Coupling-based Criteria for Integration Testing *

Zhenyi Jin and A. Jefferson Offutt
Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030-4444
email: {zjin,ofut}@isse.gmu.edu

July 29, 1998

Abstract

Integration testing is an important part of the testing process, but few integration testing techniques have been systematically studied or defined. The goal of this research is to develop practical, effective, formalizable, automatable techniques for testing of connections between components during software integration. This paper presents an integration testing technique that is based on couplings between software components. This technique can be used to support integration testing of software components, and satisfies part of the USA's Federal Aviation Authority's (FAA) requirements for structural coverage analysis of software. The coupling-based testing technique is described, and the coverage criteria for three types of couplings are defined. Techniques and algorithms for developing coverage analyzers to measure the extent to which a test set satisfies the criteria are presented, and results from a comparative case study are presented.

Keywords: Integration testing, Software coupling, Software testing.

1 Introduction

Testing software is one of the most common methods for assuring quality of complex computer software systems. The general purpose of the research reported in this paper is to formalize, via new coverage criteria, routine aspects of testing at the integration level. Formal coverage criteria offer the tester ways to decide what test inputs to use during testing, making it more likely that the tester will find any faults in the program and providing greater assurance that the software is of high quality and reliability. Such criteria also provide stopping rules and repeatability.

A program unit, or procedure, is one or more contiguous program statements having a name by which other parts of the software can invoke it [SMC74]. A module[1] is a collection of related units, collected in a file, package, module, class, etc. [Som92]. *Unit and module testing* (or just unit testing) is considered

*Partially supported by the National Science Foundation under grants CCR-93-11967 and CCR-98-04111.

[1] Previous papers considered the terms module and unit to be synonymous, thus used the term module when this paper uses unit. This paper chooses to differentiate between the two terms and use module to emphasize the modularity in design, particularly with regard to the trend of data abstraction starting with Parnas' classic paper [Par72] and continuing through the current OO languages.

to be to be testing of program units and modules independently from the rest of the software system. In some cases, such as when building general-purpose library modules, unit testing is done without knowledge of the encapsulating software system. *Integration testing* refers to testing interfaces between units and modules to assure that they have consistent assumptions and communicate correctly [Bei90]. *System testing* is testing applied to an entire integrated system.

Although much is known about unit and module testing, and many techniques and tools are available, the literature does not offer many formally defined techniques that are designed to be used during integration testing. Thus testers are often left with performing integration testing in ways that are ad-hoc and ineffective, leading to less reliable software.

The recent trend towards data abstraction and object-oriented software has led to an increased emphasis on modularity and reuse. This means that integration testing is becoming more important than in the past. One of the benefits of modularity is that the software components can be tested independently, which is usually done by programmers during unit and module testing. Unit testing techniques are sometimes applied during integration, but using these techniques for integration testing suffers from two problems. First, the unit testing techniques are usually too expensive to be practically applied during integration, and second, there is no reason to believe that they will find the kinds of faults that appear during integration. Some software faults cannot be detected during unit testing; these are often faults in the interfaces between units. Thus, specific tests must be designed to detect integration faults. The USA's Federal Aviation Authority (FAA) has recognized the increased importance of modularity and integration testing by imposing requirements on structural coverage analysis of software that "the analysis should confirm the data coupling and control coupling between the code components" [SC-92], pg. 33, section 6.4.4.2.

In this paper, it is suggested that integration testing can be improved by using a *structural coverage criterion*. A method is presented for doing this that is based on software couplings. *Coupling* between two units measures the dependency relations between two units by reflecting the interconnections between units; faults in one unit may affect the coupled unit [CY79]. Coupling provides summary information about the design and the structure of the software. Since faults are found during integration testing exactly where couplings typically occur, a new coupling-based testing technique is proposed. Criteria are defined that require that each connection between program units be covered. These criteria are designed to improve integration testing, and can be used to satisfy part of the FAA requirements on structural coverage analysis.

2 Coupling

A good software system should exhibit high cohesion in a module and low coupling between units. Coupling between two units increases the interconnections between the two units and increases the likelihood

that a fault in one unit may affect others. Also, increased coupling may lower the understandability and maintainability of a software system. Coupling was ordered into eight different levels by Page-Jones [PJ80] according to their effects on the understandability, maintainability, modifiability, and reusability of the coupled units. For each coupling level, the shared data (parameters, global variables, etc.) are classified by the way they are used.

The coupling levels are used to evaluate the complexity of software system designs. Troy and Zweben [TZ81] empirically related coupling levels to the number of faults in software. The experiment showed that coupling between units is an important factor in software quality and a good indicator of the number of faults in the software, but the study was based on subjective interpretations of design documents instead of actual code. Offutt et al. [OHK93] designed algorithms to automatically measure the coupling levels between each pair of units in a program. The eight levels of coupling were also extended to twelve levels, providing a finer grained measure of coupling.

The coupling levels are defined between pairs of units, say A and B. In Offutt and Harrold's algorithms [OHK93], for each coupling level, the parameters are classified by the way they are used. Uses are classified into computation uses (C-uses), predicate uses (P-uses) (as defined in data flow testing [FW88]), and indirect uses (I-uses) (as defined by Offutt and Harrold [OHK93]). A *C-use* occurs when a variable is used on the right side of an assignment statement, in an output statement, or a procedure call. A *P-use* occurs when a variable is used in a predicate statement. An *I-use* occurs when a variable is used in an assignment to another variable, and the defined variable is later used in a predicate; the I-use is considered to be in the predicate rather than in the assignment. The 12 levels of coupling are listed below. Most of the terminology comes from the early work by Yourdon and Constantine [CY79] and Page-Jones [PJ80], and some comes from Offutt and Harrold's paper.

0. Independent coupling – A does not call B and B does not call A, and there are no common variable references or common references to external media between A and B.

1. Call coupling – A calls B or B calls A but there are no parameters, common variable references, or common references to external media between A and B.

2. Scalar data coupling – Some scalar variable in A is passed as an actual parameter to B and it has a C-use but no P-use or I-use.

3. Stamp data coupling – A record in A is passed as an actual parameter to B and it has a C-use but no P-use or I-use.

4. Scalar control coupling – Some scalar variable in A is passed as an actual parameter to B and it has a P-use.

5. Stamp control coupling – A record in A is passed as an actual parameter to B and it has a P-use.

6. Scalar data/control coupling – Some scalar variable in A is passed as an actual parameter to B and it has an I-use but no P-use.

7. Stamp data/control coupling – A record in A is passed as an actual parameter to B and it has an I-use but no P-use.

8. External coupling – A and B communicate through an external medium such as a file.

9. Nonlocal coupling – A and B share references to the same nonlocal variables. A nonlocal variable is visible to a subset of the units in the system, typically within one module. For example, a variable declared in the body part of an Ada package is nonlocal for that package.

10. Global coupling – A and B share references to the same global variable; a global variable is visible to the entire system.

11. Tramp coupling – A formal parameter in A is passed to B as an actual parameter; B subsequently passes the corresponding formal parameter to another unit without B having accessed or changed the variable.

