

An Approach to Fault Modeling and Fault Seeding Using the Program Dependence Graph¹

Mary Jean Harrold
Computer and Information Sciences
Ohio State University
Columbus, OH
harrold@cis.ohio-state.edu

A. Jefferson Offutt
ISSE
George Mason University
Fairfax, VA
ofut@isse.gmu.edu

Kanupriya Tewary
Sun Microsystems
MS-UMPK16-303
Mountain View, CA
kanu@eng.sun.com

Abstract

We present a fault-classification scheme and a fault-seeding method that are based on the manifestation of faults in the program dependence graph (PDG). We enhance the domain/computation fault-classification scheme developed by Howden to further characterize faults as structural and statement-level, depending on the differences between the PDG for the original program and the PDG for the faulty program. We perform transformations on the PDG to produce the different types of faults described in our PDG-based fault-classification scheme. To demonstrate the usefulness of our technique, we implemented a fault seeder to embed faults in C programs. Our fault seeder makes controlled fault transformations to the PDG for a C program, and generates C code from the transformed PDG. The current version of the fault seeder creates multiple fault-seeded versions of the original program, each with one known fault. To demonstrate the operation of the fault seeder, we used it to perform a study of the effectiveness of dataflow testing and mutation testing using a set of faulty programs generated by our fault seeder. We also used the faulty programs to determine the mutation and dataflow adequacy of the fault-detecting test sets.

Keywords: Fault, fault modeling, fault seeding, mutation testing, dataflow testing.

1 Introduction

Software testing is an expensive component of the software development cycle [33] that exercises a program to demonstrate the presence of errors and to provide confidence in the program's correctness. Detecting program faults, which are sections of code that may result in incorrect output, is an important goal of software testing. A meaningful measure of a testing technique is its fault-detection ability or effectiveness [14]. We measure the effectiveness of a testing technique in terms of its ability to detect certain types of faults; we also measure the effectiveness of a testing technique relative to the effectiveness of other testing techniques.

Although there have been studies of fault categories [4, 11, 18, 22], there is no established correlation between categories of faults and testing techniques that expose those faults. Furthermore, although there have been formal and empirical studies [4, 17, 18, 30] on the relative effectiveness of testing techniques, there is no conclusive evidence that one testing technique is superior in fault-detection ability. Performing studies on fault detection is difficult because of the lack of fault-seeding techniques that automatically and systematically insert faults into a program. One fault-seeding method, *mutation testing*, inserts small syntactic changes or faults into a program [10]. However, this approach is expensive and results in a large number of

¹This work was partially supported by grants from Microsoft, Inc., Data General Corp., by NSF under Grants CCR-9109531 and CCR-9357811 to Clemson University and Ohio State University. A preliminary presentation of this work appeared as "Fault Modeling using the Program Dependence Graph" by K. Tewary and M. J. Harrold in *Proceedings of International Symposium on Software Reliability '94 (ISSRE'94)*, November 1994.

fault-seeded programs that must be tested. Additionally, mutation testing inserts only simple faults (simple syntactic code changes) into a program instead of complex faults (multiple syntactic changes or structural changes).

To facilitate the study of testing techniques, we developed a fault classification and fault-seeding approach that models both simple and complex program faults; we described our preliminary results in Reference [34]. Our program representation is a program dependence graph (PDG) that explicitly represents both data and control dependencies in a program [15]. We enhanced the domain/computation fault-classification scheme that was developed by Howden [22] to further characterize faults as structural or statement-level, depending on the differences between the PDG for the original program and the PDG for the faulty program. Structural faults are differences in the control dependence or data dependence information in the PDGs, whereas statement-level faults are differences in the information within PDG nodes. We developed transformations on the PDG to produce the different types of faults described in our PDG-based fault-classification scheme. Because statement-level faults that do not change the structure of the PDG are modeled by fault-based testing methods like mutation testing, we restrict our transformations to those that result in structural changes to the PDG.

To demonstrate the usefulness of our technique, we implemented a fault seeder to embed faults into C programs. Our fault seeder makes controlled fault transformations on the PDG, and generates code from the transformed PDG to create multiple fault-seeded versions of the original program, each with one known fault. To demonstrate the usefulness of our fault seeder, we used it to perform a small empirical study of the relative effectiveness of two unit testing techniques: dataflow testing and mutation testing. We used our fault seeder to seed faults into a set of programs, and tested the programs using both dataflow testing and mutation testing. We also determined mutation adequacy and dataflow adequacy of the fault-detecting test sets; these studies replicate earlier, more extensive studies [31, 39], and our results agree with theirs.

The advantage of our extended characterization of the domain/computation scheme is that we classify each fault type on the basis of its manifestation in the PDG. This fault representation is easy to apply to the implementation of a fault-seeding tool. There are several uses of our fault seeder. First, our fault seeder can be used as a research tool to measure both the effectiveness and relative effectiveness of unit testing techniques. Second, experiments can be conducted with software that contains specific types of faults to determine the testing technique that is best suited for finding certain types of faults. Finally, like mutation testing, our fault seeder can be used for fault-based testing, in which faults are seeded in a program, and a test suite is developed and used to test the faulty programs.

In the next section, we provide background information on program faults, the program dependence graph, dataflow testing, and mutation testing. Section 3 details our fault categorization method and our method of using the PDG to transform programs. Section 4 outlines the implementation of our prototype fault seeder, and Section 5 describes our case study. In Section 6, we discuss related work, and Section 7 gives conclusions and future directions.

2 Background

This section first describes faults and fault categories, and then discusses the program dependence graph, on which our fault modeling is based. Finally, the section describes dataflow testing and mutation testing, which are the two types of unit testing that we used for our case study.

Faults and Fault Categories

According to the IEEE standard definitions [24], a *fault* or “bug” is an incorrect step, instruction or data definition in a program. A fault may result in a *failure*, which is observed when the system exhibits incorrect external behavior. An *error* is an internal difference between the computed, observed, or measured values or conditions, and the true, specified, or theoretically correct values or conditions. Finally, a *mistake* is a human misconception that results in a fault.

To illustrate, suppose a programmer assumes that the upper bound of an array of size 50 is at index 50. For an array of size 50 whose indices start at 0, this is a mistake. A fault in the program due to this mistake may be a statement such as `array[50] = value;`. The failure in this case may be a memory violation and core dump. If the incorrectly accessed location `array[50]` contains a numerical value that is used in some computation, the output will be different from the expected output, and this difference is an error.

Howden [22] originally classified program faults, and Zeil [40] extended Howden’s classification. There have been other fault classifications based on observed instances of faults and on the complexity of faults [12]. According to the Howden/Zeil classifications, program faults² are categorized as domain faults or computation faults. A *domain fault* occurs when, due to an error in control flow, a program generates incorrect output. A *computation fault* occurs when a program takes the correct path, but generates incorrect output because of faults in the computations along that path. Domain faults are further classified into two categories. A *missing path* fault is caused by a missing conditional or clause and the associated statements, and a *path selection* fault is caused by an incorrect decision at a predicate. Path-selection faults can result from an incorrect predicate (*predicate fault*) or from an incorrect assignment statement that propagates to a control point, leading to an incorrect decision (*assignment fault*).

For fault-seeding purposes, faults should be “representative” of naturally-occurring faults; otherwise, any results obtained from the seeded faults may be inaccurate or biased. Unfortunately, to date, there is no accepted model with which to determine whether seeded faults are representative. Offutt and Hayes [30] suggest a model for program faults that they use to identify representative collections of faults. Their model characterizes a fault syntactically as the changes needed to correct the fault, and characterizes a fault semantically by viewing the faulty program as containing a computation that produces incorrect output over some subset of the input domain. Using this characterization, they define the *size* of a fault: the *syntactic size* of a fault is the fewest number of statements or tokens that must be changed to correct the fault; the *semantic size* of a fault is the relative size of the input subdomain for which the program produces incorrect output. They use their semantic size model to claim that a collection of faults is “realistic” if the collection has a distribution of semantic sizes that is similar to that of real faults. However, there is little data about distributions of semantic fault sizes for naturally occurring faults.

²Howden/Zeil use the term error instead of fault in their domain/computation classification. We shall use the term fault because we are referring to sections of code that may result in failures.

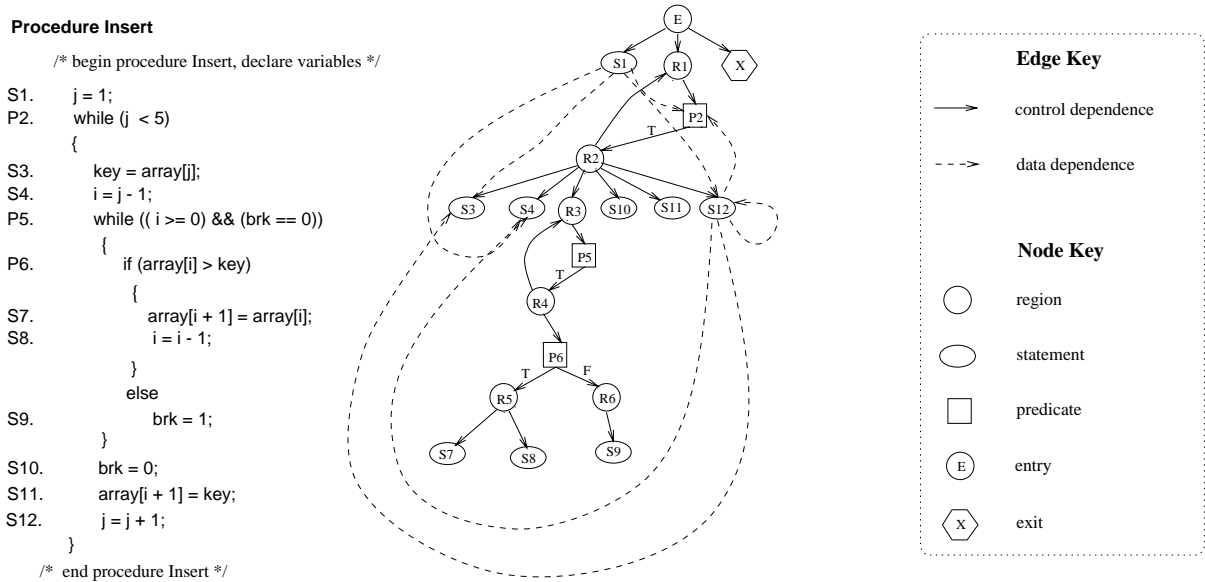


Figure 1: Procedure `Insert` is given on the left and its partial PDG is given on the right.

