Generic Coverage Tool (GCT)
User's Guide

Brian Marick
Testing Foundations

Documentation for version 1.4 of GCT
Document version 1.5

This manual describes the Generic Coverage Tool, a tool that instruments C code to provide various measures of test suite completeness. Readers of this manual (and users of the tool) should be experienced C programmers. The tool itself runs in almost any UNIX[1] environment. The instrumented code depends on UNIX library routines and system calls, but in only certain well-isolated procedures. It is relatively simple to replace those, allowing use of the tool when testing embedded systems.

The document also describes GCT Expansion Kit 1, a separate product.

[1] UNIX is a trademark of Bell Laboratories.

Preface

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 manual was originally based on *Branch Coverage Tool Documentation*, by Thomas Hoch, Brian Marick, and K. Wolfram Schafer.

Introduction

This manual describes the most common uses of GCT. You should read *A Tutorial Introduction to GCT* first. When using GCT, you'll want to refer to the manpages (which should have been installed when GCT was) and *GCT Troubleshooting*. For conciseness, this manual does not describe some types of instrumentation. See *Using Weak Mutation Coverage with GCT* and *Using Race Coverage with GCT*.

You may have also purchased the separate GCT Expansion Kit 1. Because the expansion kit is intended to blend smoothly with GCT, it is best explained in this document, rather than a separate one. If you do not have the Expansion Kit, you may skip the sections devoted to it.

1. Test Suite Coverage

How do you tell when you're done testing? You can test until you run out of time, test until you can't think of any more test cases that seem useful, or test until some objective stopping criterion is met. GCT implements a variety of such criteria. Branch, loop, multiconditional, and relational coverage are introduced in *A Tutorial Introduction to GCT*; the remainder are introduced here. All are described in detail later in this document.

1.1. Routine Coverage

Routine coverage measures whether a routine has been entered. It is most useful for determining the coverage of large test suites running against entire systems.

1.2. Call Coverage

Call coverage measures whether a certain function call has been made. 100% call coverage means that all calls between functions have been exercised at least once: every function has called every other that it can call.

1.3. Other Coverage

Two other types of coverage are described in the companion documents *Using Weak Mutation Coverage with GCT* and *Using Race Coverage with GCT*.

2. Some Terminology

A **test condition** is a description of something that must be tested. For example, "pass the routine a null pointer" or "give variable A a negative value". A test case may satisfy many test conditions, and a single test condition may be satisfied by many test cases.

Test conditions can be gotten from many places. People often derive them by looking at the program's user's manual (or other black-box description). GCT derives them from the code. GCT-generated test conditions are called **coverage conditions**.

The **condition count** tells how often a program under test has been executed in a way that satisfies a particular coverage condition. The coverage condition is a characteristic of a program; the condition count is a characteristic of a test suite for that program.

Instrumentation is code added to the program to increment the condition counts.

Ordinary Instrumentation

This chapter describes the normal way you'd instrument a program. The next chapter describes what the coverage data means. The chapter after that describes alternate approaches to instrumentation that people with special needs might use. The final chapter describes some scenarios; they may be useful examples if your system is complex.

3. The Control File

The *control file* determines which files and routines are instrumented and what kind of coverage is measured. It is usually called **gct-ctrl**, but the name can be changed with **gct -test-control**.

The control file is stored in the *master directory*. This is usually the directory that contains the main makefile for your system, though it can be anywhere.

The control file consists of a number of entries, arbitrary white space, and comments. A comment begins with a '#'; it persists until the end of the line. The file contains the following four elements, called *top level directives*, arranged in any order:

option directives
coverage directives
file directives
logfile directive

Although none of the directives are required, an empty control file is useless.

3.1. Option Directives

Options set boolean values within GCT. An option directive has this form:

(options *option*...)

By naming an option, you turn it on. By preceding it with a minus sign, you turn it off. Hence,

(options option1 option2)

turns both option1 and option2 on, whereas

(options -option1 -option2)

turns them off.

Options are described throughout the text and summarized in an appendix.

3.2. Coverage Directives

A coverage directive tells GCT what kinds of instrumentation to perform (branch, loop, etc.). Coverage directives are like options: they are turned on by naming them, and turned off by preceding their names with a minus sign:

(coverage branch -loop)

Branch instrumentation is to be inserted; loop instrumentation is not.

All possible coverage[2] would be turned on with

[2] I am ignoring race and weak mutation coverage, as they're not covered by this document.

```
(coverage branch loop multi relational routine call)
```

By default, no coverage is turned on.

3.3. File Directives

A file directive has three forms:

1) Simply naming a file causes it to be instrumented. Filenames that do not begin with a slash are interpreted relative to the master directory, not the directory where GCT runs. If the source is in a subdirectory of the master directory, for example, entries should look like this:

```
src/program1.c
src/program2.c
```

2) Preceding a filename with a minus sign causes it not to be instrumented.

3) If a file name is enclosed within parentheses, it is instrumented. Further, option directives and coverage directives may also be specified; they override top-level directives. Thus, the following control file specifies that *option1* applies to all files except *macros.c*.

```
(options option1)
(macros.c (options -option1))
```

These options are said to apply at the *file level*.

3.4. Routine Directives

A file directive may also contain one or more routine directives. A routine directive has one of three forms:

1) Simply naming a routine causes it to be instrumented.

2) Preceding its name with a minus sign causes it not to be instrumented.

3) If the routine name is within parentheses, it is instrumented. Option and coverage directives may also be specified. They override those used at the top level or within filename directives.

The routine name may be preceded by the keyword "routine"; this allows you to instrument routines named "options", "coverage", or "routine":

```
(file.c (options option1)
    (routine options (options -option1)))
```

These options are said to apply at the *routine level*.

3.5. Logfile Directive

The logfile directive names the logfile in which to store coverage data. By default, it is *GCTLOG*. To change it to *LOG*, you would write

```
(logfile LOG)
```

The argument is independent of the master directory. If it is a relative pathname, it is relative to wherever you run the executable.

3.6. Common Options

instrument Default: OFF Set at: top level, file level, or routine level

Normally, files are not instrumented unless they're explicitly named. Setting the instrument option at the top level causes files to be instrumented unless they're explicitly turned off. For example this control file

```
(options instrument)
-test.c
```

causes all files to be instrumented except *test.c*.

Normally, instrumentation is turned off by prefacing a file or routine name with a minus sign. It can also be turned off by applying the instrument option at that level. For example, the effect of the above control file can also be had with

```
(options instrument)
(test.c (options instrument))
```

This is especially useful when you only want to instrument one routine in a file. Example: Suppose we want to instrument all routines in file1.c, only routine instrument_me() in file2.c, all routines in file3.c except routine do_not_instrument(), and no other files.

```
# The default is to instrument no file.
file1.c
(file2.c (options -instrument) instrument_me)
(file3.c -do_not_instrument)
```

macros Default: OFF Set at: top level or file level

Consider the **getc()** macro. It expands into 298 characters of code. The results of instrumenting this could be quite confusing; for a line like

```
for (c = getc(stream); c >= 'A'; c = getc(stream))
```

greport might refer to the first ">=" in the line, and you very likely would not realize that the operator referred to is in the expansion of getc. Further, it is inappropriate to test the expansion of getc: having been used millions of times, it's pretty likely to be correct and should be treated just like a function call. Therefore, by default GCT does not instrument code within macro expansions.

The **macros** option makes GCT instrument even code within macros. The main use of this option is for function-defining macros. Suppose you want to measure branch coverage for this code:

```
#define NEWFUNC(NAME, RETURN-TYPE) \
RETURN-TYPE NAME (Stream) \
FILE *stream; \
{ \
    char a = getc(Stream); \
    if ('0' == a) \
        return 1; \
    return 0; \
}
NEWFUNC(myfunc, float)
```

Normally myfunc() would be completely uninstrumented. To instrument it, you must use the **macros** option:

(file.c (myfunc (options macros)))

You will have to put up with **greport** messages about the contents of *getc*; this is unavoidable. If the body of the function were not defined within the macro, the option would be unneeded.

instrument-included-files Default: OFF Set at: top level or file level

Normally, GCT ignores any routines in files included by the file passed to it. If you wish to instrument included files, set this option. Note that if a particular file is included by several source files, it will be instrumented separately in each of them.

The most common use for this option is when the source being instrumented is produced by another program. Consider, for example, a grammar parsed by YACC or BISON. The original grammar might be *c-parse.y* and the resulting C file *c-parse.tab.c*. GCT is invoked on *c-parse.tab.c*. Because of the way such files are usually constructed, *c-parse.y* appears to GCT as an include file in *c-parse.tab.c*. It will be ignored unless you set instrument-included-files.

