Using Weak Mutation Coverage with GCT

Brian Marick
Testing Foundations

Documentation for version 1.4 of GCT
Document version 1.2

The *Generic Coverage Tool (GCT) User's Manual* describes the most common uses of GCT. This manual explains how to use GCT to measure weak mutation coverage, a "researchy" coverage measure.

Preface

GCT is free software; you are encouraged to redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 1, or (at your option) any later version.

GCT is sometimes bundled with Expansion Kits, also distributed in source form but licensed separately. If you redistribute GCT, please take care to redistribute *only* GCT.

GCT and its Expansion Kits are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. For more details, refer to the GNU General Public License (the file COPYING in the distribution) or to the separate license you received with an Expansion Kit.

For more information about GCT, other products, or other services, contact:

Brian Marick
Testing Foundations
809 Balboa
Champaign, Illinois 61820

(217) 351-7228
Email: marick@cs.uiuc.edu, testing!marick@uunet.uu.net

You can join the GCT mailing list by sending mail to gct-request@cs.uiuc.edu.

This document is Copyright © 1992 by Brian Marick. Portions are Copyright © 1991 by Motorola, Inc. Under an agreement between Motorola, Inc., and Brian Marick, Brian Marick is granted rights to these portions.

Brian Marick hereby permits you to reproduce this document verbatim for personal use. You may not reproduce it for profit.

Introduction

1. Operator Mutation Coverage

You can consider weak mutation coverage a generalization of relational operator coverage (though, in fact, weak mutation coverage came first). Relational coverage hypothesizes that the program contains a misuse of a relational operator and requires tests that detect that misuse. The same can be done with misuses of other kinds of operators, in which case we call the approach *operator mutation coverage*.

For example, it may be that a line of code is

```
if (A && B)
```

but should be

```
if (A || B)
```

The coverage condition that distinguishes the correct program from the buggy program is (A&&B)!=(A||B). One of the arguments must be true, the other false. Notice that multi-condition coverage will require A true and B false (because of C's short-circuit evaluation of &&). In this case, multi-condition coverage also satisfies weak mutation coverage. That's not true of all operators, though. Another potential fault is using

```
A + B
```

instead of

```
A - B
```

This is ruled out when A+B != A-B, or, simplifying, B!=0.

Such measures have been described in [Howden82], [DeMillo78], [Hamlet77], and many later papers.

2. Operand Mutation Coverage

Another common fault is to use the wrong variable or constant. For example, consider the assignment statement A=C, which should have been A=B. The fault will never cause a failure if B always equals C, so we must write at least one test in which the two variables have different values.

These measures are due to [Howden82] and [DeMillo78]. For commentary on the cost of weak mutation coverage, see [Marick91a].

3. The Rest of This Document

The next chapter describes control file options important to users of weak mutation. The chapter after that describes in detail what weak mutation does and what the reporting tools say about it. The final chapter will help you when troubleshooting.

The Control File

4. Turning on Weak Mutation

The two types of weak mutation coverage may be turned on separately:

(coverage operator operand)
(coverage operator)
(coverage operand)

These two types of coverage are not compatible with race, routine, or call coverage. You may use weak mutation coverage together with any other type of coverage. To get as much coverage as you can, use

(coverage branch loop multi operator operand)

You do not need to call for relational coverage, since it's a subset of operator coverage.

5. Options

The following options may be set (or turned off) in the control file. They won't make sense until you've read the following chapter.

macros Default: off

Under operand coverage, macros that expand to a constant, variable name, array reference, structure dereference, or pointer dereference are partially instrumented. The top level construct is instrumented, but not any components. That is, given

```
#define ACCESS(ARRAY, OFFSET, VAL)  ARRAY[OFFSET+abs(myfun(VAL))]

ACCESS(a, o, v)
```

condition counts for the array access will be kept, but not for the computations within the square brackets. **greport** will make reports like

```
"lc.c", line 252: operand ACCESS might be index.
"lc.c", line 252: operand ACCESS might be constant.
"lc.c", line 252: operand ACCESS never differs from the previous element.
```

