of the system as a whole
Acceptance testing: validation of the software against the user requirements
Regression testing: testing of new versions of the software
Integration Testing

Testing component interactions; several strategies:

- **Top-down**
  - Integrate external interface first
  - Requires stubs
- **Bottom-up**
  - Integrate low-level components first
  - Requires drivers
- **Big-bang**
  - Integrate all components at once
  - ...

To simplify error localization, systems should be incrementally integrated

System Testing

Must include non-functional tests

- Performance
- Load
- Usability
- Security
- Portability
- Maintainability (complexity metrics)
- Coding standards
Regression Testing

Rerun tests after updates
  • identify “critical” regression tests
Need to automate (scripts, testing tools, harness of tests and I/O data…)
One of the main reasons why maintenance is costly

Alpha and Beta Testing

\( \alpha \) (alpha) test: in-house, early
\( \beta \) (beta) test: in the field, pre-release

Also, Sanity check / smoke tests: quick -and-dirty, just before shipping
Oracle

An oracle predicts the expected results of a test and is used to assess whether a test is successful or not.

There are different kinds of oracles:
- Human (tedious, error prone)
- Automated (difficult, expensive)

Testing and debugging

Testing and debugging are distinct processes

Testing is concerned with finding defects in a program

Debugging is concerned with locating and repairing these errors
Test Selection Criteria

Test Selection Criterion C: a rule for selecting the subset of D to place in T

We want a C that gives test suites that guarantee correctness

We settle for a C that gives test suites that improve confidence

Types of criteria (complementary):

• Functional (black-box): based on a specification
• Structural (white-box): based on the code

Test Adequacy Criteria

Test selection criteria guide the selection of a test suite T: we select test cases that covers some percentage of “coverable” items (as defined by the criteria).

The same criteria can also be used as test adequacy criteria: the adequacy score of T is the percentage of “coverable” items (as defined by the criteria) covered by T
Test Requirements, Test Specifications

**Test Requirements**: those aspects of the program that must be covered according to the considered criterion.

**Test Specification**: constraints on the input that will cause a test requirement to be satisfied.

Functional vs. Structural Testing
Functional vs. Structural Testing

- Based on a SW spec
- Cover as much specified behavior as possible
- Cannot reveal errors due to implementation details

- Based on the code
- Cover as much coded behavior as possible
- Cannot reveal errors due to missing paths

Structural Testing Example

**Specification**: function that inputs an integer and prints it

**Implementation**:

1. void foo(int param) {
2.     if (param < 1024) printf("%d", param);
3.     else printf("%d KB", param/124);
4. }

- Function foo contains a typo
- From the functional perspective, ints < 1024 and ints > 1024 are equivalent, but they are treated differently in the code
- The fault may be missed by black-box testing
- The fault would be easily revealed by white-box testing (e.g., by statement coverage)
Functional Testing Example

**Specification:** function that inputs an integer \( \text{param} \) and returns half its value if \( \text{param} \) is even, \( \text{param} \) otherwise

**Implementation:**

1. int foo(int param) {
2.     int result;
3.     result = param/2;
4.     return (result); }

- Function \( \text{foo} \) works correctly only for even integers
- The fault may be missed by structural testing (100% coverage with any value)
- The fault would be easily revealed by black-box testing (typically, we would use at least one odd and one even input)