The same will apply to intermediate C files produced by C++ translators.

This option is checked only once per routine, at the beginning. If part of the routine’s text comes from an include file, GCT will not notice that.

More sophisticated control over included files may be provided in later releases. Your suggestions are welcome.

4. Performing the Instrumentation

To instrument, you must perform the following steps:

(1) Remove all object files so that each source file will be recompiled with GCT. If you do not do this, the log may not be written when the program exits, or it may not be read in when the program starts. GCT inserts code to do this reading and writing wherever necessary; if a file is not processed, the code cannot be inserted. (The "Advanced Instrumentation" section describes how to prevent this code from being inserted.)

(2) In the master directory, type

```
% gct-init
```

This will create the following files. Most can be ignored, except for special circumstances; however, all are described here for reference.

gct-map
: The mapfile records what instrumentation was added.

gct-defs.h
: This contains definitions used in the instrumented files. It is copied into the directory once and is usually never changed.

gct-ps-defs.h
: Each invocation of GCT uses this file to pass information to later invocations.

gct-ps-defs.c
: This file contains the definition of the table used to store condition counts. It must be recompiled every time a new instrumented executable is built.

gct-write.c
: This file contains routines to read and write the condition counts. This file is useful only for application programs; those testing embedded systems will have to read and write condition counts some other way. (A relevant scenario appears later in this document.)

(3) Compile using GCT instead of the normal C compiler. Quite often, you will only have to do this:

```
% make CC=gct
```

For this to work, the makefile must use the $(CC) macro, rather than invoking **cc** directly.

4.1. Multiple Directories

If the source is contained in several directories, you must do two other things:

(1) If the main makefile uses makefiles in the other directories to do part of the work, it must pass the $(CC) macro to them:

```
cd subdir1; make "CC=$(CC)"
```

The other makefiles must also use $(CC).

(2) The name of the master directory has to be passed to the other makefiles. Use the -test-dir option to GCT. You should invoke the main makefile like this:

```
% make "CC=gct -test-dir ‘pwd‘"
```

The -test-dir argument will be passed to all the other makefiles.

4.2. Using Other C Compilers

GCT normally instruments the program and places the instrumented source into a temporary file, then invisibly calls a C compiler to compile the instrumented source and produce an object file or executable.

When GCT is installed, **gct** is configured to use your default C compiler (usually **cc**). If other C compilers are used on your system, the installer may have configured special GCT interface scripts. These are conventionally named **gct-***compiler*. For example, if you want GCT to use the GNU C compiler to compile the instrumented file, you can probably use

```
% make CC=gct-gcc
```

Such scripts use the -test-cc argument to identify the compiler to use. See the installation documentation for more information about setting up such scripts. (The installation documentation can be found in the documentation source directory.)

4.3. Reinstrumenting after Changes

In the basic version of GCT, any change to a file requires that the entire instrumentation process be repeated, starting with gct-init. GCT Expansion Kit 1 permits incremental reinstrumentation.

In most cases, reinstrumentation is no different than recompiling. You change the source and type

```
% make CC=gct
```

again, without starting with **gct-init**. Generally speaking, reinstrumentation preserves the effects of **gedit(1)** when it’s sensible to do so. Full details are given in the section on "Advanced Instrumentation".

The Types of Coverage

GCT can instrument the program to measure any of these kinds of coverage:

branch:	branch coverage
multi:	multi-conditional coverage
loop:	loop coverage
relational:	relational operator coverage
routine:	routine entry coverage
call:	call coverage

This chapter describes precisely what is measured for each of these kinds. It also describes what the **greport** output will look like.

4.4. More Terminology and Background

The condition count is incremented by instrumentation code inserted into the program. Whether instrumentation is inserted is controlled in two ways:

(1) The **instrumentation** option. Normally, everything is instrumented if the file is processed.

(2) The **macros** option. Normally, code within the expansion of a macro is not instrumented.

When is code "within" a macro’s expansion? Instrumentation applies to whole statements or expressions in the program (an IF statement, or a relational expression). However, GCT decides whether to instrument by examining a single token, called the *point of instrumentation.*

Consider this code:

```
#define IF     if

IF (x < 5)
then x = 3;
```

The IF statement will **not** be instrumented, because the point of instrumentation (the `if`) is within a macro, even though the rest of the statement is not. The following sections define all the points of instrumentation.

The arguments to a macro are part of its expansion. For example, in this code

```
#define TEST(tst)  if ((tst)) {
#define ENDTEST    }

TEST(a < b)
  printf("a is less than b");
ENDTEST
```

a<b is not instrumented, for the same reason the `if` is not.

Most of the time, these subtleties are invisible. Since the code isn’t instrumented, it isn’t mentioned in **greport** output, so you never think about it.

4.5. General Notes about Greport Output

greport lines look like this:

```
"example1.c", line 9: operator < might be <=. (L == R)
"example1.c", line 10: if was taken TRUE 1, FALSE 1 times.
```

"if" and "<" are tags that identify what the line refers to. If there were more than one "if" or "<" on the line, the tags would be ambiguous. In this case, the second and later instances would be numbered:

"example1.c", line 9: operator < might be <=. (L == R)
"example1.c", line 9: operator < (2) might be <=. (L == R)
"example1.c", line 10: if was taken TRUE 1, FALSE 1 times.
"example1.c", line 10: if (2) was taken TRUE 0, FALSE 1 times.

In some cases, the identification tag is abbreviated to save space on the line. See the section on multicondition coverage, which is the type of coverage that most uses abbreviations.

5. Branch Coverage

When branch instrumentation is turned on, these operators must have their tests evaluate to both zero and non-zero values:

Operator	Point of instrumentation
if	if
for	for
while	while
do-while	do
question	?

In a switch, each case label must be branched to. The default case must also be taken (even if one is not explicitly present). Fall-throughs and gotos into a switch do not increment condition counts; only the execution of the switch expression is considered to cause a true branch. For example:

```
switch(tag)
{
        case 1:
        case 2:
                <some code>
                break;
        case 3:
                goto_target:
                   <more code>
        case 4:
                <more code>
                break;
}
```

- If tag equals 1, the first case will be marked as taken. The second will not be marked as taken until the switch is executed with tag equal to 2.
- A goto to goto_target does not mean the third case has been taken. Even though it will cause a fall-through into the fourth case, it does not count as a branch to that case.
- Since this switch statement has no default case, GCT will insert one. That is, you are required to have a value for tag different from 1, 2, 3, and 4.
- The point of instrumentation is the case or default token. (In the case of the automatically inserted default, it's the closing bracket of the switch.)

As an example of the **greport** output for branch coverage, consider this program:

```
1    #include <stdio.h>
2
3    main ()
4    {
5      int arg;
6
7      printf (" This is an example\n");
8      arg = 1;
9      while ( arg <= 2 ) {
10        if ( arg == 1 ) { if ( 4 == 2*2 ) printf (" Second \n"); };
11        arg++;
12     };
13     switch (arg) {
14     case 2: /* some code */  break;
15     case 3: printf (" Third \n");
16     }
17   }
```

greport’s default output describes unsatisfied conditions:

```
"example1.c", line 10: if (2) was taken TRUE 1, FALSE 0 times.
"example1.c", line 14: case was taken 0 times.
"example1.c", line 16: default was taken 0 times.
```

There is more than one `if` statement on line 10, so **greport** adds "(2)". The system numbers such ambiguous keywords from left to right.

Note that a default case is reported for line 16, the line of the switch’s closing bracket.

If all conditions are required, **greport -all** should be used:

```
"example1.c", line 9: while was taken TRUE 2, FALSE 1 times.
"example1.c", line 10: if was taken TRUE 1, FALSE 1 times.
"example1.c", line 10: if(2) was taken TRUE 1, FALSE 0 times.
"example1.c", line 14: case was taken 0 times.
"example1.c", line 15: case was taken 1 times.
"example1.c", line 16: default was taken 0 times.
```

Examining this, we can see that the while loop was executed twice, and was left because its test evaluated false.