The Program Dependence Graph

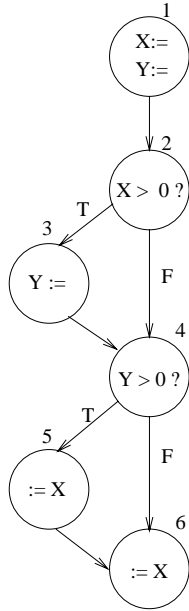
The *program dependence graph* (PDG) explicitly represents both data dependencies and control dependencies for a program [15]. In a PDG, nodes represent statements and predicate expressions, and edges represent either data dependencies or control dependencies. Thus, the PDG actually consists of a *data-dependence subgraph* (DDS) and a *control-dependence subgraph* (CDS). Figure 1 gives an example procedure `Insert`, an insertion sort for a five element array, and its PDG; for brevity, we omit declarations and procedure blocks from the figure. For our discussion, we assume that a program has a single entry and a single exit, which implies that all procedures return to their callers. Thus, we treat call sites as simple statements. These assumptions do not limit our technique, which can be generalized easily.

The PDG contains several types of nodes. Statement nodes, shown as ellipses in Figure 1, represent simple program statements such as assignment, input/output, or call. Predicate nodes, shown as squares in the figure, represent statements from which two edges may originate. An exit node, shown as a hexagon in the figure, represents the exit from the procedure. Region nodes, shown as circles in the figure, represent regions, which summarize the control-dependence conditions necessary to reach statements in the region. We distinguish several types of region nodes. In Figure 1, R1 and R3 are *loop header regions*, because they summarize the control dependencies for `while` loops. R2, R4, and R5 are *true regions*, because they summarize control dependence on the true branch of a predicate. Similarly, R6 is a *false region*, because it summarizes control dependence on the false branch of a predicate.

Two nodes in a CDS are connected with an edge if one node is control dependent on the other. In a control flow graph, if there are two nodes X and Y, X is *control dependent* on Y if there are two paths from Y, and taking one path will always lead to X, whereas taking the other path may not lead to X. For example, in Figure 1, S7 and S8 are control dependent on P6-T; region node R5 summarizes this control dependence.

A *data dependence*³ exists between statements X and Y in a program if X defines a variable *v*, Y uses

³Although there are several types of data dependence defined in the literature, we consider only *flow* dependence.



Data Flow Criteria	Program Components to be Covered	Minimal Adequate Test Set(s)	Program Path(s) Traversed
all-nodes	nodes: 1,2,3,4,5,6	{(2,2)}	123456
all-edges	edges: (1,2), (2,3), (2,4), (3,4), (4,5), (4,6), (5,6)	{(2,2), (0,0)}	123456, 1246
all-uses	definition-use pairs: (X,1,(2,3)), (X,1,(2,4)), (X,1,5), (X,1,6), (Y,1,(4,5)), (Y,1,(4,6)), (Y,3,(4,5)), (Y,3,(4,6))	{(0,0), (0,2), (2,0)}	1246, 12456, 12346

Figure 2: Control flow graph on the left and a table listing three dataflow testing criteria, program components to be covered to satisfy these criteria, test sets, and paths in the control flow graph that are traversed.

v , and there is a path from X to Y on which v is not defined. Dataflow analysis [2, 26] is used to gather dataflow information. The DDS represents the data dependencies in the program. In Figure 1, the dashed lines are data dependence edges for variable j .

Dataflow Testing

Dataflow testing considers the associations between variable definitions and variable uses. A *definition* of a variable modifies a data item, whereas a *use* of a variable references a data item without modifying it. A definition of a variable reaches a use of that variable if there is a subpath in the program from the definition to the use on which the variable is not redefined. A *definition-use pair* is an association of a definition of a variable and a use of that variable that the definition reaches.

Dataflow testing is a family of adequacy criteria [16] that includes all-nodes, all-edges, and all-uses. A test set satisfies the *all-nodes* criterion if every node in the control flow graph⁴ is executed by some test in the test set. A test set satisfies the *all-edges* criterion if each edge in the control flow graph is executed by some test in the test set. A test set satisfies the *all-uses* criterion if each definition-use pair in the program is executed by some test in the test set.

Figure 2 gives an example control flow graph for a partial program; only the definitions and uses for variables X and Y are shown. Definitions are shown with the variable on the left-hand side of assignments ($X:=$) and uses are shown with the variable on the right-hand side of assignments ($:= X$). The table to the right of the figure shows the program components and program paths that are traversed for each of the three dataflow criteria defined above, along with minimal adequate test sets for each criterion.

⁴A *control flow graph* for a program consists of a node for each statement in the program and edges representing the flow of control between statements.

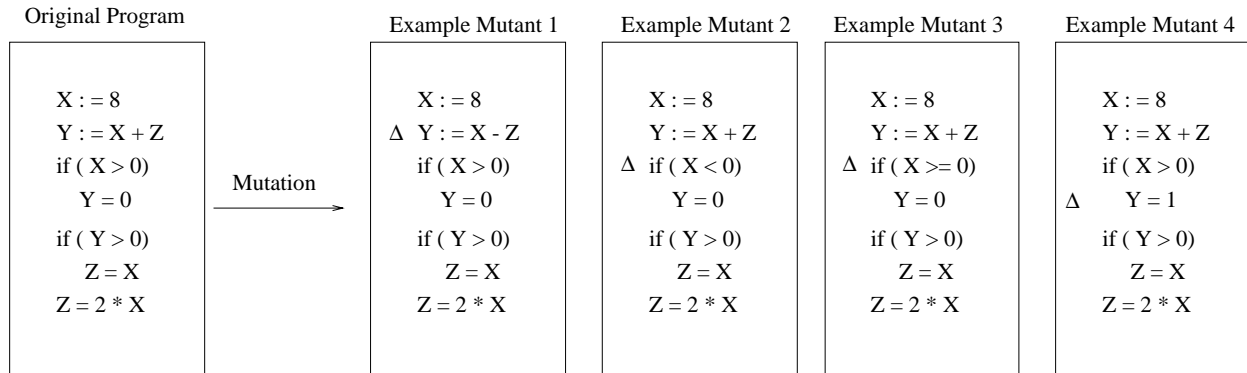


Figure 3: Mutation operators are applied to the the original program on the left to create mutant programs; only 4 mutants have been shown. The Δ marks the statement at which the mutation is applied.

Mutation Testing

Mutation testing is a fault-based testing technique [10, 19] that is based on the assumption that a program is well tested if all simple faults are predicted and removed; complex faults are *coupled* to simple faults and are thus, detected by tests that detects simple faults [10, 28, 38]. *Mutation operators* introduce simple faults into the program. Each change or *mutation* that is performed by a mutation operator produces a mutant program or simply, a *mutant*. A mutant is *killed* by a test that forces it to produce output that is different from the original program.

Figure 3 shows an example program with four mutants, in which each mutant has exactly one mutation. *Equivalent mutants* are mutant programs that are functionally equivalent to the original program and therefore cannot be killed by any test. In Figure 3, mutant 3 is an equivalent mutant because, if the value of X is 8, it makes no difference whether we check $X > 0$ or $X >= 0$.

Budd[5] defined *mutation analysis* as a method for evaluating the degree to which a set of tests exercise a program. His suggested procedure extends the test set until all nonequivalent mutants have been killed. Because the method of test generation is not significant here, we use the terms mutation analysis and mutation testing interchangeably.

3 Modeling Faults using the PDG

We characterize the types of faults that we model using the PDG as extensions to the domain/computation classification scheme developed by Howden [22]. Our goal in extending this scheme is to establish a relationship between existing fault categories and their PDG representations - not to provide an exhaustive fault classification. Section 3.1 details our extensions to the domain/computation classification that are based on fault manifestations in the PDG. The relationship between our PDG-based fault classification and the domain/computation scheme is shown in our extended fault classification, given in Figure 4. We consider only faults that occur within procedures; we do not consider interface faults. In section 3.2, we describe algorithms that transform the PDG to model the structural fault types detailed in Section 3.1. For these transformation algorithms, we consider only fault types that affect the PDG structure; we do not consider statement-level faults because they are modeled extensively by other methods such as mutation.

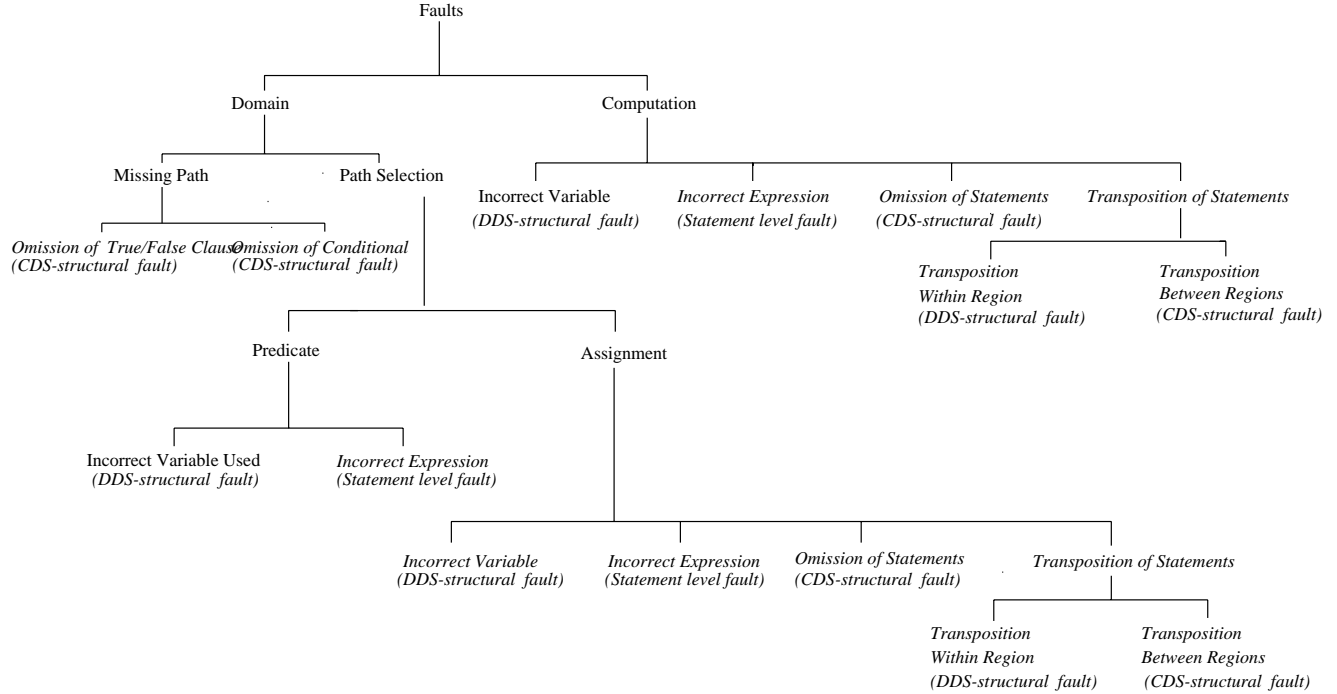


Figure 4: PDG-based fault classification: Italicized entries are our extensions to the domain/computation classification; parenthesized entries indicate the general PDG fault class for each fault type.