The previous literature intended this to be an ordered list, with lower coupling levels indicating a higher quality design. While this high level of detail is useful for qualitative analysis of designs, it has been found that 12 levels of coupling are not needed for testing. For testing purposes, many of the 12 levels can be combined and classified into four unordered types. Furthermore, an ordered list of qualitative coupling levels is not needed. For testing, these couplings can be viewed as unordered types. The four coupling types are now defined as:

- *Call coupling* is the same as in the previous levels.
- *Parameter coupling* refers to all parameter passing. This type combines scalar data coupling, stamp data coupling, scalar control coupling, stamp control, scalar data/control coupling, stamp data/control coupling and tramp coupling.
- *Shared data coupling* refers to procedures that both refer to the same data objects. This type combines nonlocal coupling and global coupling.
- *External device coupling* refers to procedures that both access the same external medium. This type is analogous to external coupling.

3 Coupling-based Testing Criteria

An important problem in software testing is deciding when to stop. Test cases are run on test programs to find failures. Unfortunately, the entire domain of the program (which in most cases is effectively

infinite) cannot be exhaustively searched. Adequacy criteria are therefore defined for testers to decide whether software has been adequately tested for a specific testing criterion [FW88].

Test requirements are specific things that must be satisfied or covered, for example, reaching statements are the requirements for statement coverage, killing mutants are the requirements for mutation, and executing DU pairs are the requirements in data flow testing. A *testing criterion* is a rule or collection of rules that impose requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percent of requirements that are satisfied.

To make integration testing a manageable process, testing must be guided by the modularization of the software. Each module to be integrated should pass an isolated test. Integration testing must be performed at a higher level of abstraction – looking at program units as atomic building blocks and focusing on their interconnections. In this paper, criteria for integration testing are proposed. These criteria are expected to be used both to guide the testers during integration testing and to help the testers find a rational, mathematical-based point at which to stop testing. First, some basic definitions from data flow analysis are provided. Then the kinds of testing that should be done on the coupling types are described, and then four different test criteria are defined, each of which requires a different amount of testing.

3.1 Basic Definitions

The coupling-based testing criteria are based on the design and data structures of the program, and on the data flow between the program units. Thus, data flow definitions are needed to support coupling testing criteria definitions. Some of the traditional definitions are given here; most of them are taken from White [Whi87]. New definitions are given later.

A *basic block* is a maximum sequence of program statements such that if any one statement of the block is executed, then all statements in the block are executed. A basic block has only one entry point and one exit point. A *control flow graph (CFG)* of a program is a directed graph that represents the structure of the program. Nodes are basic blocks, and edges represent potential control flow from node to node.

A *definition (def)* is a location in the program where a value for a variable is stored into memory (assignment, input, etc.). A *use* is a location where a variable's value is accessed. A computation use (*C-use*) is a node where a variable is used in a computation, as a functional parameter or in an output statement. A predicate use (*P-use*) is an edge where a variable is used in a decision. A *def-clear path* for a variable X through the CFG is a sequence of nodes that do not contain a definition of X.

A *caller* is a unit that invokes another unit, the *callee*. An *actual parameter* is in the caller, its value is assigned to a *formal parameter* in the callee. The *interface* between two units is the mapping of actual to formal parameters.

3.2 Coupling-based Testing

Coupling-based testing requires that the program execute from definitions of actual parameters through calls to uses of the formal parameters. Therefore, different coupling paths are defined. These are based on the three types of couplings defined in Section 2. Each coupling path is precisely defined in this section. The underlying premise of the coupling-based testing criteria is that to achieve confidence in the interfaces between integrated program units, it must be ensured that variables defined in caller units be appropriately used in callee units. Because this technique is limited to the unit interfaces, it is only concerned with definitions of variables just **before** calls and returns to other units, and uses of variables just **after** calls and returns from the called unit.

3.2.1 Coupling-based Testing Definitions

Several definitions are introduced to formally define the coupling-based testing criteria. P is an arbitrary unit in the system, and definitions refer to the specific units P_1, P_2, P_3, and P_4. P_1 calls P_2, and P_3 and P_4 are units that do not call each other, but they share global/nonlocal variable g, and they both refer to the same external device f. When P is referred to, it is understood that the statement applies equally to all P_i. x is an actual parameter in P_1 that maps to a formal parameter y in P_2. There could be more than one parameter, but only one parameter at a time is considered. For reference, these units and their call relationships are shown in Figure 1.

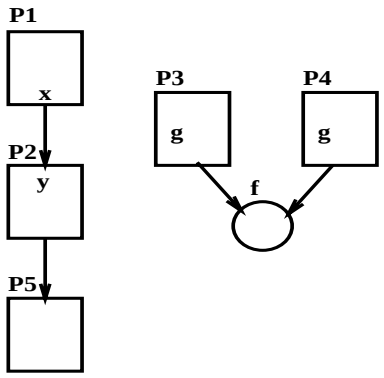

Figure 1: Example Call Graphs for Definitions

It is assumed that the control-flow graph for each unit is present, so $G_P = (N_P, E_P)$, where N_P is the set of nodes in P and E_P is the set of edges. V_P is the set of all variables referenced in P. $def(P,V)$ is the set of nodes in unit P that contain a definition of a variable V, and $use(P,V)$ is the set of nodes in P that contain a use of V. *Call_site* is a node in P_1 from which P_2 is called. One more unit, P_5 is also referred to, which is called by P_2 in the case of tramp coupling (it is the case when a formal parameter in P_1 is passed to P_2 as an actual parameter; P_2 subsequently passes the corresponding formal parameter to another unit without using or defining the variable). The following definitions and predicates are

used to define the criteria.

- **Call (P_1, P_2, call_site, $x \rightarrow y$)**: TRUE if unit P_1 calls P_2 at call_site and actual parameter x is mapped to formal parameter y. This is variable specific; if there is more than one parameter, they are analyzed one at a time. The value is FALSE if there is no such call at the given call_site.

- **Return (v)**: Nodes from which values for v are returned in a unit. Note that this includes explicit `return` statements as well as implicit returns at the end of units.

- **Start (P)**: The first node in P. It is assumed that there is one entry point.