6. Multi-condition Coverage

For the following operators, both the left-hand and right-hand sides must be both zero and non-zero:

<table>
<tr><th>Operator</th><th>Point of instrumentation</th></tr>
<tr><td>logical and</td><td>&&</td></tr>
<tr><td>logical or</td><td>||</td></tr>
</table>

greport reports on each half of each condition separately. For a line of code like

```
if (a && (b < 12 || b > c))
```

greport output would look like:

```
"test.c", line 4: condition 1 (a, 1) was taken TRUE 1, FALSE 0 times.
"test.c", line 4: condition 2 (b, 1) was taken TRUE 1, FALSE 0 times.
"test.c", line 4: condition 3 (b, 2) was taken TRUE 1, FALSE 0 times.
"test.c", line 4: condition 4 (b, 2) was taken TRUE 0, FALSE 0 times.
```

The condition number identifies the condition in the line, numbered from left to right. The items in parentheses make it easier to find the condition. The first item is the first token in one of the sides of the logical expression. The second item is the nesting level in the tree of boolean operators that makes up the whole expression. Thus, in the above example, we have these conditions:

<table>
<tr><td>1</td><td>a</td><td>(&&-expression, left side)</td></tr>
<tr><td>2</td><td>(b < 12 || b > c)</td><td>(&&-expression, right side)</td></tr>
<tr><td>3</td><td>b<12</td><td>(||-expression, left side)</td></tr>
<tr><td>4</td><td>b>c</td><td>(||-expression, right side)</td></tr>
</table>

Another test condition that has proven useful is to insist that a function that returns a boolean value should return both zero and non-zero. Since C has no boolean type, GCT must deduce when a particular integer-returning function really returns boolean values. This is done by considering the expression that calculates the return value:

```
return 1 + 1;       /* Returns integer. */
return a && b;      /* Returns boolean. */
return f(a);        /* We don’t know. */
```

Here are the operators that produce boolean return values:

<table>
<tr><td><</td><td><=</td><td>></td><td>>=</td><td>==</td><td>!=</td><td>!</td></tr>
<tr><td>&&</td><td>||</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Of these, we need no special instrumentation for && and ||: satisfying multi-condition coverage for the left- and right-hand sides will necessarily yield both true and false values for the whole expression.

Of course, a boolean return value might be one that was calculated earlier. Therefore, if the right-hand-side of an assignment statement is one of the following types of expressions, the result must be both zero and non-zero:

<table>
<tr><td><</td><td><=</td><td>></td><td>>=</td><td>==</td><td>!=</td><td>!</td></tr>
<tr><td>&&</td><td>||</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Again, we need no special instrumentation for && and ||.

Suppose you have this code:

```
  flag = ! (b || c);
  return !flag;
```

greport -all might show this:

```
"test.c", line 4: condition 1 (= expression) was taken TRUE 0, FALSE 3 times.
"test.c", line 4: condition 2 (b, 1) was taken TRUE 2, FALSE 1 times.
"test.c", line 4: condition 3 (c, 1) was taken TRUE 1, FALSE 0 times.
"test.c", line 5: condition 1 (return) was taken TRUE 3, FALSE 0 times.
```

Declarations are treated like assignment statements. Given this (odd) code:

```
  float c1 = (c < d);
```

greport -all might show this if multicondition and relational operator coverage are turned on:

```
"multi.c", line 7: condition 1 (declaration) was taken TRUE 1, FALSE 1 times.
"multi.c", line 7: operator < might be >. (L!=R)  [1]
"multi.c", line 7: operator < might be <=. (L==R)  [1]
"multi.c", line 7: operator < needs boundary L==R-1.  [0]
```

The line number used by **greport** is that of the point of instrumentation. In code like

```
line 6:  if (A
line 7:        && B)
```

expect to see

```
"test.c", line 7: condition 1 (A, 1) was taken TRUE 1, FALSE 0 times.
"test.c", line 7: condition 2 (B, 1) was taken TRUE 1, FALSE 0 times.
```

even though condition 1 is strictly on line 6.

6.1. Abbreviations in Multiple Condition Output

Operands in C may be arbitrarily complicated:

```
Array[x->y.z].m->next
```

so GCT abbreviates to save space on the line.

Array references

Array references are printed as *arrayname*[...]:

```
"example1.c", line 9: condition 1 (A[...], 1) was taken TRUE 2, FALSE 1 times.
```

If the arrayname is itself more complex than a single identifier, it is printed as <...>. Thus, the array reference (p->array)[5] might yield this message:

```
"example1.c", line 9: condition 1 (<...>[...], 1) was taken TRUE 2, FALSE 1 times.
```

Dereference

If the pointer being dereferenced is an identifier, the message will be

```
"example1.c", line 9: condition 1 (*p, 1) was taken TRUE 2, FALSE 1 times.
```

Otherwise, it will be

"example1.c", line 9: condition 1 (*<...>, 1) was taken TRUE 2, FALSE 1 times.

if, say, the C code were "*(p+1)".

Structures and unions

If the structure is an identifier, its name is used; otherwise, the usual <...> notation is used. The field is always explicitly printed. Examples:

A.field is printed as "A.field".
B->field is printed as "B->field".
A[4]->field is printed as "<...>->field".

7. Loop Coverage

These are the looping statements:

Operator	Point of instrumentation
for	for
while	while
do-while	do

Loops constructed with gotos are not measured.

The **greport** message for a while or for loop might look like this:

"example1.c", line 9: loop zero times: 1, one time: 2, many times: 3.

The output refers to the number of times the body of the loop was entered. The "zero" entry is the number of times the loop test failed on the first try; the "one" entry is the number of times it succeeded exactly once; the "many" entry is the number of times it succeeded at least twice.

The body of a do-while loop is always entered at least once, since the test is at the bottom of the loop. Therefore, the **greport** message looks like this:

"example1.c", line 9: loop one time: 10, many times: 2.

The loop test failed on the first try ten times; it failed on a later try twice.

Loop coverage is measured at the loop test. Suppose a while loop is "started" with a goto into the body, and then the test at the top of the loop immediately fails. This counts as a "loop zero times"; it is no different than if the test failed as soon as the loop was entered normally.

A break or goto out of a loop does not affect the count. In a loop like

```
do
{
  code
  if if-test break;
  code
} while (loop-test);
```

the count is the same whether the loop exits because the *if-test* succeeds or because the *loop-test* fails. Once the body of the loop has been entered, it doesn't matter how it's exited.

The only case in which non-local exits make a difference is if a loop is exited with a goto or break and then entered with another goto. In this case, GCT cannot tell that the loop was ever left.

8. Relational Operator Coverage

Relational operator coverage requires tests that probe off-by-one errors. Many such errors are caused by using the wrong operator. Suppose we have the following code:

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

A likely mistake would be using < when <= is correct. Many tests that satisfy branch coverage would cause the correct and incorrect programs to take the same branch in the same direction - giving very little evidence that the common mistake wasn't make. A better test would cause the two programs to take the branch in different directions. That will only happen if the coverage condition A+B==C+D is satisfied.

GCT also requires that < be distinguished from >. This is a less likely error, but it is extremely easy to rule out -- the only value that *doesn't* is A+B==C+D.

Relational coverage requires one more condition. Consider

```
if (length >= 100)
```

branch coverage requires some length smaller than 100. However, we don't want to try just any such length -- we'd like one that is just below the boundary. That length is the one most likely to discover if 100 is the wrong value. Therefore, relational coverage also requires a test value just to one side of the "equality boundary". (That test will satisfy the previous, < for >, condition as well.)

The following table summarizes the test conditions for all relational operators. In all cases, **epsilon** is 1 (in the current implementation). The point of instrumentation is the relational operator.

L < R		
Condition	Rule out with	Special case for Integers
>	L!=R	
<=	L==R	
left slightly smaller	L < R <= L + epsilon	L == R - 1

L <= R		
Condition	Rule out with	Special case for Integers
<	L==R	
>=	L!=R	
left slightly larger	L > R >= L - epsilon	L == R + 1

L > R		
Condition	Rule out with	Special case for Integers
<	L!=R	
>=	L==R	
left slightly larger	L > R >= L - epsilon	L == R + 1

L >= R		
Condition	Rule out with	Special case for Integers
>	L==R	
<=	L!=R	
left slightly smaller	L < R <= L + epsilon	L = R - 1

There are no coverage conditions for == and !=. The most likely fault is using one when the other is correct; the condition for this fault is satisfied whenever the operator is executed (as is ensured by multi-condition coverage).

