

An Analysis Tool for Coupling-based Integration Testing *

A. Jefferson Offutt, Aynur Abdurazik and Roger T. Alexander
George Mason University
Department of Information and Software Engineering
Software Engineering Research Laboratory
Fairfax, VA
{ofut,aynur,ralexand}@ise.gmu.edu

The Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00), pages 172–178, Tokyo Japan, September 2000.

Abstract

This research is part of a project to develop practical, effective, formalizable, automatable techniques for integration testing. Integration testing is an important part of the testing process, but few integration testing techniques have been systematically studied or defined. This paper discusses the design and implementation of an analysis tool for measuring the amount of coverage achieved by a set of test data according to a set of previously defined coupling criteria. This tool can be used to support integration testing of software components. The coupling-based testing technique, which has been described elsewhere, is summarized, and coverage algorithms are discussed. The focus of this paper is on the instrumentation techniques and an analysis tool built for Java programs. It was built in Java using the general Java parser JavaCC and the Java Tree Builder (JTB). We are currently using this tool to gather experimental data on the efficacy and the usefulness of the technique.

1. Introduction

Testing software is one of the most common methods for assuring quality of complex computer software systems. To be confident of the results of testing, testers need formally defined procedures that provide mechanisms for creating test data and for deciding when testing can stop. *Test requirements* are specific things that must be satisfied or covered during 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 percentage of requirements that are satisfied. Formal coverage criteria are important for several reasons. Often, when faced with testing a program, developers do not know what to test or where to start. First, formal criteria help this situation by providing a basis for specifying test requirements. In turn, test requirements guide the testing process by providing a clear description of what must be tested. This provides a mechanism for deciding when to stop testing and a basis for adding repeatability to the testing effort. Second, formal criteria offer the tester ways to decide what test inputs to use during testing, making it more likely that the tester will find faults in the program and providing greater assurance that the software is of high quality and reliability.

This paper uses the general term *component* to refer to a separately compilable portion of the program. Depending on the language and context, a component may be a function, procedure, module, package, or class. The tool described in this paper analyzes Java programs, hence the primary component we are concerned with is the class. *Integration testing* refers to testing interfaces between components to assure that they have consistent assumptions and communicate correctly [3].

The current emphasis on data abstraction and object-oriented software has led to an increased emphasis on modularity and reuse. A major result of this change in emphasis is that the complexity in our software, and the associated likelihood of making mistakes, is moving from the functions and procedures to the classes and the integration connections among the classes. This change is even more pronounced with component-based software development, in which components are often assumed correct without access to the source, and component integration becomes most of the implementation effort. Instead of procedures that have complicated control structures, object-oriented software often has simple procedures, with the complexity being in how the procedures and components are connected. Al-

* This work is supported in part by the U.S. National Science Foundation under grant CCR-98-04111.

though abstract data types can help achieve a higher quality design, their use may affect how software is tested. Thus, it has been our experience and the experience of many of our colleagues in academia and industry that testers are finding that less emphasis is needed on unit testing and more on integration testing. There are few techniques or tools that are specifically designed to be support integration testing. Testers are often left with performing integration testing in ways that are ad-hoc and ineffective, leading to less reliable software. 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" [16], pg. 33, Section 6.4.4.2.

Previous papers [9, 10] suggested that integration testing can be conducted by using a *structural coverage criterion*, and presented a set of integration testing criteria based on software couplings. *Coupling* between two components measures the dependency relations between the two components by reflecting the interconnections between methods; faults in one unit may affect the coupled unit [6]. 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, the coupling-based testing criteria require that various connections between program components be covered during testing.

1.1. Coupling

As part of their structured design methodology, Constantine and Yourdon [6] introduced the concept of software coupling. Coupling between two methods increases the interconnections between the two methods 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.

Although the previous literature [6, 14, 12] presented up to twelve various types of coupling in an ordered list in terms of severity, our testing criteria are defined to handle several types uniformly, thus the list is collapsed into four general unordered types. These coupling types are defined as [10]:

- *Call coupling*: component A calls another component B without passing parameters, and A and B do not share any common variable references, or common references to external media.
- *Parameter coupling*: A calls B and passes one or more data items as a parameter.
- *Shared data coupling*: A calls B and they both refer to the same data object (either globally or non-locally).