- **Coupling-def**: A *coupling – def* is a node that contains a definition that can reach a use in another unit on at least one execution path. There are three types of coupling-defs:

  1. **Last-def-before-call**: The set of nodes that define x and for which there is a def-clear path from the node to the call_site in P_1. More formally:
     - **lbc-def (P_1, call_site, x)** = $\{i, i \in N_{P_1} \mid$ node i has a definition of variable $x \bigwedge$ there is a def-clear path with respect to x from node i to call_site$\}$
  2. **Last-def-before-return**: When values are returned, (e.g., through call-by-reference parameters or a return statement), then last-def-before-return is the set of nodes that define the returned variable y, and for which there is a def-clear path from the node to the return statement. This is formally defined as:

     lbr-def (P_2, y) = $\{j, j \in N_{P_2} \mid y$ is defined in node $j \bigwedge$ there is a def-clear path with respect to y from j to Return $(y)\}$
  3. **Shared-data-def**: In the case of shared data coupling, coupling-def is defined as the set of nodes that define a nonlocal or global variable g in P_3 that is used in P_4, and for which there is a def-clear path from the def to the use. Note that it is not necessary that either P_3 call P_4 or that P_4 call P_3; the def-clear path can go through an arbitrary sequence of calls and returns. It is formally defined as:

     Shared-def (P_3, P_4, g)= $\{i, i \in N_{P_3} \mid i \in$ def $(P_3, g) \bigwedge g$ is a nonlocal or global variable $\bigwedge$ there is a def-clear path with respect to g from the definition in P_3 to the use in $P_4\}$

- **Coupling-use**: A *coupling-use* is a node that contains a use that can be reached by a definition in another unit on at least one execution path. There are three types of coupling-uses:

  1. **First-use-after-call**: In the case of call-by-reference parameters, first-use-after-call is the set of nodes i in P_1 that have uses of x and for which there exists a def-clear path with no other uses between the call statement for P_1 and these nodes. This is formally defined as:

fac-use (P_1, call_site, x) = $\{i, i \in N_{P_1} \mid$ node i has a use of variable $x \bigwedge$ there are no other uses or defs between call_site and node $i\}$

2. **First-use-in-callee**: In the case of call-by-value parameters, first-use-in-callee is the set of nodes for which parameter y in P_2 has a use, and there is at least one def-clear path with no other uses from the start statement to this use. This is formally defined as:

   fic-use (P_2, y) = $\{j, j \in N_{P_2} \mid ((y$ has a C-use at node $j) \bigvee (y$ has an I-use on edge $(i, j), i \in N_{P_2}) \bigvee (y$ has a P-use on edge $(i, j), i \in N_{P_2})) \bigwedge$ there is a path with no other use or def of y between Start (P_2) and node $j\}$

3. **Shared-data-use**: In the case of shared data coupling, coupling-use is defined as the set of nodes that use a nonlocal or global variable x. This is defined as:

   Shared-use (P_4, g) = $\{i, i \in N_{P_4} \mid i \in$ use $(P_4, g) \bigwedge g$ is a nonlocal or global variable$\}$

- **External-reference**: In the case of external coupling, the pair of references (i, j) to the same external file is called an external-reference. It is defined as:

  - **External-ref (i, j)** = $\{(i, j), i, j \in P, i, j$ reference the same external file or device$\}$

3.2.2 Coupling-defs and Coupling-uses Examples

An Ada program is used to illustrate the definitions. Ada parameters [Def83] of a subprogram can have one of three modes: *in*, *out*, and *in-out*. An *in* parameter acts as a local constant whose value is provided by the corresponding actual parameter. This roughly corresponds to the call-by-value case in the above definitions. In *out* mode, the parameter acts as a local variable whose value is assigned to the corresponding actual parameter as a result of the execution of the subprogram. This corresponds to the call-by-reference case. Ada also has an *in-out* mode, in which the parameter acts as a local variable and permits access and assignment to the corresponding actual parameter. This will correspond to both the call-by-reference and call-by-value cases. In general, these definitions work for all programming languages, but the semantics of specific language features may necessitate slight changes.

Figure 2 shows an Ada program that calculates quadratic roots. Procedure $QUADRATIC$ gives two sets of values for X, Y and Z under different control-flag values, then calls procedure $ROOT$ to calculate the roots. Solutions of the quadratic are printed if they are available, otherwise an error message is given. Figure 2 shows the **last-def-before-call**s and **first-use-in-callee** of *in* actual parameters X, Y, Z and their corresponding formal parameters A, B and C. For *out* actual parameters R_1, R_2, OK and corresponding formal parameters $ROOT_1$, $ROOT_2$ and $Result$, their **last-def-before-return** and **first-use-after-call** sets are given.

To take a close look at a particular *in* variable, it must be known where the actual parameter is last defined before the call, and where its corresponding formal parameter is first used in the callee procedure.

In $QUADRATIC$, statement 13 is a call-site for actual variable X. X is last defined in statements 4 and 8 under different control-flag values. Because there are no other definitions of X before the call-site, statements 4 and 8 are the two **last-def-before-call**s of X. In $ROOT$, A is the corresponding formal parameter of X. The first time A is used in this procedure is statement 2. Thus, statement 2 is the **first-use-in-callee** of A. As for an *out* variable, it must be known where the variable is last defined before the callee procedure ends and returns, and the first time the *out* variable is used after the call-site in the caller procedure. In $ROOT$, $ROOT_1$ is an *out* variable, and it is last defined in statement 7 before the procedure ends and returns to $QUADRATIC$, so statement 7 is the **last-def-before-return** of $ROOT_1$. Its actual parameter R_1 is first used in statement 15 after the call-site in $QUADRATIC$.

OK is an *in-out* variable, its **last-def-before-call** is in statement 12 of $QUADRATIC$, and its corresponding formal parameter $Result$ is first used in statement 3 of $ROOT$. Also, $Result$ is last defined in statements 4 and 9 under different cases before $ROOT$ ends. The returned value of $Result$ is passed to Ok and it is first used after the call-site in statement 14 of $QUADRATIC$. So for an *in-out* parameter, both its *in* and *out* couplings are checked.

```
                procedure QUADRATIC is
                . . .
1.              begin
2.                GET (Control_Flag);
3.                if (Control_Flag = 1) then
4.                  GET (X);          -- last-def-before-call (X)
5.                  GET (Y);          -- last-def-before-call (Y)
6.                  GET (Z);          -- last-def-before-call (Z)
7.                else
8.                  X := 10.0;        -- last-def-before-call (X)
9.                  Y := 0.0;         -- last-def-before-call (Y)
10.                 Z := 12.0;        -- last-def-before-call (Z)
11.               end if;
12.               OK := TRUE;       -- last-def-before-call (OK)
13.               ROOT (X, Y, Z, R_1, R_2, OK);    -- call-site
14.               if OK then          -- first-use-after-call (OK)
15.                 PUT (R_1);      -- first-use-after-call (R_1)
16.                 PUT (R_2);      -- first-use-after-call (R_2)
17.               else
18.                 PUT ("No solution");
19.               end if;
20.             end QUADRATIC;
```

Caller

```
procedure ROOT (A, B, C: in FLOAT; ROOT_1, ROOT_2: out FLOAT;
                Result: in out BOOLEAN) is
. . .
D : REAL;
. . .
1.  begin
2.    D := B**2-4.0*A*C;                    -- first-use-in-callee (A, B, C)
3.    if (Result and D < 0.0) then          -- first-use-in-callee (Result)
4.      Result := FALSE;                    -- last-def-before-return (Result)
5.      return;
6.    end if;
7.    ROOT_1 := (-B+sqrt(D))/(2.0*A);     -- last-def-before-return (ROOT_1)
8.    ROOT_2 := (-B-sqrt(D))/(2.0*A);     -- last-def-before-return (ROOT_2)
9.    Result := TRUE;                     -- last-def-before-return (Result)
10. end ROOT;
```

Callee

Figure 2: An Example of Coupling-uses and Coupling-defs

Figure 3 illustrates coupling-uses and coupling-defs for global and nonlocal variables. The variable g, which can be either global or nonlocal, is defined in procedures P1 and P2, and it is used in procedures Q1 and Q2. So P1 and P2 each have a shared-data-def of g, and Q1 and Q2 each have a shared-data-use of g. External references are similar to the shared-data case, except definitions and uses are writes and

reads.

