GNATcoverage Users Guide

Table of Contents

About this Document

This document introduces the fundamental principles behind GNATcoverage, a non intrusive structural coverage analysis framework, and offers a toolset user's guide.

Chapter 1 [scov-basics], page 2 provides a brief introduction to the "Structural Coverage Analysis" process nature and intent, including a short discussion on the fundamental distinction between "object" and "source" coverage criteria.

Chapter 2 [xcov-grounds], page 8 describes the GNATcoverage framework core operation mode and capabilities.

Chapter 3 [xcov-guide], page 10 is the toolset user's guide, with details on the tool command line interface, use examples for various situations and interpretation guidelines for the different sorts of reports that the tool can produce.

Various concepts are illustrated with examples throughout. Most of the program sources for these examples are taken from the "exploration robot" application, developed just for this illustrative purpose and introduced in Section 4.1 [explore], page 28.

1 Structural Coverage Analysis Basics

1.1 General Definition & Intent

Structural coverage analysis can be viewed as a software development activity aimed at examining which pieces of an application program (source and/or machine code) are exercised by executions of the application software. There may be several reasons why coverage analysis is performed. A typical case is the use in software certification processes such as the DO-178B standard enforced in the civil avionics domain. In this context, the application code and the test sequences are both derived from a common set of requirements, independently, and the analysis is meant to fulfill two complementary goals:

- Assess the quality of a testing campain, on the grounds that insufficient testing of some requirements often leads to improper coverage of the code that implement them,
- Identify pieces of the application code that aren't tied to a requirement, on the grounds that there would be no test to exercise them.

In the following sections we introduce the common components of a coverage analysis process, together with *terms* to be reused throughout this document.

1.2 Process Model

1.2.1 Basic Process Abstractions

A typical coverage analysis process comprises three principal steps:

1. A binary *executable program* is produced from a set of program *sources* by a development toolchain (compiler, linker, etc).
2. The executable program runs within an *execution environment*, and this execution produces *raw coverage data* about paths it exercises.
3. The raw coverage data is interpreted or *mapped* into some user readable representation.

As an illustrative example, the common GCC/GCOV process is exactly along those lines: the program is compiled and linked by GCC [*gcc*] with special command-line options, execution produces a binary data file and GCOV is then used to generate annotated sources from the original files, the executable and the execution data. Below is an example with a simple test of the Explore queues abstraction, compiled with the GNAT toolchain for Ada:

```
# Build with gcov related options - produces executable program

    $ gnatmake test_queues.adb -fprofile-arcs -ftest-coverage

# Run program - produces raw coverage data files (queues.gcda, ...)

    $ ./test_queues

# Map to user representation - produces annotated sources (queues.adb.gcov, ...)

    $ gcov test_queues
```

The annotations are in the first column for each source line: '-' indicates there is no associated object code, numbers indicates the number of times code for this line was executed, and '#' signs indicates object code never executed:

```
    1:    9:procedure Test_Queues is
    -:   10:   package Integer_Queues is new Queues (Data_Type => Integer);
    -:   11:   use Integer_Queues;
    -:   12:
    -:   13:   X : Integer;
    1:   14:   Q : Integer_Queues.Queue (Capacity => 1);
    -:   15:begin
    1:   16:   Push (12, Q);
    1:   17:   Pop (X, Q);
    1:   18:   if X /= 12 then
#####:   19:      raise Program_Error;
    -:   20:   end if;
    -:   21:end;
```

In this excerpt, the never executed code on line 19 is expected, as it is intended to trigger only when the test didn't behave as it should. This points at an important distinction to make: the `Test_Queues` procedure in this example is *testing code* written to exercise pieces of the `Queues` abstraction, and only the latter will be an actual part of the application. Most of the time, we're only interested in the coverage results for such *applicative* code.

The form of the raw information depends on the coverage analysis toolset technology. This is most often binary data. The production of raw coverage data at run time always involves some sort of *instrumentation* to have the execution produce information it normally wouldn't produce. This may be achieved in several possible manners:

- *Source Instrumentation*: The coverage analysis toolset adds explicit statements and data structures to the program source to maintain the coverage state. This is what many commercial products do.
- *Object Instrumentation*: The development toolchain inserts extra machine state and instructions in the program executable object code. This is the GCC/GCOV approach.
- *Environment Instrumentation*: The execution environment is setup to produce a trace of the program paths taken at the machine instruction level, leaving the program code untouched. This is what solutions based on hardware probes or on instrumented virtual execution environments do.

There are variants of each technique in the field, each with its own set of advantages and drawbacks compared to others.

1.2.2 Data Capitalization & Consolidation

Proper coverage of applicative code often requires several test executions on possibly disjoint pieces of the final system, with each test providing its own partial coverage outcome. *Capitalization* denotes the capability to store and manage the partial results, and *Consolidation* denotes the construction of a unified view from partial results gathered together.

Different tests could for instance be several executions of the same program with behavior differences caused by external input variations. They could also be executions of different programs exercising different units of the applicative code or common applicative code with different parameters.

To illustrate, consider a common data structure implementation such as the bounded `Queues` Ada package in the Explore example. To honor a common requirement, it contains simple error handling code so that an "Ada exception X is raised on attempts to extract an item out of an empty queue", and we expect this code not to be exercised in regular executions. It remains applicative code, still, and even the weakest DO-178B certification level requires tests to cover it, to make sure that at least minimal checks on its behavior with respect to requirements were performed. Something has to be done outside nominal executions in this case. One possibility is to construct a separate program dedicated to just testing this abstraction, which would force an artificial queue underflow.

The point is that one-shot full coverage is generally not possible in complex situations and the example shows it is already partially impossible with only regular Explore executions. System integration is most often a delicate process, not possible before late stages of the project, and it is often useful or simply unavoidable to perform coverage analysis on segregated pieces first.

Besides, even if full coverage of some applicative components could theortically be achieved from a single execution, it is often just more convenient or sensible to be able to reach the goal in an incremental fashion. In the Explore case, for instance, a strategy like "purpose is to maximize the entire application coverage by running a minimal number of sessions" would be a pain and actually go against the requirements based testing philosophy. One instead typically runs different sessions to verify different specific application requirements, each session produces its own coverage data.

Consolidation denotes the process of gathering the capitalized coverage information for the various pieces of a system into a unified view, with explicit input on what pieces are expected to have been covered. Pieces of no interest, or which might differ between the various testing scenarii (e.g. unit test harness) may be abstracted away.

The need for coverage data consolidation often correlates with testing strategies: whether coverage data is obtained from unit testing of individual components, from integration testing of the system as a whole, from some intermediate organisation, or possibly from a mix of all these.

1.2.3 Process Integration

As hinted by the previous sections, coverage analysis is a potentially complex activity, which requires potentially complex metrics on potentially complex software involved in potentially complex project development cycles.

Process integration refers to the organization of the analysis toolset that will provide consistent and easy access to all the features of interest for a given project. The toolset needs to be both powerful enough to provide the desired functionalities and flexible enough to accomodate the various possible project organizations in the field.

1.3 Coverage Analysis Classification

Coverage analysis always involves the evaluation of various *coverage quantifiers* or *metrics* such as “what percentage of my program source statements or of the corresponding machine code was exercised (covered) by this set of executions?”. In practice, this is most often refined down at the module or subprogram level and comes together with detailed reports about the bits which were exercised and those that were not. The process is typically driven by specific objectives like “tests should result in coverage of 100% of the application program source statements”. Every toolset offers its own spectrum of analysis possibilities, with variations in the implementation schemes. We distinguish two broad classes of activities: *source* and *object* coverage analysis.

1.3.1 Object Coverage Analysis

Object Coverage Analysis focuses on machine object code coverage, with two essential quantifiers:

- *Object Instruction Coverage (OIC)* ; which/how-much of the program machine instructions were exercised by a set of program executions.
- *Object Branch Coverage (OBC)* ; OIC + indications on the machine decisions taken at each machine conditional branch instruction.

Results can be rendered on a representation of the machine code, for example as an annotated assembly output. They can also be rendered on a representation of the program sources, for example by way of annotations for each source line to synthesize information about all the machine code generated for that line. The focus is always on machine code coverage, in any case, and source annotations in this context are just a means to organize and hilight machine code properties of interest for the end user.

1.3.2 Source Coverage Analysis

Source Coverage Analysis focuses on user source code and simply abstracts the machine code away. The DO-178B structural coverage criteria operate at this level, with quantifiers defined over three core elements:

- *Source statement*, in the usual programming language sense.
- *Decision*, defined as “a top-level Boolean expression (that is not an operand of a Boolean operator)”.
- *Condition*, defined as “an elementary Boolean expression (which is not a Boolean operator applied to some subexpressions)”. Note that if a given subexpression occurs more than once in a decision, each occurrence is a distinct condition.

The quantifiers are as follows:

- *Source Statement Coverage (SSC)* ; which/how-much of the source statements were exercised by a set of program executions.
- *Source Decision Coverage (SDC)* ; SSC + indications on the values taken by each logical decision and of which entry/exit points were exercised.
- *Source Modified Condition/Decision Coverage (SMCDC)* ; SDC + indications on which conditions took their two possible outcome and which were shown to have independent effect on their decisions out of a set of program executions.

The quantifier names are often used standalone to denote coverage objectives, for instance “achieving Source Statement Coverage” denotes covering 100% of the program source statements. The “source” part is often omitted and implicitly assumed, and DO-178B attaches specific structural coverage objectives to different certification levels this way: full *Statement Coverage* at level C, *Decision Coverage* at level B and *Modified Condition/Decision Coverage* at level A.

Below is an illustration of the principal differences between the criteria over a simple example function out of an early version of the Explore sources:

```
--  Whether execution of CTRL by Robot R is unsafe

function Unsafe
  (Ctrl : Robot_Control; R : Robot_Access) return Boolean
is
   Situ : Situation;
begin
   --  Probe the current situation in SITU and evaluate.
   --  Start by assuming CTRL is safe and adjust.

   Devices.Probe (Situ, R.H.DH);
   declare
      Is_Unsafe : Boolean := False;
   begin
      --  Stepping ahead into a rock block or a water pit is unsafe

      if Ctrl.Code = Step_Forward
         and then (Situ.Sqa = Block or else Situ.Sqa = Water)
      then
         Is_Unsafe := True;
      end if;

      return Is_Unsafe;
   end;
end;
```

Statement Coverage of the `Unsafe` function requires execution of all the source statements at least once. This can be achieved with a single call to the function, as soon as the boolean decision controlling the `if` statement evaluates to `True`.

