Demand-Driven Structural Testing with Dynamic Instrumentation

Jonathan Misurda[†], James A. Clause[†], Juliya L. Reed[†], Bruce R. Childers[†], and Mary Lou Soffa[‡]

[†]Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania 15260
{jmisurda, clausej, juliya, childers}@cs.pitt.edu

[‡]Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904
soffa@cs.virginia.edu

ABSTRACT

Producing reliable and robust software has become one of the most important software development concerns in recent years. Testing is a process by which software quality can be assured through the collection of information. While testing can improve software reliability, current tools typically are inflexible and have high overheads, making it challenging to test large software projects. In this paper, we describe a new scalable and flexible framework for testing programs with a novel demand-driven approach based on execution paths to implement test coverage. This technique uses dynamic instrumentation on the binary code that can be inserted and removed on-the-fly to keep performance and memory overheads low. We describe and evaluate implementations of the framework for branch, node and def-use testing of Java programs. Experimental results for branch testing show that our approach has, on average, a 1.6 speed up over static instrumentation and also uses less memory.

Categories and Subject Descriptors

D.2.5. [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.3.3. [**Programming Languages**]: Language Constructs and Features—*Program instrumentation, run-time environments*

General Terms

Experimentation, Measurement, Verification

Keywords

Testing, Code Coverage, Structural Testing, Demand-Driven Instrumentation, Java Programming Language

ICSE'05, May 15-21, 2005, St. Louis, Missouri, USA.

1. INTRODUCTION

In the last several years, the importance of producing high quality and robust software has become paramount [15]. Testing is an important process to support quality assurance by gathering information about the behavior of the software being developed or modified. It is, in general, extremely labor and resource intensive, accounting for 50-60% of the total cost of software development [17]. Given the importance of testing, it is imperative that there are appropriate testing tools and frameworks. In order to adequately test software, a number of different testing techniques must be performed. One class of testing techniques used extensively is structural testing in which properties of the software code are used to ensure a certain code coverage.Structural testing techniques include branch testing, node testing, path testing, and def-use testing [6,7,8,17,19].

Typically, a testing tool targets one type of structural test, and the software unit is the program, file or particular methods. In order to apply various structural testing techniques, different tools must be used. If a tool for a particular type of structural testing is not available, the tester would need to either implement it or not use that testing technique. The tester would also be constrained by the region of code to be tested, as determined by the tool implementor. For example, it may not be possible for the tester to focus on a particular region of code, such as a series of loops, complicated conditionals, or particular variables if def-use testing is desired. The user may want to have higher coverage on frequently executed regions of code. Users may want to define their own way of testing. For example, all branches should be covered 10 times rather than once in all loops.

In structural testing, instrumentation is placed at certain code points (*probes*). Whenever such a program point is reached, code that performs the function for the test (*payload*) is executed. The probes in def-use testing are dictated by the definitions and uses of variables and the payload is to mark that a definition or use in a def-use pair has been covered. Thus for each type of structural testing, there is a testing "plan". A *test plan* is a

"recipe" that describes where probes should be placed and what should be done when a probe is reached.

In most tools, the instrumentation is placed in the binary code before execution and remains in the code until execution terminates. This type of instrumentation can be expensive in both time and space. The instrumentation causes code growth and thus instrumenting a complete program may not be possible. Also, even though coverage may only require one instantiation of a code element, the instrumentation stays in the code, causing unnecessary time overhead.

In this paper, we describe a testing framework that addresses both flexibility and scalability for structural testing. Our approach enables testers to use different testing strategies, including custom testing, in an efficient and automatic way. The key ideas in our approach are a *test planner* that generates a plan from a test specification and an instrumenter that (1) inserts instrumentation when needed in a demand-driven fashion as the program executes and (2) deletes the instrumentation when no longer needed, according to a test plan. The approach is path specific and uses the actual execution paths of an application to drive the instrumentation and testing. The granularity of the instrumentation is flexible and includes statement level and structure level (e.g., loops, methods, program).

To ensure flexibility, we developed a specification language from which a test plan can be automatically generated by a test planner. The test specification describes what tests to apply and under what conditions to apply them. The specification language has both a visual representation and textual form. The visual language is expressed through a graphical user interface (GUI). The GUI is also able to display test results and present them to the user with a test analyzer, highlighting relevant parts of the application with the test results.

We implemented the test framework—the GUI, test planner, dynamic instrumenter, and test analyzer—and incorporated them into the Eclipse integrated development environment [5] and the IBM Jikes Java Research Virtual Machine [2]. Our prototype tool, called Jazz, can perform branch, node and def-use coverage testing over multiple code regions in a Java program, as desired by the tester. The prototype demonstrates the feasibility and practicality of our approach. Our results are very encouraging, with both very low run-time overhead and memory usage.

This paper makes several contributions, including:

- A novel and low cost approach for instrumenting a program along an execution path to perform different types of tests;
- A new framework for generating structural software testing tools that use dynamic instrumentation;
- A technique that enables dynamic insertion and removal of test instrumentation on demand; and
- An implementation and experimental evaluation of a tool that implements our approach for testing Java programs.

In the next section, we give an overview of our framework including a user scenario. In Section 3, we discuss the test planner and the dynamic instrumenter. Section 4 describes particular test planners, and experimental results are presented in Section 5. The paper concludes with related work and a summary.