The **greport** output for relational operator coverage looks like this:

```
"example1.c", line 9: operator < might be <=. (L==R)
"example1.c", line 9: operator < needs boundary L==R-1.
```

L stands for the left-hand side of the relational expression; R for the right-hand side. Satisfying the first line’s coverage condition requires that the left-hand side be precisely equal to the right-hand side. For integers, the second line requires that the left-hand side be precisely equal to the right-hand size minus one. If either side is a floating point number, larger values for the left-hand side will also work, provided they are strictly smaller than the right-hand side. That is, if the right hand side is 3.2, the left-hand side must satisfy

(3.2 - 1) <= left-hand-side < 3.2

Once the coverage condition is satisfied, nothing regarding it will appear in the report unless the **-all** option is given:

```
% greport -all GCTLOG
"lc.c", line 275: operator <= might be >=.  (L!=R) [1]
"lc.c", line 275: operator <= might be <.  (L==R) [3]
```

The number in brackets is the condition count, the number of times the test condition has been satisfied.

8.1. Warnings

GCT issues a warning that many versions of *lint(1)* do not:

```
a == b;
```

yields

"Warning: == can have no effect."

9. Routine Coverage

When an instrumented routine is entered, the condition count is incremented. Routine coverage is not affected by macros; coverage is measured even for routines defined entirely within a macro.

The **greport** output is straightforward:

```
"example1.c", line 9: routine caller is never entered.
```

In this case, line 9 is the first statement of the routine (after any declarations). If **greport -all** is used, you might see

```
"example1.c", line 9: routine caller is never entered. [0]
"example1.c", line 30: routine main is never entered. [1]
```

which indicates that main has been called once, caller never.

There is one special case: routines containing no statements, such as

```
foo()
{
}
```

or

```
foo(a)
int a;
{
  int only_a_declaration = bar(a);
}
```

do not have routine instrumentation added.

10. Call Coverage

When a function call is made, its condition count is incremented. The point of instrumentation is the function name (which may be a complex structure, such as an element of an array of function pointers).

greport identifies both the function called and the function that makes the call:

```
"example1.c", line 9: call of callme (in caller) never made.
```

If the function name is complex, it is abbreviated as with multicondition coverage:

```
"example1.c", line 9: call of calltable[...] (in caller) never made.
```

Note that there is one condition count for all of calltable, not one for each element of the array (whose size may not be known at compile time).

Advanced Instrumentation

This chapter covers the following topics:

(1) How to gain more control over GCT's processing and over what it does with the instrumented source.

(2) A complete description of the mapfile and how it is edited to control **greport** and **gsummary** output. You need to understand this section if you want to suppress or ignore entire routines or files (instead of individual coverage conditions).

(3) How Expansion Kit 1 allows re-instrumentation of individual files, with the updating of edit and coverage information.

11. Controlling Instrumentation and Compilation

11.1. Controlling When the Logfile is Read or Written

Normally, the executable automatically reads the logfile on startup and writes it when it finishes. GCT enables this by inserting

```
gct_readlog("GCTLOG");
```

as the first statement in main. (The control file's logfile directive can be used to change the filename.) It also inserts

```
gct_writelog("GCTLOG");
```

before every call to exit() or abort(), and before every return from main().

If you want to write the log before calls to other routines, edit the GCT source file *gct-exit.c* and add entries to the Process_ending_routines list. You should also edit that file if you want the log read at other entry points.

gct_readlog and gct_writelog are defined in *gct-write.c*, which **gct-init** installs in the master directory. Normally, GCT compiles this file and links it into any executable it builds.

In some cases, such as instrumenting embedded programs, UNIX file I/O is inappropriate. To turn off insertion of both calls, use

```
(options -readlog -writelog)
```

These options may be applied to individual files as well as to all files. To turn off linking of *gct-write.c*, use

```
(options -link-log)
```

at the top level.

You are now responsible for getting the log from memory to disk some other way. The scenario "Instrumenting the UNIX Kernel (and Other Embedded Systems)" describes how to do that.

11.1.1. Miscellaneous Notes

gct-init does not overwrite *gct-write.c*. You can safely modify it.

gct_readlog actually only schedules the read. The on-disk log is left unread until gct_writelog is called, whereupon it is read, merged into the in-core log, and then replaced. See the "Simultaneous Execution and Locking" scenario for a case where this behavior is useful.

gct_writelog is only inserted before direct calls to exit() or abort(). If one of these is called through a function pointer, GCT will not notice it.

11.2. Capturing Instrumented Source

GCT normally calls the C compiler for you. You never see the instrumented source. Sometimes you want to. For example, you might want to use GCT on a UNIX system, then send the instrumented code to be compiled on some other operating system.

Another possibility: GCT normally passes flags given to it to the C compiler (unless the flag begins with "test-"). However, GCT will not pass down flags it does not recognize. Thus, invoking GCT as

```
gct -K -c test.c
```

will produce *test.o*; however, the C compiler will have been invoked without -K, as "cc -c *instrumented-file*.c". GCT will warn you that -K was unused. *Troubleshooting GCT* describes how to modify GCT to pass -K to the C compiler. If you are unable to do that, you must capture the instrumented source and compile it separately.

Whatever the reason, here are the steps you need to follow when handling intermediate source yourself:

(1) Place this line in the control file

```
(options -produce-object)
```

This instructs GCT to *replace* the original source with instrumented source. The existing source is saved in a subdirectory, *gct_backup*.

(2) Compile the file in two stages. You instrument, then compile the instrumented source.

```
% gct -K -c test.c
% cc -K -c test.c
```

(3) Compile and link two utility files that GCT normally handles invisibly for you:

```
% cc -c gct-ps-defs.c
% cc -c gct-write.c
% cc -o test.o gct-ps-defs.o gct-write.o
```

gct-ps-defs.c must be compiled after all instrumentation is complete.

(4) Restore the original source:

```
% grestore
```

Rarely will you do this by hand. Normally you'll add a target like this to your makefile:

```
first_gct:
	gct-init;
	# Replace the source.
	$(MAKE) all "CC=gct"
	$(CC) -c gct-ps-defs.c
	$(CC) -c gct-write.c
	# Compile the instrumented source and link the executable.
	$(MAKE) all "OBJ=$(OBJ) gct-ps-defs.o gct-write.o"
	grestore
```

We were able to avoid changing the rest of the makefile only because it used an $(OBJ) makefile macro to identify all the objects in the executable. We linked in *gct-ps-defs.o* and *gct-write.o* by adding them to $(OBJ). If your makefile doesn't use such a macro, I recommend you add one.

11.2.1. Miscellaneous Details

If you used (options -produce-object) compilation because of an unrecognized command-line option, GCT will still still warn you about the unrecognized option, just in case you mistyped an option that GCT *does* recognize. The warning has no effect on GCT’s behavior.

If the make fails, **grestore** will not be executed. This can be useful when diagnosing the problem. You should run **grestore** by hand before retrying the make. If you forget, GCT will refuse to reinstrument the already-instrumented files.

BEWARE: If you don’t do the grestore, be careful not to remove the files in the **gct_backup** directory. They may be the only uninstrumented versions. (When using this style of instrumentation, it is generally safer to copy the source before instrumenting it.)

If you are working in a hierarchical directory structure, GCT will create a **gct_backup** directory in each subdirectory. **grestore**, run in the master directory, knows about all these directories and will copy back all the files.

The produce-object option can be set at the file level, as well as the top level. Thus, certain files might be turned directly into object files, while others might produce instrumented source to be compiled later. In this case, you will almost certainly want to use (options -link) - see below.

11.3. Other Miscellaneous Control File Options

ignore Default: ON Set at: top level or file level

GCT may process a file for three reasons: to instrument it to measure coverage, to add code to read the log, and to add code to write the log. Since the latter two are done by default, GCT normally processes every file. Just prefacing a filename with a minus sign does not cause GCT to ignore it. It turns off instrumentation, but the code to read and write the log is still added.

If you want a file completely untouched, you could use this:

```
(options instrument)
(test.c (options -instrument -readlog -writelog))
```

For convenience, the ignore option at the file level has exactly the same effect:

```
(options instrument)
(test.c (options ignore))
```

The ignore option has a different interpretation at the top level. It is on by default. If turned off, with

```
(options -ignore)
```

it instructs GCT to process the file *even if* instrument, readlog, and writelog are turned off. GCT preprocesses the file and passes it unchanged to the C compiler. The only possible use for this option is to ensure that all source is handled by the same preprocessor.

To emphasize: the meaning of the ignore option differs at different levels. At the top level, it describes what to do as a default action if no processing is needed. When applied to a file, it means to do no processing, even if processing would otherwise be needed.

link Default: ON Set at: top level only

GCT normally produces object files and executables. If this option is turned off, GCT will only produce objects. If link is off and the makefile produces these calls:

```
gct  -c  lc.c
gct  -c  get.c
gct -o lc lc.o get.o
```

the final call does nothing. No executable **lc** will be produced; you'll have to do it yourself later. In this case, you must be sure to compile the file *gct-ps-defs.c* and link it into the executable. If the readlog and writelog options are set (the default), you must do the same for *gct-write.c*. This option is meaningful only if produce-object is on at the top level. If that option is turned off, no compiling (hence linking) is ever done.

