GCT Troubleshooting

Brian Marick
Testing Foundations

Documentation for version 1.4 of GCT
Document version 1.6

This manual gives solutions to common GCT problems. If something's missing, tell me (at the address on next page), so that I can improve the next version of this document.

Troubles specific to weak mutation (operand and operator) coverage are handled in *Using Weak Mutation Coverage with GCT*.

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.

If you have a problem with GCT, search for the symptom (in bold, below), then read the following text. More than one symptom may match your problem.

1. Trouble Upgrading to Version 1.4

What happened to (options files)?

(options instrument) now plays roughly the same role.

gct-write.c does not compile

The default versions of *gct-write.c* and *gct-defs.h* have changed in GCT 1.4. Because a few users customize them, **gct-init** does not normally remove them. You have to delete them by hand before instrumentation.

Incremental update erases the old mapfile

(This only applies to Expansion Kit 1.) Reinstrumenting a file may not update the mapfile. It may instead produce a mapfile with only a single entry, the reinstrumented file. The cause is that you've called **gct-init** before the reinstrumentation, likely because you forgot to change an old Makefile. **Gct-init** should only be called before the first, full, instrumentation.

Routines in included files are no longer instrumented

Use (options instrument-included-files) if you want this behavior.

2. Trouble Instrumenting your System

Files not instrumented

Is the file in the control file?
If you are using a makefile, is the source file newer than the object file?

Control file not found with multi-directory makefiles

Have you given the -test-dir argument to **gct**?

Is the argument correct? When using ‘pwd‘ in your makefile to find the current directory, you must make sure it is evaluated at the right time. For example, if your makefile does not call itself, but rather changes its directory and calls another makefile, the following will *not* work:

```
gct:
        gct-init;
        cd subdir; $(MAKE) all "CC=gct -test-dir ‘pwd‘"
        ...
```

pwd is not evaluated until after the **cd**. A correct solution is

```
gct:
        gct-init;
        MYDIR=‘pwd‘; cd subdir; $(MAKE) all "CC=gct -test-dir $$MYDIR"
        ...
```

(Note the double dollar signs, which prevent **make** from evaluating "$M" as one of its variables.)

Files unexpectedly instrumented

When a file is actually a link (symbolic or otherwise) to another file, the control file options applying to the second file may also apply to the first. In particular, if the second file appears in the mapfile, but the first does not, the first file will still be instrumented. This is side-effect of how GCT determines whether a file named in its command line appears in the control file. The only solution is to make the two files copies, not links.

gct-init: gct-init: No such file or directory

You may see a message like this when calling **gct-init** from a makefile. If so, your version of **make** has a bug. Add a semicolon after the call to **gct-init**.

#pragmas within functions are dangerous.

All pragmas in the original source are copied into the instrumented source. Instrumentation rearranges code within functions. If the pragma is within a function and is intended to apply to a following token (or expression, or statement), that token might have been moved. Some other token might follow the pragma. The original token might even have been moved before the pragma.

Instrumentation never rearranges code outside of functions, so most pragmas are safe.

Undefined symbols: Gct_per_file_table_pointer_7

The linker may print errors about these undefined symbols if you are not using Expansion Kit 1 and attempt to reinstrument and relink a single file (perhaps after a compile error). Without the Expansion Kit, you must remove all instrumented object files, call **gct-init**, and reinstrument the entire system whenever any file changes.

You may also see this error if you have a set of object files that you've forgotten are instrumented and try to use them as if they were ordinary object files. If you're building an uninstrumented system, make sure to clean up instrumented object files first.

gct finds compile errors.

GCC, the compiler that GCT is based on, sometimes finds legitimate errors that other C compilers miss. You may have to correct the source.

Incorrect installations are another source of compile errors. Here are examples:

(1) If GCT is configured to expect an ANSI C compiler, but your C compiler is a "traditional" C compiler, programs that use comments for token concatenation may result in errors.

(2) If GCT is configured to expect an ANSI C compiler, but your C compiler is a traditional C compiler, ANSI function prototypes will cause syntax error messages.