3.1 PDG-based Fault Classification

Based on the PDG, we characterize each type of fault in the domain/computation classification scheme as *structural* or *statement-level*. Structural faults are “macro-level” faults, and are manifested as differences in the structure of the PDGs of the original program and the faulty version. If this difference in structure is due to a difference in the control dependence successors or predecessors of nodes in the PDGs, then the fault is a *CDS-structural fault*; if the difference in structure is due to a difference in the data-dependence successors or predecessors of nodes in the PDGs, then the fault is a *DDS-structural fault*.

A statement-level fault is a textual difference between a faulty statement and the original statement that is not a difference in the structure of the PDGs. For these types of faults, the CDS and DDS structure of the faulty program are the same as the CDS and DDS structure of the original program. However, the “micro-level” information inside some node differs from the information in the corresponding node in the original PDG. We identify two types of statement-level faults: *Incorrect Operators*, such as ‘<’ instead of ‘>’, and *Incorrect Precedence of Operation*, which occurs because of a lack of, or misuse of, parentheses.

Our model classifies faults as either domain or computation faults, depending on whether the fault relates to flow of control or computations along a particular path. In the following discussion, we describe our PDG-based fault classification, and give some examples.

Domain Faults

We further classify domain faults as missing path faults and path selection faults.

Missing Path Faults occur when the correct predicate is executed, but the required path is missing. There are two manifestations of this type of fault in the PDG: (1) Omission of True/False Clause and (2) Omission of Conditional.

Procedure Insert with Omission of Clause Fault

```

/* begin procedure Insert, declare variables */
S1.  j = 1;
P2.  while (j < 5)
{
S3.  key = array[j];
S4.  i = j - 1;
P5.  while ((i >= 0) && (brk == 0))
{
P6.  if (array[i] > key)
{
S7.  array[i + 1] = array[i];
S8.  i = i - 1;
}
}
S10. brk = 0;
S11. array[i + 1] = key;
S12. j = j + 1;
}
/* end procedure Insert */

```

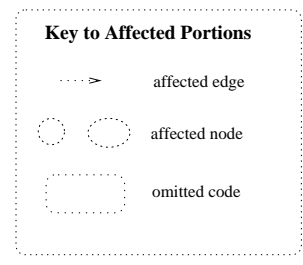
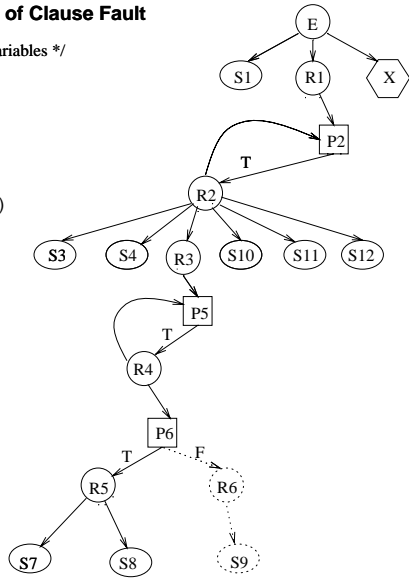


Figure 5: Example CDS for Insert for *Omission of Clause Fault*.

1. *Omission of True/False Clause Faults* involve the omission of one branch of a conditional statement, and result in the omission of a true or false clause from a conditional. This fault is a CDS-structural fault represented in the PDG by the omission of a true or false region node and its successors. An example of this type of fault is shown in Figure 5, where the false branch of P6 of Figure 1 is omitted.
2. *Omission of Conditional Faults* involve the omission of an entire conditional statement, such as a **while** statement or **if** statement. An omission of an error checking conditional is an example of this type of fault. This fault is a CDS-structural fault represented in the PDG by the omission of a predicate or loop header region and its subgraph. Figure 6 shows an example of this type of fault, where the **if** statement in P6 is omitted.

Procedure Insert with Omission of Conditional Fault

```

/* begin procedure Insert, declare variables */
S1.  j = 1;
P2.  while (j < 5)
{
S3.  key = array[j];
S4.  i = j - 1;
P5.  while ((i >= 0) && (brk == 0))
{
}
S10. brk = 0;
S11. array[i + 1] = key;
S12. j = j + 1;
}
/* end procedure Insert */

```

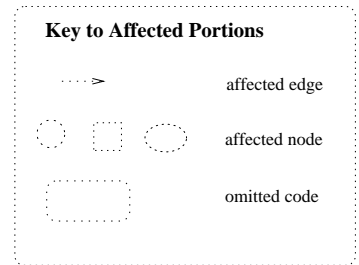
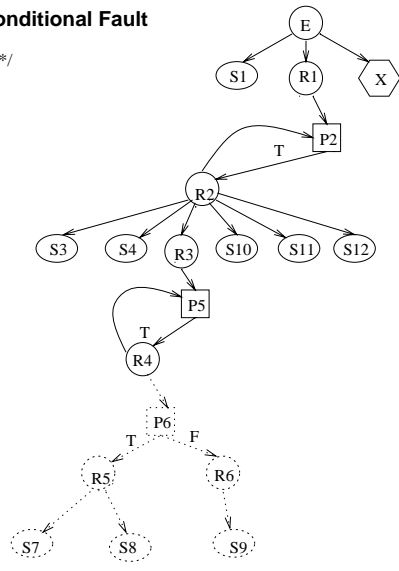


Figure 6: Example CDS for Insert for *Omission of Condition Fault*.

Procedure Insert with Incorrect Predicate Variable Fault

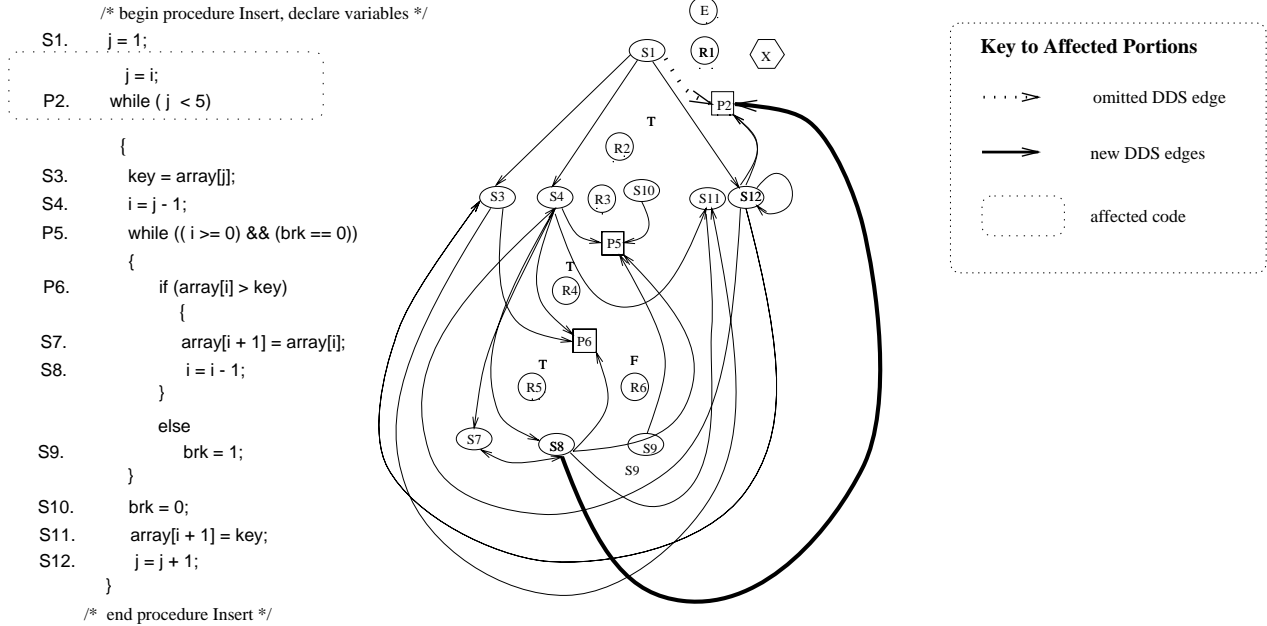


Figure 7: Example DDS for `Insert` for *Incorrect Variable Used Fault (Predicate)*.

Path Selection Faults occur when paths originating at a predicate are correct, but an incorrect decision at the predicate results in selection of the wrong path. The fault may be in the predicate itself, or it may be in an assignment statement that affects the predicate.

1. Predicate Faults

- (a) *Incorrect Variable Used Faults (Predicate)* involve the use of an incorrect variable at a predicate node. For example if the predicate expression is ‘ $(a > c)$ ’ instead of ‘ $(a > b)$ ’, the use of variable c is incorrect. This fault is a DDS-structural fault in which the data-dependence edges at a predicate node are violated. Figure 7 shows an example of this type of fault, where an extra statement, ‘ $j = i$ ’, alters the data-dependence between nodes $S1$ and $P2$, because it results in the incorrect value of j to be used in $P2$.
- (b) *Incorrect Expression Faults (Predicate)* involve the use of one or more incorrect operators. An example of an incorrect predicate expression is ‘ $(a < b)$ ’ instead of ‘ $(a > b)$ ’. This fault is a statement-level fault that is represented by an incorrect expression at a PDG node.

2. Assignment Faults

- (a) *Incorrect Variable Faults (Assignment)* are violations of the data-dependence relations in the program. The difference may be the use of an incorrect variable (*Incorrect Variable Used Faults*), or the definition of an incorrect variable (*Incorrect Variable Defined Faults*). These faults are DDS-structural faults, and are violations of data-dependence edges into an assignment node that affects a predicate node. Figure 8 is an example of an *Incorrect Variable Defined Fault* in which the variable j replaces variable key (being defined) in assignment statement node $S3$.