2. FRAMEWORK OVERVIEW

Our test framework is designed to be scalable and flexible, allowing the development of tools that can implement structural tests, using a path-specific approach. Figure 1 shows the major components in the framework, including a *test specifier*, a *test planner*, a *dynamic instrumenter*, and a *test analyzer*. The framework includes a language, *testspec*, for specifying a software test process. The specification includes the relevant parts of the program to be tested and the actions needed in the testing process. Testers can either write a specification in *testspec* or, better, use the GUI, which automatically generates a specification in *testspec*. A test planner consumes the *testspec* specification and generates a *test plan* for testing the program given the specification. Using the generated plan, the dynamic instrumenter inserts probes into a program at run-time to conduct the specified tests. Finally, the framework has a test analyzer for reporting results to the user. In this paper, we focus on the novel aspects of the framework, which are the test planner and dynamic instrumenter.

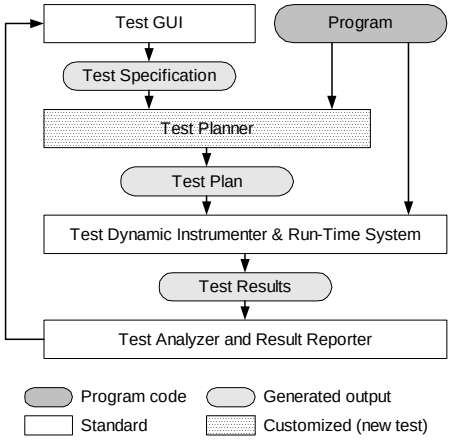


Figure 1: Framework for Demand-Driven Testing

2.1. Usage Scenario

In this section, we provide an example of using the framework. Assume a tester, Tracy, wants to test a large program using different testing strategies. Assume she wants to first test the entire program using branch testing. Using the GUI, Tracy specifies that branch testing is to be applied to the entire program. Our framework will automatically set up the correct instrumentation for this testing strategy. Because the instrumentation is dynamically inserted and deleted in a demand-driven fashion, branch testing can be performed on the entire program. Tracy then decides to further test a selected set of classes (a "test region") using def-use testing with high coverage. This testing is also carried out automatically by our framework. Then Tracy uses the GUI to indicate that a selected loop is to be tested using branch coverage but

defines coverage to be 10 instances of the loop. She also decides to test a function using def-use testing at the same time. Our framework automatically places the correct instrumentation to accomplish this. Lastly, Tracy designs a unique form of a testing strategy that has not been implemented. She uses the specification language to define the testing strategy. Using this specification, the planner generates plans to accomplish this, allowing Tracy to then test the program using a new testing strategy. This scenario indicates that our framework enables testing that is both scalable because of the demand-driven instrumentation and flexible due to the planner. More detail about the GUI and the overall framework is available in [3] and [14].

2.2. Demand-Driven Instrumentation

A unique characteristic of our framework, and the reason it is scalable, is the way that instrumentation is inserted in the executing program. Rather than insert all of the instrumentation **before** the program executes (static), we insert the instrumentation **during** program execution and only the necessary instrumentation. Likewise, we dynamically delete instrumentation when it is no longer needed. Thus, both insertion and deletion are done in a demand-driven fashion. The demand is guided by the paths that the program takes during execution. When an instrumentation point is reached, it is responsible for documenting the coverage, inserting other instrumentation, and deleting instrumentation. The insertion and deletion process is described in Section 3.3.

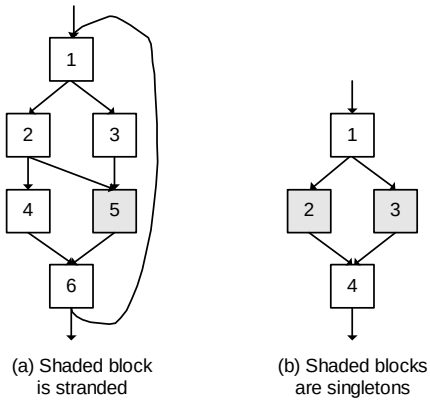


Figure 2: Example control flow graphs

Consider the control flow graph (CFG) of basic blocks for a program segment in Figure 2(a) and assume that branch testing is being performed. Before this code segment executes, one instrumentation point is placed at block 1 (either statically if the code segment is the initial block or by another block when it is executed). When the probe in block 1 is reached, the payload is responsible for inserting instrumentation in both blocks 2 and 3 since one of these blocks has to be on the executing path. Assume block 2 is reached. The instrumentation at block 2 would insert instrumentation at blocks 4 and 5. In addition, it would remove the instrumentation in block 2 since it is no longer needed, as the edge between 1 and 2 is covered. When the execution traverses the path from block 1 to block 3, then both the instrumentation at blocks 1 and 3 would be removed. Block 1 no longer needs to be instrumented as there are no other edges from block 1 that can be reached. Thus, the instrumentation is both inserted when needed and removed when no longer needed. Experimental results indicate less overhead in space because of fewer instrumentation points at any one time than static instrumentation. It is also less time expensive because instrumentation is only hit when it is needed. In the next section, we discuss the test planner and its generation of the test plan in detail.

3. TEST PLANNER

The main function of the test planner is to determine where and how to test a code region. Using the specification and the intermediate code for a test region, the test planner determines the actions necessary to carry out tests. These actions are the run-time activities that collect coverage information and instrument the test region. The actions form the basis for the test plan. In the next sections, we discuss some of the test planner challenges and implementation strategies.

3.1. Challenges

To generate a test plan, a planner needs to determine when to insert probes, where to instrument a test code region, and what to do at a probe. There are three cases the planner has to consider when deciding when to insert and delete instrumentation. First, it must identify which probes are seeds. *Seeds* are those probes which are initially inserted in a test region. Second, it needs to determine which probes are used for coverage and can be inserted and removed on-demand along a path. Finally, the planner has to determine the lifetime of a probe and whether it must be re-inserted after being hit by its “control flow successor” basic blocks.