Operators that expand into operators are ignored. That is, in

```
#define ADD(A+B)
ADD(a, b)+5
```

operator coverage will be applied to the `+`, operand coverage will be applied to the `5`, and the expansion of `ADD(a,b)` will be ignored.

dereference Default: on

Pointers are a problem for operand coverage. Consider a statement like

```
A := *P
```

We would like to rule out the case where P might have been Q by requiring a test case in which *P!=*Q. This is dangerous if Q is not a valid pointer. GCT attempts to avoid the dangers by (for

example) initializing all uninitialized pointers to zero and ruling out null pointers without dereferencing them. Not all dangers can be avoided, but GCT's effectiveness would be markedly diminished if it never dereferenced suspect pointers. Consequently, the default is to dereference pointers when needed. If a pointer is invalid, a core dump may result. In this case, you might modify the program (to use calloc instead of malloc when allocating arrays of pointers, for example) or turn off the *dereference* option.

array-substitutions Default: on

This is like the *dereference* option, but it turns off fewer substitutions. In an expression like

```
array[i]
```

turning *array-substitutions* off will eliminate a coverage condition that would require

```
array[i] != other_array[i]
```

constants Default: on

By default, operand coverage requires every variable to actually vary, to be non-constant. In practice, a large number of variables in correct programs do not vary. (Typically, these are variables whose values are calculated only once.) This results in a great number of messages from **greport**, most of which are impossible to eliminate. Sifting through them is a waste of time. The **constants** option can be used to turn them off.

The Types of Coverage

6. Changes to Multicondition Coverage

When multicondition coverage is selected alongside weak mutation coverage, it applies to more operators. The complete set of operators is

Tree type	Point of instrumentation
and	&&
or	||
bitwise and	&
bitwise or	|
bitwise exclusive or	^

As before, both the left-hand and right-hand sides must be both 0 and non-zero.

Similarly, the set of operators that yield "boolean" values has been expanded:

<	<=	>	>=	==	!=	
&	|	^	&&	||	~	!
&=	|=	^=				

These operators must yield both zero and non-zero when used as the argument to return or an assignment statement.

Further, these kinds of assignment statements must yield both zero and non-zero values:

&=	|=	^=

These extensions seldom add much cost to testing, since the operators are not commonly used, and the coverage conditions are often ruled out without your ever seeing them. Still, it's doubtful they add much benefit, and they may disappear in a later release. Your opinion is welcome.

7. Operator Coverage

All operator coverage is expressed in terms of a hypothesized correct operation that can't be ruled out. Suppose we have the following code:

```
if (a + b < c + d )
```

To rule out the fault where the first + should have been a -, B must not be 0, (since a+0=a-0). For the same reason, D must not be 0. To rule out the fault where < might be <=, there must be a case where A+B==C+D. If a fault is ruled out, nothing regarding it will appear in the report unless the **-all** option is given.

A report will look like:

```
"example1.c", line 9: operator + might be -.
"example1.c", line 9: operator < might be <=.
"example1.c", line 9: operator + (2) might be -.
```

If -all is used, the number of times the condition is satisfied is reported:

```
"lc.c", line 275: operator <= might be >=  [1]
"lc.c", line 275: operator <= might be <.  [3]
```

7.1. The Point of Instrumentation

In all cases, the point of instrumentation is the operator itself.

7.2. Arithmetic Operators

expr1 + expr2	
Alternate	Rule out with
-	expr2!=0
*	expr1+expr2!=expr1*expr2

expr1 - expr2	
Alternate	Rule out with
+	expr2!=0

expr1 * expr2	
Alternate	Rule out with
+	expr1+expr2!=expr1*expr2

expr1 / expr2	
Alternate	Rule out with
%	expr1 / expr2 != expr1 % expr2

expr1 % expr2	
Alternate	Rule out with
/	expr1 / expr2 != expr1 % expr2

- expr1	
Alternate	Rule out with
non-negated expression	expr1!=0

7.3. Boolean Operators

expr1 && expr2	
Alternate	Rule out with
||	expr1&&expr2 != expr1||expr2

Ruling out & is too hard to satisfy, so that coverage condition is omitted.

expr1 || expr2	
Alternate	Rule out with
&&	expr1 || expr2 != expr1 && expr2

Ruling out | is, again, too hard to satisfy.

