Subsumption of Condition Coverage Techniques by Mutation Testing

Technical Report ISSE-TR-96-01

A. Jefferson Offutt *
ISSE Department
George Mason University
Fairfax, VA 22030
phone: 703-993-1654
fax: 703-993-1638
email: ofut@isse.gmu.edu

Jeffrey M. Voas
Reliable Software Technologies Corp.
Suite 250
11150 Sunset Hills Road
PO Box 2393
Reston, VA 22090
phone: 703-742-8873
fax: 703-742-9836
email: jmvoas@isse.gmu.edu

January 1996

— DRAFT —

Keywords—branch testing, condition coverage, software testing, mutation testing, subsumption.

Abstract

Condition coverage testing is a family of testing techniques that are based on the logical flow of control through a program. The condition coverage techniques include a variety of requirements, including that each statement in the program is executed and that each branch is executed. Mutation testing is a fault-based testing technique that is widely considered to be very powerful, and that imposes requirements on testing that include, and go beyond, many other techniques. In this paper, we consider the six common condition coverage techniques, and formally show that these techniques are subsumed by mutation testing, in the sense that if mutation testing is satisfied, then the condition coverage techniques are also satisfied. The fact that condition coverage techniques are subsumed by mutation has immediate practical significance because the extensive research that has already been done for mutation can be used to support condition coverage techniques, including automated tools for performing mutation testing and generating test cases.

1 INTRODUCTION

A *testing criterion* is a rule or set of rules that impose requirements on a set of test cases used to test a program. *Path-testing* criteria state requirements in terms of the execution paths in the program. Test engineers measure the extent to which we have satisfied a criterion in terms of *coverage*; a test set achieves 100% coverage if it completely satisfies the criterion. Myers [Mye79] gives a variety of coverage criteria that are based on conditional expressions within the program. These criteria have been widely used, and are currently required by the FAA for certification of flight-critical software [SC-92].

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

Myers defined six condition coverage criteria. We briefly introduce them here, and define them more formally in Section 2. A *condition* in a program is a pair of algebraic expressions related by one of the relational operators $\{>, <, =, \geq, \leq, \neq\}$. Conditions evaluate to one of the binary values TRUE or FALSE and can be modified by the negation operator NOT. A *decision* is a list of one or more conditions connected by the two logical operators AND ($\wedge$) and OR ($\vee$) and used in a statement that affects the flow of control of the program. Decisions represent branches in the control flow of the program.

Statement Coverage (SC) requires that every statement in the program be executed at least once. *Decision Coverage (DC)* requires that every decision evaluate to both TRUE and FALSE at least once. DC is also known as *branch testing* and *all-edges* [Whi87]. *Condition Coverage (CC)* requires that each condition in each decision evaluate to both TRUE and FALSE at least once. *Decision / Condition Coverage (DCC)* requires that each condition in each decision evaluate to both TRUE and FALSE at least once, and that every decision evaluate to both TRUE and FALSE at least once. DCC combines DC and CC. *Modified Condition / Decision Coverage (MC/DC)* requires that every decision and every condition within the decision has taken every outcome at least once, and every condition has been shown to independently affect its decision. *Multiple-Condition Coverage (MCC)* requires that all possible combinations of condition outcomes in each decision be covered, that is, the entire truth table for the decision has been satisfied. MCC is also known as *extended branch coverage* [Whi87].

In this paper, we show that mutation testing subsumes each of the condition coverage criteria. We also explore the possibility of automating multiple-condition coverage.

1.1 Mutation Testing

Fault-based testing techniques guide the tester to develop test cases that detect a well-defined class of faults. Mutation testing is a fault-based testing technique introduced by DeMillo et al. [DLS78] and Hamlet [Ham77]. Mutation testing is based on the assumption that a program will be well tested if all so called "simple faults" are detected and removed. The coupling effect [DLS78, Off92] states that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults.

Simple faults, called *mutations*, are introduced into the program by *mutation operators*. Each mutation produced by a mutation operator produces a *mutant program*. A mutant is *killed* by a test case that causes the mutant program to produce incorrect output. A test case that kills a mutant is considered to be *effective* at finding faults in the program, and the mutant(s) it kills are not executed against later test cases. *Equivalent mutants* are mutant programs that are functionally equivalent to the original program and therefore cannot be killed by any test case. Determination of equivalent mutants is usually done by hand. The goal of mutation is to find test cases that kill all non-equivalent mutants; a test set that does so is said to be *adequate relative* to mutation. The *mutation score* is the ratio of the number of dead mutants to the number of non-equivalent mutants; it measures the adequacy of the test case set.