(3) Compile errors that refer to `__builtin_alloca` are a sign that GCT has been configured to expect that the GNU C compiler will be used to compile the instrumented code (but some other compiler was actually used).

The installation instructions describe how to configure GCT for particular types of C compilers. (This is done by modifying a shell script template.)

Non-standard constructs cause compile errors.

If you are writing in an extended dialect of C, GCT will probably complain when you use the extensions. Pragmas can sometimes be used as a workaround. Suppose the problem is caused by embedded assembly language statements.

(1) Use `(options -produce-object)` to prevent GCT from compiling the instrumented code.
(2) Preprocess the source to preface each of those lines with a #pragma; GCT will pass the whole line through unchanged.
(3) Postprocess the instrumented source to remove the #pragmas.
(4) Compile the instrumented source.

unrecognized option -*x*

GCT issues this warning whenever it discovers a command line option it knows nothing about. If the `produce-object` control file option has been turned off, the warning can safely be ignored. If, however, GCT invokes a C compiler, the unknown option is *not* passed to that C compiler. To get the option to the C compiler, you could turn off `produce-object` and compile the file separately; however, a more permanent solution is to add the option to the list of options GCT knows about.

tm.h in GCT's source directory is a link to a file in the *config* subdirectory. Suppose your option is

-K. Add the following line to that file:

#define GCT_COMPILER_SPEC "%{K}"

If the switch takes an argument, you must also define a macro called SWITCH_TAKES_ARG. To see how to define it, look for that macro's definition in *gcc.c*. Just copy it into your configuration file and add a new entry.

If the switch is more than one character long, you instead define a macro called WORD_SWITCH_TAKES_ARG. Its default definition is also in *gcc.c*.

Once you have made these changes, remove *gcc.o* and *gct.exec* and remake.

A note on arguments: GCT expects space between switches and their arguments. Thus, if you add -K as described above, GCT must be invoked as

```
gct -K arg file.c
```

not

```
gct -Karg file.c
```

If you normally call the compiler with the latter form, define GCT_COMPILER_SPEC as "%{K*}. This tells GCT to pass all options beginning with 'K' to the compiler. Do not add 'K' to the SWITCH_TAKES_ARG macro, since the argument is part of the switch. Note that you will not be able to use the "-K arg" form.

Compilation errors on comparisons of enumerated types

If relational coverage is on and GCT is given code with comparisons of enumerated types, like

```
enum numbers {one , two } number1, number2;
...

number1 < number2;
```

GCT produces instrumentation that is not accepted by a few C compilers. If your compiler is one of these, you will see a compile error when compiling your instrumented program. Add (options -enum-relational) to your control file. It will prevent instrumentation of relational operations like the above. Relational operations not involving enumerated types will be instrumented as usual.

To GCT, enumerated constants look just like integers. Even if (options -enum-relational) is set, GCT will instrument code like

```
one < 5;
```

(Here we are using an enumerated constant, not a variable.) This may be a problem, depending on your C compiler. If it is, turn off relational instrumentation for that routine.

***Initialization*:1: warning: __GNUC__ redefined**

This warning is printed if you're using version 2 of the GNU C compiler, a file is not instrumented, and you use (options -writelog -readlog). You can ignore it - __GNUC__ is given the correct value.

Logfile warnings from intermediate executables

By default, GCT adds calls to gct_readlog and gct_writelog to every source file it processes, even if it adds no instrumentation. If your makefile builds intermediate executables, and you use (options instrument), those executables will create logfiles and may print out messages about logfiles being out of date. These messages are harmless. To prevent them - and prevent logfiles from being created - use

```
(intermediate.c (options ignore))
```

It is not sufficient merely to use

```
-intermediate.c
```

That only turns off instrumentation.