!expr	
Alternate	Rule out with
~	!expr != ~expr

The only case that does *not* rule out ! is a word with all ones.

7.4. Bitwise Operators

expr1 & expr2	
Alternate	Rule out with
|	expr1!=expr2
&&	!!(expr&expr2) != expr && expr2

In the second case, notice that the *truth value* of expr1&expr2 must be distinguishable from expr1&&expr2. The weaker condition (expr1&expr2 != expr1 && expr2) would be satisfied by

expr1 = 0x5, expr2 = 0x3

The differences between the original expression and the variant are more likely to cause a noticeable effect if one of them is forced to 0.

expr1 | expr2	
Alternate	Rule out with
&	expr1!=expr2
||	(expr1 | expr2) != (expr1 || expr2)

Unlike the previous case, the truth value of the left hand side is not used -- it always has the same value as the right hand side.

expr1 ^ expr2	
Alternate	Rule out with
|	expr1|expr2!=expr1^expr2
&	expr1^expr2!=expr1&expr2

The second case is used because &-for-^ is a likely typo -- the two keys are next to each other on the keyboard.

~ expr	
Alternate	Rule out with
!	~expr != !expr

The only case that does *not* rule out ! is a word with all ones.

expr1 << expr2, expr1 >> expr2	
Alternate	Rule out with
each other	expr1!=0

7.5. Assignment

Coverage conditions rule out alternate operators, as usual. They also force test cases in which the assignment statement makes a change in the program state. For example, if, in "A = B", A and B always have the same value, perhaps the assignment statement is never needed.

In the descriptions of how to rule out alternates, *lval_new* refers to the post-assignment value of the variable on the left-hand-side of the assignment statement. *lval_old* refers to its pre-assignment value.

lval = expr	
Alternate	Rule out with
never needed	lval_old!=lval_new
==	(!!lval_new) != (lval_old == expr)

The double negation converts the value of the assignment statement into a truth value, which must be different from the truth value of the expression gotten when = is replaced with ==. This is because this fault is often found in expressions like:

```
if (a = b)                    /* Meant a == b */
```

and the comparison of truth values forces the other branch to be taken, making it more likely that the effect is observable.

lval += expr	
Alternate	Rule out with
-=, never needed	lval_old!=lval_new
*	lval_new != lval_old*expr
==	(!!lval_new) != (lval_old == expr)

== is used because it's a likely typo.

lval -= expr	
Alternate	Rule out with
+=, never needed	lval_old!=lval_new

lval *= expr	
Alternate	Rule out with
/=, never needed	lval_old!=lval_new
+=	lval_new != lval_old + expr
&=	lval_new != lval_old & expr

&= is used because it's a likely typo.

lval /= expr	
Alternate	Rule out with
*=, never needed	lval_old!=lval_new
%=	lval_new != lval_old % expr

lval %= expr	
Alternate	Rule out with
never needed	lval_old!=lval_new
/=	lval_new != lval_old / expr
^=	lval_new != lval_old ^ expr

^= is a likely typo.

lval <<= expr, lval >>= expr	
Alternate	Rule out with
the other, never needed	lval_old!=lval_new

lval &= expr	
Alternate	Rule out with
never needed	lval_old!=lval_new
|=	lval_new != lval_old | expr
*=	lval_new != lval_old * expr
^=	lval_new != lval_old ^ expr

^= and *= are likely typos.

lval |= expr	
Alternate	Rule out with
never needed	lval_old!=lval_new
&=	lval_new != lval_old & expr
+=	lval_new != lval_old + expr

+= is a likely typo.

lval ^= expr	
Alternate	Rule out with
never needed	lval_old!=lval_new
|=	lval_new != lval_old | expr
%=	expr == 0 || (lval_new != lval_old % expr)
&=	lval_new != lval_old & expr

%= and &= are likely typos.

expr == 0 rules out the %-substitution because such an alternate program would get a divide-by-zero error and hence could not be correct.

lval++, lval--, ++lval, --lval	
Alternate	Rule out with

There is no instrumentation. Any evaluation of these expressions has a different value from any of the others; hence, simple execution rules out those substitutions.

