Integration Testing Based on Software Couplings *

Zhenyi Jin and A. Jefferson Offutt
ISSE Department
George Mason University
Fairfax, VA 22030
phone: 703-993-1654
fax: 703-993-1638
email: {zjin,ofut}@isse.gmu.edu

January 27, 1995

Abstract

Integration testing is an important part of the testing process, but few integration testing techniques have been systematically studied or defined. This paper presents an integration testing technique based on couplings between software components. The coupling-based testing technique is described, and 12 coverage criteria are defined. The coupling-based technique is compared with the category-partition method. Results show that the coupling-based technique detected more faults with fewer test cases than category-partition on a subject program. This modest result indicates that the coupling-based testing approach can benefit practitioners who are performing integration testing on software. While it is our intention to develop algorithms to fully automate this technique, it is relatively easy to apply it by hand.

Keywords: Category-partition, Integration testing, Software module coupling, Software testing.

COMPASS Areas: Software Reliability, Measurement & Metrics.

1 Introduction

Testing software is one of the most common, though imperfect, methods for assuring software quality. The general purpose of the research reported in this paper is to formalize, via new coverage criteria, routine aspects of testing, particularly 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

*Partially supported by the National Science Foundation under grant CCR-93-11967.

the tester will find any faults in the program and providing greater assurance that the software is of high quality and reliability.

The emphasis on modularity in software design has influenced designers to build software systems by dividing them into components to master their complexity. One of the benefits of modularity is that the software components can be tested independently, which is usually done by the programmer during unit and module testing. Although much is known about unit and module testing and many techniques and tools are available, the integrated components also need to be tested. 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, we must specifically test for integration faults. Unfortunately, there are not many techniques in the literature that are designed to be used during integration testing. Structural testing techniques and tools are needed to support integration testing of software systems.

We consider a program unit to be 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]. We consider *unit and module testing* (or just unit testing) 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 parts of the system as it is built out of units. *System testing* is then testing applied to an entire integrated system.

Integration testing can be based on specifications, design information, or the code itself. Most of the current integration testing techniques are based on the specifications and functionality of the software [OSW86, How87], and thus could be called "black-box" techniques, because they treat the software's structure as a black box. At the unit level, we say that a testing technique is white-box if it uses information about the actual code in the unit. At the integration level, a white-box technique would use information about the individual units, although not necessarily the

[1] Older literature considered the terms module and unit to be synonymous, thus used the term module when we use unit. We choose 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.

statements. Specifically, we consider an integration testing technique to be a white-box technique if it is based on the design information and the structure of the software. Using this information should facilitate making precise statements about the adequacy and thoroughness of testing.

Modularity of software design can be measured by two properties, cohesion and coupling [CY79]. *Cohesion* describes a unit or module's functionality, while coupling measures the dependency relations between two units. *Coupling* between two units reflects the interconnections between units; faults in one unit may affect the coupled unit. Coupling provides summary information about the design and the structure of the software.

Since couplings are exactly where faults found during integration testing typically occur, we propose a new coupling-based testing technique. We define criteria that require that each connection between program units be covered. Coupling-based testing criteria are listed in Section 3 and Section 5 presents empirical results that demonstrate the usefulness of the concept. This is done by comparing with another integration testing technique, category-partition. In Section 6, conclusions and future work are presented.

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 levels of coupling are used to evaluate the complexity of software system designs. Troy and Zweben [TZ81] relate coupling levels to the number of faults in software. Their 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 their study was based on subjective interpretations of design documents instead of actual code. Offutt et al. [OHK93] extended the eight levels of coupling to twelve levels, providing a finer grained measure of coupling. They also designed algorithms to automatically measure the coupling levels between each pair of units in a program.

The coupling levels are defined between pairs of units, say A and B. For each coupling level, the call return parameters are classified by the way they are used. Classification of uses are 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 or as an output statement. 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 on the predicate. The 12 levels of coupling are listed as follows:

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 module without B having accessed or changed the variable.

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, we cannot exhaustively search the entire domain D (which in most cases is effectively infinite). 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; e.g., 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.

3.1 Data Flow 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 an occurrence of a variable where a value is stored into memory (assignment,

input, etc.). A *use* is an occurrence of a variable where its 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 I/O 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. An *oracle* can recognize the correct outcome of a set of tests as applied to a tested object.

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. These coupling paths are defined based on the 12 levels of coupling listed in Section 2, and are defined precisely in this section. The underlying premise of the coupling-based testing criteria is that to achieve confidence in the interfaces between integrated program units, we must ensure that variables defined in caller units be appropriately used in callee units. Because we limit our technique to the unit interfaces, we are only concerned with definitions of variables just **before** calls to other units, and uses of variables just **after** returns from the called unit.