The Mothra mutation system [DGK+88, DO91] is the latest mutation testing system. It automates the process of mutation testing by creating and executing mutants, managing test cases, and computing

Type	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement end replacement
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis (replacement by TRAP)
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Table 1: Mothra Mutation Operators.

the mutation score. Mutation operators are represented as a set of rules that describe syntactic changes to elements of the program. Mothra uses twenty-two *mutation operators* [KO91].

The complete set of mutation operators used by the Mothra mutation system is shown in Table 1. These were derived from studies of programmer errors and correspond to mistakes that programmers typically make and enforce common testing heuristics (such as execute every statement). This particular set of mutation operators represents more than ten years of refinement through several mutation systems. The operators in this set not only require that the test data meet statement and branch coverage criteria, extremal values criteria, and domain perturbation, but also directly model many types of errors. Each of the 22 mutation operators is represented by the three-letter acronym given on the left. For example, the "*a*rray reference for *a*rray reference *r*eplacement" (AAR) mutation operator causes each array reference in a program to be replaced by each other distinct array reference in the program.

1.2 Overview

In the remainder of this paper, we give formal proofs that mutation subsumes each of the condition coverage criteria defined in Myers [Mye79]. We then give data from using the Mothra mutation testing tool to achieve Multiple-Condition Coverage, and give example mutants for a small function. Finally, we give a related proof that mutation subsumes one of the data flow testing criteria, All Definitions data flow, and present conclusions to the paper.

2 SUBSUMPTION OF CONDITION COVERAGE BY MUTATION

Subsumption has been widely used as a way to analytically compare testing techniques. We follow Weiss [Wei89] and Frankl and Weyuker [FW88] for our definition of subsumption. A criterion C_1 *subsumes* another criterion C_2 iff for every program, any test set T that satisfies C_1 also satisfies C_2 [FW88] [1].

We show that mutation subsumes the condition coverage criteria by showing that specific mutation operators impose requirements that are identical to a specific coverage criterion. For each specific requirement of a criterion (e.g., each decision), there is a single mutant that can only be killed by test cases that satisfy the requirement. Therefore, the coverage criterion is satisfied iff the mutants associated with the requirements for the criterion are killed. We say that the mutation operators that ensure coverage of a criteria *cover* the criteria. If a criteria is covered by one or more mutation operators, then mutation testing subsumes the criteria. We begin to show in Section 2.2 exactly how mutation operators can ensure various structural coverages.

There is one problem with relating mutation testing to the condition coverage criteria. The condition coverage criteria impose only a local requirement; for example, decision coverage requires that each branch in the program be executed. Mutation, on the other hand, imposes global requirements in addition to local requirements. That is, mutation also requires that the mutant program produce incorrect output. For DC, there are specific mutants that can only be killed if each branch is executed and the final output of the mutant is incorrect. Although this means that mutation imposes stronger requirements than do the condition coverage criteria, this also means that in some cases a test set that satisfies a coverage criteria will not kill all the associated mutants. Thus, mutation as defined earlier will not strictly subsume the condition coverage criteria.

We solve this problem by basing our subsumptions on *weak mutation*. Weak mutation was originally suggested by Howden [How82] and has been experimentally verified [OL94]. Weak mutation is exactly the same as mutation (called *strong* mutation) except that only the local requirements are imposed. That is, instead of the final output of the program being incorrect, only an intermediate state of the mutant program needs to be incorrect. Thus, we actually show that the coverage criteria are subsumed by weak mutation, not strong mutation. In practice, there is seldom any real difference [OL94].

Although our proofs refer to only two conditions per predicate, we can easily show that these arguments hold in general, because we have a one-to-one mapping between the mutants and the individual criteria. For instance, suppose we have: **if a and b then**, and we wish to satisfy branch coverage. Since we are required to take both branches, we create two mutants:

if (a and b == FALSE) then mutant1 = killed

and

if (a and b == TRUE) then mutant2 = killed.