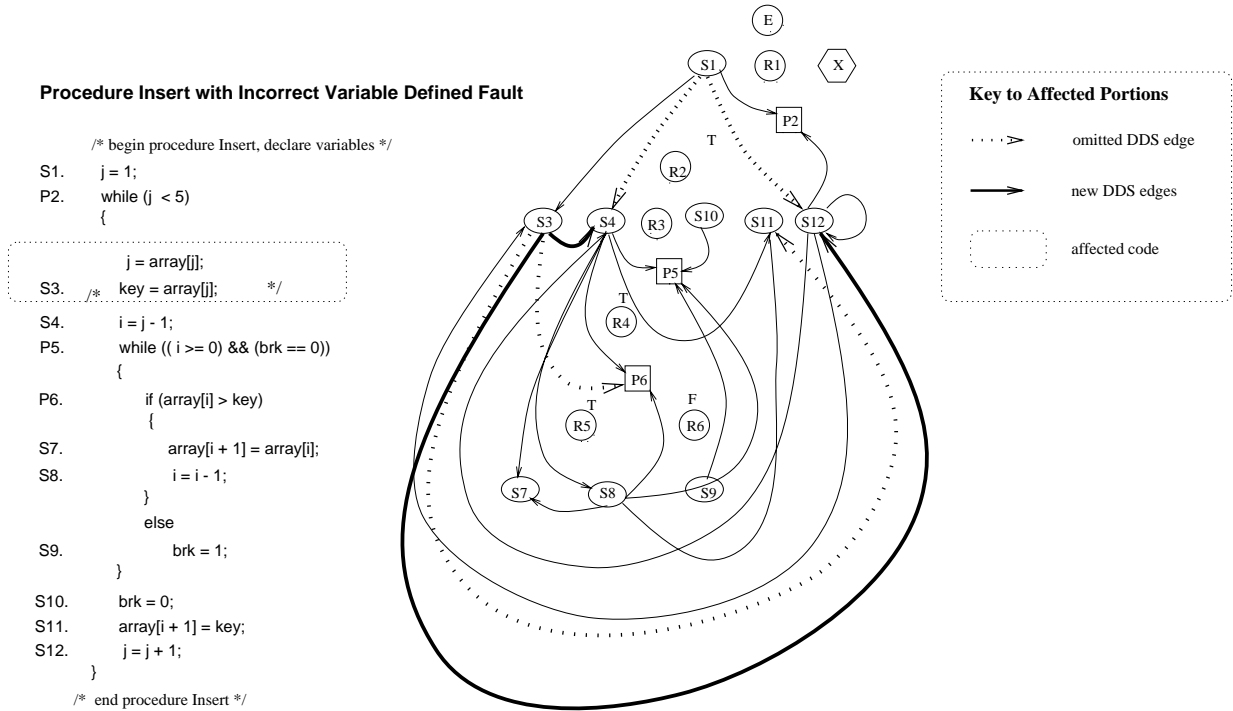


Figure 8: Example DDS for `Insert` for *Incorrect Variable Defined Fault (Assignment)*.

(b) *Incorrect Expression Faults (Assignment)* involve the use of one or more incorrect operators. An example of an incorrect expression is ‘`a = b + c`’ instead of ‘`a = b - c`’. This fault is a statement-level fault that is represented by an incorrect expression in a PDG node.

(c) *Omission of Statement Faults (Assignment)* involve missing assignment statements. These faults are CDS-structural faults, and are represented by the omission of statement nodes in the PDG. Figure 9 shows an example of this type of structural fault.

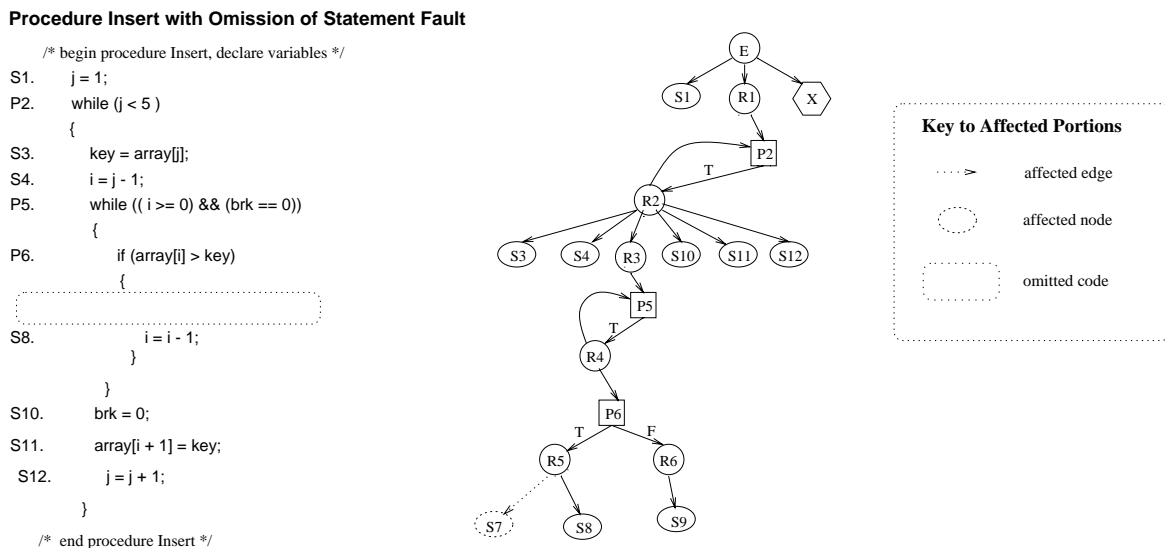


Figure 9: Example CDS for `Insert` for *Omission of Statement Fault*.

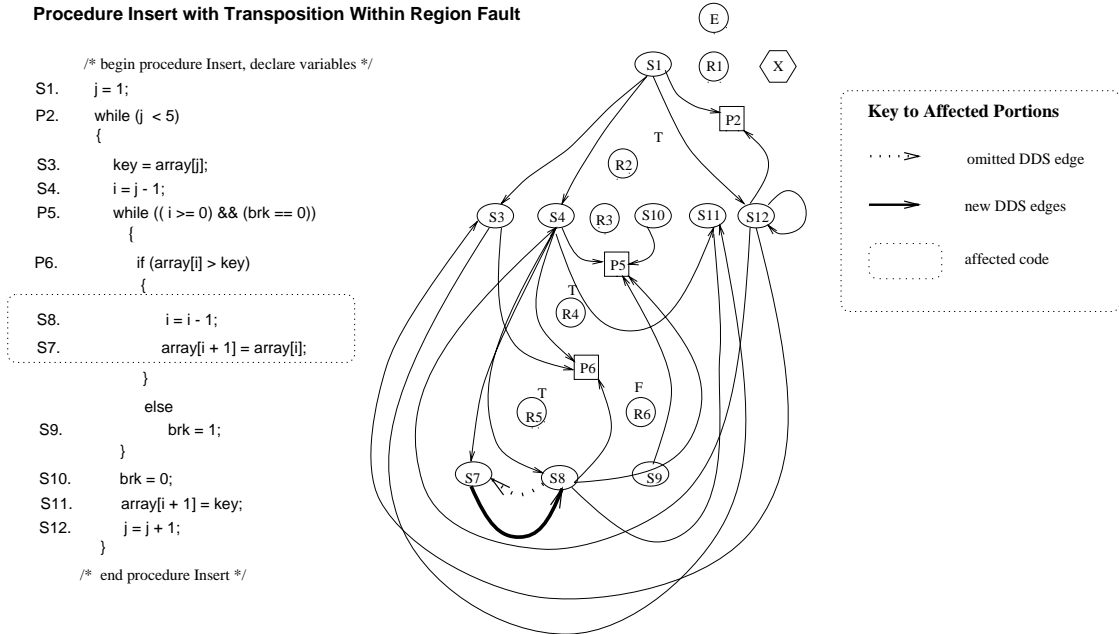


Figure 10: Example DDS for *Insert* for *Transposition Within Region Fault (Assignment)*.

(d) *Transposition of Statements Faults (Assignment)* are of two types, depending on whether the transposition occurs within the same region, or in different regions, of control dependence.

- *Transposition Within Region Faults (Assignment)* occur when two assignment statements that have data dependencies between them are transposed. This fault is a DDS-structural fault and is a violation in the data-dependence edges between the two statements. For example, suppose that statement *s1* assigns to variable *v*, statement *s2* uses variable *v*, and *s1* and *s2* are adjacent to each other in the code. Then there is a data-dependence edge from *s1* to *s2*. A fault occurs when *s1* and *s2* are reversed in the code, so that the original data dependencies are violated. An example is shown in Figure 10.
- *Transposition Between Regions Faults (Assignment)* occur when a statement appears in the incorrect region of control dependence. An example of this fault is omission of compound statement braces in C that causes a conditional to affect only the statement immediately after it. This fault is a CDS-structural fault, modeled by an omission of a node or branch from one point in the PDG and its appearance in another section of the PDG. An example is shown in Figure 11; nodes S4-S12, in R2 in Figure 1, are successors of E in the faulty version.

Computation Faults

Computation faults occur in statements that lie on the correct control flow path in the program. We see from Figure 4 that computation faults include the same fault categories as Domain-Path-Selection-Assignment faults, because both types of faults involve assignment statements. Thus, computation faults manifest themselves in the PDG in the same way as Domain-Path-Selection-Assignment faults, which we described in Section 3.1. The important difference between these fault types is that computation faults affect program output by computing incorrect values, instead of causing an incorrect path to be selected. A fault

Procedure Insert with Transposition Between Regions Fault

```

/* begin procedure Insert, declare variables */
S1.  j = 1;
P2.  while(j < 5)
    {
S3.      key = array[j];
S4.      i = j - 1;
P5.      while(( i >= 0) && (brk == 0))
        {
P6.          if(array[i] > key)
            {
S7.              array[i + 1] = array[i];
S8.              i = i - 1;
            }
        else
S9.            brk = 1;
        }
S10.     brk = 0;
S11.     array[i + 1] = key;
S12.     j = j + 1;
/* end procedure Insert */

```

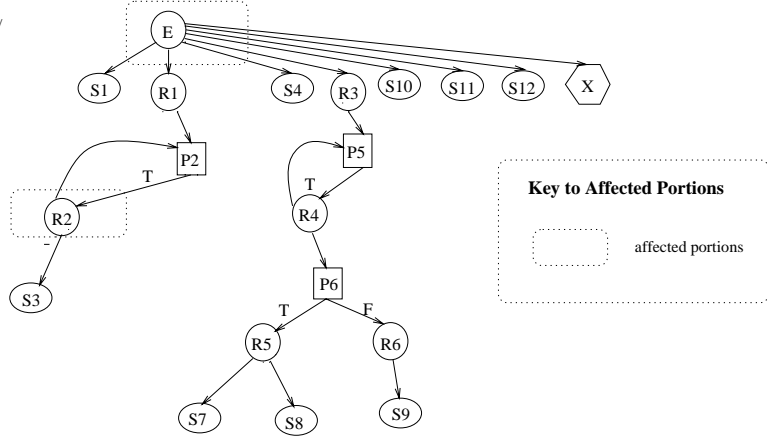


Figure 11: Example CDS for Insert for *Transposition Between Regions Fault (Assignment)*.

in an assignment in which the value is used in a computation is a computation fault, whereas a fault in an assignment that affects a predicate is an assignment fault. A fault can be both a computation and an assignment fault.