The test planner also must identify the locations of probes in a test region. These locations correspond to seed, coverage, and control flow probes. Seed locations are blocks where control enters a test region. Coverage locations correspond to basic blocks that have coverage probes. Finally, control flow locations are successors to blocks that have coverage probes which need to be re-inserted. Seed locations must be marked in a table to tell a dynamic instrumenter where probes should be inserted initially. Coverage and control flow locations also have entries in a table to hold information needed by the probes. Coverage locations usually have an entry in a results table to hold coverage information.

The last task of the planner is to determine what actions should take place at a probe. In some cases, different payloads or combinations of payloads may be used at different probes and the planner needs to select the appropriate payload.

3.2. Planner Actions

Actions in a test plan are implemented with a test probe and payload. Probes can be inserted in a code region at any basic block where test actions need to take place. A test plan

may have multiple payloads, which can be invoked by different probes, and multiple probes may be inserted at the same location to call different payloads. The test plan uses a *probe location table* (PLT) to encode probes and their locations. A PLT entry has a probe type, a payload, and a list of probes to insert (and in some cases, to remove). Additional fields can be added to the PLT by the planner. The test plan also has data storage, including global memory that is persistent with program scope (i.e., there is a single global storage area) and local storage with method scope. Global storage is used to hold test results for multiple testing runs (i.e., what has been covered) and the local storage is used to hold temporary values needed by a payload. Other storage scopes can also be incorporated into a plan (e.g., thread or class scope).

As an example, consider node coverage, which records the basic blocks that are executed in a test region. Figure 2(b) shows an example test region and Figure 3(a) shows A corresponding *testspec* specification. The specification indicates that node testing should be done on the method `counter` in class `Counter` from the Java source file `Counter.java`. Figure 3(b) shows the test plan for the test specification in (a). As shown, the test plan has a global array, `covered`, that records which blocks are executed. The PLT lists the basic blocks to instrument and the probes in these blocks call `node()` to update `covered`, insert probes in successor blocks and remove the current probe.

Once the test plan is created, it is passed to the dynamic instrumenter, which automatically inserts probes at locations that are marked as seeds in the PLT. For this example, an initial probe is inserted in block 1. Now, consider what happens when 1 is hit: `node()` executes, which inserts probes into successors blocks 2 and 3, marks 1 as covered, and removes block 1's probe. If block 2 executes next, then it is marked as covered, a probe inserted in 4, and 2 probe removed. When control exits, `covered` lists 1, 2, and 4 as covered.

The test planner automatically generates the PLT, determines global and local storage, and links payloads to probes. A test planner is implemented by a tool developer and the framework allows a developer to build a library of planners, which can be selectively invoked. A test planner can be developed to integrate different tests into a single plan. To ease the development new planners, the framework has a parser and intermediate representation for *testspec*, and interfaces for inserting and removing probes in binary code, managing test plan memory, and generating test result reports.

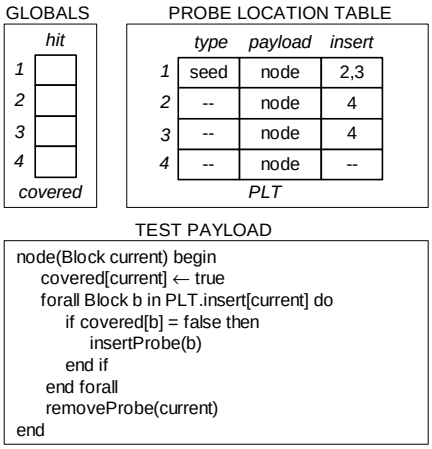
From our experience, we have found that the framework's capabilities significantly ease the development of a test planner. For example, our def-use planner took two weeks to develop and debug by a graduate student that had no previous experience with the framework. The node planner took a half-day to develop and debug.

3.3. Dynamic Instrumenter

Dynamic instrumentation requires probes that can be inserted and removed on the binary machine code. The dynamic instrumenter provides an application programmer interface (API) that abstracts and hides instruction and machine details about instrumentation. It provides for the dynamic insertion and deletion of probes and the management of global and local storage in a test plan. This API allows for flexible instrumentation that can be specified in a variety of ways. The instrumentation constructed with the API is also highly scalable since only relevant portions of the program are instrumented for only as long as needed.

```
DEFINITIONS {
  NAME: c_method, REGION_D,
  LOCATION: FILE Counter.java {
    CLASS Counter, METHOD count
  }
}
BODY {
  DO NODE_TEST ON REGION c_method UNTIL: 85%
}
```

(a) Example testspec specification



(b) Test plan for the example specification

Figure 3: Example test plan for node coverage

The interfaces for insertion and removal of test probes provide several capabilities. With the interface, probes can be associated with particular basic blocks in a test region. The interface hides and automatically handles program instruction addresses, modification of the binary instructions to insert/remove a probe, and the insertion/removal of multiple probes at the same location. The interface also provides for inserting seed probes in a test region.

The management of global and local memory is similarly abstracted. A test plan can allocate and deallocate and access elements in the global storage with simple interfaces. The instrumenter will also automatically allocate local memory on method entry and deallocate on method exit. Other aspects such as handling multithreaded programs are similarly hidden from the test plan and the developer of the test planner.

4. TEST PLANNERS

In this section, we first discuss using our framework for branch testing because branch testing illustrates many of the issues for demand-driven testing. Next, we briefly discuss node and def-use testing.