Decision Coverage requires that every decision has evaluated at least once to `True` and at least once `False`, which necessitates at least two calls in our example to exercise the `if` controlling expression both ways. It also requires going through every possible entry and exit point, without further impact of note on the simple example at hand.

Modified Condition/Decision Coverage requires additional variations over the conditions, and combinations to show that each condition can affect the decision outcome in an independent manner. This is expected to be possible with Nconditions+1 evaluations, so enforces a more precise testing of the expressions structure while keeping the test base complexity linear with the number of conditions. There exist several variants of the MCDC criteria, with differences in the way independence may be shown - [*ar0118*], [*cast6*].

1.3.3 Source vs Object Quantifiers

Object and Source coverage quantifiers are of very different natures. Both have both pros and cons, some very dependent on the evaluation context and purpose.

An interesting study is that of the implication relationships between criteria, to determine if satisfying one criteria may be used as a means to claim another. These correlations are not at all trivial in the general case. Below are a few points to illustrate.

As a starter example, we may consider that Object Branch or even Instruction Coverage implies Statement Coverage while the opposite is not true. To illustrate the basic idea, take the case of a modulo computation: it is expressed with a single statement in C or Ada, and the machine code typically features different paths to honor variations conditioned on the sign of the operands. A single pass trough this code will cover the source statement and not the full instruction set. Conversely, covering the full set of machine instructions necessitates several passes through the code, hence coverage of the source statement. For the general case, statements for which no machine code is produced need care but don't introduce fundamental difficulties.

Along similar lines, we may consider that full Object Branch Coverage implies Decision Coverage while the opposite is not true. We also observe that Object Instruction Coverage does not imply Decision Coverage, and that Obect Branch Coverage does not imply MC/DC in the general case - [*AR07/20*], [*obc-mcdc*]. Besides, when it does imply MC/DC, Object Branch Coverage often requires more extensive testing so is not necessarily an interesting alternate.

In any case, assumptions validity need to be complemented with practical consequences in industrial applications. In particular, using one criteria as a means to achieve another when an implication holds (e.g. seeking OBC to achieve DC) might call for unrequired significant additional testing efforts.

2 GNATcoverage Fundamentals

2.1 Instrumentation mode

The core principle in the GNATcoverage framework is to leverage the generation of raw coverage data by a virtual execution environment instrumented to produce machine level traces about the code it executes. We refer to these as *execution traces*.

The environment typically is a representative emulator of a real target microprocessor, possibly augmented with extensions to let it communicate with external devices. For common architectures, we leverage qemu [*qemu*] for this purpose, as a reliable and efficient free-software emulator we can instrument to generate the traces.

The environment may also be a pure virtual machine such as existing ones for Java or Caml like languages. In any case, the program itself isn't instrumented, so coverage measurements can be performed on target code, as embedded eventually, and the virtual environment runs on development hosts, which offers a lot of flexibility.

The raw coverage data out of the execution environment is very low level information about the executed instruction and branch sequences at the machine level. The actual contents structure may vary, depending on the kind of analysis anticipated.

2.2 Object Coverage Analysis

To start with, GNATcoverage allows the confrontation of execution traces with the full machine code available from program files, hence precise object coverage analysis with both instruction and branch coverage capabilities. This is achievable with simple traces that can be gathered and represented in a very efficient manner, schematically as a flat compact map of status per executed instruction or linear sequence.

The results may first be rendered at the assembly language level, with annotations for each machine instruction to indicate whether it was executed or not, and for each conditional branch whether it has been taken, not taken or both.

Then, provided extra information to establish instruction to source line correspondance, GNATcoverage is also able to render the object coverage outcome through source annotations, with source line annnotations derived from those of all the associated machine instructions. Typically, a source line is marked as *fully*/*partially* covered when all/part of the associated machine instructions were executed, and the instruction/line correspondance is extracted from standard DWARF debug information or alike.

2.3 Source Coverage Analysis

GNATcoverage is also designed to allow Source Coverage Analysis, with central focus on user source code and support for the three DO-178B criteria: Statement, Decision and Modified Condition/Decision Coverage. For MCDC, the framework sets up the necessary elements to be able to reconstruct the exercised run-time condition/decision vectors. An important part is the introduction of *Source Coverage Obligations*, or SCOs, compact tables generated to indicate the source elements of relevance to coverage analysis activities. SCOs are designed to be independent from the target certification level, which only influences the way a given trace is determined to meet.

For qemu targets and the gcc compilation toochain, the toolset uses SCOs and precise debug information to associate conditional branches with conditions, then traces are extended to track the history of run-time behavior at those branch points. Indeed, the object coverage flat execution traces aren't precise enough in this case unless very strong constraints are met by the source constructs. Using extended traces or flat ones with source constraints, the MCDC capabilities of GNATcoverage rely on the presence of a conditional branch instruction for each non-constant condition. We provide sets of compilation options suitable for both this particular purpose and for the Source Coverage analysis activity in general.

2.4 Modularity and Flexibility

Different teams have different organizations and software development infrastructures. GNATcoverage is designed as a modular set of light tools, intended to be adaptable to various operational contexts.

3 GNATcoverage Users Guide

3.1 Getting Started

Below is a verbatim copy of the distribution `README` file, which provides a brief description of the package contents, installation instructions and a Quick Start section, basic introduction to the toolset architecture and interface:

```
PACKAGE CONTENTS
================

This package provides the xcov front-end to coverage analysis activities.  It
may be used both as a wrapper to an instrumented execution environment able to
produce machine-level execution traces (xcov run) and as a trace analyzer able
to render coverage results in various output formats (xcov coverage).

Instrumented execution environments are provided as separate packages. As of
today, we leverage instrumented versions of Qemu, an open source processor
emulator.

As part of the examples subdirectory, the package also includes:

* A light Qemu Board Support Package to link with your executable to let
  it run within the emulated environment,

* A couple of very basic library components for Ada (simple IO, memory
  copy/set/compare, ...) in case they are not available otherwise,

* A number of example programs you can exercise to get familiar with the
  toolset, with Makefiles to illustrate build/run/analyze sequences.

The "Explore" example is introduced in the user's guide and used for
illustration purposes there. It features both an interactive program and AUnit
based tests for some of the program units.

The "Engines" example is used to provide a quick first contact with the
tools, through the QUICK START section below.

INSTALL - binary distribution
=============================

If you retrieved this README from a binary distribution (zip or tar.gz archive
with -bin in the name), you have

  * An <unpack-subdir>/share subdirectory with the doc and examples;

  * An <unpack-subdir>/bin subdirectory with the "xcov" program.

The xcov binary is standalone, so you may access it by simply adjusting your
PATH environment variable. A prerequisite to using xcov run is to have the
instrumented environment available already.

If this is more convenient for you, you may transfer the contents of the
"shared" and "bin" subdirectories into their corresponding entries within a
pre-existing installation tree, where a compiler toolset is located for
instance.
```

```
INSTALL - source distribution
=============================

If you retrieved this README from a source distribution (repository, zip or
tar.gz archive with -src in the name), you have

  * Sources of the front-end straight at hand, together with this README
    and a Makefile;

  * Examples and the documentation in the "examples" and "doc" subdirectories
    respectively.

To build the xcov binary, just invoke "make". This requires an Ada 2005
capable compiler and the Makefile resorts to GNAT for this purpose.

To build pdf and other versions of the documentation, invoke "make doc".

QUICK START - OBJECT COVERAGE
=============================

A quick starter is the "engines" basic example, assuming you have installed
your target compilation toolchain, this package, and the instrumented qemu
(qemu-system-ppc|sparc from binary distributions or built from the proper
source tree).

The first step is to setup the PATH environment variable to include locations
for both the target compiler and the xcov/qemu "bin" directory. For example
with a bash like shell:

  $ PATH=/usr/local/gnat/6.1.2/bin:/usr/local/xcov/bin:$PATH

Then switch to the Xcov "engines" example directory and exercise the
"test_engines" test there for your target, thanks to the local Makefile:

  engines $ make TARGET=powerpc-elf

Or if you want to test on a Leon board (sparc based):

  engines $ make TARGET=leon-elf

This performs a build/run/analyze sequence out of which object coverage
results rendered in sources are produced in html format, browsable from
an index page in index-test_engines.html.

Here is a brief description of what is going on:

# Step 1:  Build an executable program suitable for Qemu. This is a
# -------  regular Ada build with a couple of extra bsp components for
#          startup, io & a dedicated linker script:

powerpc-elf-gcc -c -o start.o ../support/powerpc-elf/start.s
...
powerpc-elf-gcc -c -g -O1 -fpreserve-control-flow test_engines.adb
...

# Step 2:  Run the executable program within the instrumented Qemu
# -------  to get an execution '.trace' file:
```

```
xcov run --target=powerpc-elf --level=branch test_engines

# Step 3:  Ask xcov to analyze the trace and produce an html version of
# -------  the results, with object coverage branch info rendered on source:

xcov coverage --level=branch --annotate=html+ test_engines.trace
mv index.html index-test_engines.html

The local Makefile actually just includes a generic Makefile, common to all
the examples and which you may reuse and adapt to your specific needs.

QUICK START - SOURCE COVERAGE
=============================

Source coverage analysis involves a similar process, with few differences:

* It is requested by specific values of the --level argument: stmt,
  stmt+decision or stmt+mcdc ;

* Sources should be compiled with -g -fpreserve-control-flow -gnateS,
  and currently with -O0. We don't support optimized code yet ;

* The list of units for which analysis is desired needs to be passed as
  a list of ALIs with the '--scos' option to both xcov run and xcov coverage,
  for instance via a response file (--scos=@file_with_list_of_ALIs)

This can be exercised for our examples by passing an extra XCOVLEVEL
argument to 'make' invocations, for example 'make XCOVLEVEL=stmt'.

FURTHER DOCUMENTATION
=====================

Further documentation is available from the "Xcov Fundamentals & Users Guide"
in the <unpack/installation-root>/share/doc/xcov directory of this package.
```

As suggested by the previous introduction, GNATcoverage offers a front-end to various coverage analysis related functionalities, each activated by a toplevel of command line option:

- '`run`' ; run code within an instrumented environment to produce execution traces.
- '`coverage`' ; process execution traces to produce user-level results.