Similarly, any evaluation of --lval is distinguishable from -lval because the value of lval is different.

7.6. Flow of Control

while	
Alternate	Rule out with
if	two or more loop executions

In the current implementation, operand coverage for while statements is the same as loop coverage. The **greport** message is the same:

"example1.c", line 9: loop zero times: 1, one time: 0, many times: 0.

Executing the while test more than once rules out the possibility that it should have been an if.

Unfortunately, there's no equivalent condition that rules out the more common fault where an if should have been a while.

goto, continue, break	
Alternate	Rule out with

There is no instrumentation for goto; if the goto target is incorrect, the error will likely be detected by a test that executes the goto, which branch coverage requires.

Similarly, the error where a continue should have been a break, or a break a continue, will likely be detected if executed, especially if loop coverage is required.

8. Operand Coverage

In operand coverage, values of variables or constants are compared against each other. The following are all operands:

A
1
Array[1]
Array[1+1]
S.field
S->field
*ptr
*(ptr+S.field)

Operand coverage is a requirement on pairs of operands. Suppose we have a program containing this code:

function(A);

Operand coverage might require the following:

A != B
A != C
A != D

This could be accomplished in one test in which the variables had these values:

A == 0, B == 1, C == 1, D == 1

or it might take three tests:

A == 0, B == 1, C == 0, D == 0 => Rules out B (A!=B).

A == 1, B == 0, C == 1, D == 0 => Rules out B (A!=B) and D (A!=D).
A == 0, B == 0, C == 1, D == 0 => Rules out C (A!=C).

The idea of operand coverage is that if a particular operand has the same value as another in every test, we have no evidence that the correct program shouldn't use the other operand. (Of course, if the operand has the same value in every possible execution, it doesn't matter which we use.)

Output is in terms of the existing operand and the hypothesized substitution. For example, given

```
if (A < F(5, A))
```

output might be

```
"example1.c", line 9: operand A might be B.
"example1.c", line 9: operand A might be C.
"example1.c", line 9: operand A might be constant.
"example1.c", line 9: operand 5 might be A.
"example1.c", line 9: operand A (2) might be B.
```

(In the third line, notice that there is no way to indicate which constant A might be.)

8.1. Compatibility

A pair of operands can be compared only if C defines inequality between them. For example, integers can be compared to floats, but floats cannot be compared to whole structures. (There is almost no circumstance in which a float should have been a structure without generating a compiler error or warning.)

This is not ideal. For example, consider the following code:

```
func(s1, s2)
char *s1, *s2;

if (func2(s1))
    return 5;
...
```

GCT will rule out the substitution of S2 for S1 if S2!=S1. However, this comparison of pointers is not optimal. In this context, S1 and S2 are intended to be strings. A better comparison would be to rule out the substitution if

```
string_not_equal(s1, s2)
```

However, this cannot be done automatically, since GCT has no way of knowing that these character pointers represent null-terminated strings.

8.2. Selecting Alternates

There may still be an infinite number of type-compatible operands. How are alternates selected? For example, should A be compared to all of these possible operands?

```
Array[0]
Array[1]
Array[2]
...
Array[A]
Array[B]
Array[C]
```

```
...
Array[A+B]
...
```

Clearly, this is not useful. GCT uses two general rules to pick alternate operands. Each kind of operand (e.g., array references, pointer dereferences) extends, restricts, or modifies the set of alternates produced by those two rules.

constants

For any operand, all constants of a compatible type are potential alternates. (For variables, this infinite set of comparisons can be implemented by requiring that the variable actually take on at least two values.)

local variables

All variables of a compatible type, defined in the same function, are potential alternates. Global variables are *not* alternates. There are often a surprising number of them (hidden in include files) and not testing for them doesn't (yet) seem to lead to any loss of effectiveness.