Wrong include files found with GNU C, Version 2

In its default configuration, version 2 of the GNU C compiler searches */usr/local/include* before */usr/include*. Version 1 searches in the other order. Other compilers do not search */usr/local/include* at all. GCT behaves like GCC Version 1, making it compatible with the latter two cases, but not with GCC Version 2. If your program depends on GCC Version 2's behavior, you will have problems. There is no good solution. Perhaps the least bad is to change your makefile to compile with

```
$(CC) -I/usr/local/include file.c
```

Another solution is to edit the include_defaults array in GCT's source file *cccp.c*; make it match GCC Version 2's behavior.

GCT dumps core after a parse error

In some cases, GCT's reaction to a syntax error is to print an error message and then later dump core. This is because the original GCT design assumed that it would be given only syntactically correct programs; this assumption lingers on in some places. The solution is to fix the syntax error and then use GCT again.

3. Trouble Instrumenting your System (options -produce-object)

You can ignore this section if you didn't use (options -produce-object) in the control file.

Make fails on non-compilation commands

You normally use GCT like this:

```
make all "CC=gct"
```

In some cases, the actions performed by the makefile fail unless $(CC) creates an object or executable file. A typical example is a target like this:

```
all: $(OBJ)
   $(CC) -o temp $(OBJ)
   mv temp final
```

The **mv** fails because *temp* is not created. In many versions of **make**, you can use **make -k** to print error messages and continue. This also ignores legitimate errors, so examine the make log carefully.

In some cases, the recursive call can be preceded by a minus sign:

```
-$(MAKE) all "CC=gct"
```

but in other cases the makefile must be changed more extensively to put minus signs in front of many non-compilation actions.

4. Errors in greport output

No Logfile written

If you've used (options -writelog), does the program call gct_writelog before exiting?

Deleted file still in greport output

This is only applicable to Expansion Kit 1, which allows incremental updates. Suppose you remake an executable by deleting some files and recompiling others. GCT cannot tell the difference between a file not used in the executable and one simply not recompiled; therefore, lines about the deleted file will appear in **greport** output. The solution is to edit the mapfile to ignore that file's !File: header.

This can also happen in the following way. Your control file originally looks like this:

```
(options -readlog -writelog)
file.c
other-files.c
```

(It is unusual for a control file to turn off readlog and writelog; it is usually only done on embedded systems.) If you now remove file.c from the control file and recompile it, even through GCT, a bug prevents the mapfile's old entries for that file from being removed. Edit the mapfile to ignore the whole file.

Coverage is unbelievably low

(This only applies if you do not use Expansion Kit 1.) The same file may appear twice in the mapfile. The first instance never records any coverage, so many spurious entries appear in **greport** output (and make the **gsummary** percentages low). This happens if you reinstrument a file without using **gct-init**. Without Expansion Kit 1, you must use **gct-init** before reinstrumenting any file. You must also reinstrument any other files that were previously instrumented.

File appears twice in greport output

See the previous item.

No coverage at all for a particular file

See the previous item.

5. Trouble Working with greport output

Test causes no increase in coverage

If you've used (options -writelog), does the program call gct_writelog() before *every* call to exit()? Does it call gct_writelog() at the end of the main() routine?

Does the program call abort() or similar routines? If so, you need to call gct_writelog() before them as well. The user's manual describes how to make GCT do this automatically; you can also do it manually.

Lines about included files are not in greport output

By default, GCT only instruments routines in the original source file, not in files it includes. Use (options instrument-included-files) to instrument those as well.

Nothing about preprocessed file in greport output

In some cases, a C file is produced from some other source file, such as YACC or bison grammars, or C++ source. In this case, the file passed to GCT may not have the name of the original source. Typically, the original source appears as if it were an include file. GCT won't instrument it unless (options instrument-included-files) is set.

Cannot rule out right-hand multicondition.

In a test like A && B, people often are surprised that B can be both true and false, but GCT will insist that

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

GCT obeys C’s rules for short-circuit evaluation. The value of B is never tested unless A is true. Not every execution where B is false will rule out this coverage condition, only one where A!=0, B==0.

It would not be safe to evaluate the right-hand side to see if coverage would be satisfied. Consider

```
ptr!=0 && ptr->field==5
```