- *External device coupling*: A calls B and they both access the same external medium (for example, a file or sensor).

1.2. Coupling-based testing

Coupling-based testing requires that the program execute from definitions of actual parameters through calls to uses of the formal parameters. The criteria are based on the notion of a coupling path. Informally, a *coupling path* is a sequence of statements that, when executed, proceed from a definition of a variable, through a call to a method or a return from a method, to a use of that variable. A statement that contains a definition of a variable that can reach a call-site or a return is called a *last-def*. When a value is transmitted into or out of a method (through a parameter, a return value, or a non-local variable reference), the first time it is used on an execution path after the method is entered or exited is called a *first-use*. Note that there can be more than one last-def and first-use of a given variable and call-site.

The underlying premise of the coupling-based testing criteria is that to achieve confidence in the interfaces between integrated program methods, it must be ensured that variables defined in caller methods be appropriately used in callee methods. Although these coverage criteria could certainly be defined so that coupling definitions reach **all** uses instead of just **first** uses, we have chosen not to do so. The conjecture behind this decision is that if a fault exists, it will probably be triggered if the data value is used once and subsequent uses are less likely to be important. Of course, it would be valuable to empirically validate this conjecture, and it is possible that the extra cost that would be incurred if subsequent uses in the callee must be covered would be worthwhile. Note that the first-use requirement is not a dynamic requirement, but static. That is, a use that is a first-use on one execution path might be a *subsequent-use* on another path. Thus, it is possible for one test case to cover more than one first-use in the same execution path.

The coupling-based criteria are similar in nature to the standard unit-level data flow coverage [7]. For brevity, informal versions of these definitions are given here. Assume that there is a call from component C_1 to component C_2 , and x is an actual parameter in C_1 that maps to a formal parameter y in C_2 , and the program is tested with a set of test cases T .

- *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 last-def of each actual parameter x in C_1 , the set of paths executed by the test set T contains at least one coupling path to **at least one** first-use of y in C_2 .

- *All-coupling-uses* requires that for each last-def of x in C_1 , the set of paths executed by the test set T contains at least one coupling path to **each** first-use of y in C_2 .
- *All-coupling-paths* is more complicated to define. Definitions of similar data flow testing criteria [5, 7, 11, 15] have imposed related requirements, including requiring that **all** paths be executed. The problem with this is that if there is a loop along a path, the number of paths becomes infinite. The other requirements suffered from similar problems.

We define a *subpath set* 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 set* is the set of nodes on a coupling path.

All-coupling-paths requires that for each definition of x , the set of paths executed by T contains all **coupling path sets** from the definition to all reachable 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 number of times.

2. Measuring Coupling Coverage

Structural coverage analysis is used to determine whether couplings have been covered. The coupling-based testing criteria provide test requirements. Test cases can then be generated specifically to satisfy each test requirement, or they can be generated by some other method and then analysis techniques can be used to check whether all requirements have been satisfied. This paper focuses on the latter approach, determining whether a testing criterion has been satisfied by a set of externally provided test cases. One difficulty that arises in almost all formal testing criteria is that of infeasible requirements. In coupling-based testing, some definition-use pairs cannot be covered because there is no executable def-clear path from the definition to the use. This problem has been analyzed [7] and partial solutions have been proposed [13] elsewhere. In this work, we accept that some requirements will be unsatisfiable, so 100% coverage is not completely possible. Identifying infeasible coupling definition-use pairs is currently left up to the tester.

Given a coupling criterion and a set of test cases, algorithms can be used to measure whether the test set satisfies the criterion's test requirements. These measurements are typically performed using *instrumentation*, which are statements added to the program for analysis purposes. We have developed a tool that uses instrumentation to measure

coverage for call-coupling, all-coupling-defs, all-coupling-uses, and all-coupling-paths. The instrumentation techniques have been outlined elsewhere [10], this paper provides more details about the instrumentation and focuses on the implementation.