[1] Frankl and Weyuker actually used the term *includes*. The term subsumption was defined by Clarke et al.: A criterion C_1 *subsumes* a criterion C_2 iff every set of execution paths P that satisfies C_1 also satisfies C_2 [CPRZ85]. The term subsumption is currently the more widely used and the two definitions are equivalent; we follow Weiss's suggestion to use the term *subsumes* to refer Frankl and Weyuker's definition.

When both of these mutants are killed, branch coverage at this statement is satisfied. Further, suppose that we have a decision of the form: **if a and b and c then**. Since there is an implicit precedence in this example, either **(a and b) and c** or **a and (b and c)**, then we can make the four condition-adequate mutants:

if (a and b == FALSE) then mutant1 = killed,
if (a and b == TRUE) then mutant2 = killed,
if (c == FALSE) then mutant3 = killed, and
if (c == TRUE) then mutant4 = killed

for our first precedence example, and:

if (b and c == FALSE) then mutant1 = killed,
if (b and c == TRUE) then mutant2 = killed,
if (a == FALSE) then mutant3 = killed, and
if (a == TRUE) then mutant4 = killed

for the second precedence. Thus our mutants can guarantee that each condition has each outcome exercised, regardless of the complexity of the decision.

2.1 Control Flow Graph

We formally define the coverage criteria in terms of the control flow graph of a program. A program unit P is considered to be an individual subprogram (main program, procedure, or function). A subprogram is decomposed into a set of *basic blocks (BB)*, each of which is a maximal sequence of simple statements with one entry point such that if the first statement is executed, all statements in the block will be executed. The subprogram is represented by a *control flow graph (CFG)*, in which the nodes are basic blocks and the edges correspond to possible flow of control between the basic blocks. For simplicity, we will assume that each node has at most two outgoing edges; that is, all decisions are binary-valued. For language constructs that have multi-way branches (such as Fortran's arithmetic-if), we will assume that the CFG is built by splitting the multi-way branches into several nodes with two outgoing edges.

As an example, consider the following subprogram:

```
Function CFG_Example (A : Integer, B : Integer) : Integer
O1, O2 : Integer;
BEGIN
   IF (A > 0) THEN
      O1 = B*B;
   ELSE
      O1 = B+B;
      WHILE (B < A) LOOP
         O1 = O1+1;
         B  = B+B;
      END LOOP
```

```
      O2 = O1 - A;
   END IF
   O3 = O2 + O1*2;
   PRINT (O3);
END CFG_Example
```

The corresponding CFG for this subprogram is shown in Figure 1. Although they are not strictly part of the CFG, we annotate the edges with the appropriate program predicates and computations. Such information is often desirable and included with the graph.