3.2 Fault Transformation Algorithms

We model all structural faults presented in our fault classification scheme, shown in Figure 4, by transformations on the PDG. In this section, we overview our algorithms for transforming the CDS and DDS. For simplicity, we present the handling of structured programs; the transformation techniques can also be applied to unstructured programs. In all cases, we assume that a high level driver routine traverses the PDG to determine the node to transform. The PDG node structure used in all algorithms is:

Field	Type	Description
id	integer	node id
type	integer	type of node (region, predicate, statement etc.)
src	string	source string mapped to node
cd_succ_list, cd_pred_list	list of node pointers	control-dependence-successor/-predecessor lists
dd_succ_list, dd_pred_list	list of node pointers	data-dependence-successor/-predecessor lists
def, use	list of string pointers	variables defined and used
subgraph_count	integer	count of nodes in subgraph

3.2.1 CDS-structural Transformations

We perform CDS-structural fault transformations using two basic operations on the CDS: (1) *pruning* to remove nodes and branches, and (2) *grafting* to remove nodes and branches from one branch, and join them to another branch. Faults of omission require only pruning, whereas faults of transposition require grafting.

Pruning algorithm `OmitPath`, shown in Figure 12, is used to model *Omission of Clause*, *Omission of Conditional*, and *Omission of Statements* faults. `OmitPath` seeds faults by pruning statement nodes, or region or predicate branches. `OmitPath` calls `GetNode` to return `node_ptr`, a pointer to the region or predicate that is to be deleted. `DeleteList` removes this region or predicate node and its successor nodes. `OmitPath`

```

Algorithm OmitPath
input    node_id                id of region/predicate node for transformation
globals cd_info                CDS
output   modified cd_info and reduced node_count
declare  nnode_ptr              pointer to node
          GetNode(node_id)        returns pointer to node
          DeleteList(list, node_id) deletes node from list

begin OmitPath
  node_ptr = GetNode(node_id)
  DeleteList(node_ptr.pred_list.successor_list, node_id)
  remove node referenced by node_ptr from successor list of parent
  node_count = node_count - node_ptr.subgraph_count
  return node_count
end OmitPath

```

Figure 12: Fault transformation algorithm `OmitPath`.

deletes the node referenced by `node_ptr` from the successor list of its parent; this deletion also causes all nodes in the subgraph rooted at this node to be deleted. Next, the algorithm decrements the PDG node count, `node_count`, by the number of nodes in the deleted subgraph; the subgraph node count is part of the node structure. Finally, `OmitPath` returns `node_count`. For example, to model the fault in Figure 5, given the PDG in Figure 1, `OmitPath` prunes region node R6 and decrements the PDG node count, `node_count`, by 2. `OmitPath` has runtime that is linear in the number of nodes in the PDG.

Our technique seeds *Omission of Conditional* faults by pruning all conditional branches. This transformation is similar to the *Omission of Clause* transformation except that, while the PDG is being traversed, conditional and iterative regions are removed. Algorithm `OmitConditional` performs this transformation; we omit `OmitConditional` because it is similar to algorithm `OmitPath`, given in Figure 12.

Algorithm `TransposeBetweenRegions`, shown in Figure 13, is a grafting algorithm that seeds *Transpose Between Regions (Assignment)* faults. This transformation results in the transposition of nodes from one point in the PDG to another. Procedure `GetNode` returns a pointer, `node_ptr`, to the region node from which nodes must be transposed. A new parent is found for the transposable nodes (nodes that will be moved out of the region) by `GetViableParent`, which searches and returns a new parent region node to which new control-dependence successors will be grafted. `GetNode` returns a pointer, `parent_ptr`, to the new parent. For each transposable successor `n` in `transpose_list`, the region node’s successor count is decremented and the new parent’s successor count is incremented. After `GetNode` returns a pointer to `n`, `InsertList` updates the successor and predecessor lists of the new parent and the transposed node, to form the new control-dependence edge. Finally, `DeleteList` severs the link between region node and transposed node. After grafting, the transformed PDG has the same number of nodes as the original PDG. Figure 11 gives an example of this transformation; R2 is the region node from which nodes S4 to S12 are transposed onto the new parent region, E. `TransposeBetweenRegions` has runtime that is linear in the number of nodes in the PDG.

3.2.2 DDS-structural Transformations

The DDS-structural transformations involve a violation of one or more edges in the DDS, in conjunction with the addition of new edges. We assume that the required dataflow information is available at each node. The algorithms have runtime that is linear in the number of nodes in the PDG.

```

Algorithm TransposeBetweenRegions
input    node_id : id of region node for transformation
           transpose_list : list of node ids to transpose
globals  cd_info : CDS
output   transformed cd_info
declare  node_ptr, n_ptr, parent_ptr : pointers to nodes
           new_parent_id, n : node ids
           GetNode(node_id) : returns pointer to node
           DeleteList(list, node_id), InsertList(list, node_id) : deletes/inserts node(s)
           GetViableParent(node_id) : returns new parent id

begin TransposeBetweenRegions
  node_ptr = GetNode(node_id)
  new_parent_id = GetViableParent(node_id)
  parent_ptr = GetNode(new_parent_id)
  foreach(n ∈ transpose_list) do
    node_ptr.succ_count--
    parent_ptr.succ_count++
    n_ptr = GetNode(n)
    InsertList(n_ptr.pred_list, new_parent_id); InsertList(parent_ptr.cd_succ_list, n_ptr)
    DeleteList(n_ptr.cd_pred_list, node_ptr); DeleteList(node_ptr.cd_succ_list, n)
  end TransposeBetweenRegions

```

Figure 13: Fault transformation algorithm TransposeBetweenRegions.

Figure 14 shows our first DDS-structural transformation algorithm that we use to seed *Incorrect Variable Used Faults (Predicate/Assignment/Computation)*. In algorithm `IncorrectVarUse`, after `GetNode` returns a pointer to the node being transformed, procedure `DeleteEndList` deletes a DDS predecessor of this node. This violation of an incoming DDS edge represents the removal of a variable used in the node. `GetReachingDefs` selects another node (`def_ptr`) containing a variable definition reaching this node. Using `InsertList`, the algorithm constructs a new DDS edge from the new variable definition node `def_ptr`. `PrefixString` modifies the current node source string, by attaching the required assignment statement (new variable used is assigned to old variable used). Figure 7 shows the result of applying `IncorrectVarUse` to the PDG in Figure 1. Node P2 is transformed by violating the DDS edge from S1 (definition of `j`), and adding the DDS edge from S8 (definition of `i`); the assignment '`j = i`' is prefixed to the source string at node P2.

The second DDS transformation is more complex. Figure 15 shows algorithm `IncorrectVarDef` that

```

Algorithm IncorrectVarUse
input    node_id : id of statement node for transformation
globals  cd_info, dd_info : CDS and DDS
output   modified dd_info
declare  node_ptr, succ_ptr, def_ptr : pointers to nodes
           GetNode(node_id) : returns pointer to node
           DeleteEndList(list), InsertList : deletes/inserts node(s)
           GetReachingDefs(node_ptr) : returns pointer to each node with reaching def
           PrefixString(string pointer) : prefixes string to source string

begin IncorrectVarUse
  node_ptr = GetNode(node_id)
  succ_ptr = DeleteEndList(node_ptr.dd_pred_list)
  def_ptr = GetReachingDefs(node_ptr)
  InsertList(def_ptr.dd_succ_list, node_ptr); InsertList(node_ptr.dd_pred_list, def_ptr)
  PrefixString('succ_ptr.def = def_ptr.def', node_ptr.src)
end IncorrectVarUse

```

Figure 14: Fault transformation algorithm IncorrectVarUse.

```

Algorithm IncorrectVarDef
input    node_id : id of statement node for transformation
globals  cd_info, dd_info : CDS, DDS
output   modified dd_info
declare  node_ptr, succ_ptr, n_ptr : pointers to node
           n : id of node
           use_ptr, def_ptr : pointer to node with reachable use/new def
           GetNode(node_id) : returns pointer to node
           DeleteList(list, node_id), InsertList(list, node_ptr) : deletes/inserts node(s)
           GetReachingDefs(node_ptr) : returns pointer to each node with reaching defs
           GetReachableUses(node_ptr) : returns pointer to each node with reachable uses
           ChangeLValue(string pointer, lhs) : substitutes lhs in string

begin IncorrectVarDef
  node_ptr = GetNode(node_id)
  DeleteList(node_ptr.dd_succ_list, all-nodes)
  def_ptr = GetReachingDefs(node_ptr)
  node_ptr.def = def_ptr.def
  ChangeLValue(node_ptr.src, node_ptr.def)
  while(use_ptr =GetReachableUses(node_ptr) is valid node) do
    InsertList(node_ptr.dd_succ_list, use_ptr); InsertList(use_ptr.dd_pred_list, node_ptr)
  foreach (n ∈ def_ptr.dd_succ_list)do
    if (n ∈ node_ptr.dd_succ_list) then
      DeleteList(def_ptr.dd_succ_list, n)
      n_ptr = GetNode(n)
      DeleteList(n_ptr.dd_pred_list, def_ptr.id)
  end OmitIncorrectVarDef

```

Figure 15: Fault transformation algorithm `IncorrectVarDef`.

models *Incorrect Variable Defined Faults (Assignment/Computation)*. First, `GetNode` returns a pointer to the statement node. `DeleteList` removes all outgoing DDS edges at this node because the definition will change. `GetReachingDefs` returns a node pointer (`def_ptr`) to another variable definition. Next, the algorithm replaces the original variable defined (in the `def` field) by the new variable in `def_ptr`. The source string at (`src`) is changed by `ChangeLValue`. `GetReachableUses` returns all reachable uses of the new definition, and for each of these uses, `InsertList` constructs new DDS edges. Finally, because the new definition may “kill” some other definitions of the variable, all DDS edges from the “killed” definition must be removed. Removal of these edges is done in the `foreach` loop at the end of `IncorrectVarDef`. If any DDS successors of the node `def_ptr` are now successors of the new definition at `node_ptr`, they are deleted from the successor list of `def_ptr`, because that definition no longer reaches. Figure 8 shows an example of this transformation, where the PDG in Figure 1 has an *Incorrect Variable Defined Fault (Assignment)*. Node S3 is transformed, and the variable `key` is changed to `j`. This change results in the omission of DDS edges to P6 and S11 because S3 no longer defines `key`. New edges to S4 and S12 are added, as these nodes have uses of `j`. Also, edges (S1, S4) and (S1, S12) are removed, because the definition at S1 no longer reaches S4 and S12.