To make integration testing a manageable process, testing must be guided by the modularization of the software. Each unit 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.

3.2.1 Testing Steps

Generation of test data based on couplings is performed in two major steps: static analysis of the program units to obtain information about test case requirements and dynamic analysis to generate test cases.

Static analysis includes several steps. First, the mapping of parameter variables between units are

syntactically determined. Second, data flow analysis is applied to each unit to find appropriate definitions of actual parameters and uses of formal parameters. Third, the dependency relations between units are found. This includes their couplings through calls as well as through global variables.

Dynamic analysis focuses on test cases. First, test cases are created that satisfy the 12 coupling-based testing criteria. Then the test cases are run on the software and failures are noted.

3.2.2 Coupling-Based Testing Definitions

To formally define the coupling-based testing criteria, we introduce several definitions. A and B are two units, and A calls B. x is an actual parameter in A mapping to a formal parameter y in B. There could be more than one parameter, but we consider only one parameter at a time. Our definitions use the following variables:

M : A or B.
V_A : set of variables in A.
V_B : set of variables in B.
N_A : set of nodes in A.
N_B : set of nodes in B.
E_A : set of edges in A.
E_B : set of edges in B.
v : a variable in V_A or V_B.
C : a unit that is called by unit B in the case of tramp coupling.
def(M,v) : a definition of a variable v in unit M.
use(M,v) : a use of a variable v in unit M.
call_site : A node in A where B is called.

Call(A, B, call_site, $x \rightarrow y$**)**: TRUE if unit A calls B at call_site and actual parameter x maps 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.

Record(v): TRUE if v is a record, FALSE otherwise. This is used in stamp coupling.

Return(v): TRUE if v is used in the return(v) statement in a unit, FALSE otherwise.

Start(M): The first node in M.

Last-def-before-call: The set of nodes i that define x and for which there is a def-clear path from the node to the call statement for A. This is defined as:

- **ldbc-def(A, call_site,** x**)** = $\{i, i \in N_A \mid$ 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$\}$

First-use-after-call: The set of nodes i in A that have uses of x and for which there are no other uses between the call statement for A and these nodes. This is defined as:

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

Last-def-before-return: If B is a function that returns a value to A, then Last-def-before-return is the set of nodes that define the returned variable y, and for which there is also a def-clear path from the node to the return statement. This is defined as:

- **ldbr-def(B,** y**)** = $\{j \in N_B \mid y$ is defined in node $j \bigwedge$ there is a def-clear path with respect to y from j to return(y)$\}$

First-P-use-in-callee: The set of edges in B that have a predicate use of y such that there is at least one path from Start(B) to the edge with no other P-uses of y. This is defined as:

- **fp-use(B,** y**)** = $\{(j_1, j_2) \in E_B \mid y$ has a P-use at edge(j_1, j_2) $\bigwedge \exists$ a path from Start(B) to edge(j_1, j_2) with no uses of $y\}$

First-C-use-in-callee: The set of nodes in B that have a computation use of y, such that there is at least one path from Start(B) to the node with no other C-uses of y. This is defined as:

- **fc-use(B,** y**)** = $\{j \in N_B \mid y$ has a C-use in node $j \bigwedge$ there is a path with no uses of y between Start(B) and node $j\} \exists$ a path from Start(B) to node j with no uses of $y\}$

First-I-use-in-callee: The set of edges in B that have an indirect use of y (I-use), such that there is at least one path from Start(B) to the edge with no other I-uses of y. It is defined as:

- **fi-use(B, y)** = $\{j \in \mathrm{E}_B \mid y$ has an I-use at edge $j \bigwedge \exists$ a path from Start(B) to edge j with no uses of $y\}$

First-use-in-callee: The set of nodes for which parameter y in B has a computation use, or an incoming edge has a predicate use or an indirect use, and there is at least one path from the begin statement to the use. This is defined as:

- **f-use(B, y)** = $\{j \in \mathrm{N}_B \mid ((y$ has a C-use at node $j) \bigvee (y$ has an I-use on edge $(i, j), i \in \mathrm{N}_B) \bigvee (y$ has a P-use on edge $(i, j)), i \in \mathrm{N}_B) \bigwedge$ there is a path with no other uses of y between Start(B) and node $j\}$

3.2.3 Coupling-Based Testing Criteria

The 12 coupling-based testing criteria are based on the definitions above. For each, we describe the criterion, then define it formally.