4.1. Branch Coverage Planner

The branch coverage test planner instruments a region to ensure that all edges can be marked as covered when they are traversed. The planner generates a test plan that instruments on-demand along an execution path and removes instrumentation as soon as possible. To generate the test plan, the planner has to determine which blocks are seeds, when to insert and permanently remove probes, and what payload to use at a probe.

For branch coverage, the seed blocks are the entry points into a test region. These seed blocks insert instrumentation when control passes through an entry. Seeds are identified as basic blocks that have one or more predecessors outside of the test region.

A more difficult issue is how to record which edges are executed, and when probes need to be inserted and removed. To cover an edge, two probes are executed: one in the edge's source and one in the sink node. The probe in the source records the beginning of an edge and the probe in the sink marks the edge as covered. The difficulty is identifying what instrumentation to insert and delete when a block is hit.

In general, when a probe in a source node is executed, it inserts a probe into its successors of uncovered edges and removes itself. The successors are determined by the planner and added to the PLT entry for a node. In this way, as control flows through a test region, probes are inserted and deleted to follow execution paths. Although in many cases this strategy is sufficient, in other cases a probe has to remain until *all* edges from itself to its successors are covered. As an example, consider the CFG in Figure 2(a) and assume that block 1 is a seed. The approach as described cannot mark some edges as covered when the loop executes several iterations and takes certain paths, such as:

(1→2→4→6)→(1→3→5→6)→(1→2→5→6)

In the first iteration, block 1 inserts probes in 2 and 3 and removes itself. The payload for the probe in 2 marks the edge *1→2* as covered and inserts probes in blocks 4 and 5. The probe in 2 is also removed. Similarly, 4 and 6 mark edges *2→4* and *4→6* as covered, insert probes in successors, and remove themselves. At the end of the first iteration, there are probes in 1, 3, and 5.

When the second iteration begins, block 1 is hit, removes itself, and does not insert any probes: edge *1→2* is already covered and there is a probe in 3. Block 3 executes and marks edge *1→3*, but it does not insert a probe in 5. When block 5 executes, it marks edge *3→5* and inserts a probe in block 6. Finally, when block 6 executes, it marks edge *5→6*. However, it will not insert a probe in block 1 because *6→1* was already covered by the previous iteration. Hence, on the third iteration, when control flows reaches *2→5*, no instrumentation has been inserted to capture that edge.

The problem is block 2 needs a probe to record it as a predecessor to block 5. Block 5 is *stranded* because an edge to it cannot be covered. Stranded blocks occur when a block has multiple predecessors and at least one of those predecessors has multiple successors.

The planner identifies stranded blocks by inspecting the CFG. If a block is stranded, then the probes in the stranded block's predecessors can not be removed until all outgoing edges of the predecessor are covered. The planner ensures that these probes are permanent by marking them in the test plan that they have to be re-inserted until the stranded block is fully covered.

Another problem occurs for *singleton blocks*, as shown in Figure 2(b). In this case, edge *1→3* is not marked as covered when path *1→2→4* is followed by path *1→3→4*. When execution reaches block 1 on path *1→3→4*, its entry edge is already covered and no instrumentation is inserted in block 1. That is, there will be no probe in 1 to record the successor to block 3. Hence, edge *1→3* cannot be marked as covered.

To handle singleton basic blocks, the planner identifies blocks with a single predecessor and inserts a probe that encodes the edge. In the figure, when block 1 first executes, it inserts a probe in block 3 that knows predecessor is block 1. When path *1→3→4* executes, the probe at 3 is hit, records edge *1→3* as covered, and inserts a probe in block 4 to cover edge *3→4*.

Line	Pseudocode
1	CFG G ← buildCFG(testRegion)
2	**forall** Block b **in** G.nodes **do**
3	PLT[b].insert ← b.successors
4	PLT[b].payload ← regularPayload
5	*// Check if block b is an entry block (a seed)*
6	**if** b.predecessors ⊄ G.nodes **then**
7	PLT[b].seed ← true
8	*// Check if block b is a singleton*
9	**else if** |b.predecessors| = 1 **then**
10	GLOBAL[sources][b] ← b.predecessors
11	PLT[b].payload ← singletonPayload
12	**end if**
13	*// Check if block b is stranded*
14	**forall** Block p **in** b.predecessors **do**
15	**if** |p.successors|>1 ∧ |b.predecessors|>1 **then**
16	PLT[b].insert ← PLT[b].insert ∪ b.predecessors
17	**end for**
18	**end for**

Table 1: Branch coverage planner

The algorithm for the branch coverage planner is shown in Table 1. For brevity, we do not show the payloads—their actions are as described in this section. The planner creates a CFG for the test region on line 1. Next, it iterates over basic blocks to determine whether they are a seed, a singleton, stranded, or a regular block. Initially, on lines 3 and 4, a block is treated as a regular block that inserts probes in its successors blocks. Lines 6 and 7 check whether the block has any predecessors that are not in the test region and sets the PLT field `seed` to true, if so. When a block has a single predecessor and it is not a seed, then it is a singleton, as shown on lines 9-12. In this case, the singleton's predecessor is recorded in a table in global memory. At run-time, when the payload `singletonPayload()` is invoked, it accesses the table to get its predecessor. Finally, lines 14 to 17 check for stranded basic blocks. The planner treats stranded blocks as regular blocks (i.e., it uses the normal payload), except its predecessor are added to the insertion list to ensure they are re-inserted at run-time.