Figure 16 shows our third DDS transformation algorithm. This transformation models the *Transposition Within Region Fault (Assignment)*. The algorithm `TransposeWithinRegion` accepts as input the statement node to be transposed; `GetNode` returns a pointer to this statement node. `GetSibling` obtains each sibling node of `node_id`, and checks to see if the sibling node is the DDS successor of `node_id`. If the sibling node is the DDS successor of `node_id`, the direction of the edge is reversed using `DeleteList` and `InsertList` routines. Also, the information within the two nodes is exchanged using the `ExchangeInfo` routine. For example in the PDG in Figure 1, there is a data-dependence edge (S8, S7) (not shown). Thus, exchanging

```

Algorithm TransposeWithinRegion
input    node_id : id of statement node for transformation
globals  cd_info, dd_info : CDS, DDS
output   modified dd_info
declare  node_ptr, sibling_ptr : pointers to node
           GetNode(node_id) : returns pointer to node
           DeleteList(list, node_id), InsertList(list, node_ptr) : deletes/inserts node(s)
           GetSibling(node_id) : returns pointer to sibling node
           ExchangeInfo(node_ptr, node_ptr) : swaps node info

begin TransposeWithinRegion
  node_ptr = GetNode(node_id)
  while (sibling_ptr = GetSibling(node_id) != invalid pointer) do
    if (sibling_ptr.id ∈ node_ptr.dd_succ_list)
      DeleteList(node_ptr.dd_succ_list, sibling_ptr.id);
      DeleteList(sibling_ptr.dd_pred_list, node_ptr.id)
      InsertList(node_ptr.dd_pred_list, sibling_ptr);
      InsertList(sibling_ptr.dd_succ_list, node_ptr)
      ExchangeInfo(node_ptr, sibling_ptr)
  end TransposeWithinRegion

```

Figure 16: Fault transformation algorithm `TransposeWithinRegion`.

the information in these nodes results in a fault. The DDS edge (S8, S7) is removed, and a new edge (S7, S8) is created. Reachable uses and reaching definitions information for these nodes is also exchanged.

4 Prototype Fault Seeder

To demonstrate the usefulness of our transformation algorithms that model program faults, we built a prototype fault seeder that inserts structural faults into C programs. We developed our fault seeder in C on Sun SPARC machines⁵, using our analysis system, `Aristotle` [21], to gather program information and generate the faulty programs. Figure 17 shows a dataflow diagram representing our fault seeder, and illustrates its use of some of `Aristotle`'s components. `Aristotle`'s parser/analyzer, `CParse`, takes the C program, `pgm.c`, as the input to be seeded with faults. `CParse` parses the program, and produces various types of analysis information including the PDG, which is used by our fault seeder. The fault-transformation component of our prototype, `FSeed`, inputs the PDG for `pgm.c`, along with the fault type that the user desires. `FSeed` seeds as many faults as possible of this fault type, and produces a new PDG for each fault it seeds. The transformed PDGs are then input to `CGen`, which is a source code generator and instrumenter available in `Aristotle`. For each transformed PDG, `CGen` generates a C program, and stores it in a file. The result is a set of n fault-seeded program versions, `pgm.fseed.0.c ...pgm.fseed.n.c`, each with exactly one fault of the user specified fault type⁶.

`FSeed` implements the fault-transformation algorithms presented in Section 3.2, each of which requires information about the portion of the PDG to transform. To provide this information, `FSeed` has a high-level fault-seeder driver that calls the transformation algorithms with appropriate information. Figure 18 presents the partial fault-seeder driver, `FaultSeederDriver`; for brevity, we detail two cases in this algorithm, mention two other cases, and omit the remaining cases.

⁵SPARCstation is a trademark of Sun Microsystems, Inc.

⁶We used this design for our prototype fault seeder. A tool for general distribution might let the user determine, for each fault classification, the percentage of faults to be generated.

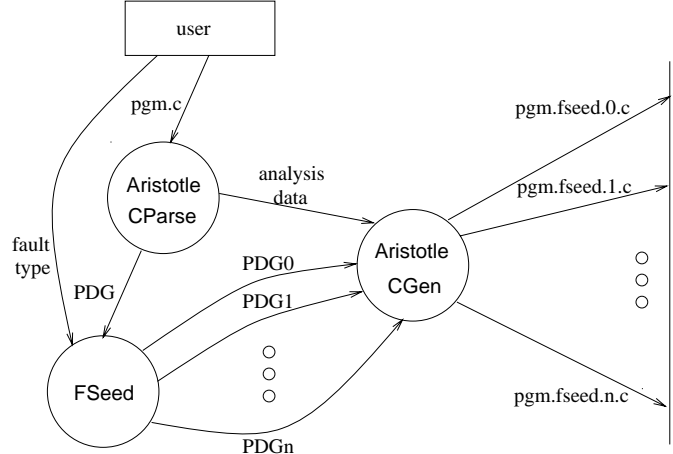


Figure 17: Dataflow diagram of our PDG-based fault-seeder system. Circles represent system modules, edges between the circles indicate the flow of data between modules, and the parallel lines represent external storage.

The goal of our prototype fault seeder is to produce multiple fault-seeded versions of `pgm.c` each with one fault of the fault type. Thus, each `case` in algorithm `FaultSeederDriver` corresponds to a structural fault in our classification scheme in Figure 4; each `case` label represents a concatenation of nodes in the classification tree. The user inputs the `fault_type` value, and `FaultSeederDriver` traverses the PDG to find as many places as possible to perform the fault transformation `fault_type`. Hence, in each case, `FaultSeederDriver` iterates over all the nodes in the PDG.

For each node in the PDG, `FaultSeederDriver` checks for certain conditions to determine the appropriate transformation position. For example, to seed the domain fault *Omission of Clause*, `FaultSeederDriver` finds all conditional predicate nodes, and calls the `OmitPath` routine to omit true and false regions. Because the fault categories for computation faults and assignment faults are the same, the only difference in the action taken by `FaultSeederDriver` is to set the `selection_type` variable to `C_USE` or `P_USE`. The driver performs some additional analysis to determine the uses (c-uses or p-uses⁷) of the definition in the node, to differentiate between computation and assignment faults. For example, if the input `fault_type` is `Computation.IncorrectVariable`, `FaultSeederDriver` sets the `selection_type` to `C_USE`, and the algorithm iterates over all nodes and selects a statement node with c-uses for transformation. The algorithm calls `IncorrectVarUse` and `IncorrectVarDef` to perform DDS-structural transformations. The output of `FSeed` consists of the transformed PDGs for `pgm.c`, which are shown in Figure 17 as `PDG0`, `PDG1`, ... `PDGn`.

The transformed PDGs are then input to `CGen`, which generates C programs `pgm.fseed.0.c` ... `pgm.fseed.n.c`. `CGen` consists of two main components: a CDS walk routine and a node-level code generator. The CDS walk routine traverses the CDS to generate a *walk sequence*, which is a sequence of nodes obtained using a graph traversal algorithm such that a correct program is generated from this sequence. Using the walk sequence, the node-level code generator produces code for each node in the CDS, based on the node type and the statements associated with the node. At this point, the fault-seeder tool becomes language dependent; `CGen` is written for C.

⁷A c-use occurs on the right-hand side of an assignment or in an output statement, whereas a p-use occurs in a predicate expression.

```

Algorithm FaultSeederDriver
input    fault_type, node_count, program_file
globals  cd_info, dd_info : CDS, DDS
output   fault-seeded programs of type specified
declare  Transformation algorithms from section 3.2
           node_id : node id being processed
           node_ptr : pointer to node
           GetNode(node_id) : returns pointer to node
           RightRegion(node_ptr), LeftRegion(node_ptr) : return left/right region node id for predicate
           GetUse(node_ptr) : returns list of c-use/p-uses
           CGen : source code generator
begin FaultSeederDriver
  switch(fault_type)
    case Domain.MissingPath.OmissionofClause :
      foreach(node_id ∈ 1..node_count)do node_ptr = GetNode(node_id)
        if(node_ptr.type==if-predicate)then
          OmitPath(RightRegion(node_ptr)) CGen
          OmitPath(LeftRegion(node_ptr)) CGen
        elseif(node_ptr.type==while/do/for predicate)then
          OmitPath(node_id) CGen
      return
    case Computation.IncorrectVariable: selection_type= C_USE
      foreach(node_id ∈ 1..node_count)do node_ptr = GetNode(node_id)
        if(node_ptr.type ==statement)and(selection_type ∈ GetUse(node_ptr))then
          IncorrectVarUse(node_id) CGen
          IncorrectVarDef(node_id) CGen
      return
    case Domain.PathSelection.Assignment.OmissionofStatements: selection_type= P_USE (omitted)
    case Computation.OmissionofStatements: selection_type= C_USE (omitted)
    .....
end FaultSeederDriver

```

Figure 18: Driver routine for FSeed; only two cases are detailed.

5 Fault Seeding Case Study

We performed a case study with our prototype fault seeder to demonstrate its operation. In the first part of our study, we compared the fault-detection abilities of dataflow testing and mutation testing; in particular, we compared the all-uses dataflow criterion to mutation testing. Several previous studies [31, 27] have compared the relative effectiveness of dataflow testing and mutation testing. However, none of these studies used an automatic fault-seeding tool to insert faults into the subject programs. In the second part of our case study, we evaluated the dataflow adequacy and mutation adequacy of the test suite created by our fault seeder.

5.1 Case Study Procedure

We applied our fault seeder to a set of subject C programs that are listed in Table 1. We then used *Combat* [20] to dataflow test each C program, and we used *Mothra* [9] to mutation test equivalent Fortran-77 versions of the subject programs. We now describe the steps involved in the case study.

Step 1

In the first step of the case study, we prepared the subject programs. Because *Mothra* required Fortran-77 programs as input and *Combat* required C programs, we first translated the subject programs so that both language versions would be available. We started with Fortran-77 versions of the programs, and first made

Program	PDG Nodes	PDG Edges	DU Pairs	Mutants	Infeasible DU Pairs	Equivalent Mutants
bisect	13	17	46	511	0	62
bub	15	19	37	338	1	35
cal	19	28	59	3010	0	266
euclid	4	4	13	196	1	24
find	28	38	148	1022	13	75
insert	16	21	44	437	1	46
mid	10	14	31	183	0	13
newton	9	12	33	394	0	48
quad	4	4	16	359	0	31
search	14	18	37	370	1	32
secant	9	12	30	694	0	84
tritype	19	45	112	951	14	109

Table 1: Details of our subject programs.

sure that the programs did not use any features of Fortran-77 that would not translate directly into C. Then, we hand translated the programs into C taking care to use as direct a translation as possible so as not to introduce any bias into our results because we used different programs. We attempted to preserve the control structure of the programs during translation; however, we had to introduce some differences in the programs due to the requirements of **Mothra** and **Combat**. For example, **Combat** requires input and output statements as part of the procedure being tested. Thus, for a program using arrays, the results was two extra loops to read and print the arrays, and hence extra faults. However, these faults in the input and output loops were trivial and easy to detect and therefore, do not affect the results of our study. We tested our translations by running both versions on a set of functional tests, and comparing the outputs of the two versions.

In this first step of the case study, we manually determined the equivalent-mutant and infeasible-path information for the subject programs. Although this process was time consuming, it was essential in the computation of the adequacy scores. For the subject programs, Table 1 shows the number of PDG nodes and edges, the number of definition-use (DU) pairs, the number of mutants, the number of infeasible definition-use (DU) pairs, and the number of equivalent mutants.