The tool relies primarily on the *coupling graph*, $C = (M, E, F, A)$. M is a finite multi-set of nodes that represent software methods. 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 coupling relations and are called *call edges*. F is a finite multi-set of nodes that represent external files a unit may write to or read from. When a unit U calls unit W , the edge starts from node U and ends at node W . 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*. Finally, A is a set of annotations on nodes. Some of the nodes may reference non-local 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 has a unique root node, which is the main program method. The methods called by the root form its successors. Parameter coupling relations are represented by the sequence of edges of a depth-first search of the coupling graph (excluding the external device nodes). The coupling graph is used to measure call-coverage by calculating the number of the edges or nodes covered in the coupling graph. If E is the number of edges in a coupling graph, CD is the number of coupling defs, CU is the number of coupling def-use pairs, and CP is the number of coupling paths sets, then coupling coverage is measured by the following formulas:

$$\begin{aligned} \text{call-coupling} &= \frac{E_{\text{covered}}}{E} \\ \text{coupling-def} &= \frac{CD_{\text{covered}}}{CD} \\ \text{coupling-use} &= \frac{CU_{\text{covered}}}{CU} \\ \text{coupling-path} &= \frac{CP_{\text{covered}}}{CP} \end{aligned}$$

2.1. Using instrumentation to measure coupling coverage

a program P is modified to produce an instrumented version P' . P' has all the functionality of P , plus extra statements to keep track of whether specific portions of the software has been executed. For coupling-based testing, these extra statements primarily keep track of calls, last definitions, and first uses. We first describe the instrumentation

for call coupling, which is the simplest of the four criteria, then describe how the instrumentation for the other criteria work by building on call coupling.

Call coupling. Consider the graph shown in Figure 1 (A). A calls B, C, and D, and B calls E and F. C returns a value to A (hence the bidirectional arrow). Figure 1 (B) shows the same call graph with instrumentation added. CTab[] is an integer coverage array that is indexed by call-sites and initialized to 0. The statement “CTab[i] ++” is added to every call-site. After the program has been executed, if CTab[i] = 0, the call-site associated with i has not yet been executed. Of course, CTab[] must be saved to disk and re-read between executions to allow the coverage information to be accumulated across multiple test cases.

All-coupling-defs. All-coupling-defs requires slightly more complicated instrumentation, and the use of CTab[] must be expanded. First, CTab[] is changed to be indexed by last definitions instead of call-sites. The definition can be in the caller before the call, or in the callee before the return. There must be a definition-clear path from the definition to the call (or return). A new table is introduced for all-coupling-defs to keep track of where the most recent definition of variables was made. At each last 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. Each element of 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. This instrumentation is illustrated in Figure 2, where the actual parameter X is passed to a method B and associated with the formal parameter y. LastDef[] is set at the last defs at nodes 2 and 3. Note there is no assignment to LastDef[] at node 1 because there is no def-clear path from node 1 to the call-site at node 4.

CTab[] is updated at the first uses, but this update is complicated by parameter passing. If a method B(y) is called from more than one location, the formal parameter of y may be associated with a different actual variable in each call location. To handle parameter passing, an *actual-formal mapping table* is introduced. The Actual table keeps track of the current mapping of formal to actual parameters. So at each call-site, the function call “SetActual (y, A);” is added for every parameter to the called procedure. In addition, it is necessary to keep track of return values of functions using the same Actual table. After the program is executed a number of times, each entry in CTab[] indicates the number of times the corresponding last definition in a caller reached a use in a callee.

All-coupling-uses. The instrumentation for all-coupling-uses is a straightforward extension to that of all-

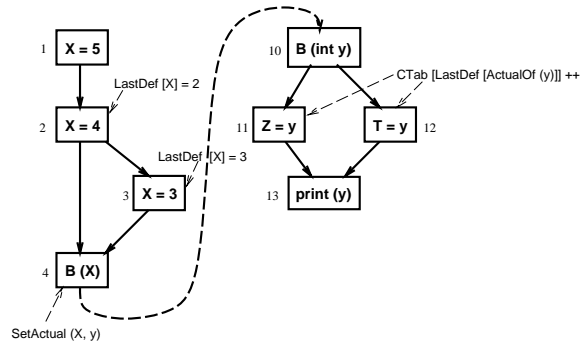


Figure 2. All-Coupling-Defs Instrumentation.