The following sections provide further details on the various modes of operation, first for simple cases where a single trace is to be produced and analyzed, then for more sophisticated needs requiring coverage data capitalization and consolidation.

3.2 Instrumented Execution

xcov run offers a unified interface to launch programs for a specific target machine within the appropriate instrumented execution environment to produce execution traces.

The Quick Start example in the distribution README illustrates a simple use for a powerpc-elf or a leon-elf target, using the dedicated GCC toolchain to build from sources and the '--target' execution engine selector. The general interface synopsis is available from xcov --help, as follows:

```
run [OPTIONS] FILE [-eargs EARGS...]
  Options are:
  -t TARGET  --target=TARGET   Set the target
    targets: powerpc-elf leon-elf i386-pok i386-linux prepare
  -v --verbose                 Be verbose
  -T TAG  --tag=TAG            Put TAG in tracefile
  -o FILE  --output=FILE       Write traces to FILE
  -eargs EARGS                 Pass EARGS to the simulator
```

'-v' requests verbose output, in particular the commands to run the program within the underlying instrumented environment.

The FILE argument is the executable program file name. This name is stored as-provided in the output trace header, where it is retrieved later by xcov coverage for analysis purposes.

By default, xcov run writes the execution trace in the current directory, in a file named like the executable input with a .trace suffix. For example xcov run /path/to/myexecfile produces a myexecfile.trace file in the current directory. '--output' allows the selection of an alternate output file name.

The '--tag' option expects a string argument and stores it verbatim as a trace tag attribute in the output trace header. The tag so associated with a trace can be retrieved from trace dumps and is output as part of some analysis reports. It is useful as a flexible trace identification facility, structured as users see fit for custom trace management purposes.

3.3 Object Coverage Analysis

Over execution traces, various levels of object coverage analysis may be performed with `xcov coverage`. An analysis variant first needs to be selected with the '`--level`' option:

`=insn` requests *Object Instruction Coverage* data, with an indication for every instruction of whether it has been executed or not.

`=branch` requests *Object Branch Coverage* data, with extra details about the directions taken by conditional branch instructions.

An additional '`--annotate`' option selects the output format:

`=asm` annotated assembly code on standard output.

`=xcov[+]` annotated source files, with the object code for each source line interspersed if the `+` variant is selected.

`=html[+]` html index of per source file coverage summary, with links to annotated sources [`+` code expandable from each source line].

`=report` synthetic report of per subprogram coverage results.

The following sections provides extra details and examples for each situation. In principle, this is all pretty independent of the program compilation options. Aggressive optimizations very often make source to object code associations more difficult, however. Besides, if source coverage analysis is to be performed as well, the whole process is simpler if the same compilation options are used, and these have to be strictly controlled for source coverage.

3.3.1 Machine level reports, `--annotate=asm`

For object coverage analysis purposes, '`--annotate=asm`' produces annotated assembly code for all the program routines on standard output. The annotations are visible as a special character at the beginning of each machine code line to convey information about the corresponding instruction, with variants for instruction or branch coverage modes. We call *simple* those machine instructions which are not *conditional branch* instructions.

For *Object Instruction Coverage*, with '`--level=insn`', we define:

Note	*Means ...*
'`-`'	instruction was never executed
'`+`'	instruction was executed

For *Object Branch coverage* ('`--level=branch`'), the '`+`' case is refined for conditional branch instructions and two additional notes are possible:

Note	*Means ...*
'`-`'	instruction never was executed
'`+`'	instruction was executed, taken both ways for a conditional branch
'`>`'	conditional branch was executed, always taken
'`v`'	conditional branch was executed, never taken

We qualify instructions marked with '+' as *fully covered*, those marked with '-' as *uncovered* and the others as *partially covered*.

To illustrate, we will consider the Branch Coverage outcome for a piece of the Explore example, produced out of a couple of runs within QEMU for the PowerPC architecture. The original source of interest is the `if` statement which controls the Station processing termination, upon a Quit request from the user. The control is performed by a single decision, composed by two connected conditions to expose a case insensitive interface:

```
procedure Run (Sta : Station_Access) is
   ...
   Put ("'P'robe, 'S'tep, Rotate 'L'eft/'R'ight, 'Q'uit ? ");
   Flush;
   Get (C);

   if C = 'Q' or else C = 'q' then
      Kill (Sta.all);
      return;
   else
   ...
```

We first run a sample session to exercise Probe, then Quit with 'Q', and request branch coverage data in assembly format:

```
... $ xcov run --target=powerpc-elf explore
[Explore runs in QEMU
 - type 'p', then 'Q']

... $ xcov coverage --level=branch --annotate=asm explore.trace
```

For the code associated with the source bits of interest, this yields the following assembly coverage report excerpt:

```
...
<stations__run>:
...
fffc1c0c +:     4b ff e6 7d   bl     0xfffc0288 <text_io__get>
fffc1c10 +:     2f 83 00 51   cmpiw  cr7,r3,0x0051
fffc1c14 +:     41 9e 00 0c   beq-   cr7,0xfffc1c20 <stations__run+00000078>
fffc1c18 +:     2f 83 00 71   cmpiw  cr7,r3,0x0071
fffc1c1c >:     40 9e 00 10   bne-   cr7,0xfffc1c2c <stations__run+00000084>
fffc1c20 +:     7f e3 fb 78   or     r3,r31,r31
fffc1c24 +:     4b ff e7 d1   bl     0xfffc03f4 <actors__kill>
...
```

The `beq` and `bne` instructions are two conditional branches corresponding to the two conditions. In addition to straightforward coverage of the rest of the code, the '+' for the first branch indicates that it is fully covered and the '>' for the second branch indicates partial coverage only. Indeed, both conditions were evaluated to False on the 'p' input, then on 'Q' the first condition was evaluated to True and the second one was short-circuited.

We run a second experiment, when the user quits with 'Q' immediatly. We observe that the first conditional branch is only partially covered and the second one is not even exercised:

```
...
<stations__run>:
...
fffc1c0c +:     4b ff e6 7d   bl     0xfffc0288 <text_io__get>
fffc1c10 +:     2f 83 00 51   cmpiw  cr7,r3,0x0051
fffc1c14 >:     41 9e 00 0c   beq-   cr7,0xfffc1c20 <stations__run+00000078>
fffc1c18 -:     2f 83 00 71   cmpiw  cr7,r3,0x0071
fffc1c1c -:     40 9e 00 10   bne-   cr7,0xfffc1c2c <stations__run+00000084>
fffc1c20 +:     7f e3 fb 78   or     r3,r31,r31
fffc1c24 +:     4b ff e7 d1   bl     0xfffc03f4 <actors__kill>
...
```

3.3.2 In-Source text reports, --annotate=xcov[+]

For object coverage analysis, ‘--annotate=xcov’ produces annotated source files with the .xcov extension in the current directory, one per original compilation unit. An alternate output directory may be selected by passing a ‘--output-dir=<directory name>’ command line option as well.

The annotations are visible as a special character at the beginning of every source line, which synthesizes the coverage status of all the machine instructions generated for this line. The machine instructions are printed next to their associated source line when the ‘+’ option extension is used. Eventhough the annotations are rendered on source lines in this case, they are really meant to convey object code properties, hence are of a different nature than what the DO-178B structural coverage criteria refer to.

We defined a uniform synthesis of source line from object code annotations for both instruction and branch coverage:

Note	*Means ...*
’.’	no associated machine code for this line
’-’	all the instructions associated with the line are ’-’ (uncovered)
’+’	all the instructions associated with the line are ’+’ (fully covered)
’!’	otherwise

To lines with associated object code we apply qualifiers similar to those for individual instructions: ’-’, ’+’ and ’!’ denote *uncovered*, *fully covered* or *partially covered* lines respectively.

At this stage, gnatcov relies on dwarf debug information to associate machine instructions with their corresponding source lines, so these annotations are only possible when this is available. In GCC parlance, this requires compilation with the ‘-g’ command line switch, designed never to influence the generated code.

3.3.3 In-Source html reports, --annotate=html[+]

‘--annotate=html’ produces one .html browsable annotated source file per original compilation unit in the current directory. The annotations are identical to the ‘=xcov’ ones, and an alternate output directory may be selected with ‘--output-dir’ as well. Each source line is colorized to reflect its associated object code coverage completeness, with green, orange and red for full, partial or null coverage respectively.

An index.html page summarizes the coverage results and provide links to the annotated sources. With the + extension, the annotated machine code for each line may be expanded below it by a mouse click on the line.

3.3.4 Synthetic reports, `--annotate=report`

For object coverage analysis, '`--annotate=report`' produces a synthetic summary of per function coverage results, with a single annotation assigned to each function in the same way it is to each source line in the '`=xcov`' or '`=html`' cases.

3.3.5 Inlined and Template/Generic entities

The generated code for an inlined subprogram or a generic instantiation implicitely associates with two source locations: the entity source itself (what code materializes) and where the instantiation takes place (where the generated code goes). Choices were made for In-Source reports. Behind the scenes, xcov uses standard debug information to establish the links between object code and original source, so the choice stems from this information essentially. The next paragraphs are specific to the GNAT/GCC chains in this respect.

For inlined calls, the GCC debug information associates the generated machine code with the inlined source positions, so the related object coverage information is reported there. This scheme has all the instances reported at a centralized location and allows use of the full inlined subprogram source structure to organize the results. Consider for example the following excerpt of branch coverage report for the Station control code in Explore. A call to an `Update` subprogram is inlined in `Process_Pending_Inputs`. We observe that the code reported in the `Update` sources is coming from the `process_pending_inputs` symbol, where it was inlined, and that absence of code is reported at the call site, since indeed all the code for this call is attached to the inlined entity.

```
  53 .:       procedure Update (Map : in out Geomap; Situ : Situation) is
  54 +:          Posa : constant Position := Pos_Ahead_Of (Situ);
<stations__run__process_pending_inputs.1939+fffc1bb4>:+
fffc1c04 +:  4b ff ed c1  bl     0xfffc09c4 <geomaps__pos_ahead_of>
fffc1c08 +:  90 61 00 30  stw    r3,0x0030(r1)
  55 .:       begin
  56 +:          Map (Posa.X, Posa.Y) := Situ.Sqa;
<stations__run__process_pending_inputs.1939+fffc1bc4>:+
fffc1c28 +:  88 01 00 19  lbz    r0,0x0019(r1)
fffc1c2c +:  98 03 00 0f  stb    r0,0x000f(r3)
  [...]
  63 +:       procedure Process_Pending_Inputs (Sta : Station_Access) is
  [...]
  68 .:             Update (Sta.Map, Situ);
```

Similar principles apply to template instantiations such as those of Ada generic units, and the centralized view property is well illustrated this way. The excerpt below provides an example with the `Queues` abstraction in Explore, instantiated in several places. The corresponding code sequences are all attached to original unit source, with an indication of their instantiation locations via the symbol names in the start-of-sequence addresses:

```
  39 +:    function Empty (Q : Queue) return Boolean is
<robot_control_links__data_queue_p__empty+fffc02fc>:+
fffc02fc +:  94 21 ff f0  stwu   r1,-0x0010(r1)
 [...]
<geomaps__situation_links__data_queue_p__empty+fffc0878>:+
fffc0878 +:  94 21 ff f0  stwu   r1,-0x0010(r1)
 [...]
```

3.3.6 Focusing on subprograms of interest

GNATcoverage provides a number of facilities to allow filtering results so that only those of actual interest show up.