For the example above, A would be compared to B, C, D, but not to any array reference. An array reference like Array[A] would be compared to A, B, C, D, etc., but not to any other array reference. (Under this rule -- we'll see in the section after next how comparisons between array references are generated.)

8.3. Weak Sufficiency

It's not always enough just to check for differing values. Consider, for example,

```
if (var1 == var2)
```

Suppose var2 should be var3. Suppose further that var1 == 1, var2 == 2, and var3 == 3. var2!=var3, so operand coverage is satisfied. However, the if statement has the same value in both cases, so the fault might quickly "damp out" and become unlikely to be detected. We really want the relational expression to take on a different value.

Consequently, if a variable comparison is made when the variable in question is the ENTIRE right-hand or left-hand side of a relational expression, then not only must the comparison reveal a difference, but the results of the relational operator must be different.

This rule, whose technical name is *weak sufficiency* [Marick91b], will certainly cause you confusion at first. You will have an expression like

```
f(a, b, c)
{
  if (a < b)
  ...
```

and see this message from **greport**

"example1.c", line 9: operand B might be C.

and say, "That's impossible -- the test case was F(1, 2, 3) and 2!=3", having forgotten that, because of weak sufficiency, you must instead satisfy this coverage condition:

```
(a < b) != (a < c)
```

The early annoyance will pay off in better tests.

8.4. Weak Sufficiency for Compound Operands

Weak sufficiency also applies to compound operands like array references, structure field references, and so forth. The rule is similar: Whenever an operand is immediately contained in a larger operand, the entire containing operand must have a different value from the operand formed by substituting the alternate contained operand. Some examples will make this clear:

Array[A] must differ from Array[B]. It is not enough for A to differ from B.

Array[Array2[A]] must differ from Array[Array2[B]].

*PTR must differ from *PTR2. It is not enough for PTR to differ from PTR2.

However, in the case of *(ptr+1), we only require PTR to differ from PTR2. We do **not** require *(PTR+1) to differ from *(PTR2+1), because PTR is not immediately contained in the dereference. (The increased difficulty of devising test cases for such comparisons doesn't seem to be worthwhile.)

8.5. Refinements for Particular Kinds of Operands

8.5.1. Constants

Constants follow the general local variable rule. Thus, you might expect to see messages like

"lc.c", line 153: operand 3 might be c.
"lc.c", line 252: operand '{' might be index.
"lc.c", line 919: operand "panic: %s" might be reason.

The constant rule would seem to be unneeded: what's the point of comparing a constant to another constant? In general, none. But there is a point when weak sufficiency holds. Consider

if (a < 5)

GCT will generate tests that require you to demonstrate that no other constant but 5 would do. In this case, setting A==5 rules out any constant greater than 5. (If 5 should have been 6, say, the if statement would branch the other way.) Setting A=4 rules out any constant less than 4. (If 5 should have been 4, the if statement would again branch the other way.) Hence, 5 is likely to be correct.

In such cases, the message from **greport** will be "5 might be another constant". Here are the various cases, and coverage conditions that will eliminate them:

```
A < 5
        A==4
        A==5
A <= 5
        A==6
        A==5
A > 5
        A==6
        A==5
A >= 5
        A==4
        A==5
A != 5
        A==5
A == 5
        A==5
```

8.5.2. Sizeof

The *sizeof* operator is a constant. Constant substitutions are done for *sizeof* operators used in relational expressions. Variable substitution tests aren't done -- it's unlikely that a sizeof expression should have been a variable.

8.5.3. Array References

The usual constant and variable substitution tests are used for array references. Expect to see messages like

"lc.c", line 869: operand chars[...] might be ch.
"lc.c", line 869: operand chars[...] might be constant.