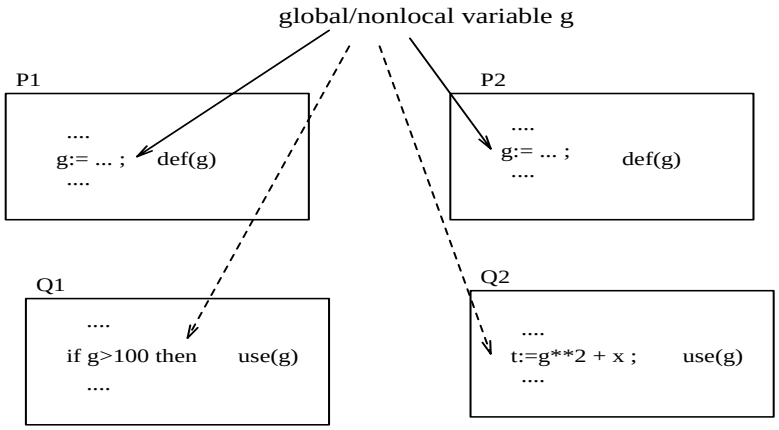

Figure 3: An Example of Shared-data Defs and Uses

3.2.3 Coupling-based Testing Path Definitions

A coupling path is a path between two program units from a definition to a use, or between two references, and that satisfies certain other requirements. The other requirements depend on the type of coupling between the two units. The three types of coupling paths use the definitions above. Three types of coupling paths are now described, first informally, then formally.

1. Parameter coupling path: For each actual parameter x, and each last definition of x before a call_site, there is a parameter coupling path from the last definition, to the call_site, and to each first use of the formal parameter y in P_2. This is defined as:

   - parameter-coupling (P_1, P_2, call_site, x, y) = $\{(i, j), i \in N_{P_1}, j \in N_{P_2} \mid i \in$ lbc-def (P_1, call_site, x) $\bigwedge j \in$ fic-use (P_2, y)$\}$

   If a parameter x is call-by-reference, then there is also a parameter coupling path from each last definition before return of the formal parameter y in P_2 to each first use after the call of x in P_1. This is defined as:

   - parameter-coupling (P_1, P_2, call_site, x, y) = $\{(j, i), i \in N_{P_1}, j \in N_{P_2} \mid j \in$ lbr-def (P_2, y) $\bigwedge i \in$ fac-use (P_1, call_site, x)$\}$

2. Shared data coupling path: For each nonlocal or global variable g that is defined in P_3 and used in P_4, and each definition of g in P_3, there is a shared data coupling path that is definition-clear with respect to g from the definition to each first use of g in P_4. It is defined as:

- Shared-data-coupling $(P_3, P_4, g) = \{(i, j), i \in N_{P_3}, j \in N_{P_4} \mid i \in \text{shared-def } (P_3, g) \bigwedge j \in \text{shared-use } (P_4, g)\}$

3. External device coupling path: For each pair of references (i, j) to the same external device, an external device coupling path executes both i and j on the same execution path.

The external device coupling path is quite a bit less restrictive than the other types of coupling paths. This is because in file accesses, and even more so in other external devices, considering definitions and uses does not make sense. In data flow, definitions and uses are considered because of the philosophy that values of definitions should be used; in files, even writes and reads are not necessarily related. Thus the notions of definition-clear and definition-use pairs do not apply to external device coupling.

3.2.4 Coupling Path Examples

Figure 4 illustrates the parameter coupling paths for several parameters in the $QUADRATIC$ program. Consider the variable X, which is passed to the formal parameter A. Test cases should execute paths from **last-def-before-call**s to **first-use-in-callee**s. The corresponding paths, with statement numbers in ROOT shown in boldface, are (4-13-**1**-**2**) and (8-13-**1**-**2**). The path (**7**-**10**-15) represents the parameter coupling path for *out* variable $R_1/ROOT_1$ that starts from the **last-def-before-return** to the **first-use-after-call**. Also shown are the paths for *in-out* variable $OK/Result$. Test cases should execute paths from the **last-def-before-call** to the **first-use-in-callee** path, (12-13-**1**-**3**), as well as from **last-def-before-return**s to the **first-use-after-call** paths, (**4**-**10**-14) and (**9**-**10**-14).

Figure 5 illustrates the shared data coupling paths for global data g from the example in Figure 3. The solid lines represent the paths from two shared-data-defs of g in P1 and P2 to the shared-data-use of g in Q1, and the dashed-line represents paths to the shared-data-use of g in Q2. External device coupling paths are similar except that they are between different references to the file.

3.3 Testing Criteria

The data flow testing criteria are extended to the following four coupling test criteria. These criteria are similar to the standard data flow levels [FW88]. They provide an increasing amount of coverage, at more cost. It is hoped that this will provide a practical range of cost/benefit choices. These definitions are written in a general form, so as to apply to all three types of coupling defined in the previous section. In the definitions, P_1 and P_2 are units in the software system.

- *Call coupling* requires that the set of paths executed by the test set T covers all call_sites in the system.
- *All-coupling-defs* requires that for each coupling-def of a variable x in P_1, the set of paths executed by the test set T contains at least one coupling path to **at least one** reachable coupling-use.