(options -link) is exactly equivalent to calling GCT with the -test-nolink option.

link-def Default: ON Set at: top level only

As GCT instruments, it writes important definitions into the file *gct-ps-defs.c*. When GCT is to produce an executable, it silently compiles that file and links it into the executable. By turning this off, you prevent that behavior.

I can't think of a case where this behavior would be useful. It's provided just for symmetry with

-link-log (which, you will recall, prevents *gct-write.o* from being linked into the executable).

This option is meaningful only if produce-object and link are on.

12. Editing a Mapfile

The *Tutorial Introduction to GCT* and *gedit(1)* manpage describe how to add *line edits* to a GCT mapfile. You should read both of those before reading this section.

Line edits are used to suppress or ignore certain coverage conditions. You can also edit the mapfile directly to suppress or ignore larger groups of coverage conditions.

Here is a sample GCT mapfile:

```
!GCT-Mapfile-Version: 2.0
!Timestamp: Tue Aug 25 13:10:47 1992
!File: test.c -
!Routine: main -
!Checksum: 1018535575
!Instr-Checksum: 221876993
- 4 if
- 4 &
!File: test2.c -
!Routine: four -
!Checksum: 2108929743
!Instr-Checksum: 1472975722
- 3 routine four is never entered.
- 3 if
- 3 &
```

The first line identifies the mapfile. All the lines beginning with exclamation points are *header* lines; they describe parts of the source file. The lines beginning with dashes identify particular coverage conditions.

All **gedit** does is change a dash to either an 'S' or an 'I'. For example, the first dashed line represents the true case of an if on line 4. Were it suppressed, it would look like:

```
S 4 if
```

and **greport** output might look like:

```
"test.c", line 4: if was taken TRUE 1S, FALSE 0 times.
"test2.c", line 3: if was taken TRUE 1, FALSE 0 times.
```

Note the 'S' in the true case of the first line.

There are other dashes that can be changed. If you change the dash at the end of *test.c*'s !File header to 'I', thusly:

```
!File: test.c   I
```

then run **greport**, you'll see:

```
"test2.c", line 3: if was taken TRUE 1, FALSE 0 times.
```

All the coverage conditions for *test.c* are completely ignored.

Particular routines can be edited by changing the dash at the end of the !Routine header. To suppress everything in routine four, we’d make this edit:

!Routine: four S

greport alone would show no output because everything is suppressed. **greport -all** would show

"test2.c", line 3: routine four is never entered. [1S]
"test2.c", line 3: if was taken TRUE 1S, FALSE 0S times.

You might also see mapfile lines like this:

!Internal-File: test2.c -

All the mapfile entries after such a line come from an included file. You can suppress or ignore the entire contents of the included file by changing the dash. The end of an included file is marked by another !Internal-File or a new !File.

Whenever you change a dash, make sure you change it to a capital ’I’ or a capital ’S’. Lower-case characters are not acceptable.

12.1. Which Edits Take Precedence

The smaller the scope of an edit, the higher its precedence. If a file is ignored, but a routine within it is suppressed, **greport -all** will show output for that routine, but not for the rest of the file. If, however, **gedit** was used to ignore a particular line within the routine, that line edit would take precedence over the routine’s suppression. The line would never be shown, even with **greport -all**.

In addition to the above edits, replacing a dash with a ’V’ forces a line of output to be visible. It overrides an edit on a larger scope. Like ’I’ (but unlike ’S’), ’V’ applies to an entire line of **greport** output, not just a particular condition in a multi-condition line. (This is because you can’t make only part of a branch or loop line visible or invisible - you either see the whole line or none of it.)

You use ’V’ when you’re testing a particular feature spread out throughout the entire system. GCT, for example, is an addition to the GNU C compiler, with only a few changes to the original source. I measure the coverage of those changes by instrumenting the changed routines, Ignoring the entire routines, but making Visible the lines I changed.

Included files are considered to have a wider scope than functions. Though it’s possible to have an included file that’s entirely within a function, that’s rare. Overlap is also possible (especially in code written by parser generators), but this ordering still works well in practice.

12.2. Miscellany

Mapfile entries for loops look peculiar. The **greport** output for a do looks like:

"do.c", line 5: loop one time: 1, many times: 2.

but the mapfile entries look like:

- 1133 do-loop
- 1133 &
- 1133 &
- 1133 &

Why are there two entries in the output but four in the mapfile? The answer is "that’s an implementation detail". All you need to know is that the last three all sum to the "many times" number. You can suppress any of them to suppress that coverage condition. (**gedit** suppresses all of them, just for consistency.)

The **greport** output for a for or while looks like this:

"while.c", line 5: loop zero times: 1, one time: 1, many times: 2.

but it also has four mapfile entries. The second and third of them sum to the "one time" test condition.

What happens if you edit one of the "one time" loop entries to ignore and the other to suppress? If two edits apply at the same level, ’V’ takes precedence over ’I’, which takes precedence over ’S’, which takes precedence over ’-’. It’s hard to see why you would want to do this.

[NOTE: This behavior has an obscure, but perhaps marginally, useful effect when updating mapfiles. It may change in later releases. Do not depend on it.]

13. Incrementally Updating Mapfiles and Logfiles

This section is useful only to users of Expansion Kit 1.

To understand how incremental update works, a bit of detail on how GCT works is required:

(1) **gct-init** produces an empty mapfile and a starting version of *gct-ps-defs.c* and *gct-ps-defs.h*.

(2) As each source file is processed by GCT, information about it is appended to the end of the mapfile and *gct-ps-defs.c*. *gct-ps-defs.h* is kept updated with the size of GCT’s internal log.

(3) At any moment, a C compiler can compile *gct-ps-defs.c* and link that with the instrumented objects to form a working executable. The executable does not have to be created using GCT. Using GCT has the advantage that *gct-ps-defs.c* is compiled and linked for you.

If, after linking, GCT is used to reinstrument a file, its behavior is no different: information is appended. Now, however, there are two entries for that source file. Without Expansion Kit 1, relinking the executable would result in two sections of the log being devoted to that file. Since the old object file has been discarded, the old section will remain zero (and will be quite noticeable in **greport** output). Equally bad, any edits applying to the old version of the file will not be applied to the new one.

Expansion Kit 1 solves this problem. At link time, GCT calls **gct-remap** to update the mapfile. **gct-remap** finds any duplicate file entries, applies edits from older versions to the newest, then removes older versions.

13.1. Updating Edits

If you edited the old version of the mapfile, those edits might be carried forward to the new version:

(1) Edits to !Routine, !File, and !Internal-File header lines are always preserved.

(2) Line edits (made with **gedit**) are carried forward for any routine that hasn’t changed "too much".

"Too much" is defined later. For now, consider it to mean that the mapfile entries for the routine no longer describe the same instrumentation. For example, if branch coverage is being measured and the same branching constructs occur in the same order, the new version is compatible with the old and the old edits apply to the new routine. If a new if is added, the old edits are discarded. If an if is changed to a while, the old edits are discarded.

Changing

```
for (i = 0; i < 5; i++)
```

to

for (i = 0; i < **a_variable**; i++)

will not cause any edits for that for to be discarded. This is problematic. A "zero times" loop condition was impossible in the old version and could be suppressed. In the new version, it might very well be possible, so why should the suppression be retained?

Most of the time, changing source does not change impossibility. So a sensible approach is to retain the edits while the source is changing and not spend time reevaluating impossibility after every change. When the source has stabilized, reevaluate *all* the suppressed conditions one last time.

However, you can be more cautious if you desire. The following options provide finer control over GCT’s behavior. They can be set only at the top level of the control file; you cannot control behavior for individual files or routines.

remap-strict

If the remap-strict option is turned on, line edits apply to a new version only if a routine is completely unchanged (except for whitespace and comments). Edits to headers are still retained. These are usually used to ignore files or parts of files; generally you still want to ignore them. If you want to eliminate such edits, you have to do it by hand, or by using **gct-init** to reinstrument the whole program.

remap-forget

If you are truly cautious, you will be worried that any change might invalidate the suppression of any coverage condition anywhere, even in an unchanged file. (After all, such unexpected implications are what nasty bugs are made of.) In this case, setting the remap-forget option causes every line edit in the old mapfile to be dropped, even those for untouched files. Edits to headers are retained.

remap

By turning this option off, you can prevent incremental update. Turning off the link or produce-object options also prevents updating.