4.2. Node Coverage Planner

The node coverage planner is the simplest of the planners described in this paper. The planner iterates over basic blocks in a test region, adding each block to the PLT and marking each one as a seed. That is, *all* probes are inserted before the test region is executed. The planner links a payload with each probe that records coverage. When a probe is hit, the payload marks the block covered and deletes itself on demand. In this way, the deletion of probes is demand-driven, but the insertion of probes is not. In comparison to the node coverage approach in Section 3, this approach reduces payload complexity.

4.3. Def-use Coverage Planner

The goal of def-use testing is to determine the coverage among pairs of variable definitions and uses. The def-use coverage planner inserts probes at definitions to record when a variable is assigned a value. Probes are also inserted at locations where variables are used. These probes mark a def-use pair as covered by examining which definition was the most recently executed.

To instrument a test region for def-use, the planner first determines all definitions in a region and inserts seed probes at those definitions. When a definition is hit, its payload inserts probes at all reachable uses. Probes at definitions must remain in the test region until all reachable uses are covered. These probes are needed because they record which definition has been recently executed. To keep a probe at a definition, the planner generates a test plan that re-inserts the probe. The test plan puts probes in control flow successors of blocks with definitions, which re-insert the original probe and remove themselves.

Probes at uses can be deleted immediately once they are hit. When a probe for a use, say u_1, of variable x is inserted, there must have been a definition of x, called d_1, because the probe at u_1 is inserted by d_1. Hence, once the def-use pair (d_1, u_1) is covered, the probe at u_1 can go away. If another definition, d_2, of x also reaches u_1, then a new probe is inserted at u_1 by d_2.

A challenge for the def-use planner is that a CFG node can have many definitions and uses of different variables. The node may even be a control flow successor to several nodes with definitions. The planner treats all definitions, uses, and control flow probes independently, and in effect, inserts several probes in a block. The effect of multiple probes is done by combining the probes into one probe that invokes several payloads.

Table 2 shows the algorithm for the def-use planner without support for composite data structures (payloads are not shown). The algorithm first constructs the CFG and def-use chains for a test region (lines 1 and 2). Then, it proceeds in two passes. In the first pass, the planner iterates over the def-use chains to process and group definitions and uses in a basic block (lines 4-16). Each definition is marked as a seed on line 6 and the definitions and uses are grouped on lines 9-11. On line 9, the variable name for the definition in the current chain is recorded in the defining block. Line 10 records a use of the variable in the block that reads it. Because probes need to be inserted at uses, a set of blocks and variable names is maintained to track the locations where those probes should be placed (line 11). Lines 12-15 determine control flow successors for re-insertion of probes at definitions.

Line	Pseudocode
1	CFG G ← buildCFG(testRegion)
2	DUChains chains ← buildDUChains(G)
3	*// Pass 1: Group definitions, uses, & successors*
4	**forall** Chain c **in** chains **do**
5	Block defBlk ← c.defBlock()
6	PLT[defBlk].seed ← true // mark all defs as seeds
7	Block useBlk ← c.useBlock()
8	Variable v ← c.variableName()
9	defBlk.recordDefs ← defBlk.recordDefs ∪ v
10	useBlk.recordUses ← useBlk.recordUses ∪ v
11	defBlk.placeUses ← defBlk.placeUses ∪ (useBlk, v)
12	**forall** Block b **in** defBlk.successors() **do**
13	defBlk.placeSucc ← defBlk.placeSucc ∪ b
14	b.placeDefs ← b.placeDefs ∪ (defBlk, v)
15	**end for**
16	**end for**
17	*// Pass 2: Build PLT and payloads*
18	**forall** Block b **in** G **do**
19	Trampoline tramp ← **new** Trampoline()
20	**forall** Variable v **in** b.recordUses **do**
21	emitRecordUse(tramp, b, v)
22	**forall** Variable v **in** b.recordDefs **do**
23	emitRecordDef(tramp, b, v)
24	**forall** (Block b2, Variable v) **in** b.placeDefs **do**
25	emitPlaceDef(tramp, b2, v)
26	PLT[b].insert ← PLT[b].insert ∪ b2
27	**end for**
28	**forall** (Block b2, Variable v) **in** b.placeUses **do**
29	emitPlaceUse(tramp, b, b2, v)
30	PLT[b].insert ← PLT[b].insert ∪ b2
31	**end for**
32	**if** |b.placeSucc| > 0 **then**
33	emitPlaceSuccessors(tramp, b)
34	PLT[b].insert ← PLT[b].insert ∪ b.placeSucc
35	**end if**
36	PLT[b].payload ← tramp
37	**end for**

Table 2: Def-use coverage planner

Once the definitions, uses, and control flow successors are grouped, the second pass traverses the basic blocks to construct the PLT and combine payloads (lines 18-37). The payloads are actually created for each payload by generating code for a "trampoline" that has calls to functions that perform actions at a probe.

To construct the trampoline, the algorithm emits calls to functions that mark uses in a block as covered (lines 20-21) and record definitions (lines 22-23). Next, a call is emitted that re-inserts probes at definitions when the current block is a control flow successor (lines 24-27). Calls are also emitted to functions that (1) insert probes for variable uses (lines 28-31), and (2) insert probes in control flow successors (lines 32-35). Finally, on line 36, the generated trampoline is recorded in the PLT as the payload. Although not shown, the algorithm also handles definitions and uses of variables in the same block.