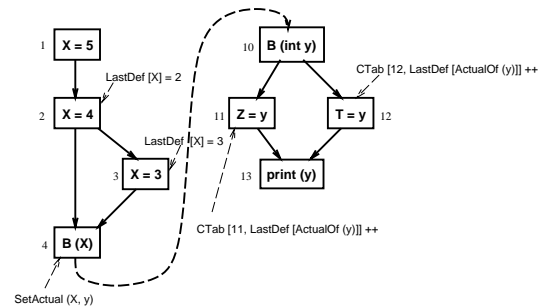


Figure 3. All-Coupling-Uses Instrumentation.

coupling-defs. The complication is that CTab[] must be indexed by a combination of last-definitions and first-uses. In Figure 2, the CTab[] update is the same at both node 11 and 12; it does not matter which use is reached by the definition of the variable. For all-coupling-uses, both uses must be reached, so CTab[] must be updated with a different index at each use. This instrumentation is shown in Figure 3.

All-coupling-subpaths. Finally, the instrumentation for all-coupling-subpaths requires the most complexity. It is necessary to keep track of all the statements that are executed between the last-def and the first-use. This is done by using a subpath set data structure. The subpath set is initialized to empty at last-defs, and subsequent statements that are executed are added to the set (if a statement is executed more than once, the set semantics ensures that it is only included once). This is illustrated in Figure 4 with calls to InitSubPathSet() at nodes 2 and 3, and calls to AddSubPathSet() at nodes 2, 3, 4, and 10.

When a first-use is reached, CTab[] is updated with the variable and the use location, and also the subpath set. Figure 4 shows updates to CTab[] at nodes 11 and 12,

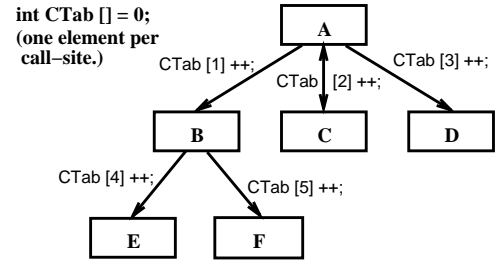
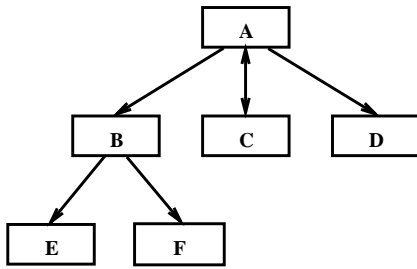


Figure 1. (A) Example Call Graph. (B) Instrumented Call Graph.

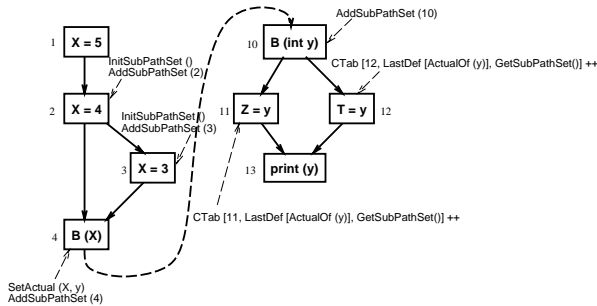


Figure 4. All-Coupling-Subpaths Instrumentation.

which include calls to `GetSubPathSet()`. This method uses a hash function to find a unique integer that represents the subpath set. Obviously, the instrumentation for all-coupling-subpaths is the most intrusive of the four criteria, and also the most complicated.

One of the most appealing things about the instrumentation for coupling coverage analysis is its relative simplicity. The modifications to each method are completely independent of modifications to other methods. This makes the analysis relatively cheap and simple, because only one method has to be analyzed at a time. In fact, the only representation of the program that is needed is an abstract syntax tree. Moreover, these instrumentations have very minor impacts on the program's performance. The most severe impact is that for all-coupling-subpaths, which requires instrumentation statements to be added at a number of nodes in the control flow graph. Moreover, the subpath set data structure requires more computation than for the other criteria. Nevertheless, the increased computation only has a constant time impact on the overall running time of the original program, and the necessary data structures are common and well understood.