13.2. Manual Updating

While **gct-remap** is normally called by GCT, it can also be invoked manually. Suppose you are using (options -produce-object), where the instrumented source replaces the original source. Here is how you must build a system after reinstrumenting a file:

```
gct -c test.c
cc -c test.c
gct-remap
cc -o exec *.o
```

By default, **gct-remap** operates on *gct-map* in the current directory. The usual command-line options can be used to change file and directory names; see the manpage. Also, the remap-strict and remap-forget control file options also have their own command-line equivalents.

13.3. Messages from gct-remap

gct-remap tells you when edits are dropped. Since it is normally invoked by GCT, the messages will appear to come from GCT. You can also make **gct-remap** tell you which edits were preserved by putting (options remap-verbose) in the control file.

13.4. The Definition of a Major Change

This definition will be in terms of what greport -all would display for a routine. There has been a major change if any of the following are true:

(1) A report for the revised routine would have more or fewer lines.

(2) The first word of the line's text has changed, such as if

"lc.c", line 153: **while** was taken TRUE 0, FALSE 29 times.

changed to

"lc.c", line 153: **if** was taken TRUE 0, FALSE 29 times.

(3) For relational coverage, a change in the operator, such as from

"lc.c", line 137: operator > might be >=. (L==R)

to

"lc.c", line 137: operator < might be <=. (L==R)

(This also applies to weak mutation coverage.)

(4) For multicondition coverage, a change in the number of conditions, such as from

if (A && B)

to

if (A && B && C)

(5) For loop coverage, changing to a do loop from another type of loop, or changing from a do loop.

(6) For call coverage, changing the name of the called routine.

These are the *only* changes that count as major changes. Changes to a while's test, no matter how severe, are not a major change. (They may be a major change if relational or multi-condition coverage are used, but only because of their effect on other **greport** lines.) Similarly, there is no major change to multicondition coverage as long as the same number of conditions are being measured. That is, changing

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

to

```
if (C || B)
```

is not a major change. Finally, if only loop coverage is being measured, changing from a for loop to a while loop is not a major change. (Branch coverage would count this as a major change.)

Note that measuring a new type of coverage will usually be a major change, even if there's been no change to the code.

13.5. Miscellaneous Notes

Any number of reinstrumentations of a file may be made before **gct-remap** is used. In particular, this sequence is harmless:

> You make an instrumented system.
> You edit *file.c*.
> gct -c file.c fails with a compile error.
> You fix the problem.
> gct -c file.c works.
> gct -o exec *.o updates the mapfile

The result is the same as if you'd not gotten the compile error.

14. Updating the Logfile

Once the mapfile has been updated, **greport** and **gsummary** will refuse to use the old logfile, which no longer corresponds to it:

```
% greport GCTLOG
The mapfile and logfile come from two different instrumentations.
The mapfile comes from one begun on Tue Aug 25 15:57:07 1992.
The logfile comes from one begun on Tue Aug 25 15:55:53 CDT 1992.
```

Further, if gct_readlog is used by the executable, it will print this :

```
% a.out
gct logging routines: Warning: logfile GCTLOG is out of date.
gct logging routines: The executable is from 'Thu Nov 5 08:59:37 1992'.
gct logging routines: The logfile is from 'Thu Nov 5 08:58:36 CST 1992'.
gct logging routines: The logfile will not be updated.
```

The program will continue to run, but the logfile will not be overwritten (to avoid losing valid information).

Often, it makes sense simply to delete the logfile and start over. But, for large systems, you may want to preserve old coverage information and continue testing. Finding out if coverage has been invalidated by changes is most efficiently done after the bulk of the changes have been made, not as each new change is made.

gct-newlog takes an old logfile and mapfile and produces a new logfile. It also uses the new mapfile, which is, as usual, assumed to be *gct-map*.

```
% gct-newlog gct-map.old GCTLOG.old > GCTLOG.new
% greport GCTLOG.new
"test.c", line 4: if was taken TRUE 1, FALSE 0 times.
"test.c", line 11: if was taken TRUE 0, FALSE 0 times.
```

gct-newlog uses the same rules for preserving logfile entries as **gct-newmap** uses for mapfile edits:

(1) By default, a routine’s entries are carried forward if the instrumentation structure is unchanged. Otherwise, they are set to zero.

(2) If the -strict option is given, entries are carried forward only for unchanged routines.

(3) There is no -forget option; the same effect is had by deleting the logfile.

gct-newlog requires that you save a copy of the mapfile before updating it. If you forget, you will be glad to know that **gct-remap** saves a backup copy of the mapfile it updates in *gct-mapfile.gbk*. This mapfile has duplicate files appended to the end, but **gct-newlog** will ignore them in order to reconstruct the original mapfile.

Be careful, though: if **gct-remap** is called twice before **gct-newlog** is called, the original mapfile will be lost forever and you’ll see:

```
testing-112% gct-newlog gct-map.gbk GCTLOG
The mapfile and logfile come from two different instrumentations.
The mapfile comes from one begun on Tue Aug 25 15:57:07 1992.
The logfile comes from one begun on Tue Aug 25 15:55:53 CDT 1992.
```

Scenarios

These scenarios describe specialized uses of GCT. Although some will seem irrelevant, I suggest you read them all, since they may mention broadly applicable GCT features that weren't covered in the *Tutorial*.

15. Multiple Executables in One Instrumentation

Suppose that you have a single makefile that produces multiple executables. If the executables are completely independent - share no source - it's quite reasonable to instrument them independently:

```
% gct-init -test-map map-1
% make "CC=gct -test-map map-1" executable1
% gct-init -test-map map-2
% make "CC=gct -test-map map-2" executable2
```

Suppose, however, that **executable1** and **executable2** share source. Each executable might have its own test suite, and you'll want to know how thoroughly each suite exercises its unique source. But you'll probably also want to know how well both test suites exercise the shared files.

The most convenient approach is to

(1) instrument all the executables at the same time, so that they all use the same logfile.

(2) use the -visible-file (or -vf) option to **greport** and **gsummary** to select subsets of the logfile.

15.1. Initial Instrumentation

The first instrumentation of the program is exactly as you'd expect:

```
% gct-init
% make "CC=gct" executable1 executable2
```

We may now run the executables and see what they do:

```
% executable1
Noargs called

% greport -all GCTLOG
"noargs.c", line 3: routine noargs is never entered.  [1]
"someargs.c", line 3: routine someargs is never entered.  [0]
"executable1.c", line 4: routine main is never entered.  [1]
"executable2.c", line 5: routine main is never entered.  [0]

% executable2
Noargs called
Someargs called

testing-187% greport -all GCTLOG
"noargs.c", line 3: routine noargs is never entered.  [2]
"someargs.c", line 3: routine someargs is never entered.  [1]
"executable1.c", line 4: routine main is never entered.  [1]
"executable2.c", line 5: routine main is never entered.  [1]
```

If we only wanted to look at coverage of the shared routines, we'd type this:

```
% greport -vf noargs.c -vf someargs.c -all GCTLOG
"noargs.c", line 3: routine noargs is never entered.  [2]
```

```
"someargs.c", line 3: routine someargs is never entered.  [1]
```

To get a file-by-file summary of only the shared files, we'd type

```
% gsummary -vf noargs.c -vf someargs.c -files GCTLOG
noargs.c      100=ALL 100=ROUT 1#
someargs.c  100=ALL 100=ROUT 1#
TOTAL       100=ALL 100=ROUT 2#
```

The first number is the percent total coverage, the next the percent routine coverage, and the final the number of coverage conditions in the file. For more information, see the **gsummary** manpage.

In a large system, these calls to **greport** and **gsummary** would be via shell scripts, to reduce typing. The shell scripts can be automatically generated from **nm(1)** output.

15.2. Warnings when Executing Intermediate Executables (Expansion Kit 1)

If the makefile builds executables which are then executed to create C files, and those *intermediate executables* are instrumented, you may see this message during the initial instrumentation:

```
gct logging routines: Warning:  logfile GCTLOG is out of date.
gct logging routines: The executable is from 'Thu Nov  5 08:59:37 1992'.
gct logging routines: The logfile is from 'Thu Nov  5 08:58:36 CST 1992'.
gct logging routines: The logfile will not be updated.
```

This is harmless, unless you want to measure the coverage attained by executing the intermediate executable. That's rare, but if you do, see the worst-case scenario described below.