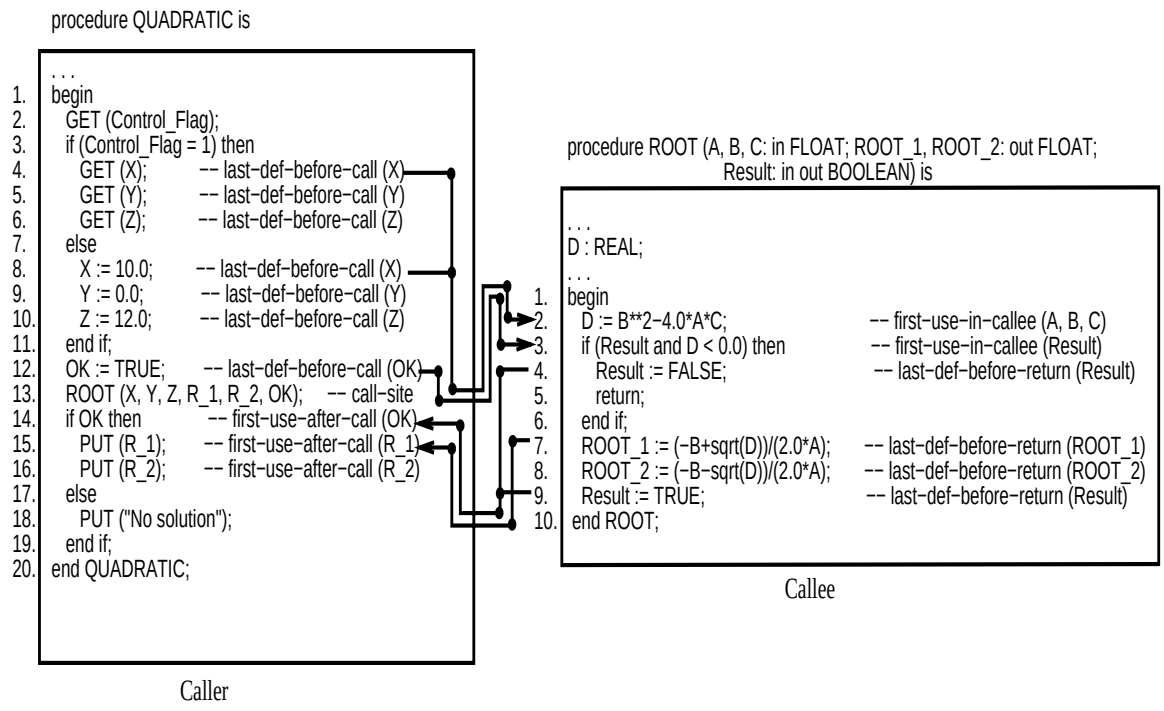

Figure 4: First Example of Coupling Paths

- *All-coupling-uses* requires that for each coupling-def of variable x in P_1, the set of paths executed by the test set T contains at least one coupling path to **each** reachable coupling-use.

- *All-coupling-paths* is intended to require that **all** coupling paths be executed, except of course for the problem that if there is a loop, the number of subpaths becomes infinite. In previous work of this nature (such as the All-DU-Paths criterion [FW88]), this problem has been handled in a variety of ways, some of which led to confusing or ambiguous definitions. A novel technique is introduced that depends on unique sets of nodes within the coupling paths.

  A *subpath set* is defined to be the set of nodes on some subpath. There is a many-to-one mapping between subpaths and subpath sets; that is, if there is a loop within the subpath, the associated subpath set is the same no matter how many iterations of the loop are taken. A coupling path is a subpath, so a *coupling path set* is the set of nodes on a coupling path.

  Between any pair of coupling-defs and coupling-uses, there are a finite number of coupling path sets. Thus, *All-coupling-paths* requires that for each coupling-def of variable x, the set of paths executed by test set T contains all **coupling path sets** from the coupling-def to all reachable coupling-uses.

  Note that if there is a loop involved, all-coupling-paths requires two test cases; one for the case when the loop body is not executed at all, and another that executes the loop body some arbitrary

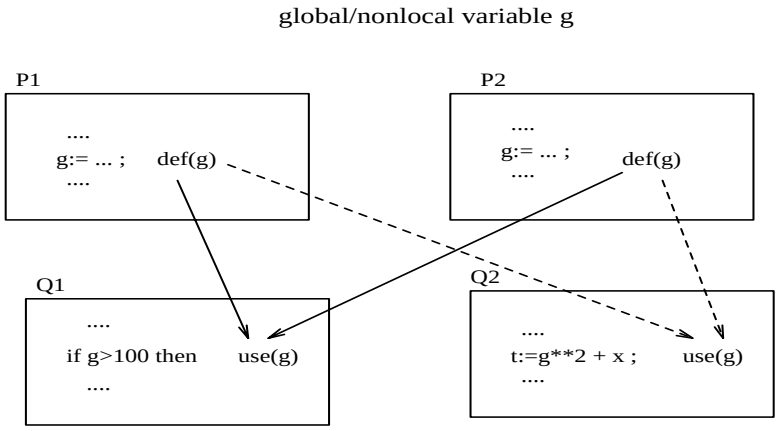

Figure 5: Second Example of Coupling Path

number of times.

Which criterion to use should be made based on the requirements of the software system as well as the cost/benefits choices. Additionally, it is possible to leave out certain types of coupling paths.

A common method to compare testing criteria is the subsumption relationship [FW88]. Criterion A is said to subsume criterion B if and only if every test set that satisfies A also satisfies B. Similar to the data flow testing subsumption hierarchy, these four coupling testing levels have the subsumption hierarchy shown in Figure 6.

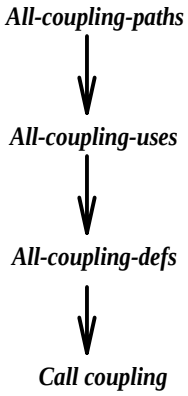

Figure 6: Coupling Testing Subsumption Hierarchy

4 Coupling Coverage Analysis

Structural coverage analysis is needed to determine whether all couplings have been covered. The coupling-based testing criteria defined in the previous section provide test requirements that must be

satisfied during testing. Test cases can then be generated specifically to satisfy each test requirement or they can be generated by some other method and techniques can be used to check whether all requirements have been satisfied. This section describes how this analysis step can be automated. The primary representation model used is the *coupling graph*, which is a directed graph: C = (M, E, F, A).

- M is a finite multi-set of nodes representing units in the system or subsystem. A node is depicted by a rectangle. Nodes in the graph are repeated when called from different sites so as to make the coupling graph more understandable.

- F is a finite multi-set of nodes representing external files that a unit may write to or read from. It is represented by small circles containing the device names.

- E is a finite set of directed edges that connect nodes in M and unit nodes to external device nodes. Edges between unit nodes indicate the coupling relations of different units and are referred to as *call edges*. When unit A calls unit B, the edge starts from node A and ends at node B. If the parameter being represented is call-by-reference, the edge is bidirectional.

  Edges from unit nodes to external device nodes indicate the unit writes to or reads from an external device. These are called *shared device edges*.

- A is a set of annotations on nodes. Some of the nodes may reference nonlocal or global data. This is indicated at the right-hand side of the node with a (dX) or (uX) indicating definitions of X or uses of X. These annotated nodes are referred to as *shared data nodes*.

A coupling graph is structured hierarchically. The root node is the main program that calls a sequence of other units; this sequence of units then becomes the next layer of the coupling graph, and they can call other sequences of units and so on. At each layer, the left-most node is the first one to be called by its parent (first is based on a static analysis), and the right-most one is the last one to be called. Parameter coupling relations are represented by the sequence of edges of a depth-first search of the coupling graph (excluding the external device nodes). For the sake of simplicity, recursive calls and loops in the coupling relations are indicated as one edge.

Figure 7 shows a coupling graph where unit A calls B, C and D. Unit B calls E and F, and unit D calls units B, C and G. B appears twice in the graph because it is called from two different places (A and D). The double-boxed instance indicates that B has appeared before and the subgraph for B is not repeated.

Nodes F and D both define a global variable g, and unit C uses g. Units E and G both write to external device f1. Unit A calls function C, so a value is returned to A.