5. JAZZ: A STRUCTURAL TESTING TOOL

To investigate the efficiency and effectiveness of demand-driven structural testing, we implemented our framework and built a tool with it, called Jazz. The tool does branch, node, and def-use coverage and implements a GUI, test planners, dynamic instrumentation, and a test analyzer. Jazz is incorporated in Eclipse [5] and Jikes for the Intel x86 [2].

5.1. Jikes RVM

To integrate the framework into Jikes, we had to address how the test planner gets control, multi-threading, and the interaction of garbage collection (GC) and instrumentation. The first issue was handled by adding a callback to Jikes' just-in-time compiler to invoke the planner. The planner is called after the bytecode has been translated into x86 instructions. At this point, a method's CFG, symbol table, and line number map are available. Once a plan is generated, the dynamic instrumenter inserts seed probes on the binary code.

Jazz supports multi-threading as found in Java programs. Because test information may be local to a thread, it has to be saved and restored at a thread switch. For example, when marking edges in branch coverage, two successive probes pass information to indicate the edge. If a thread switch happens in between the probes, then this information needs to be saved. The test plan indicates what information to switch by allocating it in local memory. For branch coverage, the "previous hit block" is a local and saved/restored at a context switch.

To switch the test plan's local memory, our dynamic instrumenter modifies a method's activation frame to include a hidden variable, called `local_pool`, that is a pointer to a separate memory pool. Local information in a test plan is kept in this buffer and referenced as offsets from `local_pool`. The memory pool is managed as an activation stack: An activation is allocated and deallocated in a method's prologue and epilogue, and `local_pool` is set to the current activation. On a thread switch, the RVM switches a thread's stack, and hence, `local_pool` will be switched, causing the memory pool to also be switched.

The concern with GC is where to allocate data and code space for the instrumentation. If the storage is allocated as part of the application context, then there may be interactions with GC. In particular, it is difficult for GC to track references involving binary-level instrumentation inserted without its knowledge. To avoid this problem, the dynamic instrumenter allocates its own memory from the operating system to hold instrumentation code and data. This memory buffer is not visible to the RVM and avoids any interactions with GC.

5.2. Dynamic Instrumentation for the x86

To implement test probes, the dynamic instrumenter uses *fast breakpoints* [12]. A fast breakpoint replaces an instruction in the target machine code with a jump to a breakpoint handler. The breakpoint handler calls the test payload and it executes the original instruction that was replaced by the jump. We use fast breakpoints because they have low overhead and can be easily inserted and removed on binary code. When implementing fast breakpoints there are essentially two choices. The first choice is to execute the original instruction as part of the breakpoint handler. The second choice copies the instruction back to its original location where it is executed when the breakpoint handler completes. Hence, these breakpoints are "transient" and similar to the invisible breakpoints used by debuggers to transparently track program values and paths.

A consequence of transient breakpoints is probes do not remain in a test region once executed. If a permanent probe is needed, then the test planner has to re-insert the probe. Re-insertion can be done by placing probes in the successors to a block that needs a permanent probe. The successor probes re-insert the original probe when executed and remove themselves and their siblings. While fast breakpoints can be implemented to make them permanent, variable length instruction sets complicate the implementation. Instead, transient breakpoints simplify and increase the portability of the instrumentation interfaces.

On the x86, copying the instrumented instruction back to its original location works better than executing the instruction in the handler. If the instrumented instruction is executed in the handler, then instructions have to be decoded to find instruction boundaries because an entire instruction must be copied to the handler. Indeed, in some cases, multiple instructions may have to be copied and executed in the handler because the breakpoint jump can span several instructions. The breakpoints do not know anything about the instructions where a breakpoint is inserted, which significantly simplified their implementation. The trade-off is for a breakpoint to remain, it must be re-inserted after the original instruction is executed.

6. EXPERIMENTS

Using SPECjvm98 benchmarks [20], we performed experiments to measure Jazz's performance and memory needs. The experiments were run on a unloaded 2.4 GHz Pentium IV with 1 GB of memory and RedHat Linux 7.3. All results are averages over three program runs. The test specification for the experiments covers all loaded methods. For def-use testing, the specification selects all variables and all def-use pairs. The test inputs are the data sets provided in SPECjvm98.

6.1. Branch Coverage Testing

To investigate the efficiency of demand-driven testing, we compared the performance and memory requirements of our technique to a traditional approach based on static instrumentation. For branch testing, the coverage on the benchmarks was 38.9% to 58%. Both approaches reported the same coverages.

The performance and memory demands of the two approaches are shown in Table 3. The second column is the run time of the benchmark without instrumentation. The

Program	Base Time (Sec.)	Performance (slowdown)		Memory (kilobytes)	
		Demand	Static	Demand	Static
compress	28.1	1.1	3.42	7.9	7.5
jess	21.5	1.19	1.71	50.2	60.3
db	44.7	0.98	1.12	9.7	8.9
javac	26.2	1.23	1.38	178.9	186.0
mpegaudio	25.4	1.01	2.2	24.7	29.5
mtrt	13.8	1.56	2.3	22.4	23.0
jack	17.8	1.16	1.16	73.4	78.0

Table 3: Branch coverage overhead

third and fourth columns compare the slowdown of demand-driven testing along a path ("Demand") and static instrumentation ("Static"). The slowdown is the ratio of the run time with testing over the run time without instrumentation. The fifth and sixth columns compare the memory requirements of the two approaches.

The results for static instrumentation were gathered from a tool that we implemented. This tool instruments a program's binary code before run time and does not remove the instrumentation. It is similar to tools such as Rational PurifyPlus [10], JCover [11], and Clover [4]. We implemented our own tool to make it easier to compare the performance and memory overheads of the demand-driven and static instrumentation approaches on the same framework and benchmarks. Both tools do the same actions at a probe, except the tool with static instrumentation does not insert or remove probes.