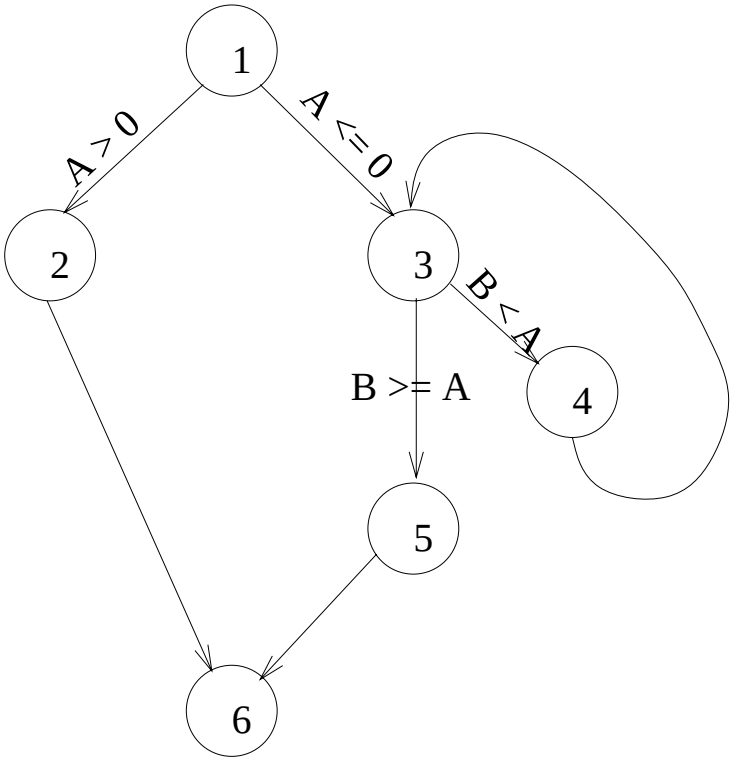

Figure 1: **CFG for program CFG_Example.**

2.2 Statement Coverage

The statement coverage criterion (SC) requires that each node in the CFG be executed at least once. Note that by executing a node, we are guaranteed to execute each statement in the corresponding basic block. The *s*tatement *an*alysis mutation operator (SAN) replaces the entry statement of each basic block in the program by a special TRAP statement. The semantics of the TRAP statement is that when it is executed, it always signals a failure, which immediately kills the mutant. To kill all SAN mutants, we are required to find test cases that reach each basic block. Since this is exactly the requirement of SC, the SAN operator covers SC.

2.3 Decision Coverage

The decision coverage criterion (DC) requires that each edge in the CFG be executed at least once. The *l*ogical *c*onnector *r*eplacement mutation operator (LCR), among other modifications, replaces each decision in a program by TRUE and FALSE. To kill the TRUE mutant, a test case must take the FALSE branch, and to kill the FALSE mutant, a test case must take the TRUE branch. Thus, the LCR operator forces us to execute each branch in the CFG, and it covers DC.

2.4 Condition Coverage

For decisions that have more than one condition, decision coverage can be satisfied without fully testing each condition. For example, if we have the decision `if` $(A \wedge B)$, decision coverage can be satisfied with the two test cases: $(A = T, B = T)$ and $(A = T, B = F)$, neither of which ever cause A to be FALSE. To make up for this, condition coverage requires that each condition in each decision evaluate to TRUE and FALSE at least once.

The *r*elational *o*perator *r*eplacement mutation operator (ROR), among other modifications, replaces each condition in each decision by TRUE and FALSE. To show that ROR covers CC, we must look at the two cases where the logical operator is AND and OR. By the argument above, if CC is covered with 2 conditions, it is also covered for arbitrary numbers of conditions.

1. `AND` Case.

   We will assume the decision is of the form: `if` $(A \wedge B)$, where A and B are boolean conditions. Below is the truth table for $(A \wedge B)$, and two ROR mutations. Since $TRUE \wedge B$ has the same result as the original decision for three of the four truth assignments, we must have the truth assignment (F T) to kill that mutant. Likewise, the mutant $A \wedge TRUE$ requires the truth assignment (T F). Thus, these two mutants are killed if and only if CC is satisfied, and ROR covers CC for the AND case.

	T T	T F	F T	F F
$A \wedge B$	T	F	F	F
$TRUE \wedge B$	T	F	**T**	F
$A \wedge TRUE$	T	**T**	F	F

2. `OR` Case.

   We will assume the decision is of the form: `if` $(A \vee B)$, where A and B are boolean conditions. Below is the truth table for $(A \vee B)$, and two ROR mutations. Since $FALSE \vee B$ has the same result as the original decision for three of the four truth assignments, we must have the truth assignment (T F) to kill that mutant. Likewise, the mutant $A \vee FALSE$ requires the truth assignment (F T). Thus, these two mutants are killed if and only if CC is satisfied, and ROR covers CC for the OR case.

	T T	T F	F T	F F
$A \vee B$	T	T	T	F
$FALSE \vee B$	T	**F**	T	F
$A \vee FALSE$	T	T	**F**	F

$$\text{if } A \wedge B \text{ then}_{MC/DC} = \begin{cases} \textit{show that two decision outcomes occur:} \\ \textbf{if } A \wedge B = \textbf{TRUE} \\ \textbf{if } A \wedge B = \textbf{FALSE} \\ \textit{show that two condition outcomes for condition a occur:} \\ \textbf{if } A = TRUE \\ \textbf{if } A = FALSE \\ \textit{show that two condition outcomes for condition b occur:} \\ \textbf{if } B = TRUE \\ \textbf{if } B = FALSE \\ \textit{show that two conditions independently affect decision outcome:} \\ \textbf{if } A \wedge TRUE = \textbf{if } A \wedge B \textbf{ then} \\ \textbf{if } B \wedge TRUE = \textbf{if } A \wedge B \textbf{ then} \end{cases}$$

$$\text{if } A \vee B \text{ then}_{MC/DC} = \begin{cases} \textbf{if } A \vee B = TRUE \\ \textbf{if } A \vee B = FALSE \\ \textbf{if } A = TRUE \\ \textbf{if } A = FALSE \\ \textbf{if } B = TRUE \\ \textbf{if } B = FALSE \\ \textbf{if } A \vee FALSE = \textbf{if } A \vee B \textbf{ then} \\ \textbf{if } B \vee FALSE = \textbf{if } A \vee B \textbf{ then} \end{cases}$$