For an operand like Array[A] you will also see messages like

"lc.c", line 869: operand A might be B.

As discussed in section 4.6.4, this means that Array[A] must differ from Array[B]. Notice, however, that GCT cannot blindly insert such comparisons: suppose B had the value -100,000,000? They are made only if 0 <= B < A. Expect to be confused several times because you know that B is different from A, but you still get the **greport** message that A might be B. You've forgotten that B must both be different than A and also safe as an index.

GCT will also consider B ruled out if B<0, though this is less reliable.

For any array reference, GCT also requires a test case in which Array[A] differs from Array[A-1]. This encourages non-homogeneous test arrays, which are much more likely to find faults. Of course, such a test can only be made if A>0.

GCT also requires test cases where Array[5] differs from Array2[5] (or Array2 is null). These test cases are dangerous if Array2 is uninitialized. The **-array-substitutions** or **-dereference** options turn them off.

8.5.4. FIELD REFERENCES

This discussion applies to both references of the form A.field and also P->field.

The usual constant and variable substitution tests are used, so you'll see messages like

"lc.c", line 476: operand name_tally->pure_code might be constant.
"lc.c", line 476: operand name_tally->pure_code might be B.

In addition, the tests must rule out substitutions of a different field in the same structure. For example, if the structure is defined as

```
struct S
{
  int field1;
  int field2;
};
```

an operand like p->field1 will yield this message:

"lc.c", line 476: operand p->field1 might use field field2.

The normal variable substitution on P, when combined with weak sufficiency, will require that you rule out possible substitutions like

```
q->field1
```

Failing to do so will yield a message like

"lc.c", line 476: operand p might be q.

Notice that

p->field is compared with p->field2
and
p->field is compared with q->field
but
p->field is *not* compared with q->field2.

Double typos of this sort are rare and are likely to be caught by tests adequate for single typos [Offutt92].

8.5.5. Dereferences

Pointer dereferences use the standard constancy and variable substitution tests. Expect to see messages like:

"lc.c", line 250: operand *p might be argc.
"lc.c", line 250: operand *p might be constant.

You will also see messages like

"lc.c", line 250: operand p might be q.

Because of weak sufficiency, you must construct test cases such that *p != *q.

8.6. Miscellaneous

There is no instrumentation for the address-of operator. Any likely alternate operand for &var (such as &var2) will almost always yield a different value; hence, merely executing the expression is enough of a coverage condition.

There is no instrumentation for casts. Although there are faults associated with cases, I don’t have enough of them yet.

Troubleshooting

See *GCT Troubleshooting* for help with general GCT problems.

If you have trouble that you think others might share, please mail a description of the problem to marick@cs.uiuc.edu, so that this chapter might be improved.

9. Trouble Instrumenting

Compiler runs out of space

You are most likely using operand coverage and an old C compiler. You'll have to turn off operand coverage.

GCT panics.

GCT will panic if given an expression like

```
((int) p) = 5;
```

This is not legal C, but some code contains it. The only solution is to change the source.

Warnings about comparisons.

Suppose we have this code:

```
(i < 10000000)
```

Suppose `c` is a character. We will end up with this test, because characters are comparison-compatible with integers:

```
_G(i < 10000000 != c < 1000000)
```

Because the constant is larger than any character, this will lead to warnings:

Sun CC:
"file.c", line 208: warning: constant 1007 is out of range of char comparison
"file.c", line 208: warning: result of comparison is always true

GCC:
file.c: In function remove_intermediates:
file.c:208: warning: comparison is always 1 due to limited range of data type

It can also lead to warnings from the assembler. You can safely ignore all these warnings.

10. Trouble Handling Greport Output

Cannot figure out which complex operand is meant.

If you see

"example1.c", line 9: operand <...>[...] (2) might be C.

this refers to the second complex array reference on the line (even if they are not identical). For example, in this line:

```
p->array[5] + Array[p->array[4]];
```

`p->array[5]` is printed as `operand <...>[...]`.
`Array[p->array[4]]` is printed as `operand Array[...]`.
`p->array[4]` is printed as `operand <...>[...] (2)`.