The memory results in Table 3 include the space for local and global storage, the PLT, and the breakpoint handler and payload code. Because the memory demands change as probes are inserted and removed in the demand-driven approach, the memory sizes are maximums over a program run.

Performance. The slowdown over uninstrumented code for the demand-driven approach varies from 0.98 on *db* to 1.56 on *mtrt*, with a 1.18 average slowdown. The performance overhead is related to how quickly branches are covered. The benchmarks with the best performance, *compress*, *db*, and *mpegaudio*, have tight loops that cover edges quickly. In other cases, such as *mtrt*, many edges can be covered (50% for *mtrt*), but some probes are not as removed quickly and incur overhead. For example, a probe in *mtrt* stays 27 times longer than a probe in *mpegaudio*. Programs with many try-catch blocks, such as *mtrt*, can exhibit this behavior.

In comparison to branch testing with static instrumentation, the demand-driven technique is 1.01 to 3.11 times faster (average is 1.63). The ability to remove probes is important to reducing overhead, particularly in loops and when coverage converges quickly (i.e., the same paths are taken). Indeed, probes in the dynamic approach have a much higher cost (average 806 ns) than the static probes (average 32 ns), which can be inlined and do not modify instructions at run time. Yet, the ability to remove the probes far outweighs their higher cost.

Memory Requirements. Table 4 shows that the demand-driven approach needs 7.9 to 178.9 (average 52.5) kilobytes of memory. The memory requirements depend on two factors. First, the size of the result and PLT tables is important. The table sizes are determined by the number of basic blocks and how many probes are inserted/removed in a block. Second, the requirements depend on the total size of the breakpoint handlers, which is determined by the maximum number of probes that are active in the program at any one time. For example, *javac* has 1,116 active probes, where each probe needs 31 bytes and the memory footprint of the breakpoint handlers is 34,596 bytes. *compress*, on the other hand, has only 71 active probes, requiring 2,201 bytes.

As Table 3 shows, demand-driven testing usually has smaller memory requirements than the static approach. Although PLTs are larger with the dynamic technique, there are many fewer active probes, resulting in a smaller memory footprint. In fact, the ability to both insert and remove probes on-demand keeps the number of active probes low. For example, *jack* has 2,025 active probes in the static technique and requires 78 KB of memory, while with dynamic instrumentation, it has a maximum of 473 active probes and 73 KB of memory.

From the results in this section, we conclude that demand-driven branch coverage testing is effective in both performance and memory demands. The technique has much less performance overhead than an equivalent approach with static instrumentation and its memory needs are on par or better than the static technique.

6.2. Node and Def-Use Testing

To show the flexibility of our approach to support other structural tests, we implemented test planners for node and def-use coverage. For these tests, our tool reported 75% to 90.6% node coverage and 66.9% to 90.5% def-use coverage. We also measured performance and memory requirements, as shown in Table 4.

Performance. Node testing has a small performance impact, with a maximum slowdown of 1.04 and an average of 1.03 (excluding *mpegaudio*). In this test, the overhead is minimal because probes are removed on-demand and executed only once. Similar to branch testing, tight loops with large iteration counts quickly amortize the cost of executing a probe only once. In *mpegaudio*, performance is improved slightly because the execution of probes positively affected machine behavior, such as the instruction cache hit rate.

Def-use has slowdowns from 1.04 to 3.79, with an average of 2.27. The slowdown depends on how quickly probes for definitions can be removed. A probe at a definition remains until *all* reachable uses are covered. A probe for a use, on the other hand, can be removed immediately once it is hit. Hence, probes at definitions cause most of the test overhead. The number of def-use pairs is also a factor; it typically take longer to cover a larger number of pairs. Finally def-use probes are more expensive (1065 ns average cost vs. 780 ns for branch coverage), which also contributes to the overhead.

Interestingly, def-use pairs can take a long time to cover, even when nodes and branches are covered quickly. Although *jess* has a 1.04 slowdown for node testing and 1.19

Program	Performance (slowdown)		Memory (kilobytes)	
	Node	Def-Use	Node	Def-Use
compress	1.0	1.41	4.4	43
jess	1.04	3.79	34.1	258.9
db	1.0	1.04	5.0	77.2
javac	1.04	3.07	107.3	1332.7
mpegaudio	0.99	2.45	15.0	114.0
mtrt	1.03	2.05	12.5	90.4
jack	1.03	2.05	46.5	405.6

Table 4: Node and def-use overhead

for branch testing, it has a 3.79 slowdown for def-use. This benchmark has a large number of variables with many def-use pairs that are never covered, which causes probes to remain and incur overhead. Indeed, *jess* executes 47 times more probes per second of run-time than *db*, leading to a larger slowdown.

Memory Requirements. Node coverage has small memory demands (4.4–107.3 KB, average 32.1 KB) because its PLT is small. Def-use has larger memory requirements, ranging from 43 to 1,332.2 KB (average 332 KB). Typically, def-use inserts and removes more probes at a location than branch or node coverage, and hence, the size of a PLT entry and breakpoint handler is larger. It also takes longer for probes to be removed, which results in more active probes (e.g., *javac* has 1,116 active probes in branch testing and 1,663 probes in def-use testing) and higher memory demands.

As this section has demonstrated, our approach is flexible and can accommodate several types of coverage testing. The overheads are particularly encouraging for such a general and flexible framework.