Step 2

We performed the fault seeding in the second step of our procedure. We ran our fault-seeder system on each subject C program, and created several fault-seeded versions of the program. We selected four structural faults from our classification (Figure 4) to be seeded: Omission of Clause, Omission of Conditional, Transposition Between Regions, and Transposition Within Region. The second column in Table 2, (labeled Total Faults), gives the total number of faults seeded. The next four columns give the numbers of faults of each type that were seeded.

Step 3

In Step 3, we hand translated each of the faulty C programs into its Fortran-77 counterpart. We used a procedure similar to that used in Step 1 to guarantee equivalence of the faulty Fortran-77 and C versions.

Program	Total Faults	<i>Omission of Clause</i>	<i>Omission of Conditional</i>	<i>Transposition Between Regions</i>	<i>Transposition Within Region</i>	Dataflow	Mutation
bisect	8	2	3	3	0	8	8
bub	8	0	4	2	2	8	8
cal	7	2	2	2	1	7	7
euclid	3	0	1	1	1	2	3
find	8	0	7	0	1	8	8
insert	5	0	3	1	1	5	5
mid	7	2	5	0	0	6	7
newton	5	1	2	1	1	5	5
quad	3	0	1	2	0	3	3
search	6	1	3	2	0	6	6
secant	4	1	2	0	1	4	4
tritype	10	1	9	0	0	10	10
TOTAL	74	10	42	14	8	72	74

Table 2: The faults detected by mutation and dataflow testing.

Step 4

In this step, we tested each set of faulty versions using **Combat** for dataflow testing and **Mothra** for mutation testing. Figure 19 shows the flow chart for the procedure we used. For each faulty version of a subject program, we generated tests to satisfy the adequacy criterion (dataflow or mutation). We generated the tests manually by considering the definition-use pairs to be satisfied or mutants to be killed; we made no attempt to minimize the test sets. If some test in the adequate test set caused a program failure, we isolated the test as a fault-detecting test, and declared the fault detectable by the adequate test set. This one-to-one correspondence between test, fault, and failure was a natural outcome of the fact that each program version has exactly one fault, and that fault must be failure causing. It is important to note that when we generated the tests, we used no information about the current fault in the program version being tested; this avoided a bias in the generation of tests. The ideal way to remove any such bias would be to fully automate the test generation procedure. For **Mothra**, a tool such as **Godzilla** generates a large number of tests to attempt to satisfy the mutation adequacy criterion; however, a number of tests must be hand picked. For **Combat** such a tool does not currently exist. Therefore, for this study, we manually generated all tests for both **Mothra** and **Combat**. The last two columns in Table 2 give the total number of faults detected by dataflow testing and mutation testing.

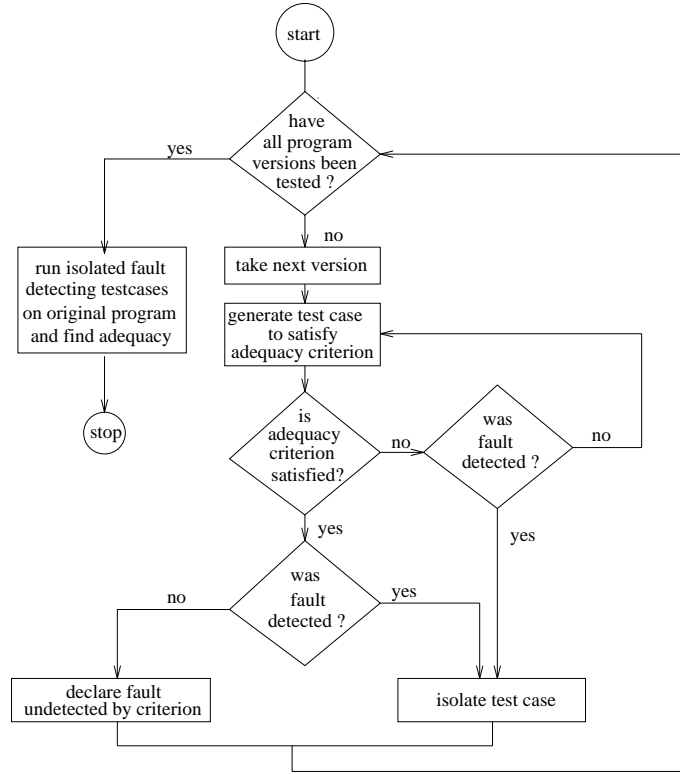


Figure 19: Procedure used for our case study.

Step 5

Having determined the number of faults found using mutation and dataflow testing, we wanted to determine how close the fault-detecting test sets came to satisfying the dataflow adequacy criteria and the mutation adequacy criterion. That is, we wanted to compute the dataflow adequacy and the mutation adequacy of the test sets that were, by construction, 100% effective. For this purpose, we tested the original programs using the isolated fault-detecting test sets, and determined mutation and dataflow adequacy scores. For additional information we noted the all-nodes and all-edges as well as the all-uses adequacy of the fault detecting test sets. Table 3 shows the results of this study for dataflow adequacy and Table 4 shows the results of the study for mutation adequacy. We computed the adequacy scores as $\frac{N_s}{N_t - N_u}$, where N_s is the number of nodes, edges or definition-use pair satisfied, or the number of mutants killed; N_t is the total number of nodes, edges, definition-use pairs or mutants, and N_u is the number of unsatisfiable entities: nodes, edges or definition-use pairs that lie on infeasible paths, or equivalent mutants.

The second through fourth columns of Table 3 show the adequacy scores of the fault detecting test sets that were obtained using three dataflow criteria. The last two columns compare test set sizes. The fifth column gives the test set size for an all-uses adequate test set and the sixth column gives the size of the structural fault-detecting test set obtained by fault seeding, for which the adequacy scores in this table were recorded. The dataflow adequate test sets were not minimized.

The second column of Table 4 gives the overall number of mutants generated and the third column gives the overall mutation adequacy scores. The fourth through ninth columns give the numbers and adequacy scores for each of the three mutation superclasses. The last two columns compare the test sets sizes for

Program	All-Nodes	All-Edges	All-Uses	Adeq Set	Fseed Set
bisect	92.31	82.35	93.48	6	8
bub	100.00	100.00	100.00	1	8
cal	100.00	80.00	90.56	8	7
euclid	100.00	100.00	100.00	4	3
find	100.00	97.36	97.30	1	8
insert	100.00	100.00	100.00	1	5
mid	80.00	64.28	74.93	6	7
newton	88.89	66.67	78.78	4	5
quad	100.00	100.00	100.00	2	3
search	92.85	88.89	89.19	2	6
secant	100.00	91.67	90.91	4	4
tritype	53.84	68.42	56.81	16	10

Table 3: The dataflow adequacy scores of fault detecting test sets.

Program	All	Score	cca	Score	pda	Score	sal	Score	Adeq Set	Fseed Set
bisect	511	89.4	293	95.4	195	75.9	23	100.0	24	8
bub	338	23.4	189	30.4	119	7.7	30	28.6	6	8
cal	3010	89.2	2687	89.4	272	86.1	51	91.8	47	7
euclid	196	97.7	98	96.8	72	98.1	26	100.0	3	3
find	1022	96.8	602	97.7	255	92.7	165	98.7	13	8
insert	437	91.6	244	94.2	140	85.5	53	92.5	3	5
mid	183	68.8	50	80.9	115	61.0	18	83.3	25	7
newton	394	80.3	205	86.7	175	71.7	14	71.4	23	5
quad	359	77.7	229	81.9	114	67.0	16	100.0	12	3
search	370	95.6	232	99.5	112	84.7	26	100.0	7	6
secant	694	77.1	427	84.0	251	64.2	16	73.3	25	4
tritype	951	72.7	474	69.6	429	74.5	48	91.1	44	10

Table 4: The mutation adequacy scores of fault detecting test sets.

mutation adequate test sets (tenth column), and fault-detecting test sets obtained by fault seeding, for which the adequacy scores in this table have been recorded (eleventh column). The mutation adequate test sets have not been minimized.

5.2 Analysis of Results

Table 2 shows that dataflow testing detected seventy-two of the seventy-four faults seeded, whereas mutation testing detected all faults seeded. The two faults that were not detected by dataflow testing were, somewhat surprisingly, trivial faults. Both were *Omission of Conditional* type of faults, and both resulted in the control flow graph of the program being reduced to a single basic block. In practice, such faults are not *persistent* in that they do not remain undetected for long. It appears from our results that both dataflow and mutation testing are nearly 100% effective in detecting most types of PDG structural faults. For mutation testing, our results indicate that the coupling effect, which was suggested and empirically investigated by mutation testing researchers[10, 29], may hold for structural faults as well. Thus, although mutation operators have

been designed to find simple syntactic faults, the coupling effect results in complex faults being linked to simple ones, and hence in their being discovered.

The results of finding the dataflow and mutation adequacy of the fault-detecting test sets reveal some interesting facts. The second through fourth columns in Table 3 show the all-nodes, all-edges, and all-uses adequacy of the fault-detecting test sets. For nine out of twelve subject programs, the fault-detecting test sets were more than 90% dataflow adequate. For four of these nine programs, the fault-detecting test sets were 100% dataflow adequate. The two subject programs for which the fault-detecting test sets were poorly dataflow adequate were `mid` and `tritype`. An inspection of the fault types seeded in the third through sixth columns of Table 2 reveals that the faults seeded for `mid` and `tritype` are not well distributed: there are no *Transposition Between Regions* or *Transposition Within Region* type of faults. Thus, there may be a relation between types of faults seeded and dataflow adequacy of test sets that expose these faults. The dataflow adequacy scores for `newton` show that the fault-detecting test set was only about 80% all-uses adequate, although Table 2 shows that `newton` had a good distribution of seeded faults. This anomaly might be due to the specific test sets that were selected for `newton`. Such anomalies can be resolved by increasing the number of subject programs and the size of the programs seeded with faults. Columns 5 and 6 in Table 3 show that the fault-detecting test sets do not provide an improvement in test set size, because the adequate test sets satisfying the all-uses dataflow criterion, are small.

The third column of Table 4 (Score) shows the overall mutation adequacy scores for the fault-detecting test sets. For half of the subject programs, the fault-detecting test sets at least 89% mutation adequate. These subject programs were `euclid`, `search`, `insert`, `bisect`, `cal`, and `find`. These programs cover a range of sizes and numbers of mutants; thus, the adequacy results for the fault-detecting test sets are important, because they show that we can construct *nearly* mutation adequate test sets that are also 100% structural fault-detecting, using a small number of tests. For comparison, Table 4 shows the sizes of the (100%) mutation adequate test sets (in column Adeg Set) and the sizes of the fault-detecting test sets (in column Fseed Set). Comparing these two test sets indicates that it may be possible to get a good mutation adequacy with small test sets. For program `find`, the fault detecting test set consisting of eight tests is nearly 100% mutation adequate, whereas the compared to a 100% mutation adequate test set with thirteen tests. More interesting, is program `cal` for which the mutation adequate test set is very large (forty-seven tests), yet the fault-detecting 90% mutation adequate test set is small (seven tests). The results for `bisect` are similar. This raises the issue of whether it is acceptable to have small (and hence cheap) nearly-mutation-adequate test sets, that are 100% effective in finding structural faults. We can also consider building small fault-detecting test sets and adding tests to kill certain classes of mutants, thus streamlining the process of selecting mutation adequate test sets.