Figure 2: Modified Condition/Decision Adequate Mutants for `OR` and `AND`.

2.5 Decision / Condition Coverage

Although CC forces conditions to take on all possible values, it says nothing about the decisions. Thus, CC can be satisfied without executing all edges; for example, in the OR case above, the decision evaluates to TRUE in both cases. Decision / condition coverage combines CC and DC and requires that each condition in each decision be evaluated to TRUE and FALSE at least once, and each edge in the CFG be taken at least once.

Since DCC is simply a combination of DC and CC, and DC is covered by the LCR operator, and CC is covered by the ROR operator, DCC is covered by LCR and ROR.

2.6 Modified Condition / Decision Coverage

MC/DC is defined in FAA standard DO178-B [SC-92] such that:

> "Every point of entry and exit in the program has been invoked at least once, and every condition in a decision in the program has taken on all possible outcomes at least once, every decision in the program has taken on all possible outcomes at least once, and each condition in a decision has been shown to *independently* affect that decision's outcome. (A condition is shown to independently affect a decision's outcome by varying just that condition while holding all other conditions possible fixed.)"

To show that each condition independently affects a decision's outcome, we need to assure that if a compiler short-circuits while evaluating a decision, the condition that we are concerned with receives evaluation first.[2] The mutation rules for doing this are shown in Figure 2. Here we are showing the rules that provide MC/DC for the decisions $A \wedge B$ and $A \vee B$. Note that we ensure both outcomes of the decision, that both internal conditions are exercised for both $TRUE$ and $FALSE$. And finally we show the mutants that provide that each condition independently affect the outcome of the decision for these two examples.

2.7 Multiple-Condition Coverage

MCC recognizes that for a decision with N conditions, there are 2^N possible combinations of values for the conditions. These combinations correspond to the full truth table of the decision. Whereas DC, CC, DCC, and MC/DC choose subsets of these combinations, MCC requires that all combinations of decision truth assignments be exercised.

Again, we use the ROR operator and the LCR operator to cover this criterion. To show that these operators cover MCC, we must look at the two cases where the logical operator is AND and OR.

1. `AND` Case.

   If the original predicate is $A \wedge B$, then the mutants required to ensure MCC are in the following table. The entry above the line, $A \wedge B$, gives the truth values of the original predicate for each of the possible values of A and B. Each of the following rows represent a mutant of the predicate, and give the resulting truth values of the mutated predicate. The cells where the mutant's value differs from the original predicate indicate truth assignments for A and B that would kill that mutant.

   Since the mutant $TRUE \wedge B$ has the same result as the original decision for three of the four truth assignments, we must have the truth assignment (F T) to kill it. Likewise, $FALSE \wedge B$ requires the truth assignment (T T), $A \wedge TRUE$ requires (T F), and $A = B$ requires (F F). Thus, these four mutants are killed if and only if MCC is satisfied, and ROR and LCR covers MCC for the AND case.

		T T	T F	F T	F F
	$A \wedge B$	T	F	F	F
1	$TRUE \wedge B$	T	F	**T**	F
2	$FALSE \wedge B$	**F**	F	F	F
3	$A \wedge TRUE$	T	**T**	F	F
4	$A = B$	T	F	F	**T**

   $A \neq B$ is actually `NEQV`, which means "not equivalent", or exclusive `OR`.

2. `OR` Case.

   If the original predicate is $A \vee B$, then the mutants required to ensure MCC are in the following truth table. The entry above the line, $A \vee B$, gives the resulting truth values of the original predicate for each of the possible values of A and B. Each of the following rows represent a mutant of the predicate,

[2] A compiler short-circuits a decision evaluation by halting the evaluation after enough information has been evaluated to determine the value of the entire decision.

and give the resulting truth values of the mutated predicate. The cells where the mutant's value differs from the original predicate indicate truth assignments for A and B that would kill that mutant.