3. Tool Architecture and Implementation

The overall architecture for the coupling coverage analysis tool is shown in Figure 5. There are three major components: (1) the Coupling Instrumentor (`CoupInst`) tool, (2) the instrumented version of the test program, and (3) the Coupling Analysis (`CoupAnal`) report generator tool. The coupling instrumentor uses a parser (`JavaCC Parser`) and an abstract syntax tree (`AST`), which is a collection of classes that are produced by `JavaCC`. The coupling instrumentor accepts a test program and generates several components. The first is the instrumented version of the test program, and the second is the coverage table (`CTab`) that keeps track of instrumentation. These are sufficient for call coupling, the other criteria need the `LastDef` table, the `Actual` table, and the `SubPath` set.

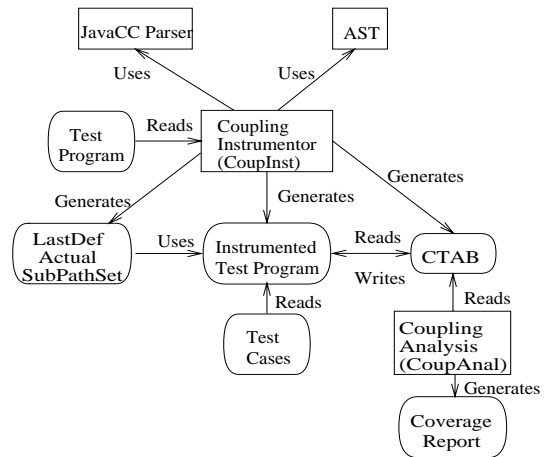


Figure 5. Coupling Coverage Analysis Tool Architecture.

The instrumented test program is then compiled and run with test cases that the user supplies. As the test program

runs, it modifies the coverage table to indicate which coupling relationships have been covered. Once the coverage table has been created, CoupAnal is run to view statistics about the coverage.

Figure 6 is a UML [4] class diagram that describes the coupling coverage instrumentor. Classes are represented by boxes that have three parts; the class name, data members that are declared in the class, and methods of the class. The main entry point (CoupInst) has three objects, a parser, a CouplingTool (which implements the instrumentor), and a tree (which contains the AST parse tree).

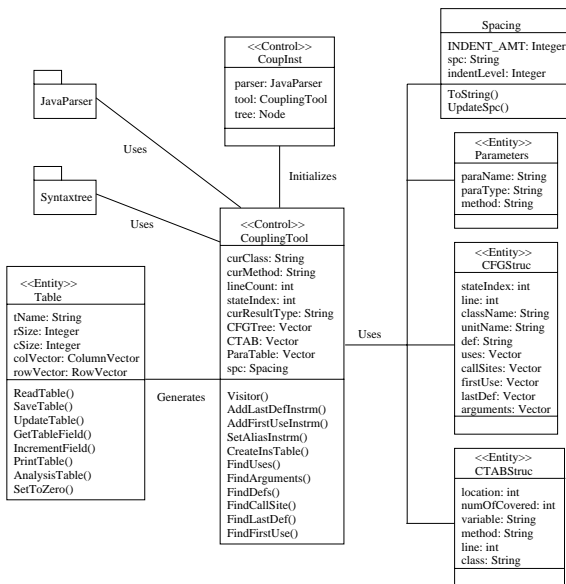


Figure 6. Coupling Coverage Instrumentor UML Diagram.

JavaParser and SyntaxTree are created by separate tools. SyntaxTree is package of classes based on a visitor-based architecture that allows access to all nodes in the tree. The central class, CouplingTool, visits every node in the syntax tree. It identifies all call-sites, coupling parameters, and coupling paths, then adds the appropriate instrumentation for each of the four criteria defined in Section 1.2. The Table class is used to implement four CTabs (one for each criterion), and related tables for last definitions, first uses, etc.

Figure 7 is a UML class diagram that describes the CoupAnal tool. The Instrumented Test Program writes the coverage information to the CTab[] table, and the CoupAnal program accesses the CTabs. The Table is a simple table that stores the information about method calls, parameters, and non-local references. It uses the Java Vector class to organize the informa-

tion. The current report generator is currently fairly simple, it accesses the information and prints a straightforward ASCII report. The reports give the number of couplings found (call-sites, coupling-defs, coupling-uses, or coupling-paths), the number covered, and the percent covered. It also lists statement numbers for all couplings that have not yet been covered.