15.3. Updating Multiple Executables (Requires Expansion Kit 1)

Since all the executables share a single logfile, all of them must be relinked whenever one of them is changed. Therefore, the executables must be deleted before any file is reinstrumented:

```
% rm executable1 executable2
```

Further, all the source must be compiled before any executable is linked. (If an executable was linked, then another source file compiled, then another executable linked, the second executable would be incompatible with the first - they cannot share the same logfile.) The -test-nolink option does this:

```
% make "CC=gct -test-nolink" executable1 executable2
```

A final step relinks everything:

```
% make "CC=gct executable1 executable2
```

Other than these extra steps, the procedure is the same as for single executables, as described in *A Tutorial Introduction to GCT*.

15.4. A Worst-Case Scenario for Updating Multiple Executables

The very worst case for incremental updates is when

(1) Your makefile builds multiple executables.

(2) Some of these executables are used to build source files, which are compiled later to form other executables.

(3) All these executables - including the intermediate executables - share source you want instrumented.

(Note: this scenario presents no problems for the initial complete instrumentation, just for updates.)

If you want to get every last iota of coverage information, including coverage yielded during the system build, the process is complicated. First, save existing coverage information:

```
% mv GCTLOG GCTLOG.pre-build
% cp gct-map gct-map.pre-build
```

Now, make the intermediate executables (here called **inter1** and **inter2**):

```
% make "CC=gct -test-nolink" inter1 inter2
% make "CC=gct" inter1 inter2
```

The mapfile has now been updated. Use those executables to build source files and compile those files (along with any others needing compilation):

```
% rm executable*
% make "CC=gct -test-nolink" executable1 executable2
```

Save the information about how the intermediate files were used when building sources:

```
% mv GCTLOG GCTLOG.inter
% cp gct-map gct-map.inter
```

Now link the final executables, updating the mapfile again:

```
% make "CC=gct" executable1 executable2
```

Finally, create a new log file that incorporates all saved information:

```
% gct-newlog gct-map.pre-build GCTLOG.pre-build > GCTLOG.P
% gct-newlog gct-map.inter GCTLOG.inter > GCTLOG.I
% gmerge GCTLOG.P GCTLOG.I > GCTLOG
```

If you don’t need to measure the coverage when running intermediate executables, simply do this:

```
% mv GCTLOG GCTLOG.pre-build
% cp gct-map gct-map.pre-build
% make "CC=gct" inter1 inter2
% rm executable*
% make "CC=gct -test-nolink" executable1 executable2
% make "CC=gct" executable1 executable2
% gct-newlog gct-map.pre-build GCTLOG.pre-build > GCTLOG
```

Because you’re allowing the intermediate executables to become out of date with each other, they’ll print warnings about not updating the logfile. Since you don’t care about the contents of the logfile, those warnings can be ignored.

15.5. An Alternate Approach (requires Expansion Kit 1)

gct-newlog can also be used to combine information from multiple executables. Consider GCT Expansion Kit 1, which shares some files with GCT. When both GCT and Expansion Kit 1 are instrumented, their separate logfiles contain information about coverage of the file *g-tools.c*. For example, here’s a terse summary of its use in GCT:

```
% gsummary -f -vf g-tools.c /usr/tmp/GCTLOG
g-tools.c      75=ALL  79=BR  78=SW  62=LP  85=ML  50=< 140#
TOTAL          75=ALL  79=BR  78=SW  62=LP  85=ML  50=< 140#
```

and here’s a summary of its use in Expansion Kit 1:

```
% gsummary -f -vf g-tools.c /usr/tmp/KITLOG
g-tools.c      38=ALL  36=BR  44=SW   8=LP  75=ML   0=< 109#
TOTAL          38=ALL  36=BR  44=SW   8=LP  75=ML   0=< 109#
```

You might notice that the condition counts are different for the two summaries - this is because some files are conditionally compiled out of the Kit 1 version.

gct-newlog can be used to convert Kit 1’s logfile to a form matching GCT’s. Almost all the coverage data will be inapplicable - except that from the routines the two programs share, which are the routines we’re interested in. This is done as follows:

```
% gct-newlog -v ../../kit1/src/gct-map /usr/tmp/KITLOG > NEWLOG
gcc.c is new; its entries will be zeroed.
gct-contro.c is new; its entries will be zeroed.
[Many more messages like this.]
Routine add_file_external_edit (in g-tools.c) is new; its entries will be zeroed.
Routine routine_external_edit (in g-tools.c) is new; its entries will be zeroed.
Routine file_external_edit (in g-tools.c) is new; its entries will be zeroed.
```

The messages are about files and routines unique to GCT; since they weren’t in the Kit 1 mapfile, they’re zeroed. The last three lines identify those routines that were compiled out of Kit 1’s *g-tools.c*.

Next, we merge the GCT version and the Kit 1 version to get the combined summary:

```
% gmerge NEWLOG /usr/tmp/GCTLOG > MERGED-LOG
% gsummary -vf g-tools.c -f MERGED-LOG
g-tools.c      77=ALL  82=BR  78=SW  62=LP  85=ML  58=< 140#
TOTAL          77=ALL  82=BR  78=SW  62=LP  85=ML  58=< 140#
```

We conclude that Kit 1 hasn’t yet exercised *g-tools.c* very differently than GCT. We could get a detailed list of what Kit 1 does that GCT doesn’t do by typing the following:

```
% gnewer /usr/tmp/GCTLOG MERGED-LOG | greport -n
```

You must beware of two pitfalls:

(1) The two executables really need to be built in different directories. Although you could give the mapfiles and logfiles different names, there is (in GCT 1.4) no way to change the name of *gct-ps-defs.c* and other auxiliary files. Keeping separate instrumentations of two executables in the same directory would require careful juggling of the sort that almost never works.

(2) **gct-newlog** requires that files have the same name in both mapfiles. If, for example, GCT’s mapfile used *g-tools.c* and Expansion Kit 1’s mapfile used *../../gct/src/g-tools.c*, the data for the two files would not be combined. This could be solved by editing one of the mapfiles before combining. (*g-tools.c* is copied to Kit 1’s directory, so the problem does not arise.)

16. Simultaneous Execution and Locking

Suppose that an instrumented executable forks another instrumented executable that uses the same logfile, waits for the child to finish, then continues. The default implementations of gct_writelog() and gct_readlog() are coded such that no information is lost.

This only works, though, because the parent process does nothing while the child writes the logfile. If two independent processes share the same logfile and try to update it at the same time, the logfile will likely be corrupted.

To avoid this problem, search for the string "Locking" in *gct-write.c* (created by **gct-init**). The following text describes how to turn on POSIX-compliant locking of logfiles. Locking may be turned on for a particular program, or the GCT installer may make it the default.

If you turn on locking in a private copy of *gct-write.c*, you are safe because **gct-init** will not overwrite the file. Beware of using **gclean**, though, which will remove *gct-write.c*. For an example of ensuring that locking stays turned on, see the **instrument** script in the GCT source.

17. Instrumenting Libraries

Instrumenting a library to test it in isolation is no different from any other program. Sometimes, though, you’d like to instrument a library, link the instrumented library into all the programs that use it, then find how the complete suite of programs exercises the library. This problem is very similar to instrumenting multiple executables.

If possible, have the executables write the logfile. Do this by "instrumenting" them using an empty control file. This will add gct_readlog and gct_writelog calls, but will otherwise not change the program.

Another possibility is to tell GCT that all entry points to the library are "main" routines. If you do this, all entry routines will call gct_readlog when they are called, then call gct_writelog before they return. This will probably make your executables run slowly. (Note: there is no problem with library routines that call entry points to the same library, except that the executables will be even slower.)

Adding new entry points is clumsy: you have to edit the routine gct_entry_routine in the GCT source file *gct-exit.c* and remake GCT.

No matter how you instrument libraries, you should guard against simultaneous access to the logfiles.

18. Instrumenting the UNIX Kernel (and Other Embedded Systems)

The UNIX kernel and other embedded systems cannot use *gct-write.c* to write the in-core log to a file. Therefore, the control file should include

```
(options -readlog -writelog)
(options -link-log)
```

You must get the in-core log to disk some other way. The log is an array named Gct_table, whose size in bytes is given by Gct_table_size. The declarations and definitions can be found in *gct-defs.h* and *gct-ps-defs.c* respectively.