Three kinds of coupling relations appear in Figure 7. Direct call coupling edges are A-B, A-C, A-D, B-E, B-F, D-B, D-C, and D-G. Shared data nodes are F, D, and C, and external device edges are E-f1,

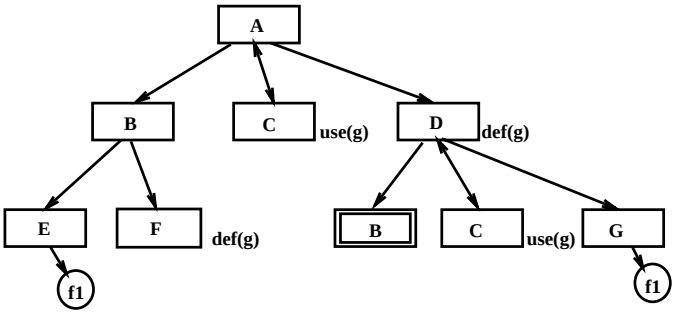

Figure 7: An Example of Coupling Graph

and G-f1.

The coupling graph can be used to measure coverage by calculating the number of the edges or nodes covered in the coupling graph. A general way to measure the coverage is given as follows:

- E = number of edges in coupling graph
- CD = number of coupling defs
- CU = number of coupling def-use pairs
- CP = number of coupling paths sets

$$call - coupling = \frac{E\_covered}{E}$$
$$coupling - def = \frac{CD\_covered}{CD}$$
$$coupling - use = \frac{CU\_covered}{CU}$$
$$coupling - path = \frac{CP\_covered}{CP}$$

4.1 Using Instrumentation to Measure Coupling Coverage

Given a coupling criterion and a set of test cases, algorithms for measuring whether the test set achieves coverage can be developed. Usually these measurements are performed using some sort of *instrumentation*, which are statements added to the program for analysis purposes. That approach is used to describe coverage measurement schemes for each of call-coupling, all-coupling-defs, all-coupling-uses, and all-coupling-paths. Because instrumentation techniques for measuring coverage are well known and common, the full algorithms are not presented. Instead, focus is on the aspects that are unique and novel to the coupling-based testing criteria. All of the procedures described in this section can be completely and efficiently automated.

- Call-coupling:

  This is the simplest criterion for coverage analysis and commercial products are available that perform this type of analysis. To measure call coupling, two things must be added to the test program. First, an integer table CTab is created that has one entry for each call-site in the program. The table is initialized to all zeros. Next, at each call-site, the following statement is added: `CTab[i] = CTab[i] + 1;` where i is an integer constant that is unique to each call-site. Note that this statement must be in the *caller*, not in the *callee*, because the procedure might be called from multiple sites, and each update to the table must be unique. After the program executes, a report is printed that states how many times each procedure call was made, and a summary statistic of the percent of calls that have been made. This report could be tabular in form, or the call graph could be shown with appropriate annotations.

  Note that the coupling table could be declared globally, as indicated in the previous paragraph, or designed in an object-oriented fashion, with operations to initialize, update, print, and save the table. The table must be saved between executions so the results can be accumulated across test cases.

- All-coupling-defs:

  To count coupling-defs, a table similar to CTab is used, except that it is indexed by last-def-before-calls and last-def-before-returns. The updating is also slightly more involved. At each definition of a variable X, the statement "`LastDef[X] = location;`" is added, where *LastDef* is a table that has one entry for each variable that is communicated between procedures, and *location* is an integer that is unique to that definition of X. *LastDef* must be initialized to a special value such as zero to indicate that X's value is undefined or from a definition that is *not* a last-def-before-return or last-def-before-call.

  The CTab is updated at the first uses, but this update is complicated by parameter passing. If a procedure $P(X)$ is called from more than one place, the formal parameter of X may be associated with a different actual variable in each call location. To handle parameter passing, an Alias table is introduced. The Alias table keeps track of the current mapping of formal to actual parameters. So at each call-site, the statement "`SetAlias(X, A);`" is added for every parameter to the called procedure. In addition, it is necessary to keep track of return values of functions. The same Alias table is used, by putting the statement "`SetAlias(X, F);`" at each return site. This alias table must be reflexive; that is, if X is aliased to A, then A is aliased to X.

  At each first use of X (first-use-after-call and first-use-in-callee), the following statement is added:

  ```
  IF LastDef[X] ∈ (last-def-before-return or last-def-before-call)
  THEN CTab[LastDef[GetAlias(X)]] = CTab[LastDef[GetAlias(X)]] +1;
  ```

This condition is *TRUE* if and only if X was defined at the *lbr* or *lbc* and the program executed a def-clear path from the def to the use.

- All-coupling-uses:

  Counting coupling-uses is very similar to counting coupling-defs, except that the CTab is indexed by *pairs* of last-defs and first-uses instead of just last-defs. The instrumentation at the defs is exactly the same, and the first-use instrumented statements are modified so CTab is indexed by a combination of `LastDef[X]` and the location of the use.

- All-coupling-paths:

  Computing coverage for all-coupling-paths is somewhat more complicated. A conceptual approach is described, then two techniques that can work in practice are derived from the approach. The technique to use should depend on what kind of information has already been computed and is already available in the system.

  The general approach is based on the following observation. **For any given path, there is at least one node in the path that is unique to that path.** Once that node is found, instrumentation can be inserted into the program to keep track of which subpath was followed. For each coupling path, the unique node u is identified (two methods for doing this are described below). At u, the statement "`SubPath[X] = location;`" is added, where `SubPath` is a table that is indexed by variables and has one entry for each coupling variable. Additionally, the same statement as used in all-coupling-defs, "`LastDef[X] = location;`", is added at the definition.

  Then, at each site fu of a first use of X (first-use-after-call and first-use-in-callee), the CTab table is updated as follows:

```
IF LastDef[X] ∈ (last-def-before-return or last-def-before-call) AND
    SubPath[X] ∈ (subpath-set)
 THEN CTab[LastDef[X], SubPath[X], fu] = CTab[LastDef[X], SubPath[X], fu] +1;
```

  This condition is *TRUE* if and only if X was defined at the *lbr* or *lbc* and the program executed a def-clear path from the def to the use along the subpath containing the node u.

  Two methods are suggested to determine the unique node u in a path. One requires a graphical representation of the program's control structure (that is, a CFG), and the other requires a set representation of the nodes on the paths.

  1. Assume a graph for which the initial node i contains the coupling-def, and the final node f contains the coupling-use. Assign the integer label of 0 to each node. Then do a breadth-first search of the graph, incrementing and decrementing the node labels as follows. If a node is

a decision node (has more than one successor), add one to each child. If a node is a junction node (has more than one predecessor) subtract one from the *smallest* predecessor and assign that value to the node. When finished, for each path, the node with the **highest** value is unique to that path.

2. Assume the subpath set exists for each path between the coupling-def and coupling-use: P_i = subpath set for path i. The unique node u for path i is then given by the formula:

$$u = P_i - (P_i \cap P_j), \forall P_j \neq P_i.$$

5 Case Study

To demonstrate the feasibility of these criteria, a study has been undertaken to compare the coupling-based testing technique with another technique that is used for integration testing, category partition. The goal was to demonstrate that the coupling criteria can be used in an effective way; it is hoped to evaluate the coupling-based testing strategies more fully in the future.