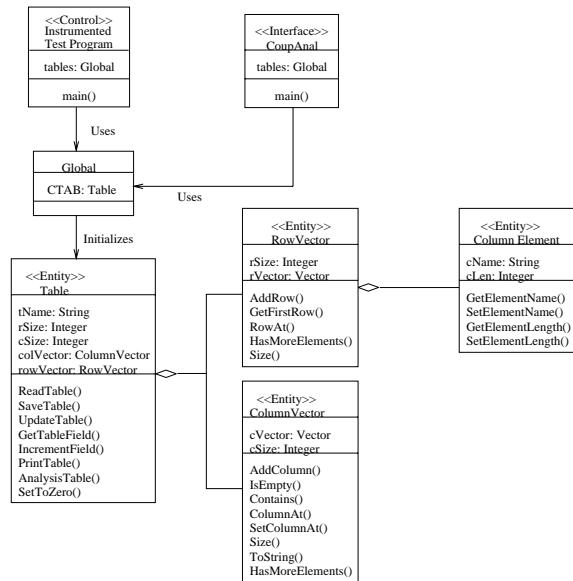


Figure 7. Instrumented Program UML Diagram.

3.1. Tool implementation

The coupling tool coverage analyzer analyzes Java programs, and is built using Java. The tool uses the Java Tree Builder (JTB) by Tao and Palsberg at Purdue University¹. JTB is a syntax tree builder that works in conjunction with Sun’s Java Compiler Compiler (JavaCC) parser generator. It takes a plain JavaCC grammar file as input and automatically generates three things. One, it creates a set of syntax tree classes that are based on the productions in the grammar that use the visitor design pattern [8]. Two, it creates a visitor superclass whose default methods visit the children of the current node. Finally, a JavaCC grammar with the proper annotations to build the syntax tree during parsing is created. The visitor class allows new subclasses to inherit, and override the default methods, performing various operations on and manipulating the syntax tree.

¹JTB can be downloaded from the web at URL <http://www.cs.purdue.edu/homes/taokr/jtb/>.

4. Status and Future Work

This paper presents extensive details on the coupling coverage analysis tool. Our goal is to present enough detail so that other researchers could build similar tools. We expect to use this tool as a basis for a number of future research projects, and also plan to make the tool available to other researchers in the near future. The analyzer has been used with the Mistix program [2, 10], a moderate size program that is based on the Unix file system. We were able to use previously generated test cases and verify full coverage with those test cases [10]. We have extensive plans for this tool, including using the coupling coverage analyzer to perform an extensive empirical investigation of the coupling coverage criteria.

We are also currently working on a project to extend the coupling coverage criteria to object-oriented software [1]. This requires extensions to handle inheritance and polymorphism, two language features that greatly complicates the coupling relationships among methods.

References

- [1] Roger T. Alexander and A. Jefferson Offutt. Analysis techniques for testing polymorphic relationships. In *Proceedings of the Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '99)*, pages 104–114, Santa Barbara CA, August 1999.
- [2] Paul Ammann and A. Jefferson 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.
- [3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.
- [4] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998. ISBN 0-201-57168-4.
- [5] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 244–251, London UK, August 1985. IEEE Computer Society Press.
- [6] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice-Hall, Englewood Cliffs NJ, 1979.
- [7] 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.
- [8] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.
- [9] Zhenyi Jin and A. Jefferson Offutt. Coupling-based integration testing. In *Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)*, pages 10–17, Montreal, Canada, October 1996. IEEE Computer Society Press. Outstanding paper award.
- [10] Zhenyi Jin and A. Jefferson Offutt. Coupling-based criteria for integration testing. *The Journal of Software Testing, Verification, and Reliability*, 8(3):133–154, September 1998.
- [11] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, May 1983.
- [12] 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.
- [13] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *The Journal of Software Testing, Verification, and Reliability*, 7(3):165–192, September 1997.
- [14] M. Page-Jones. *The Practical Guide to Structured Systems Design*. YOURDON Press, New York, NY, 1980.
- [15] S. Rapps and W. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.
- [16] RTCA Committee SC-167. Software considerations in airborne systems and equipment certification, Seventh draft to Do-178A/ED-12A, July 1992.