The primary filtering device for object coverage analysis is the '`--routines`' option to `xcov coverage`. '`--routines`' expects a single argument, to designate a set of symbols, and restricts coverage results to machine code generated for this set. The argument is either a single symbol name or the name of a file prefixed with a `@` character, expected to contain a list of symbol names.

To illustrate, the example command below produces a branch coverage report for the `Unsafe` subprogram part of the `Robots` unit in Explore. Out of a GNAT compiler, the corresponding object symbol name is `robots__unsafe`, here designated by way of a single entry in a symbol list file:

```
$ cat slist
robots__unsafe

$ xcov coverage --level=branch --annotate=asm --routines=@slist explore.trace
Coverage level: BRANCH
robots__unsafe !: fffc1074-fffc109b
fffc1074 +:  2f 83 00 02      cmpiw  cr7,r3,0x0002
fffc1078 +:  40 be 00 1c      bne+   cr7,0xfffc1094 <robots__unsafe+00000020>
[...]
```

GNATcoverage provides a '`disp-routines`' command to help the elaboration of symbol lists. The general synopsis is as follows:

```
 disp-routines {[--exclude|--include] FILES}
   Build a list of routines from object files
```

`xcov disp-routines` outputs the list of symbols in a set built from object files provided on the command line. 'Object file' is to be taken in the general sense of 'conforming to a supported object file format, such as ELF', so includes executable files as well as single compilation unit objects.

The output set is built incrementally while processing the arguments left to right. '`--include`' states "from now on, symbols defined in the forthcoming object files are to be added to the result set". '`--exclude`' states "from now on, symbols defined in the forthcoming object files are to be removed from the result set". An implicit `--include` is assumed right at the beginning, and each object file argument may actually be an `@file` containing a list of object files. Below are a few examples of commands together with a description of the set they build.

```
$ xcov disp-routines explore
  # (symbols defined in the 'explore' executable)

$ xcov disp-routines explore --exclude test_stations.o
  # (symbols from the 'explore' executable)
  # - (symbols from the 'test_stations.o' object file)

$ xcov disp-routines --include @sl1 --exclude @sl2 --include @sl3
  # (symbols from the object files listed in text file sl1)
  # - (symbols from the object files listed in text file sl2)
  # + (symbols from the object files listed in text file sl3)
```

In-source reports, when requested, are generated for sources associated with the selected symbols' object code via debug line information. Coverage synthesis notes are produced only on those designated lines. For example, `--annotate=xcov --routines=robots__unsafe` will produce a single `robots.adb.xcov` in-source report with annotations on the `Unsafe` function lines only, because the debug info maps the code of the unique symbol of interest there and only there.

Note that inlining can have surprising effects in this context, when the machine code is associated with the inlined entity and not the call site. When the code for a symbol A in unit Ua embeds code inlined from unit Ub, an in-source report for routine A only will typically produce two output files, one for Ua where the source of some of the symbol code reside, and one for Ub, for lines referenced by the machine code inlined in A.

3.4 Source Coverage Analysis

Source coverage analysis focuses on source elements such as "statements" or "decisions". Machine object code is entirely abstracted away. For source coverage assessment, `gnatcov` relies on *Source Coverage Obligation* (SCO) tables, compact descriptions of the source elements relevant to coverage analysis.

As of today, `gnatcov` supports SCOs provided as part of the Ada Library Information files generated by the GNAT compilers when invoked with the '`-gnateS`' command line option. To obtain accurate results, the code should be compiled with optimizers disabled (`-O0` in gcc parlance). Support for optimized code is being worked on for future versions.

The general process to perform source coverage analysis is similar to the one for object coverage: `xcov run` produces execution traces, and `xcov coverage` generates reports out of them. Source coverage analysis is requested thanks to variants of the `--level` option, which should be passed to both `xcov run` and `xcov coverage`.

The set of SCOs for which coverage is to be assessed is provided by way of a '`--scos`' command line option, which accepts either a single `.ali` filename argument, or an `@` prefixed filename containing a list of `ali` files. `--scos` is the source oriented version of what `--routines` offers in the object coverage case. They may not be used together. `--scos` conveys both SCO information to the analysis engine and the selection of units for which result reports are to be produced. The option may be repeated on the command line, with cumulative effects.

Source coverage results may be produced in several output formats, selected with the '`--annotate`' command line option. `xcov`, `html`, and `report` are available, with general characteristics identical to those described in the object coverage section: `xcov` is a text format with a coverage annotation on each source line, `html` features line colorization and an index page, and `report` outputs the sequence of incomplete coverage diagnostics out of the analysis performed.

3.4.1 Statement Coverage (SC)

Statement coverage is achieved with `--level=stmt`, together with `--scos` to provide the set of SCOs of interest via ALI files. The `xcov` and `html` annotation formats both generate a representation of the sources with annotations on each relevant line, according to the following table:

Note	*Means ...*
'`.`'	no SCO or no executable code for this line
'`-`'	statement uncovered (not executed) on this line
'`+`'	statement covered (executed) on this line

Below is a sample session to illustrate on the Explore example, for the `robots` unit after recompilation with '`-gnateS -O0`'. Note the '`--level`' option passed to both `run` and `coverage` invocations:

```
$ xcov run --level=stmt explore
... run session, trace goes to explore.trace by default ...

$ xcov coverage --level=stmt --scos=obj/robots.ali --annotate=xcov explore.trace
```

To analyze a full set of units at once, just fetch the list of ALI files in a list and provide an @file to --scos. For instance, in a Unix-like environment:

```
$ ls obj/*.ali > alis
$ xcov coverage --scos=@alis --level=stmt --annotate=xcov explore.trace
```

For the Stations unit, this produces a stations.adb.xcov output with:

```
Coverage level: STMT
87% of 38 lines covered
[...]
  74 .:       function Control_For (C : Character) return Robot_Control;
  75 .:       --  Map user input character C to Robot_Control command, Nop if
  76 .:       --  the input isn't recognized.
  77 .:
  78 .:       function Control_For
  79 .:         (C : Character) return Robot_Control is
  80 .:       begin
  81 +:          case C is
  82 .:             when 'p' | 'P' =>
  83 +:                return (Code => Probe, Value => 0);
  84 .:             when 's' | 'S' =>
  85 +:                return (Code => Step_Forward, Value => 0);
  86 .:             when 'l' | 'L' =>
  87 -:                return (Code => Rotate_Left, Value => 0);
  88 .:             when 'r' | 'R' =>
  89 -:                return (Code => Rotate_Right, Value => 0);
```

--annotate=report instead simply diagnoses the set of source lines with uncovered statements, for example like:

```
stations.adb:87: statement not executed
stations.adb:89: statement not executed
```

More details on the report format are available in a dedicated appendix of this documentation. By default, the report goes to standard output. It may be directed to a file instead, with the addition of a '-o <filename>' option on the command line.

3.4.2 Decision Coverage (DC)

gnatcov features combined Statement and Decision Coverage assessment capabilities with '--level=stmt+decision'. We consider to be *decisions* all the boolean expressions used to influence the control flow via explicit constructs in the source program, such as if statements or while loops. For proper operation, expressions may only resort to short-circuit operators to combine operands. The GNAT compilers offer the No_Direct_Boolean_Operator restriction pragma to make sure this rule is obeyed.

A decision is said fully covered when tests were made so that the decision has evaluated to both True and False. If only one of these two possible outcomes was exercised, the decision is said partially covered. The case where none of the possible decision outcomes was exercised happens when the enclosing statement was not executed at all, or when all the attempted evaluations were interrupted e.g. because of exceptions. Uncovered statements remain reported as such, without further details even if there are decisions therein.

The `xcov` and `html` annotation formats both generate a representation of the sources with annotations at the beginning of each relevant line, according to the following table:

Note	*Means ...*
'`.`'	no SCO or no executable code for this line
'`-`'	statement uncovered on this line
'`!`'	decision partially covered on this line
'`+`'	all the decisions on this line are fully covered

As for object coverage, additional information is available on request with an extra `+` suffix on the annotation format, that is, with `--annotate=xcov+` or `html+`. Extra details are typically provided for decisions partially covered, with information about which outcome was not exercised.

The `--annotate=report` synthetic output lists information about uncovered statements and partial decision coverage. For example, after exercising Explore to have the robot execute safe commands in both Cautious and Dumb modes, we get the expected results below on a sample of the `Robots` control code:

```
$ xcov coverage --level=stmt+decision --annotate=report
  --scos=obj/powerpc-elf/robots.ali explore.trace
...
robots.adb:56:9: decision outcome TRUE never exercised
robots.adb:75:10: decision outcome TRUE never exercised
robots.adb:78: statement not executed
```

For decision related diagnostics, the source location features both a line and a column number to designate the first token of the decision unambiguously. Below is the corresponding `--annotate=xcov+` output excerpt. Decision diagnostics are always expanded on the first line of the decision:

```
  [...]
  51 .:    function Unsafe (Cmd : Robot_Command; Sqa : Square) ...
  52 .:    begin
  53 .:       --  Stepping forward with a block or a water pit ahead is Unsafe
  54 .:
  55 +:       return
  56 !:         Cmd = Step_Forward
DECISION "Cmd = Ste..." at 56:9: outcome TRUE never exercised
  57 !:         and then (Sqa = Block or else Sqa = Water);
  58 .:    end Unsafe;
  [...]
  64 .:    procedure Process_Next_Control
  65 .:      (Port : Robot_Control_Links.IOport_Access)
  66 .:    is
  [...]
  73 .:       --  Cautious, the robot refuses to process unsafe controls
  74 .:
  75 !:       if Robot.Mode = Cautious
DECISION "Robot.Mod..." at 75:10: outcome TRUE never exercised
  76 !:         and then Unsafe (Ctrl.Code, Probe_Ahead (Robot.Hw.Rad))
  77 .:       then
  78 -:          return;
  79 .:       end if;
  [...]
```

3.4.3 Modified Condition/Decision Coverage (MCDC)

In a similar fashion to statement or decision coverage, `gnatcov` features Modified Condition/Decision Coverage assessment capabilities with '`--level=stmt+mcdc`'. In addition to the particular level specification, you should also provide `xcov run` with the set of SCOs you plan to analyze later on using the produced trace, with a `--scos` argument as for `xcov coverage`. If you plan different analysis for a single run, providing a common superset to `xcov run` is fine. Providing `xcov run` with only a subset of the SCOs you will analyze might result in pessimistic assessments later on (spurious MCDC not achieved outcome).