(1) If you’re instrumenting a UNIX kernel, the log can be read from */dev/kmem* using a program that is linked with the files *gct_write.c* and *gct-ps-defs.c*. The important part of that program would look like this:

```
struct nlist gct_nlist[] =
{
   {"Gct_table"},
   {0},
}

nlist(argv[0], gct_nlist);
fd = open("/dev/kmem", O_RDONLY)
lseek(fd, gct_nlist[0].n_value, SEEK_SET);
read(fd, Gct_table, Gct_table_size);
gct_writelog("GCTLOG");
```

(2) If your embedded system communicates over a network to a host, you might use a call like this:

```
result = write(cnn, Gct_table, Gct_table_size);
/* Other end is a program that calls gct_writelog */
```

where cnn is an open network connection.

Use the **gmerge(1)** program to combine logs from different runs into a single cumulative logfile.

When testing embedded systems, you sometimes want to separate the coverage achieved by booting the system from the coverage of a particular test. The **gnewer(1)** program can help you do that.

19. Instrumenting the Same File Twice in One Executable

In a few rare cases, a makefile may compile the same file twice. The output from such a build would look like:

```
cc -DFIRST_WAY -o first.o file.c
cc -DSECOND_WAY -o second.o file.c
cc -o executable first.o second.o
```

If Expansion Kit 1 is not installed, this will work. **greport** will produce output for both of the files, both tagged with "file.c". (You will have to decide which set of lines is due to which object file.)

If Expansion Kit 1 is installed, it will not work. To GCT, this case is indistinguishable from instrumenting *file.c* once, changing the code, then reinstrumenting - in which case you want the second instrumentation to supersede the first. You can use (options -remap) to turn off remapping; however, this means that you must reinstrument all files whenever you change one of them. It would be better to change the makefile so that it did this:

```
cp file.c first.c
cc -DFIRST_WAY -o first.o first.c
cp file.c second.c
cc -DSECOND_WAY -o second.o second.c
cc -o executable first.o second.o
```

Note: it is dangerous to make *first.c* a link to *second.c*. GCT identifies files in the control file not by file name but by true file identity (inode number, for those who know UNIX details). Therefore, a control file entry for *first.c* would would match either of the files, as would a control file entry for *second.c*. If there are entries for both *first.c* and *second.c* in the control file, it is undefined which will apply. (Currently, it's the last one, but don't depend on that.)

Appendix A: Implementation Restrictions

GCT should be portable to any UNIX system derived from either Berkeley 4.X UNIX or System V Release 3 (or later). It has also been ported to Apollo Sr10.3. The tool was implemented by modifying the GNU C compiler (version 1.37) so that it produces new C code instead of object code. Any C language text accepted by that compiler can be instrumented, except for these restrictions:

- GCT reserves to itself all global identifiers beginning with "Gct_" or "GCT_". It also uses "gct_readlog", "gct_writelog", "_G", and "_G2".

Appendix B: GCT Options

20. Controlling Modification of the File

instrument Default: OFF Set at: top level, file level, or routine level

When ON, add coverage instrumentation to all files, a particular file, or a particular routine. In addition to the options keyword, this is also turned on by naming a file or routine, or turned off by prefacing the name with a minus sign.

readlog Default: ON Set at: top level, file level, or routine level

When ON, add a call to gct_readlog as the first statement in main().

writelog Default: ON Set at: top level, file level, or routine level

When ON, add a call to gct_writelog before exit(), abort() and any return from main().

ignore Default: ON Set at: top level or file level

When ON at the top level, do no processing of files where instrument, readlog, and writelog are all OFF. By default, this means such files are simply handed to the C compiler; if produce-object is OFF, they are left untouched. When OFF at the top level, such files are processed. Although no code is added by GCT, the files are run through GCT's C preprocessor.

At the file level, turning this option ON causes the file to be ignored, regardless of the values of instrument, readlog, and writelog. It is simply compiled. Turning it OFF forces the file to be processed.

macros Default: OFF Set at: top level or file level

Normally, the contents of macro expansions are not instrumented. If this option is ON, they are.

instrument-included-files Default: OFF Set at: top level or file level

Normally, the contents of included files are not instrumented. If this option is ON, they are.

enum-relational Default: ON Set at: top level, file level, or routine level

A few compilers will produce error messages when given the code that results from instrumentation of relational operators involving enumerated types. Turning this option off prevents GCT from instrumenting such relational operators. See also *GCT Troubleshooting*.

21. Controlling Treatment of the Instrumented File

produce-object Default: ON Set at: top level or file level

If ON, the file, once instrumented, is passed to the C compiler (either **cc** or the -test-cc argument). The result of a call to GCT is an instrumented object file. If OFF, the file, once instrumented, *replaces* the original source, which is saved in the *gct_backup* directory for later restoration by **grestore**.

replace Default: ON Set at: top level or file level

This debugging option is only relevant if produce-object is OFF. It causes the instrumented output for *file.c* to be placed in T*file.c*. This is convenient if you want to see what instrumented source looks like without having to worry about using **grestore**. The file must be in the master directory.

22. Types of Instrumentation

These are not strictly options, but they’re turned on and off the same way, and it’s convenient to list them here.

branch	Default: OFF	Set at: any level
multi	Default: OFF	Set at: any level
loop	Default: OFF	Set at: any level
routine	Default: OFF	Set at: any level
relational	Default: OFF	Set at: any level
call	Default: OFF	Set at: any level
race	Default: OFF	Set at: any level

Race coverage; discussed in *Using Race Coverage with GCT*.

operator Default: OFF Set at: any level
Weak mutation coverage; discussed in *Using Weak Mutation Coverage with GCT*.

operand Default: OFF Set at: any level
Weak mutation coverage.

23. Controlling Linking

link Default: ON Set at: top level only

Normally, GCT will link an executable just like any C compiler would. If OFF, GCT only instruments. The -test-nolink option to gct may be more convenient.

link-log Default: ON Set at: top level only

Link in code to read and write the log file. Turn this off for embedded systems.

link-def Default: ON Set at: top level only

Link in the definition of GCT’s log.

24. Controlling the Logfile

This is not an option, but it is convenient to include it in this appendix.

logfile Set at: top level only

The argument is the name of the logfile. The default is *GCTLOG*. Don’t put quotes around the argument. Example:

```
(logfile /usr/tmp/GCTLOG)
```

25. Incremental Update

These options are meaningful only if Expansion Kit 1 has been installed.

remap Default: ON Set at: top level only

Incrementally update the map file at link time. (This option causes GCT to call the **gct-remap** program.)

remap-strict Default: OFF Set at: top level only

Normally, **gedit’s** line edits are forgotten only if the instrumentation structure of a routine has changed. If ON, line edits are forgotten if any change has been made to a routine.

remap-forget Default: OFF Set at: top level only

Forget all line edits in the previous version of the mapfile.

remap-verbose Default: OFF Set at: top level only

Pass the -v option to **gct-remap**.

26. Weak Mutation Options

Some options apply only to weak mutation coverage. They are documented in *Using Weak Mutation Coverage with GCT*.

27. Debugging Options

Some other options are useful when debugging GCT. They are documented in the source file *gct-opts.def*.

Appendix C: Reporting Bugs

I can spend only a limited amount of time on unpaid GCT support. Please help me spend that time productively by making your bug reports as useful as possible.

Check *GCT Troubleshooting*; you may find a solution to your problem.

As a first resort, send bug reports to me (marick@cs.uiuc.edu) in preference to the mailing list.

For a bug in the documentation, include the document version number and GCT version number (both of these are on the title page).

For a software bug, please do the following:

- Tell me what machine and version of UNIX you’re using. What argument was given to **config.gcc**?
- Include a complete input file and control file. If possible, make the input file a preprocessed input file (by using ’gct -E source.c > FILE-TO-SEND.c’). This will reduce the chance that I won’t be able to reproduce the bug because it depends on the contents of system include files.
- Rerun your example with that input file, and use ‘gct -v’ to cause GCT to print out its version number and a detailed log of what it did.
- Describe what is wrong with GCT’s behavior (even if it’s obvious).
- Include your email address in the body of the bug report.
- Put your phone number in the bug report, in case return mail to you fails.

More information, such as stack backtraces and so on, can wait. I’ll ask you for them if I need them.

Bibliography

[Beizer83]
Boris Beizer. *Software Testing Techniques*. New York: Van Nostrand Reinhold, 1983.

[Howden78]
W. E. Howden. ’An Evaluation of the Effectiveness of Symbolic Testing’. *Software - Practice and Experience*, vol. 8, no. 4, pp. 381-398, July-August, 1978.

[Marick91]
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.

[Marick92]
Brian Marick. *The Craft of Software Testing* and *Test Condition Catalog*, Testing Foundations, 1992.

[Myers79]
Glenford J. Myers. *The Art of Software Testing*. New York: John Wiley and Sons, 1979.

Table of Contents