1. No coupling: No coupling means that A and B are not connected in any way. There is no coupling path between these two units, and no test requirement is necessary.
2. Scalar data coupling path: The scalar data coupling criterion requires that for each scalar parameter x, and each last definition of x before a call_site, a test case executes at least one path from the last definition, to the call, and to each of the first C-uses of the formal parameter y in B. This is defined as:
   - Scalar-data-coupling(A, B, call_site, x, y) = $\{(i, j)\ i \in \mathrm{N}_A, j \in \mathrm{N}_B \mid i \in$ ldbc-def(A, call_site, $x) \bigwedge j \in$ fc-use(B, $y)\}$
3. Stamp data coupling path: The stamp data coupling criterion requires that for each record parameter x, and each last definition of x before a call_site, a test case executes at least one path from the last definition, to the call, and to each of the first C-uses of the formal parameter y in B. It is defined as:

- Stamp-data-coupling(A, B, call_site, x, y) = $\{(i, j)\ i \in N_A, j \in N_B \mid \text{Record}(x) \bigwedge \text{Record}(y) \bigwedge i \in \text{ldbc-def(A, call\_site, } x) \bigwedge j \in \text{fc-use(B, } y)\}$

4. Scalar control coupling path: The scalar control coupling criterion requires that for each scalar parameter x, and each last definition of x before a call_site, a test case executes at least one path from the last definition, to the call, and to each of the first P-uses of the formal parameter y in B. It is defined as:

   - Scalar-control-coupling(A, B, call_site, x, y) = $\{(i, j)\ i \in N_A, j \in N_B \mid i \in \text{ldbc-def(A, call\_site, } x) \bigwedge j \in \text{fp-use(B, } y)\}$

5. Stamp control coupling path: The stamp control coupling criterion requires that for each record parameter x, and each last definition of x before a call_site, a test case executes at least one path from the last definition, to the call, and to each of the first P-uses of the formal parameter y in B. It is defined as:

   - Stamp-control-coupling(A, B, call_site, x, y) = $\{(i, j)\ i \in N_A, j \in N_B \mid \text{Record}(x) \bigwedge \text{Record}(y) \bigwedge i \in \text{ldbc-def(A, call\_site, } x) \bigwedge j \in \text{fp-use(B, } y)\}$

6. Scalar data/control coupling path: The scalar data/control coupling criterion requires that for each scalar parameter x, and each last definition of x before a call_site, a test case executes at least one path from the last definition, to the call, and to each of the first I-uses of the formal parameter y in B. It is defined as:

   - Scalar-data/control-coupling(A, B, call_site, x, y) = $\{(i, j)\ i \in N_A, j \in N_B \mid i \in \text{ldbc-def(A, call\_site, } x) \bigwedge j \in \text{fi-use(B, } y)\}$

7. Stamp data/control coupling path: The stamp data/control coupling criterion requires that for each record parameter x, and each last definition of x before a call_site, a test case executes at least one path from the last definition, to the call, and to each of the first I-uses of the formal parameter y in B. It is defined as:

   - Stamp-data/control-coupling(A, B, call_site, x, y) = $\{(i, j)\ i \in N_A, j \in N_B \mid \text{Record}(x) \bigwedge \text{Record}(y) \bigwedge i \in \text{ldbc-def(A, call\_site, } x) \bigwedge j \in \text{fi-use(B, } y)\}$

8. External coupling path: The external coupling criterion requires that for each pair of references to the same external file, both i and j must be executed on the same execution path.

9. Nonlocal coupling path: The nonlocal coupling criterion requires that for each nonlocal variable x that is defined in A and used in B, and each definition of x in A, a test case executes at least one path from the last definition to each of the first uses of x in B. It is defined as:

   - Nonlocal-coupling(A, B, x) = $\{(i, j)\ i \in \mathrm{N}_A, j \in \mathrm{N}_B \mid i \in \mathrm{def}(\mathrm{A}, x) \bigwedge j \in \mathrm{use}(\mathrm{B}, x)\}$

10. Global coupling path: The global coupling criterion requires that for each global variable x, that is defined in A and used in B, and each definition of x in A, a test case executes at least one path from the definition to each of the first uses of x in B. It is defined as:

    - Global-coupling(A, B, x) = $\{(i, j)\ i \in \mathrm{N}_A, j \in \mathrm{N}_B \mid i \in \mathrm{def}(\mathrm{A}, x) \bigwedge j \in \mathrm{use}(\mathrm{B}, x)\}$

11. Tramp coupling path: The tramp coupling criterion requires that for each parameter x that is defined in A, passed through B, and used in C, and each last definition of x before a call_site, some test case executes at least one path from the last definition in A, through B, and to each of the first uses in C. It is defined as:

    - Tramp-coupling(A, B, C, call_site, $x \rightarrow z$) = $\{(i, j, k)\ i \in \mathrm{N}_A, j \in \mathrm{N}_B\ , k \in \mathrm{N}_C \mid i \in$ ldbc-def(A, call_site, x) $\bigwedge$ Call(B, C, call_site, $x \rightarrow z$) $\bigwedge k \in$ f-use(C, z)$\}$

To simplify the definitions, the above criteria ignore the issue of returning values via parameters – in effect, assuming every parameter is a call-by-value. The following definition is generalized and applies to all the above criteria involving parameters. If a parameter x is call-by-reference, then the criterion requires that a path be executed from each last definition before return of the formal parameter y in B to each first use after the call of x in A. This is defined as:

- $\{(j, i)\ i \in \mathrm{N}_A, j \in \mathrm{N}_B \mid j \in$ ldbr-def(B, y) $\bigwedge i \in$ fac-use(A, call_site, x)$\}$

4 Related Work

Most integration testing techniques have been black-box in nature and are thus difficult to compare with coupling-based testing, except on an empirical basis. Inter-procedural data flow testing is one white-box integration testing approach that has some similarities with coupling-based testing [HS89, HR94]. 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 DU pairs inside a module. Coupling-based testing chooses values for call parameters based on requirements that are constructed on **existing** calls within the software. 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 aliasing will not be a problem with coupling-based testing. It also seems likely that coupling-based testing will scale up more readily than inter-procedural testing, and can be applied more easily by hand.

5 Proof of Concept Study

To demonstrate the feasibility of these criteria, we have undertaken a study to compare the coupling-based testing technique with another technique that is used for integration testing, category partition. Our goal was not to undertake a full experimental analysis of the two techniques, but to demonstrate that the coupling criteria can be used in an effective way. Thus, the results of this study should be taken as preliminary.

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 the 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 Empirical Study

We compared coupling-based testing with category-partition testing in terms of their fault-detection ability. We used one moderate size program, faults created for previous experimentation [AO94], and generated test cases by hand.

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, Irv94]. We used a C version of **Mistix** on a Sun workstation. **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 previous research [AO94]. Some of the faults can only be found 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. The 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 by the first author. This was done by generating test cases manually to satisfy the coupling criteria described in Section 3.

To avoid any bias that could be created by having knowledge of faults and one set of test cases before creating the other set, 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 were recorded and used in the analysis.

5.3 Results and Analysis

The faults are summarized in Table 2, detailed testing results from the coupling-based and category-partition techniques are listed in Table 3, and a summary of the results is given in Table 4. Several of the modules were developed as reusable components, and had functionality that was not used

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 char is used and the character is equivalent to an integer.
Fault 7	Lost 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**

in **Mistix**. Thus, of the 21 faults, 8 faults are functions that are never called in **Mistix**, thus they 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 (see Tables 2 and 4).

From Table 4 we can see that the category-partition technique resulted in 72 test cases, which detected seven faults. Four faults were missed because of bad choices or no choices, 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.

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

Table 2: Inserted Faults Summary

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. 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, with which bigger and more complicated software can be experimented. Also, other integration level testing techniques [How87, HB89, HS89, LW90] should be compared with the coupling-based testing technique.

6 Conclusions

This paper has introduced a new integration testing technique, coupling-based testing. A set of twelve coupling-based criteria were defined. To demonstrate the concept of this new testing technique, coupling-based testing was applied to a moderately-sized software systems and the results were compared with the category-partition technique on their effectiveness in detecting faults. Coupling-based testing is more effective than category-partition on this program, which suggests that coupling-based testing can be a powerful testing technique for integration testing.

References

[AO93] P. Ammann and A. J. Offutt. Functional and test specifications for the MiStix file

Faults	Category-Partition	Coupling-based
Fault 1	found	found
Fault 2	found	found
Fault 3	unit never called	unit never called
Fault 4	affected data item never used	can only be found at unit testing
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 3: Faults Detected

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.

[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.

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

Table 4: Faults Detected

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

[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.

[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] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, New Orleans, LA, December 1994. ACM Press.

[HS89] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, Key West Florida, December 1989. ACM SIGSOFT 89.

[Irv94] A. Irvine. The effectiveness of category-partition testing of object-oriented software. Master's thesis, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1994.

[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.

[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.

[OSW86] T. J. Ostrand, R. Sigal, and E. J. Weyuker. Design for a tool to manage specification-based testing. In *Proceedings of the Workshop on Software Testing*, pages 41–50, Banff Alberta, July 1986. IEEE Computer Society Press.

[Par72] D. Parnas. On the criteria to be used in decomposing a system into modules. *Communications of the ACM*, December 1972.

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

[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.

[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.