Since the mutant $TRUE \vee B$ has the same result as the original decision for three of the four truth assignments, we must have the truth assignment (F F) to kill it. Likewise, $FALSE \vee B$ requires the truth assignment (T F), $A \vee FALSE$ requires (F T), and $A \neq B$ requires (T T). Thus, these four mutants are killed if and only if MCC is satisfied, and ROR and LCR covers MCC for the OR case.

		T T	T F	F T	F F
	$A \vee B$	T	T	T	F
1	$TRUE \vee B$	T	T	T	**T**
2	$FALSE \vee B$	T	**F**	T	F
3	$A \vee FALSE$	T	T	**F**	F
4	$A \neq B$	**F**	T	T	F

3 Mothra and Multiple-Condition Coverage

We used the Mothra mutation testing system [DGK+88] to investigate the effectiveness of using mutation to automatically achieve multiple-condition coverage for sixteen program units. Mothra includes an automated test data generator, Godzilla [DO91, DO93], that attempts to generate test cases to kill all mutants. We used Mothra to create the mutants that were necessary to cover multiple-condition coverage, and then Godzilla to generate test cases to attempt to kill those mutants. We used the weak mutation version of Mothra [OL94] to execute the Godzilla-generated test cases against the mutants. The results of this procedure is shown in Table 2.

Program	Executable Statements	Mutants	Live	Dead	Test Cases	Percent Killed
Banker	39	14	7	7	3	50
Bisect	21	10	4	6	3	60
BinSearch	20	7	2	5	2	71
Bubble	11	3	0	3	1	100
Cal	29	21	6	15	5	71
DeadLk	52	25	7	18	5	72
EBC	12	8	0	8	4	100
Euclid	11	2	0	2	1	100
Find	28	14	0	14	5	100
Insert	14	6	0	6	1	100
Mid	16	10	0	10	7	100
Newton	14	8	3	5	3	62
Pat	17	7	1	6	3	86
Quad	10	2	0	2	2	100
TriSmall	13	49	0	43	6	88
TriTyp	28	56	13	43	10	77
Warshall	11	5	0	5	1	100
Total	346	247	43	198	62	80

Table 2: Multiple-Condition Coverage by Godzilla.

The `Mutants` column gives the number of mutants required to ensure MCC for each program, `Live` and `Dead` indicates how many were unkilled and killed by Godzilla, and `Test Cases` gives the number of test cases that Godzilla generated. `Percent Killed` gives how many mutants were killed – these numbers are also the percent of MCC combinations that were covered. The fact that we were able use Godzilla to cover four out of five MCC combinations indicates that automation of MCC testing is feasible.

4 Example SAN, LCR, and ROR Mutants

This section contains an example program, with the LCR and ROR mutants necessary to ensure MCC. The program is a small example with two decisions, each containing two conditions:

```
Function EBC (A : Integer, B : Integer) : Integer
Rslt : Integer
   if (A > 0 || B > 0) then
      Rslt = 1;
   else
      Rslt = 0;
   endif
   if (A > 0 && B > 0) then
      Rslt = Rslt + 2;
   else
      Rslt = Rslt + 4;
   endif
   return (Rslt);
end EBC;
```

Next we show the relevant mutants embedded into the program. Each line that begins with a "#" represents a mutant program, where the preceding statement is replaced by the mutated statement. Thus, eight mutants are represented. Each mutant is annotated with its mutation operator type, and a unique number for referencing the mutant.

```
             Function EBC (A : Integer, B : Integer) : Integer
             Rslt : Integer
             if (A > 0 || B > 0) then
# ror     20    if (FALSE    || (B > 0)) then
# ror     21    if (TRUE     || (B > 0)) then
# ror     27    if ((A > 0)  || FALSE) then
# lcr      3    if ((A > 0)  != (B > 0)) then
                Rslt = 1;
             else
                Rslt = 0;
             endif
             if (A > 0 && B > 0) then
# ror     34    if (FALSE    && (B > 0)) then
# ror     35    if (TRUE     && (B > 0)) then
# ror     42    if ((A > 0)  && TRUE) then
# lcr      9    if ((A > 0)  == (B > 0)) then
```

```
   Rslt = Rslt + 2;
else
   Rslt = Rslt + 4;
endif
return (Rslt);
end EBC;
```

5 All Definitions Data Flow

Data flow testing is a family of testing criteria due to Rapps, Frankl and Weyuker [FW88]. Data flow testing assumes that to adequately test a program, we need to test combinations of definitions of data and uses of data.