To support MCDC, we introduce a distinction between two kinds of Boolean expressions:

- *Simple* Boolean expressions are Boolean atoms such as a lone Boolean variable or a function call, possibly negated.
- *Complex* Boolean expressions are those that feature at least two Boolean atoms combined with short-circuit operators, the only ones allowed for proper operation as for Decision Coverage.

In addition to simple and complex expressions used to influence control-flow statements, we treat as decisions all the complex Boolean expressions anywhere they might appear. For example, the Ada code excerpt below:

```
X := A and then not B;
if Y then [...]
```

... features two expressions subject to MCDC analysis: `A and then not B` (complex expression with two atoms), on the right hand side of the assignment to `X`, and the simple `Y` expression that controls the `if` statement. The Boolean atoms in a decision are called *conditions* in the DO-178 literature. The types involved need not be restricted to the standard Boolean type when one is defined by the language; For Ada, typically, they may subtypes or types derived from the fundamental Boolean type.

Compared to Decision Coverage, MCDC assessments incur extra verifications on the demonstration by the tests of the independent influence of conditions on decisions. Several variants of the criterion exist, with a common idea: for each condition in a decision, tests are required to expose a pair of valuations where both the condition and the decision value change while some extra property on the other conditions holds. The point is to demonstrate that every condition is significant in the decision and that the tests exercised representative combinations of the possible behaviors, while keeping the number of required tests linear with the number of conditions in a decision.

Unique Cause MCDC is a common variant where the extra property is "all of the other conditions in the decision shall remain unchanged". To illustrate, the table below expands the 4 possible condition/decision vectors for decision `A and then B`. `T/F` represent the True/False boolean values and the rightmost column indicates which vector pairs demonstrate Unique Cause independent effect of each condition.

```
| # | A  B  A && B | Indep |
|---|--------------|-------|
| 1 | T  T    T    | A  B  |
| 2 | T  F    F    |    B  |
| 3 | F  T    F    | A     |
| 4 | F  F    F    |       |
```

GNATcoverage actually implements a common variant, accepting variations of other conditions in an independence pair as long as they could for sure not possibly influence the decision outcome, e.g. due to short-circuit semantics. This variant provides additional flexibility on the set of tests required to satisfy the criterion without reducing the minimal size of this set. In the `and then` case, it becomes possible to use the #4 `+` #1 pair as well to demonstrate the independent influence of `A`, as `B` is not evaluated at all when `A` is False so the change on `B` is irrelevant in the decision switch.

Output-wise, the in-source notes for the `xcov` or `html` formats are the same as for decision coverage reports, with condition specific cases marked with '!' as well. `--annotate=report` outputs feature specific diagnostics where conditions are identified with their precise file:line:column source location. Using the same decision as in the previous example to illustrate, we run the Explore robot in Cautious mode only, try both safe and unsafe actions and get:

```
robots.adb:75:10: condition has no independent influence pair, MC/DC not achieved
```

Such condition related messages are only emitted when no more general diagnostic applies on the associated decision or statement, however. In our familiar example, attempting only safe actions in Cautious mode yields a “`decision outcome TRUE never exercised`” diagnostic, not a couple of condition related messages.

3.5 Advanced features

3.5.1 Coverage Data Capitalization & Consolidation

The `gnatcov` philosophy with respect to coverage data capitalization is to provide flexible means to allow custom trace management facilities, not to dictate a specific organization. Two devices were introduced for this purpose: trace tags let users associate an arbitrary string with each execution trace, and `xcov run` stores a reference to the executable program there as well. In addition to this, `gnatcov` features coverage consolidation capabilities, to allow coverage analysis of a provided set of routines from runs exercising them possibly in the context of different executable programs.

To illustrate, we analyze object branch coverage of the `Unsafe` function of the Explore `Robots` unit. We first run a simple interactive session which exercises the function only partially and look at the results:

```
$ xcov run --target=powerpc-elf --tag 'Safe explore session' explore
[... Probe, Step on clear square, Quit ...]

$ xcov coverage --level=branch --annotate=xcov+ --routines=robots__unsafe
  explore.trace

  robots.adb.xcov
 [...]
  57 .:  function Unsafe (Cmd : Robot_Command; Sqa : Square) return Boolean is
  58 .:  begin
  59 .:     --  Stepping forward into a rock block or a water pit is Unsafe
  60 .:
  61 .:     return Cmd = Step_Forward
  62 !:       and then (Sqa = Block or else Sqa = Water);
<robots__unsafe>:
fffc13a4 +:  2f 83 00 02  cmpiw  cr7,r3,0x0002
fffc13a8 +:  40 be 00 1c  bne+   cr7,0xfffc13c4 <robots__unsafe+00000020>
fffc13ac +:  2f 84 00 01  cmpiw  cr7,r4,0x0001
fffc13b0 v:  41 9e 00 0c  beq-   cr7,0xfffc13bc <robots__unsafe+00000018>
fffc13b4 +:  2f 84 00 02  cmpiw  cr7,r4,0x0002
fffc13b8 >:  40 be 00 0c  bne+   cr7,0xfffc13c4 <robots__unsafe+00000020>
fffc13bc -:  38 60 00 01  li     r3,0x0001
fffc13c0 -:  4e 80 00 20  blr
fffc13c4 +:  38 60 00 00  li     r3,0x0000
```

These results are as expected. The first branch is fully covered because the session featured both a Probe and a Step forward, so the `Cmd = Step_Forward` condition is exercised both ways. The two following branches are only partially covered because we never actually try any of the unsafe steps forward.

We then run the provided unit tests in addition, combine the results and observe full object branch coverage:

```
$ make UNIT_TESTS=test_explore
[...]
xcov run --target=powerpc-elf test_explore

$ xcov coverage --level=branch --annotate=xcov+ --routines=robots__unsafe
  test_explore.trace explore.trace
```

```
  robots.adb.xcov
  [...]
  57 .:   function Unsafe (Cmd : Robot_Command; Sqa : Square) return Boolean is
  58 .:   begin
  59 .:      --  Stepping forward into a rock block or a water pit is Unsafe
  60 .:
  61 .:      return Cmd = Step_Forward
  62 +:        and then (Sqa = Block or else Sqa = Water);
<robots__unsafe>:
fffc3b00 +:  2f 83 00 02  cmpiw  cr7,r3,0x0002
fffc3b04 +:  40 be 00 1c  bne+   cr7,0xfffc3b20 <robots__unsafe+00000020>
fffc3b08 +:  2f 84 00 01  cmpiw  cr7,r4,0x0001
fffc3b0c +:  41 9e 00 0c  beq-   cr7,0xfffc3b18 <robots__unsafe+00000018>
fffc3b10 +:  2f 84 00 02  cmpiw  cr7,r4,0x0002
fffc3b14 +:  40 be 00 0c  bne+   cr7,0xfffc3b20 <robots__unsafe+00000020>
fffc3b18 +:  38 60 00 01  li     r3,0x0001
fffc3b1c +:  4e 80 00 20  blr
fffc3b20 +:  38 60 00 00  li     r3,0x0000
```

In this example, the set of traces to consolidate was provided as a sequence of trace filenames on the command line. A text file containing a list of trace files, designated by an `@` prefixed filename, may also be used for this purpose. In the example at hand, we could have, say, an `explore.tracelist` text file containing

```
test_explore.trace
explore.trace
```

and pass `@explore.tracelist` to `xcov coverage` to consolidate. The trace order has no influence in either case.

To help traceability, `gnatcov` provides the full list of traces used to assess reported results, in two possible places: in the `html` indexes and in the preliminary information part of the `--annotate=report` output when such formats are requested. Each trace filename is listed together with information recorded in the trace by `xcov run`: the trace creation timestamp, the path-to-executable command line argument, and the provided `--tag` value.

The example provided here focuses on object coverage analysis for illustrative puroposes. GNATcoverage's consolidation capabilities apply identically to the source coverage analysis case, with multiple execution traces on input and `--scos` to specify the sources subset of interest.

3.5.2 XML outputs for automated processing

In addition to the `report`, `xcov[+]` and `html[+]` output formats, `gnatcov` offers XML outputs for all the supported criteria. These outputs are obtained with '`--annotate=xml`' on the command line, which generates an XML file for each source file of interest, plus an `index.xml` which includes all the others.

XML outputs are intended for automated processing by other tools, and provide a representation of `gnatcov` internal computations with full details for maximum flexibility. Their specification is provided as an appendix of this document.

3.5.3 Source Coverage Exemptions

In some circumstances, there are good and well understood reasons why proper coverage of some source statement or decision is not achievable, and it is convenient to be able to abstract these coverage violations away from the genuine defects out of a testing sequence. The GNATcoverage *exemptions* facility was designed for this purpose.

For Ada with the GNAT compilers, coverage exemptions are requested for sections of source by the insertion of dedicated pragmas. `pragma Annotate (Xcov, Exempt_On, "justification text");` starts a section, providing some exemption justification text that will be recalled in coverage reports. `pragma Annotate (Xcov, Exempt_Off);` closes the current exemption section. There may be no overlap between exemption regions.

Exempted regions are reported as blocks in both the annotated source and the synthetic text reports. In the former case, a '#' or '*' caracter annotates all the exempted lines, respectively depending on whether 0 or at least 1 violation was exempted over the whole section. In synthetic text reports, a single indication is emitted for each exempted region, and the indications for all the regions are grouped in a separate report section. More details on the format of these indications is provided in the appendix section dedicated to the synthetic text report format.