5.1 Category-Partition Testing

The category-partition testing technique [OB88, BHO89] creates functional test cases by decomposing functional specifications into test specifications for major functions of the software. It identifies those elements that influence the functionality and generates test cases by methodically varying the elements over all values of interest. Thus, it can be considered a black-box integration technique.

The category-partition method offers the test engineer a general procedure for creating test specifications. The test engineer's key job is to develop *categories*, which are defined to be major characteristics of the input domain of the function under test, and to partition each category into equivalence classes of inputs called *choices*. By definition, choices in each category must be disjoint, and together the choices in each category must cover the input domain.

5.2 Methodology

Coupling-based testing was compared with category-partition testing in terms of their fault-detection abilities. One moderate size program was used, faults created for previous experimentation [AO94] were used, and test cases were generated by hand. The all-coupling-uses criterion was chosen as a representative criterion, and the one expected to be most useful. This choice is consistent with empirical data flow researchers [FWH97, FW93, MW94] who most commonly use the all-uses criterion for empirical studies.

Mistix is based on the Unix file system, and has been used in course projects in graduate software engineering classes at George Mason University and in previous research [AO94, OI95]. A C version of

Faults	Description
Fault 1	Same integer constant value always stored in linked list.
Fault 2	Tail of linked list is lost; cannot detect at unit level
Fault 3	Character list counter is not incremented.
	Trivial, but cannot be detected at system level, because characters never used.
Fault 4	Lose the tail of a linked list. But the tail is unused.
	So it can only be found during unit testing.
Fault 5	Lookup will fail only when the integer value is not on the list.
Fault 6	Will fail when character is used and is equivalent to an integer.
Fault 7	Lose the rest of the linked list.
Fault 8	Wrong assignment for the element type tag.
Fault 9	OR operator is used when AND should be.
Fault 10	Fails when list has different types of elements.
Fault 11	Function returns an incorrect value. It returns name instead of directory.
Fault 12	Wrong prompt.
Fault 13	All directories' parents are listed as the root.
Fault 14	Wrong operation, should call FindDir function instead of IsFile function.
Fault 15	Copy files to the wrong directory.
Fault 16	Prints the wrong directory,
	should be current directory name instead of the next directory name.
Fault 17	Directory names are not given.
Fault 18	Misspelled input abbreviation.
Fault 19	Wrong global definition that allows invalid commands.
Fault 20	Linked list fails to operate when there are different types of elements on the list.
Fault 21	Linked list fails to operate when there are different types of elements on the list.

Table 1: Descriptions of 21 Faults Inserted into **Mistix**

Mistix on a Sun workstation was used. **Mistix** has 31 function units, 65 function calls, and has 533 lines of code.

An oracle version of **Mistix** had been previously written, and 21 faults had been inserted for the previous study. Some of the faults can only be detected at the integration level, and some can only be detected at the unit level. Some of the faults were inserted into functions that are not called or used in the **Mistix** program. Brief descriptions of these faults are in Table 1.

The **Mistix** program does not have any call, stamp data/control, or external coupling, but does exhibit all other kinds of coupling listed in Section 2. Results of category-partition testing on **Mistix** were provided by Ammann and Offutt [AO93], including 72 test cases and fault detection information. The coupling-based testing was applied by hand. This was done by generating test cases manually to satisfy the all-coupling-uses criterion defined in Section 3.

To avoid any bias that could be created by having knowledge of faults or one set of test cases before creating the other set, the faults were created by one person (Offutt), the category-partition tests by

another (Ammann), and the all-coupling-uses tests by a third (Jin). Additionally, the category-partition testing results were not reviewed until after coupling-based testing was finished. The coupling-based technique yielded 37 test cases, which were generated manually all at once (before any execution). Each test case was executed against the buggy version of **Mistix**. After each execution, failures (if any) were checked and corresponding faults were debugged. This process was repeated on each test case until no more failures occurred. The number of faults detected was recorded and used in the analysis.

5.3 Results and Analysis

The faults are summarized in Table 2, a summary of the results is given in Table 3, and detailed testing results from the coupling-based and category-partition techniques are listed in Table 4. Several of the units were developed as reusable components, and had functionality that was not used in **Mistix**. Thus, of the 21 faults, 8 are in functions that are never called in **Mistix**, and cannot be detected at the integration level. Fault 4 can also only be detected at the unit testing level, so there are a total of 12 faults that can be detected during integration testing.

Inserted faults	Amount
unit never called	8
unit level faults	1
could be detected	12
total	21

Table 2: Inserted Faults Summary

	Category-partition	Coupling-based
number of test cases	72	37
faults found	7	11
faults missed	5	1

Table 3: Faults Detected

From Table 3 it can be seen that the category-partition technique resulted in 72 test cases, which detected seven faults. Four faults were missed because bad choices were made during the partitioning, or an appropriate choice was not made, and one fault was related to an input abbreviation that was not used. The coupling-based technique resulted in 37 test cases that detected 11 faults. Fault 9 is related to condition predicates that have to be generated from a unit that is never called in Mistix, so no test cases were generated to represent the condition and the fault was missed. Six of the seven faults found by the category-partition tests were also found by the coupling-based tests, and the one fault that was

Faults	Category-partition	Coupling-based
Fault 1	found	found
Fault 2	found	found
Fault 3	unit never called	unit never called
Fault 4	data member not used	data member not used
Fault 5	unit never called	unit never called
Fault 6	unit never called	unit never called
Fault 7	no such choice	found
Fault 8	unit never called	unit never called
Fault 9	found	no test cases generated could cover it
Fault 10	unit never called	unit never called
Fault 11	unit never called	unit never called
Fault 12	non-functional	found
Fault 13	no such choice	found
Fault 14	found	found
Fault 15	not found	found
Fault 16	found	found
Fault 17	found	found
Fault 18	abbreviation ignored	found
Fault 19	found	found
Fault 20	unit never called	unit never called
Fault 21	unit never called	unit never called

Table 4: Faults Detected

missed by the coupling-based tests was found by the category-partition tests.

The goals of this empirical pilot study were twofold. The first goal was to see if coupling-based testing could be practically applied. The second was to make a preliminary evaluation of the merit of the coupling-based testing criteria by comparing it with the category-partition technique. Both goals were satisfied; the coupling-based technique was applied and worked well, and performed better than the category-partition method with half as many test cases. However, there are several limitations to the interpretation of the results. First, **Mistix** is of moderate size; it has only three layers of call hierarchies, and three types of coupling were not used. Longer and more complicated programs are needed. Second, the 21 faults inserted into **Mistix** were generated intuitively. More study should be carried out to reveal the types of faults that occur at the at integration level. In future studies, it is desired to have an automated test case generator to generate test cases based on the coupling criteria, which can be used to experiment with bigger and more complicated software. Also, other integration level testing techniques [How87, HB89, HS91, LW90] should be compared with the coupling-based testing technique.