Cannot rule out operand coverage when operand is in relational expression.

Greport messages of the form

"example1.c", line 9: operand A might be B.

can be frustrating to rule out because of weak sufficiency. (See the discussion earlier in this document.) If A is a part of a relational operator expression, such as

A < 3

It is not enough to provide a test case where A!=B. You must provide a test case where (A<3)!=(B<3).

Cannot rule out operand coverage when operand is in nested operand.

(See also above case.) Weak sufficiency also applies to operands built up from structures, arrays, and pointer dereferences. For example, in the expression

function(A[3].foo)

the "A might be B" message must be ruled out with A[3].foo!=B[3].foo. Even worse, if the expression is

A[3].foo < 5

the needed coverage condition is

(A[3].foo) < 5 != (B[3].foo < 5)

Cannot rule out operand coverage when operand is array index.

In A[I], "I might be J" can only be ruled out if 0<=J<I. It would be dangerous to use other values of J to index into the array. GCT will also consider J ruled out if it is less than zero. That is, to eliminate the **greport** message, you must satisfy this coverage condition:

j<0 || (0 <= j < i && A[i]!=A[j])

Wrong variable seems to be named.

In lines like

"example1.c", line 9: operand A (2) might be B.

the parenthetical number is NOT instances of A on the line, but rather the number of instances that have had instrumentation generated for them. These are almost always the same number. However, if a particular instance of A on a line has no associated coverage conditions, the numbering will be confusing. The only common case of this is in assignment. For example, in

A = A + 1

the first A has no coverage conditions. **greport** would therefore show:

"example1.c", line 9: operand A might be B.
"example1.c", line 9: operand A might be C.

referring to the *second* A on the line.

Sorry about that.

Variable is always reported to be constant.

Operand mutation requires variables to actually vary; they must take on two different values *in the same execution of the program*. It is not enough for a variable to have the value 5 when the program is run once, then value 3 when the program is run again. This can make certain variables impossible to vary, because they are set once in the beginning and never change. argv, argc, and standard I/O library buffers (stdin, etc.) exhibit this problem. You can use the -constants option to suppress constancy instrumentation entirely.

Instrumented program dumps core.

Most likely, instrumentation caused dereferencing of an uninitialized (but non-zero) pointer or array. See the discussion of the -dereference and -array-substitutions options in the section on the control file. **gcorefrom**(1) can help you locate which instrumentation caused the problem.

It's also possible the program has a bug.

Cannot rule out = might be ==.

You must have (!!B) != (A==B). The easiest thing to try for is two non-equal, non-zero numbers. (This also rules out "= is never needed".) Having both A and B be 0 also works.

Watch out for the **!!**: it means that A==2, B==2, for example, will not rule out the condition.

III. Bibliography

[DeMillo78]
R.A. Demillo, R.J. Lipton, and F.G. Sayward, 'Hints on test data selection: help for the practicing programmer'. *Computer*. vol. 11, no. 4, pp. 34-41, April, 1978.

[Hamlet77]
R.G. Hamlet. *IEEE Transactions on Software Engineering*, vol. SE-3, No. 4, pp. 279-289, 1977.

[Howden82]
W. E. Howden. *IEEE Transactions on Software Engineering*, vol. SE-8, No. 4, pp. 371-379, July, 1982.

[Marick91a]
Brian Marick. 'Experience with the Cost of Test Suite Coverage Measures'. *Pacific Northwest Software Quality Conference*, October, 1991. Also available as a compressed postscript file in cs.uiuc.edu:/pub/testing/experience.ps.Z.

[Marick91b]
Brian Marick. 'The Weak Mutation Hypothesis'. *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis, and Verification*, October, 1991.

[Offutt92]
A. Jefferson Offutt. 'Investigations of the Software Testing Coupling Effect'. *ACM Transactions on Software Engineering and Methodology*, vol. 1, No. 1, January, 1992.