4 Appendices

4.1 The "Explore" Guide Example

The Explore example is a toy Ada application we use throughout the `gnatcov` documentation to introduce and illustrate a number of concepts. Below is a short functional and organisational description, verbatim from the sources:

```
------------------------------------------------------------------------------
--                         Couverture/Explore example                       --
--                    Copyright (C) 2008-2009, AdaCore                      --
------------------------------------------------------------------------------

-------------------------
-- Functional overview --
-------------------------

--  This example implements software pieces involved in the following
--  imaginary situation:
--
--  - A pilotless robot is set on a field to explore, where we expect to
--    find either regular ground or rock blocks.
--
--  - An inhabited station off the field communicates with the robot to
--    control it and visualize the explored portions of the field.

--  The robot is equipped with several devices:
--
--  - a steering engine, able to have the robot perform a short step
--    forward or rotate 90 deg either way without advancing,
--
--  - a front radar, able to probe the field characteristics (ground or
--    block) one step ahead,
--
--  - a locator, able to evaluate the robot's position and orientation on
--    the field.
--
--  The robot communicates with the station via two communication links:
--
--  - a control link, through which the station sends commands for the
--    robot to execute (probe ahead, step forward, ...)
--
--  - a situation link, through which the robot sends info about it's
--    current situation (position, orientation, field ahead).
--
--  The field is modeled as a set of squares, each represented by a
--  single character to denote the square nature:
--
--     '#' is a rock block, ' ' is regular ground, '~' is water
--
--     '?' is a square yet unexplored
--
--     '<', '>', '^' or 'v' is the robot heading west, east, north
--     or south, respectively.

--  Below is a schematic illustration of the various parts involved
```

```
--  for a rectangular field delimited by rock blocks, a robot heading
--  south and an 'S' to materialize the Station:
--
--            field                               view
--          ##########                          ??????????
--          #  #     #  <- control link         ?  #  ? ??
--          #    v<==========================>S ??   R  ??
--          #  ~     #   situation link ->      ?  ?    ??
--          ##########                          ??????????
--
--  The Robot and Station active entities are both called Actors in this
--  world, and Links are attached to local Ports owned by these actors.
--
--  The Robot runs in one of two modes: in Cautious mode, it wont execute
--  unsafe commands like stepping forward with a rock block ahead.  In Dumb
--  mode it executes all the commands it receives.

--------------------
-- Sample session --
--------------------

--  A user running the program is like sitting at the Station, able to type
--  commands for the Robot to execute and to receive feedback on the new
--  situation, from which the local view of the field is updated.  Below is
--  sample session with explanatory comments in []:
--
--  $ ./explore
--
--  [The fake initial field is setup as a 5x5 area with blocks all around
--   and 1 in the middle, like
--
--          #####
--          #   #
--          # # #
--          #   #
--          #####
--
--   Robot is arbitrarily placed in the upperwest corner, heading east]
--
--  'C'autious mode, 'D'umb mode
--  'P'robe, 'S'tep, Rotate 'L'eft/'R'ight, 'Q'uit ? P
--
--       [Station asks for user input, user types "P" to probe ahead.
--        Robot answers with it's current situation: position (x=2, y=2),
--        looking east, square ahead is regular ground.
--        Station displays its current knowledge of the field]:
--
--  ?????
--  ?> ??
--  ?????
--  ?????
--  ?????
--
--  [Probe command processing done, Next cycle ...]
--
--  'C'autious mode, 'D'umb mode
--  'P'robe, 'S'tep, Rotate 'L'eft/'R'ight, 'Q'uit ? L
--
```

```
--       [User asks "rotate left (counterclockwise)", station sends out
--        command followed by a Probe request.
--        Robot processes both requests and answers: position unchanged,
--        now heading north, square ahead is a block.
--        Station displays its current knowledge of the field]:
--
--  ?#???
--  ?^ ??
--  ?????
--  ?????
--  ?????
--
--  [etc until user requests 'Q'uit or crashes the robot by asking
--   it to step forward into a block, in Dumb mode]
--

-----------------------------------
-- General software organization --
-----------------------------------

--  The set of units and essential abstractions involved is sketched below -
--  units underlined, associated abstractions parenthesized).
--
--  This set is a pretty straightforward mapping of the general concepts
--  involved, as described in the Functional Overview.  Only "Queues" is a
--  pure support unit, offering the well known datastructure abstraction.
--
--  See the package specs for extra descriptive details.
--
--                              =======
--                              Explore
--                              =======
--
--   Geomaps (field Map, Situations + Situation_Links)
--   =======
--
--   Actors (Actor)         Robots (Robot)        Stations (Station)
--   ======                 ======                ========
--
--   Links (IOports, IOlinks)      Queues (Queue)
--   =====                         ======
--
--   Controls (Robot_Control + Robot_Control_Links)
--   ========

package Overview is
end Overview;
```

4.2 Trace Format Definition

This information is best located and maintained in the source comments, where it naturally gets updated as the project evolves. Below is a verbatim inclusion of the relevant Ada specification:

```
------------------------------------------------------------------------------
--                                                                          --
--                              Couverture                                  --
--                                                                          --
--                     Copyright (C) 2008-2009, AdaCore                     --
--                                                                          --
-- Couverture is free software; you can redistribute it  and/or modify it   --
-- under terms of the GNU General Public License as published by the Free   --
-- Software Foundation; either version 2, or (at your option) any later     --
-- version.  Couverture is distributed in the hope that it will be useful,  --
-- but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHAN-  --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details. You  should  have  received a copy of the GNU --
-- General Public License  distributed with GNAT; see file COPYING. If not, --
-- write  to  the Free  Software  Foundation,  59 Temple Place - Suite 330, --
-- Boston, MA 02111-1307, USA.                                              --
--                                                                          --
------------------------------------------------------------------------------
with Interfaces; use Interfaces;

package Qemu_Traces is

   --  Execution of a program with 'xcov run' produces an "Execution Trace"
   --  file, possibly controlled by an internal "Trace Control" file for the
   --  simulation engine to help the support of mcdc like coverage criteria.

   --  The Trace Control simulation input contains a list of addresses ranges
   --  for which branch history is needed. This is computed by xcov from SCO
   --  decision entries, and is referred to as a Decision Map.

   --  The Execution Trace output contains a list of execution trace entries
   --  generated by the simulation engine, preceded by a list of trace
   --  information entries produced by xcov for items such as the path to the
   --  binary file or a user provided tag string.

   --  Here is a quick sketch of the information flow:

   --                          XCOV run                  Execution Trace
   --                o--------------------------------o   --------------
   --                |    gen info section -----------|-->|Info section|
   --                |                                |   |------------|
   --                |                       QEMU --|-->|Exec section|
   --  mcdc : SCO.D --|--> gen decision map -----^     |   --------------
   --                |    -----------------           |
   --                |    |Control section|           |
   --                |    -----------------           |
   --                o--------------------------------o

   --  All the files sections feature a section header followed by a sequence
   --  of entries. The section header structure is identical in all cases, and
   --  always conveys some trace related data (trace control, trace context
   --  info, or actual execution trace), identified by a Kind field.
```

```
--  The decision map file general structure is then:

--      -------
--      |SH   |    Section Header .Kind = Decision_Map
--      |TCE[]|    Sequence of Trace Control Entries
--      -------

--  And that of the output execution tracefile is:

--      -------
--      |SH   |    Section Header .Kind = Info
--      |TIE[]|    Sequence of Trace Info Entries
--      |-----|
--      |SH   |    Section Header .Kind = Flat|History
--      |ETE[]|    Sequence of Exec Trace Entries
--      -------

-------------------------
-- File Section Header --
-------------------------

--  Must be kept consistent with the C version in qemu-traces.h

subtype Magic_String is String (1 .. 12);

Qemu_Trace_Magic : constant Magic_String := "#QEMU-Traces";
--  Expected value of the Magic field.

Qemu_Trace_Version : constant Unsigned_8 := 1;
--  Current version

type Trace_Kind is (Flat, History, Info, Decision_Map);
for Trace_Kind use
  (Flat         => 0,   --  flat exec trace (qemu)
   History      => 1,   --  exec trace with history (qemu)
   Info         => 2,   --  info section (xcov)
   Decision_Map => 3);  --  history control section (xcov, internal)
for Trace_Kind'Size use 8;

type Trace_Header is record
   Magic : Magic_String;      --  Magic string
   Version : Unsigned_8;      --  Version of file format
   Kind : Trace_Kind;         --  Section kind

   Sizeof_Target_Pc : Unsigned_8;
   --  Size of Program Counter on target, in bytes

   Big_Endian : Boolean;
   --  True if the host is big endian

   Machine_Hi : Unsigned_8;
   Machine_Lo : Unsigned_8;
   --  Target ELF machine ID

   Padding : Unsigned_16;
   --  Reserved, must be set to 0
end record;
```

```
----------------------------------------------
-- Trace Information Section (.Kind = Info) --
----------------------------------------------

--  The section header fields after Kind (but big_endian) should be 0.

--  The section contents is a sequence of Trace Info Entries, each with a
--  Trace Info Header followed by data. Data interpretation depends on the
--  entry Kind found in the item header.  We expect an Info_End kind of
--  entry to finish the sequence.

type Info_Kind_Type is
  (Info_End, Exec_File_Name, Coverage_Options, User_Data, Date_Time);

type Trace_Info_Header is record
   Info_Kind   : Unsigned_32;
   --  Info_Kind_Type'Pos, in endianness indicated by file header

   Info_Length : Unsigned_32;
   --  Length of associated real data. This must be 0 for Info_End.

end record;

--  The amount of space actually occupied in the file for each entry is
--  always rounded up for alignment purposes. This is NOT reflected in
--  the Info_Length header field.

Trace_Info_Alignment : constant := 4;

--  This is the structure of a Date_Time kind of entry:

type Trace_Info_Date is record
   Year  : Unsigned_16;
   Month : Unsigned_8;   --  1 .. 12
   Day   : Unsigned_8;   --  1 .. 31
   Hour  : Unsigned_8;   --  0 .. 23
   Min   : Unsigned_8;   --  0 .. 59
   Sec   : Unsigned_8;   --  0 .. 59
   Pad   : Unsigned_8;   --  0
end record;

---------------------------------------------------
-- Execution Trace Section (.Kind = Raw|History) --
---------------------------------------------------

--  The section contents is a sequence of Trace Entries. There is no
--  explicit sequence termination entry ; we expect the section to end with
--  the container file.

--  Each trace entry conveys OPerational data about a range of machine
--  addresses, most often execution of a basic block terminated by a branch
--  instruction. These have slightly different representations for 32 and
--  64 bits targets.

--  Flat sections are meant to convey the directions taken by branches as
--  observed locally, independently of their execution context.  This
--  limits the output to at most two entries per block (one per possible
--  branch outcome) and doesn't allow mcdc computation.
```

```
   --  History sections are meant to allow mcdc computation, so report block
   --  executions and branch outcomes in the relevant cases, as directed by
   --  the simulator decision map input.

   type Trace_Entry64 is record
      Pc   : Unsigned_64;
      Size : Unsigned_16;
      Op   : Unsigned_8;
      Pad0 : Unsigned_8;
      Pad1 : Unsigned_32;
   end record;

   type Trace_Entry32 is record
      Pc   : Unsigned_32;
      Size : Unsigned_16;
      Op   : Unsigned_8;
      Pad0 : Unsigned_8;
   end record;

   --  The Operation conveyed is a bitmask of the following possibilities:

   Trace_Op_Block : constant Unsigned_8 := 16#10#;
   --  Basic block pc .. pc+size-1 was executed

   Trace_Op_Fault : constant Unsigned_8 := 16#20#;
   --  Machine fault occurred at pc

   Trace_Op_Br0 : constant Unsigned_8 := 16#01#;
   Trace_Op_Br1 : constant Unsigned_8 := 16#02#;
   --  Op_Block execution terminated with branch taken in direction 0 or 1

   -------------------------------------------
   -- Decision Map or Trace Control Section --
   -------------------------------------------

   --  The section contents is a sequence of Trace Control Entries.

   --  Entries are meant to convey range of addresses where branch history is
   --  needed for mcdc computation purposes. The structure is piggybacked on
   --  that of the Execution Trace output section, which has everything to
   --  represent address ranges already.

end Qemu_Traces;
```