A *data definition* of a variable is a location where a value is stored into memory (assignment, input, etc.), and a *data use* is a location where the value of the variable is accessed. Uses are subdivided into 2 types: a *computation use* (*c-use*) directly affects a computation or is an output, and a *predicate use* (*p-use*) directly affects the flow of control. c-uses are considered to be on the nodes in the CFG and p-uses are on the edges. Thus, in the CFG in Figure 1, there are p-uses of `A` on edges 12 and 13, and c-uses of `O1` and `B` on node 2. A *definition-clear subpath* for a variable X through the *CFG* is a sequence of nodes that does not contain a definition of X.

The All-defs data flow criterion requires that each definition of a variable reach at least one use. That is, for each definition of a variable X on node n, there must be a definition-clear subpath for X from n to a node or an edge with a use of n.

Although mutation testing does not explicitly have this requirement, the requirement is met implicitly through the *sdl* mutation operator. For each statement in the program, the *sdl* operator (statement deletion) creates a mutant by deleting the statement. To show subsumption of All-defs, we restrict our attention to only statements that contain variable definitions. Assume that M_1 is an *sdl* mutation that deletes statement S_i containing a definition of a variable X. To kill M_1 under strong mutation, a test case t must 1) cause the mutated statement to be reached (reachability), 2) cause the execution state of the program after execution of S_i to be incorrect (necessity), and 3) cause the final output of the program to be incorrect (sufficiency) [DO91]. Any test case that reaches W_i will cause an incorrect execution state, because the mutated version of S_i will not assign a value to X. For the final output of the mutant to be incorrect, there are two cases. First, if X is an output variable, t must have caused an execution of a subpath from the deleted definition of X to the output without an intervening definition. Since the output is considered a use, this satisfies the criterion. Second, if X is not an output variable, then the nondefinition of X at S_i must result in an incorrect output state. This is only possible if X is used at some later point during execution without being redefined. Thus, t will satisfy the All-defs criterion for the definition of X at S_i, and the *sdl* mutation operator ensures that mutation subsumes All-defs.

6 Conclusions

This paper has shown how weak mutation testing can be used to generate coverage adequate test suites. To do so, we have restricted the mutants to only those that guarantee that certain conditions are exercised. Since we are creating restricted mutants in this scheme, we are able to ignore problems such as semantically equivalent mutants. In our scheme, there is a mutant for each requirement that we are attempting to satisfy. We believe that our scheme can be implemented efficiently, and that it represents a cost effective of performing coverage (unit) testing. The data in section 3 supports this view.

It is interesting to note that recent experimentation [OLR+96] has indicated that only five mutation operators may be needed for effective mutation testing, including ROR and LCR. This leads us to hypothesize that MCC is in some sense a major part of mutation.

We admit that there is a down-side to automatic test case generation for structural coverage: since a machine is generating test cases versus a person manually inspecting the code to create the test cases, the possibility of a person finding faults via manual inspection no longer exists. However, the cost savings of such a scheme can still be applied to inspections or other techniques that are aimed towards fault detection and fault removal.

In this paper, we have focused our examples towards Fortran and the mutant operators of Mothra. However for practical considerations, most avionics software that must be certified according to DO178-B is not coded in Fortran, but rather C, Ada or PLM (similar to Pascal). This then includes additional constructs that alter the flow of control such as CASE statements and switches. And for non-avionics applications, we acknowledge that issues of side-effects in conditional expressions must be handled by a tool that implements this scheme.

References

[CPRZ85] 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.

[DGK+88] 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.

[DLS78] 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.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[DO93] R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, April 1993.

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

[Ham77] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[How82] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[KO91] 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.

[Mye79] G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York NY, 1979.

[Off92] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

[OL94] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[OLR+96] A. J. Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 1996. To appear.

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

[Wei89] S. N. Weiss. What to compare when comparing test data adequacy criteria. *ACM SIGSOFT Notes*, 14(6):42–49, October 1989.

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