6 Related Work

Most integration testing techniques are black-box in nature and thus are difficult to compare with coupling-based testing, except empirically. Spillner [Spi92] developed a pair of integration testing techniques based on unit testing methods. One technique adapted control flow technology to test software modules by testing as many different sequences of calls as possible. The other adapted data flow technology to test for data flow anomalies across procedure calls. No empirical results were given in the paper. Jorgensen [Jor84] used the decision-to-decision paths (DD-paths) approach from unit testing for integration testing. Module-to-Module paths (MM-paths) were defined as combinations of DD-paths. Linnenkugel and Müllerburg [LM90] also used data flow and control flow technology to develop criteria for selecting integration test data. Leung and White [LW90] applied extremal values testing concepts to integration testing.

The inter-procedural data flow testing technique of Harrold, Soffa, and Rothermel [HS91, HR94] is one white-box testing approach that has some similarities with coupling-based testing. It could be used either for module or integration testing. In standard **intra**-procedural data flow testing, test cases are created to exercise subpaths from defs of variables to uses of variables within the same unit. In **inter**-procedural data flow testing, defs in one unit are required to reach uses in another unit of the same module. Sometimes these are through direct paths via function calls, other times through a sequence of external calls to the module. The point of this approach is to create sequences of calls to the module based on definition-use pairs inside a module. Inter-procedural testing differs from coupling-based testing in several ways. One, inter-procedural testing creates sequences of calls to procedures in a module, whereas coupling-based testing attempts to cover existing calls and definition-use pairs within an existing system. Two, inter-procedural testing looks at one module, and creates calls to be used in a driver program. Coupling-based testing looks at an entire system and exercises calls and subpaths that precede and succeed calls. Three, inter-procedural testing requires analysis of the entire control-flow graph for procedures, whereas coupling-based testing only requires analysis of small portions of the control-flow graph. It is thought that this will make it easier to automatically generate test data (a problem that has not been addressed for inter-procedural data flow testing), and that for coupling-based testing, it does not matter if parameters are aliased. Coupling-based testing also provides for integration between modules, whereas inter-procedural testing only tests calls in the same module, and it is believed that coupling-based testing will scale up more readily than inter-procedural testing, and can be applied more easily by hand.

Although there are fundamental differences between the purposes and uses of coupling-based testing and inter-procedural testing, it was decided that a direct empirical comparison would be helpful. The same Mistix program from Section 5 was used, and test data was generated by hand, following the technique in Harrold, Soffa, and Rothermel's papers [HS91, HR94]. Although these papers did not

contain complete procedural descriptions to generate the tests, every effort was made to follow the concepts exactly. To avoid bias, a student who was familiar with the inter-procedural test strategy, but who was **not** familiar with coupling-based testing, category-partition testing, or the Mistix faults generated the inter-procedural tests.

This effort yielded 53 tests. These tests found eight of the Mistix faults, and missed four. All eight faults found by the inter-procedural tests were also found by the coupling-based tests, and the one fault that was missed by the coupling-based tests was also missed by the inter-procedural tests.

7 Conclusions

This paper has introduced a new integration testing technique, coupling-based testing. Four coupling-based criteria were defined, and coverage measurement analysis was described that quantitatively measures the test coverage. As part of this, the notion of a subpath set was introduced, which can also be useful in traditional data flow testing. This technique can be used to satisfy the FAA's requirements on structural coverage analysis of software [SC-92]. This technique has been compared with the category-partition method and the inter-procedural data flow testing method, and it was found that the coupling-based technique detected more faults with fewer test cases than either of these methods. Although in both cases, the faults found by the other methods were subsets of the faults found by coupling-based testing, the data is insufficient to draw general conclusions. This result indicates that this approach can benefit practitioners who are performing integration testing on software. Future work includes a coverage analysis tool for this method, and the automation of some of the test data generation for coupling-based testing. More evaluation of the effectiveness of this technique is also necessary.

8 Acknowledgments

Thanks to Roger Alexander for catching numerous small problems in the definitions, Yiwei Xiong for verifying (and fixing!) the instrumentation techniques, and JinLan Zhang for help with the inter-procedural comparison.

References

[AO93] P. Ammann and A. J. Offutt. Functional and test specifications for the MiStix file system. Technical report ISSE-TR-93-100, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1993.

[AO94] P. Ammann and A. J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94)*, pages 69–80, Gaithersburg MD, June 1994. IEEE Computer Society Press.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[BHO89] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218, Key West Florida, December 1989. ACM SIGSOFT 89.

[CY79] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

[Def83] United States Department Of Defense. *Reference Manual for the Ada Programming Language – ANSI/MIL-STD-1815A-1983*. American National Standards Institute, Inc, 1983.

[FW88] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[FW93] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[FWH97] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software*, 38(3):235–253, 1997.

[HB89] D. Hoffman and C. Brealey. Module test case generation. In *Proceedings of the Third Workshop on Software Testing, Verification and Analysis*, pages 66–74, Key West Florida, December 1989. ACM SIGSOFT.

[How87] W. E. Howden. *Functional Programing Testing and Analysis*. McGraw-Hill Book Company, New York NY, 1987.

[HR94] M. J. Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Symposium on Foundations of Software Engineering*, pages 154–163, New Orleans, LA, December 1994. ACM SIGSOFT.

[HS91] M. J. Harrold and M. L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[Jor84] P. C. Jorgensen. MM-Paths: A white-box approach to software integration testing. In *Third Annual Phoenix Conference on Computers and Communications*, pages 181–185, Phoenix Arizona, 1984.

[LM90] U. Linnenkugel and M. Müllerburg. Test data selection criteria for (software) integration testing. In *Proceedings of the 1990 Conference on Systems Integration*, Morristown New Jersey, April 1990. IEEE Computer Society Press.

[LW90] H. K. N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Conference on Software Maintenance-1990*, pages 290–301, San Diego, CA, Nov 1990.

[MW94] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *The Journal of Software Testing, Verification, and Reliability*, 4(1):9–31, March 1994.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[OHK93] A. J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *The Journal of Systems and Software*, 20(3):295–308, March 1993.

[OI95] A. J. Offutt and A. Irvine. Testing object-oriented software using the category-partition method. In *Proceedings of the Seventeenth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '95)*, pages 293–303, Santa Barbara, CA, August 1995.

[Par72] D. Parnas. On the criteria to be used in decomposing a system into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PJ80] M. Page-Jones. *The Practical Guide to Structured Systems Design*. YOURDON Press, New York, NY, 1980.

[SC-92] RTCA Committee SC-167. Software considerations in airborne systems and equipment certification, Seventh draft to Do-178A/ED-12A, July 1992.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Som92] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Company Inc., 4th edition, 1992.

[Spi92] A. Spillner. Control flow and data flow oriented integration testing methods. *The Journal of Software Testing, Verification, and Reliability*, 2(2):83–98, 1992.

[TZ81] D. A. Troy and S. H. Zweben. Measuring the quality of structured designs. *The Journal of Systems and Software*, 2:112–120, 1981.

[Whi87] L. J. White. Software testing and verification. In Marshall C. Yovits, editor, *Advances in Computers*, volume 26, pages 335–390. Academic Press, Inc, 1987.