4.3 Source Coverage Obligations Definition

Below is a verbatim inclusion of the relevant Ada specification:

```
------------------------------------------------------------------------------
--                                                                          --
--                         GNAT COMPILER COMPONENTS                         --
--                                                                          --
--                                 S C O S                                  --
--                                                                          --
--                                 S p e c                                  --
--                                                                          --
--          Copyright (C) 2009-2011, Free Software Foundation, Inc.         --
--                                                                          --
-- GNAT is free software;  you can  redistribute it  and/or modify it under --
-- terms of the  GNU General Public License as published  by the Free Soft- --
-- ware  Foundation;  either version 3,  or (at your option) any later ver- --
-- sion.  GNAT is distributed in the hope that it will be useful, but WITH- --
-- OUT ANY WARRANTY;  without even the  implied warranty of MERCHANTABILITY --
-- or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License --
-- for  more details.  You should have  received  a copy of the GNU General --
-- Public License  distributed with GNAT; see file COPYING3.  If not, go to --
-- http://www.gnu.org/licenses for a complete copy of the license.          --
--                                                                          --
-- GNAT was originally developed  by the GNAT team at  New York University. --
-- Extensive contributions were provided by Ada Core Technologies Inc.      --
--                                                                          --
------------------------------------------------------------------------------

--  This package defines tables used to store Source Coverage Obligations. It
--  is used by Par_SCO to build the SCO information before writing it out to
--  the ALI file, and by Get_SCO/Put_SCO to read and write the text form that
--  is used in the ALI file.

with Snames; use Snames;
--  Note: used for Pragma_Id only, no other feature from Snames should be used,
--  as a simplified version is maintained in Xcov.

with Types;  use Types;

with GNAT.Table;

package SCOs is

   --  SCO information can exist in one of two forms. In the ALI file, it is
   --  represented using a text format that is described in this specification.
   --  Internally it is stored using two tables SCO_Table and SCO_Unit_Table,
   --  which are also defined in this unit.

   --  Par_SCO is part of the compiler. It scans the parsed source tree and
   --  populates the internal tables.

   --  Get_SCO reads the text lines in ALI format and populates the internal
   --  tables with corresponding information.

   --  Put_SCO reads the internal tables and generates text lines in the ALI
   --  format.

   --------------------
```

```
-- SCO ALI Format --
--------------------

--  Source coverage obligations are generated on a unit-by-unit basis in the
--  ALI file, using lines that start with the identifying character C. These
--  lines are generated if the -gnateS switch is set.

--  Sloc Ranges

--    In several places in the SCO lines, Sloc ranges appear. These are used
--    to indicate the first and last Sloc of some construct in the tree and
--    they have the form:

--      line:col-line:col

--    Note that SCO's are generated only for generic templates, not for
--    generic instances (since only the first are part of the source). So
--    we don't need generic instantiation stuff in these line:col items.

--  SCO File headers

--    The SCO information follows the cross-reference information, so it
--    need not be read by tools like gnatbind, gnatmake etc. The SCO output
--    is divided into sections, one section for each unit for which SCO's
--    are generated. A SCO section has a header of the form:

--      C dependency-number filename

--        This header precedes SCO information for the unit identified by
--        dependency number and file name. The dependency number is the
--        index into the generated D lines and is ones origin (i.e. 2 =
--        reference to second generated D line).

--        Note that the filename here will reflect the original name if
--        a Source_Reference pragma was encountered (since all line number
--        references will be with respect to the original file).

--        Note: the filename is redundant in that it could be deduced from
--        the corresponding D line, but it is convenient at least for human
--        reading of the SCO information, and means that the SCO information
--        can stand on its own without needing other parts of the ALI file.

--  Statements

--    For the purpose of SCO generation, the notion of statement includes
--    simple statements and also the following declaration types:

--      type_declaration
--      subtype_declaration
--      object_declaration
--      renaming_declaration
--      generic_instantiation

--    and the following regions of the syntax tree:

--      the part of a case_statement from CASE up to the expression
--      the part of a FOR loop iteration scheme from FOR up to the
--        loop_parameter_specification
```

```
--      the part of a WHILE loop up to the condition
--      the part of an extended_return_statement from RETURN up to the
--        expression (if present) or to the return_subtype_indication (if
--        no expression)

--    and any pragma that occurs at a place where a statement or declaration
--    is allowed.

--  Statement lines

--    These lines correspond to one or more successive statements (in the
--    sense of the above list) which are always executed in sequence (in the
--    absence of exceptions or other external interruptions).

--    Entry points to such sequences are:

--      the first declaration of any declarative_part
--      the first statement of any sequence_of_statements that is not in a
--        body or block statement that has a non-empty declarative part
--      the first statement after a compound statement
--      the first statement after an EXIT, RAISE or GOTO statement
--      any statement with a label (the label itself is not part of the
--       entry point that is recorded).

--    Each entry point must appear as the first entry on a CS line.
--    The idea is that if any simple statement on a CS line is known to have
--    been executed, then all statements that appear before it on the same
--    CS line are certain to also have been executed.

--    The form of a statement line in the ALI file is:

--      CS *sloc-range [*sloc-range...]

--    where each sloc-range corresponds to a single statement, and * is
--    one of:

--      t        type declaration
--      s        subtype declaration
--      o        object declaration
--      r        renaming declaration
--      i        generic instantiation
--      C        CASE statement (from CASE through end of expression)
--      E        EXIT statement
--      F        FOR loop (from FOR through end of iteration scheme)
--      I        IF statement (from IF through end of condition)
--      P[name:] PRAGMA with the indicated name
--      R        extended RETURN statement
--      W        WHILE loop statement (from WHILE through end of condition)

--      Note: for I and W, condition above is in the RM syntax sense (this
--      condition is a decision in SCO terminology).

--    and is omitted for all other cases

--    Note: up to 6 entries can appear on a single CS line. If more than 6
--    entries appear in one logical statement sequence, continuation lines
--    are marked by Cs and appear immediately after the CS line.
```

```
--    Implementation permission: a SCO generator is permitted to emit a
--    narrower SLOC range for a statement if the corresponding code
--    generation circuitry ensures that all debug information for the code
--    implementing the statement will be labeled with SLOCs that fall within
--    that narrower range.

--  Decisions

--    Note: in the following description, logical operator includes only the
--    short-circuited forms and NOT (so can be only NOT, AND THEN, OR ELSE).
--    The reason that we can exclude AND/OR/XOR is that we expect SCO's to
--    be generated using the restriction No_Direct_Boolean_Operators if we
--    are interested in decision coverage, which does not permit the use of
--    AND/OR/XOR on boolean operands. These are permitted on modular integer
--    types, but such operations do not count as decisions in any case. If
--    we are generating SCO's only for simple coverage, then we are not
--    interested in decisions in any case.

--    Note: the reason we include NOT is for informational purposes. The
--    presence of NOT does not generate additional coverage obligations,
--    but if we know where the NOT's are, the coverage tool can generate
--    more accurate diagnostics on uncovered tests.

--    A top level boolean expression is a boolean expression that is not an
--    operand of a logical operator.

--    Decisions are either simple or complex. A simple decision is a top
--    level boolean expression that has only one condition and that occurs
--    in the context of a control structure in the source program, including
--    WHILE, IF, EXIT WHEN, or immediately within an Assert, Check,
--    Pre_Condition or Post_Condition pragma, or as the first argument of a
--    dyadic pragma Debug. Note that a top level boolean expression with
--    only one condition that occurs in any other context, for example as
--    right hand side of an assignment, is not considered to be a (simple)
--    decision.

--    A complex decision is a top level boolean expression that has more
--    than one condition. A complex decision may occur in any boolean
--    expression context.

--    So for example, if we have

--        A, B, C, D : Boolean;
--        function F (Arg : Boolean) return Boolean);
--        ...
--        A and then (B or else F (C and then D))

--    There are two (complex) decisions here:

--        1. X and then (Y or else Z)

--           where X = A, Y = B, and Z = F (C and then D)

--        2. C and then D

--    For each decision, a decision line is generated with the form:

--      C* sloc expression [chaining]
```

```
--    Here * is one of the following characters:

--      E  decision in EXIT WHEN statement
--      G  decision in entry guard
--      I  decision in IF statement or conditional expression
--      P  decision in pragma Assert/Check/Pre_Condition/Post_Condition
--      W  decision in WHILE iteration scheme
--      X  decision appearing in some other expression context

--    For E, G, I, P, W, sloc is the source location of the EXIT, ENTRY, IF,
--    PRAGMA or WHILE token, respectively

--    For X, sloc is omitted

--    The expression is a prefix polish form indicating the structure of
--    the decision, including logical operators and short-circuit forms.
--    The following is a grammar showing the structure of expression:

--      expression ::= term             (if expr is not logical operator)
--      expression ::= &sloc term term  (if expr is AND or AND THEN)
--      expression ::= |sloc term term  (if expr is OR or OR ELSE)