For information on the mutation adequacy with respect to mutant superclasses, of the fault-detecting test sets, consider the fourth through ninth columns in Table 4. The distribution of mutants in the three mutant superclasses is shown here, along with the percentage of mutants of each class that are killed. It appears that our fault-detecting test sets target the `sal` type mutants, as in most cases, the fault-detecting test sets are more than 90% adequate for the `sal` mutant class. The `pda` class of mutants is least affected by the fault-detecting test sets. For the programs: `quad`, `secant` and `newton`, the fault-detecting test sets were only between 75% and 80% mutation adequate; the sixth column shows that the percentage of `pda` mutants in this class was high, which is logical because these programs consist primarily of numerical computation

statements. Fault-detecting test sets for the programs `tritype` and `mid` are poorly mutation adequate, which might be due to the fact that these programs had a very large number of branches and hence `pda` type mutants. The fault-detecting test set for `bub` had the least mutation adequacy: only 23%. Table 4 indicates that there is nothing exceptional about the mutant distribution for `bub`. The low mutation score might be due to the specific test set selected.

Despite this variation in dataflow and mutation adequacy scores, and the recognized need for extended experimentation, the results of the conducted experiments clearly show that it is feasible to use the structural fault seeder in conjunction with established testing methods, for many applications.

6 Related Work

Howden [22] categorized faults as domain and computation faults. In his classification, domain faults were further categorized as missing path or path selection faults, and path selection faults were categorized as predicate faults or assignment faults. Zeil [40] later detailed Howden’s categorization. We have extended the domain/computation fault categorization so that it can be easily translated to a well-defined program representation and hence, automated. Basili and Selby [4] classified faults according to two schemes. The first scheme separates faults of commission from faults of omission, and the second scheme partitions faults into six classes: initialization, computation, control, interface, data, and cosmetic. This scheme does not characterize faults according to a program representation that can be used to automate the process of fault classification and fault seeding. The authors state that the fault categorization for their experimental programs was based on a “consistent interpretation” of the faults and that it is possible for another analyst to categorize the faults in a given program differently.

DeMillo and Mathur [12] presented a fault categorization scheme based on their investigation of Knuth’s Tex error log. According to their categorization, faults are “simple” or “complex”. They defined simple faults as those that are modeled by mutation operators. Complex faults, under this classification, consist of the following types: missing code, incorrect placement, incorrect algorithm, and miscellaneous. Simple faults are, by definition, easy to model and reproduce (by mutation operators), but the modeling of complex faults was not dealt with in this work. Our fault categorization method provides a clear distinction between statement and structural faults, where statement faults are modeled as first-order or multi-order mutations, or are interface faults.

Mutation analysis has been applied to several languages, and several tools have been developed to implement mutation analysis systems. The `Mothra` system [9, 13], was developed for Fortran-77. `Mothra` uses a set of mutant operators, described by King and Offutt [25], that were based on earlier mutation systems [3, 6].

More recent efforts have focused on mutation systems for the C language. Reference [1] describes a set of mutant operators for C, which we henceforth call the Purdue C operators. To date, there are two publicized systems that use these operators. Delamaro et al. developed Proteum [8, 7], and Untch implemented TUMS [35, 36] to implement the schema-based execution model [36]. The `Pisces` system [37] also uses mutation to assess testability and to perform weak mutation [23]. The `Pisces` system is partially based on the Purdue operators, and it is, to date, the only commercial tool that is based on mutation.

Offutt et al. [32] defined a set of mutant operators for the Ada language. These operators are are loosely

based on the Purdue C operators, but differ in terms of the language features and the modifications made to the C operators to fix several problems and inconsistencies with the C operators⁸.

There is some overlap between our fault classification and the C operators. Our *missing path fault* category is similar to the Fortran-77 operator Statement Deletion (SDL), the C operator Statement deletion (SSDL), and the Ada operator Statement-Replace with NULL (SRN), except that ours removes the entire subpath, whereas the mutation operators delete one statement at a time. Our *incorrect variable used fault* is modeled by the various variable replacement operators in the mutation operators. Our *incorrect expression faults* are similar to the mutation operators that mutate expressions, except that ours can introduce more than one incorrect operator. Some of the *transposition between regions faults* will be modeled by the C operator Move brace up or down (SMVB), and the Ada operator END shift (SES), which moves end markers of compound statements up or down. However, most of the *transposition between regions faults* transformations will create more complicated faults. Finally, the *transposition within region fault* is not related to any operator for Fortran-77, C, or Ada.

Perhaps the biggest difference between our fault seeder and mutation systems is based on the semantic fault model summarized earlier. Offutt and Hayes [30] point out a key difference between fault seeding and mutation analysis. When fault seeding, we want faults that approximate natural faults as closely as possible; by exhibiting a distribution of semantic fault sizes that matches the distribution of natural faults. Mutation, however, may not want a similar distribution of semantic fault size. Using faults that have smaller semantic size may lead to stronger tests.

7 Conclusions and Future Work

In this project, we used the PDG as the basis for a fault-classification scheme, and applied this scheme in building an automated fault-seeding tool. Our classification categorized faults on the basis of transformations made on the CDS and the DDS to create the PDG for the faulty program. The two broad categories of faults identified are structural (macro-level) faults and statement (micro-level) faults. Structural faults can be DDS-structural faults or CDS-structural faults. We used our fault-seeder tool to perform structural fault transformations on the PDG to create faulty versions of a given program.

We conducted a case study using the fault-seeder system, and our results indicate the possibility of various applications. The primary application of such a fault seeder is to help investigate effectiveness of testing methods, and address issues related to cost and efficiency of testing methods. We can use the fault seeder to predict the fault distribution of software: instead of making actual fault transformations, we can output the number of faults of each type that are possible, and use the appropriate testing technique to test the software. Also, we could use the fault seeder to make mutation testing more cost-efficient in the number of tests. We can do this by constructing structural fault-detecting test sets that are almost mutation adequate, and adding test cases to kill certain classes of mutants. Finally, we can build a test-case generator that generates structural fault detecting tests, using information at the point of transformation in the PDG.

⁸Untch suggested some of these inconsistencies.

Acknowledgements

Gregg Roethermel and the anonymous reviewers made many helpful comments and suggestions for various versions of this paper.

References

- [1] H. Agrawal, R. DeMillo, R. Hathaway, Wm. Hsu, Wynne Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette IN, March 1989.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.
- [3] D. M. St. Andre. Pilot mutation system (PIMS) user's manual. Technical report GIT-ICS-79/04, Georgia Institute of Technology, April 1979.
- [4] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12), December 1987.
- [5] T. A. Budd. Mutation analysis: Ideas, examples, problems, and prospects in computer program testing. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129–148. North-Holland, 1981.
- [6] T. A. Budd, R. Hess, and F. G. Sayward. EXPER implementor's guide. Technical report, Department of Computer Science, Yale University, 1980.
- [7] M. E. Delamaro and J. C. Maldonado. Proteum: A mutation testing tool for C programs. Appeared in Purdue SERC newsletter, 1995.
- [8] M. E. Delamaro, J. C. Maldonado, M. Jino, and M. Chaim. Proteum: Uma ferramenta de teste baseada na análise de mutantes (Proteum: A testing tool based on mutation analysis). In *7th Brazilian Symposium on Software Engineering*, pages 31–33, Rio de Janeiro, Brazil, October 1993. In Portuguese.
- [9] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.
- [10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [11] R. A. DeMillo and A. P. Mathur. A grammar based fault classification scheme and its application to the classification of the errors of tex. Private communication, Software Engineering Research Center, Purdue University, West Lafayette IN.
- [12] R. A. DeMillo and A. P. Mathur. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. Technical report SERC-TR-92-P, Software Engineering Research Center, Purdue University, West Lafayette IN, March 1991.
- [13] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [14] S.N. Weiss E.J. Weyuker and Dick Hamlet. Comparison of program testing strategies. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, Victoria, British Columbia, October 1991. ACM SIGSOFT 91.
- [15] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [16] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, October 1988.
- [17] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, SE-19(3):202–213, March 1993.
- [18] M. R. Girgis and M. R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Proceedings of the Workshop on Software Testing*, pages 64–73. IEEE Computer Society Press, July 1986.
- [19] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

- [20] M. J. Harrold and P. Kolte. Combat: A compiler based data flow testing system. *Proceedings of the 10th Pacific Northwest Software Quality Conference*, pages 311–323, October 1992.
- [21] M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith. Aristotle: A system for development of program analysis based tools. *Proceedings of the 33rd Annual ACM Southeast Conference*, pages 110–119, March 1995.
- [22] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.
- [23] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.
- [24] *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12.1990*. Institute of Electrical and Electronics Engineering, New York, 1990.
- [25] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software–Practice and Experience*, 21(7):685–718, July 1991.
- [26] W. A. Landi and B.G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.
- [27] 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.
- [28] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.
- [29] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.
- [30] A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *Proceedings of the 1996 International Symposium on Software Testing, and Analysis*, pages 195–200, San Diego, CA, January 1996. ACM Press.
- [31] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software–Practice and Experience*, 26(2):165–176, February 1996.
- [32] A. J. Offutt, Jeff Payne, and Jeffrey M. Voas. Mutation operators for Ada. Technical report ISSE-TR-96-06, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, March 1996.
- [33] I. Sommerville. *Software Engineering, third edition*. Addison-Wesley Publishing Company, New York NY, 1989.
- [34] K. Tewary and M. J. Harrold. Fault modeling using the program dependence graph. *IEEE International Symposium on Software Reliability '94 (ISSRE'94)*, pages 126–135, November 1994.
- [35] R. Untch. *Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method*. PhD thesis, Clemson University, Clemson SC, 1995. Clemson Department of Computer Science Technical report 95-115.
- [36] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 139–148, Cambridge MA, June 1993.
- [37] J. M. Voas, L. Morell, and K. W. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2), March 1991.
- [38] K. S. How Tai Wah. Fault coupling in finite bijective functions. *The Journal of Software Testing, Verification, and Reliability*, 5(1):3–47, March 1995.
- [39] W. E. Wong and A. P. Mathur. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, 4(1):69–83, March 1995.
- [40] S. J. Zeil. Perturbation techniques for detecting domain errors. *IEEE Transactions on Software Engineering*, 15:737–746, June 1989.