7. RELATED WORK

There are a number of commercial tools that perform coverage testing on Java programs, including JCover [11], Clover [4] and IBM's Rational PurifyPlus [10]. Of these tools, only Clover does both branch coverage and statement coverage; none do def-use coverage. Unlike our framework, these tools statically instrument a program with probes that remain for the entire execution of the tested program. Our demand-driven approach does not modify the program source code or class files. Instead, it operates on binary code, enabling the use of dynamic instrumentation. Our framework also avoids unnecessary overhead due to static instrumentation by removing instrumentation as soon as it is not needed.

Tikir and Hollingsworth [21] use a dynamic technique for node coverage with Dyninst [9]. As in Jazz, the Dyninst tool dynamically inserts instrumentation on method invocations for node coverage. Unlike our approach however, instrumentation is only removed via a garbage collection process. Instead of removing instrumentation as soon as possible, a separate thread periodically removes the instrumentation. However, this instrumentation remains until collected, even when it is not needed. In comparison to their coverage tool, Jazz works well. They report slowdowns of 1.001 to 2.37 (average 1.36) for C programs, while Jazz's slowdowns are 0.99 to 1.04 (average 1.03) for Java programs. Although it is difficult to directly compare these results, the demand-driven technique has better performance because instrumentation is inserted on paths and removed *immediately* rather than periodically.

Path profiles can be used to compute code coverage [1]. Path profiling transforms a CFG into a directed acyclic graph (DAG) and assigns values to the nodes so that each unique path from the entry to the exit of the DAG produces a unique sum. However, the instrumentation needed path profiling cannot be removed. Because the results presented in [1] do not include the overhead for edge labeling, instrumentation insertion, or path regeneration, a performance comparison is difficult.

The concept of fast breakpoints was pioneered by Kessler [12] but these breakpoints were not applied in a general manner to dynamically instrument programs for structural testing. Dynamic instrumentation systems like PIN [18], Dyninst [9] and Paradyn [13] used a technique similar to ours to instrument a program. Like our framework, Dyninst is intended to be general, with a language for specifying instrumentation [9]. However, their instrumentation techniques were not designed to support test development.

8. SUMMARY

This paper addresses the need for scalable and flexible testing tools by developing a framework where test specifications are automatically converted to an implementation of the specifications. The framework is flexible in that both standard and custom structural tests can be incorporated into tools created through the framework. Different tests, code regions, code granularities, and coverages can all be incorporated. The framework also generates tools that are scalable because the instrumentation is dynamically inserted on demand as the program executes. Instrumentation is also deleted at the time it is no longer needed. Experimental results indicate savings in both time and memory over statically placing instrumentation before program execution.

9. ACKNOWLEDGEMENTS

This research has been supported in part by the National Science Foundation, Next Generation Software, award CNS–0305198. The project was also supported in part by an IBM Eclipse Innovation Grant (2003).

10. REFERENCES

[1] T. Ball and J. R. Larus, "Efficient path profiling", *Int'l. Symp. on Microarchitecture*, 1996.

[2] M. Burke, J-D. Choi, S. Sink, et al., "The Jalapeno dynamic optimizing compiler for Java", ACM Java Grande Conference, 1999.

[3] B. Childers, M. L. Soffa, J. Beaver et al., "SoftTest: A framework for software testing of Java programs", *Eclipse Technology eXchange Workshop*, 2003.

[4] Clover, http://www.cenqua.com/clover/.

[5] Eclipse Integrated Development Environment, http://www.eclipse.org

[6] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria", *IEEE Trans. on Software Engineering*, 14(10), October 1988.

[7] P. G. Frankl, S. N. Weiss, and E. J. Weyuker, "ASSET: A system to select and evaluate tests", *Proceedings of the IEEE Conference on Software Tools*, 1985.

[8] M. J. Harrold and M. L. Soffa, "Interprocedural data flow testing", *Testing, Analysis and Verification Symp.*, 1989.

[9] J. Hollingsworth, B. Miller, M. Goncalves, et al., "MDL: A language and compiler for dynamic program instrumentation", *Conf. on Parallel Architecture and Compilation Techniques*, 1997.

[10] IBM, Rational PurifyPlus, http://www.ibm.com/rational.

[11] JCover, http://www.codework.com/JCover/

[12] P. Kessler, "Fast breakpoints: Design and implementation", *ACM SIGPLAN Conf. on Programming Languages, Design and Implementation*, 1990.

[13] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, et al., "The Paradyn parallel performance measurement tools", *IEEE Computer*, 11(28), 1995.

[14] J. Misurda, J. Clause, J. Reed, B. R. Childers, and M. L. Soffa, "Jazz: A Tool for Demand-Driven Structural Testing", *International Conference on Compiler Construction*, 2005.

[15] L. Osterweil et al., "Strategic directions in software quality", *ACM Computing Surveys*, Vol. 4, 1996.

[16] C. Pavlopoulou and M. Young, "Residual test coverage monitoring", *Int'l. Conf. on Software Engineering*, 1999.

[17] W. Perry, *Effective Methods for Software Testing*, John Wiley & Sons, Inc., New York, New York, 1995.

[18] Pin, http://rogue.colorado.edu/Pin/

[19] S. Rapps and E. Weyuker, "Selecting software test data using data flow information", *IEEE Trans. on Software Engineering*, 11(4):367–375, 1985.

[20] Standard Performance Evaluation Corporation, http://www.spec.org/jvm98

[21] M. Tikir and J. Hollingsworth, "Efficient instrumentation for code coverage testing", *Int'l. Symp. on Software Testing and Analysis*, 2002.