--      expression ::= !sloc term       (if expr is NOT)

--      In the last three cases, sloc is the source location of the AND, OR,
--      or NOT token, respectively.

--      term ::= element
--      term ::= expression

--      element ::= *sloc-range

--    where * is one of the following letters:

--      c  condition
--      t  true condition
--      f  false condition

--      t/f are used to mark a condition that has been recognized by the
--      compiler as always being true or false. c is the normal case of
--      conditions whose value is not known at compile time.

--    & indicates AND THEN connecting two conditions

--    | indicates OR ELSE connecting two conditions

--    ! indicates NOT applied to the expression

--    Note that complex decisions do NOT include non-short-circuited logical
--    operators (AND/XOR/OR). In the context of existing coverage tools the
--    No_Direct_Boolean_Operators restriction is assumed, so these operators
--    cannot appear in the source in any case.

--    The SCO line for a decision always occurs after the CS line for the
--    enclosing statement. The SCO line for a nested decision always occurs
--    after the line for the enclosing decision.

--    Note that membership tests are considered to be a single simple
```

```
--    condition, and that is true even if the Ada 2005 set membership
--    form is used, e.g. A in (2,7,11.15).

--    The expression can be followed by chaining indicators of the form
--    Tsloc-range or Fsloc-range, where the sloc-range is that of some
--    entry on a CS line.

--    T* is present when the statement with the given sloc range is executed
--    if, and only if, the decision evaluates to TRUE.

--    F* is present when the statement with the given sloc range is executed
--    if, and only if, the decision evaluates to FALSE.

--    For an IF statement or ELSIF part, a T chaining indicator is always
--    present, with the sloc range of the first statement in the
--    corresponding sequence.

--    For an ELSE part, the last decision in the IF statement (that of the
--    last ELSIF part, if any, or that of the IF statement if there is no
--    ELSIF part) has an F chaining indicator with the sloc range of the
--    first statement in the sequence of the ELSE part.

--    For a WHILE loop, a T chaining indicator is always present, with the
--    sloc range of the first statement in the loop, but no F chaining
--    indicator is ever present.

--    For an EXIT WHEN statement, an F chaining indicator is present if
--    there is an immediately following sequence in the same sequence of
--    statements.

--    In all other cases, chaining indicators are omitted

--    Implementation permission: a SCO generator is permitted to emit a
--    narrower SLOC range for a condition if the corresponding code
--    generation circuitry ensures that all debug information for the code
--    evaluating the condition will be labeled with SLOCs that fall within
--    that narrower range.

--  Case Expressions

--    For case statements, we rely on statement coverage to make sure that
--    all branches of a case statement are covered, but that does not work
--    for case expressions, since the entire expression is contained in a
--    single statement. However, for complete coverage we really should be
--    able to check that every branch of the case statement is covered, so
--    we generate a SCO of the form:

--      CC sloc-range sloc-range ...

--    where sloc-range covers the range of the case expression

--    Note: up to 6 entries can appear on a single CC line. If more than 6
--    entries appear in one logical statement sequence, continuation lines
--    are marked by Cc and appear immediately after the CC line.

--  Disabled pragmas

--    No SCO is generated for disabled pragmas
```

```
---------------------------------------------------------------------
-- Internal table used to store Source Coverage Obligations (SCOs) --
---------------------------------------------------------------------

type Source_Location is record
   Line : Logical_Line_Number;
   Col  : Column_Number;
end record;

No_Source_Location : Source_Location := (No_Line_Number, No_Column_Number);

type SCO_Table_Entry is record
   From : Source_Location := No_Source_Location;
   To   : Source_Location := No_Source_Location;
   C1   : Character       := ' ';
   C2   : Character       := ' ';
   Last : Boolean         := False;

   Pragma_Sloc : Source_Ptr := No_Location;
   --  For the statement SCO for a pragma, or for any expression SCO nested
   --  in a pragma Debug/Assert/PPC, location of PRAGMA token (used for
   --  control of SCO output, value not recorded in ALI file).

   Pragma_Name : Pragma_Id := Unknown_Pragma;
   --  For the statement SCO for a pragma, gives the pragma name
end record;

package SCO_Table is new GNAT.Table (
  Table_Component_Type => SCO_Table_Entry,
  Table_Index_Type     => Nat,
  Table_Low_Bound      => 1,
  Table_Initial        => 500,
  Table_Increment      => 300);

--  The SCO_Table_Entry values appear as follows:

--    Statements
--      C1   = 'S' for entry point, 's' otherwise
--      C2   = statement type code to appear on CS line (or ' ' if none)
--      From = starting source location
--      To   = ending source location
--      Last = False for all but the last entry, True for last entry

--    Note: successive statements (possibly interspersed with entries of
--    other kinds, that are ignored for this purpose), starting with one
--    labeled with C1 = 'S', up to and including the first one labeled with
--    Last = True, indicate the sequence to be output for a sequence of
--    statements on a single CS line (possibly followed by Cs continuation
--    lines).

--    Note: for a pragma that may be disabled (Debug, Assert, PPC, Check),
--    the entry is initially created with C2 = 'p', to mark it as disabled.
--    Later on during semantic analysis, if the pragma is enabled,
--    Set_SCO_Pragma_Enabled changes C2 to 'P' to cause the entry to be
--    emitted in Put_SCOs.

--    Decision (EXIT/entry guard/IF/WHILE)
```

```
--      C1   = 'E'/'G'/'I'/'W' (for EXIT/entry Guard/IF/WHILE)
--      C2   = ' '
--      From = EXIT/ENTRY/IF/WHILE token
--      To   = No_Source_Location
--      Last = unused

--    Decision (PRAGMA)
--      C1   = 'P'
--      C2   = ' '
--      From = PRAGMA token
--      To   = No_Source_Location
--      Last = unused

--    Note: when the parse tree is first scanned, we unconditionally build a
--    pragma decision entry for any decision in a pragma (here as always in
--    SCO contexts, the only pragmas with decisions are Assert, Check,
--    dyadic Debug, Precondition and Postcondition). These entries will
--    be omitted in output if the pragma is disabled (see comments for
--    statement entries).

--    Decision (Expression)
--      C1   = 'X'
--      C2   = ' '
--      From = No_Source_Location
--      To   = No_Source_Location
--      Last = unused

--    Operator
--      C1   = '!', '&', '|'
--      C2   = ' '
--      From = location of NOT/AND/OR token
--      To   = No_Source_Location
--      Last = False

--    Element (condition)
--      C1   = ' '
--      C2   = 'c', 't', or 'f' (condition/true/false)
--      From = starting source location
--      To   = ending source location
--      Last = False for all but the last entry, True for last entry

--    Element (chaining indicator)
--      C1   = 'H' (cHain)
--      C2   = 'T' or 'F' (chaining on decision true/false)
--      From = starting source location of chained statement
--      To   = ending source location of chained statement

--    Note: the sequence starting with a decision, and continuing with
--    operators and elements up to and including the first one labeled with
--    Last = True, indicate the sequence to be output on one decision line.

----------------
-- Unit Table --
----------------

--  This table keeps track of the units and the corresponding starting and
--  ending indexes (From, To) in the SCO table. Note that entry zero is
--  present but unused, it is for convenience in calling the sort routine.
```

```
   --  Thus the lower bound for real entries is 1.

   type SCO_Unit_Index is new Int;
   --  Used to index values in this table. Values start at 1 and are assigned
   --  sequentially as entries are constructed.

   type SCO_Unit_Table_Entry is record
      File_Name : String_Ptr;
      --  Pointer to file name in ALI file

      Dep_Num : Nat;
      --  Dependency number in ALI file

      From : Nat;
      --  Starting index in SCO_Table of SCO information for this unit

      To : Nat;
      --  Ending index in SCO_Table of SCO information for this unit
   end record;

   package SCO_Unit_Table is new GNAT.Table (
     Table_Component_Type => SCO_Unit_Table_Entry,
     Table_Index_Type     => SCO_Unit_Index,
     Table_Low_Bound      => 0, -- see note above on sorting
     Table_Initial        => 20,
     Table_Increment      => 200);

   -----------------
   -- Subprograms --
   -----------------

   procedure Initialize;
   --  Reset tables for a new compilation

end SCOs;
```

4.4 XML output specifications

Below is a verbatim inclusion of the relevant Ada specification:

```
------------------------------------------------------------------------------
--                                                                          --
--                              Couverture                                  --
--                                                                          --
--                    Copyright (C) 2009-2010, AdaCore                      --
--                                                                          --
-- Couverture is free software; you can redistribute it  and/or modify it   --
-- under terms of the GNU General Public License as published by the Free   --
-- Software Foundation; either version 2, or (at your option) any later     --
-- version.  Couverture is distributed in the hope that it will be useful,  --
-- but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHAN-  --
-- TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public --
-- License  for more details. You  should  have  received a copy of the GNU --
-- General Public License  distributed with GNAT; see file COPYING. If not, --
-- write  to  the Free  Software  Foundation,  59 Temple Place - Suite 330, --
-- Boston, MA 02111-1307, USA.                                              --
--                                                                          --
------------------------------------------------------------------------------

package Annotations.Xml is

   --  This package provides support to output coverage results in XML format.
   --  To make this easily useable by an external tool, there is only one
   --  single entry for the XML output. To avoid to make this file a monster,
   --  it is broken down into sub-units by the use of the Xinclude standard.
   --
   --  The following files are generated:
   --
   --  * an index file, named index.xml;
   --  * one file per compilation unit, named after the corresponding source
   --  file with a suffix ".xml".
   --
   --  The following sections will describe each file type. The following
   --  convention will be used to denotate possible values for attributes:
   --
   --  * COVERAGE_KIND: can be either 'insn', 'branch', 'stmt',
   --              'stmt+decision', 'stmt+mcdc'.
   --  * COVERAGE: can be either '+' (total coverage for the chosen coverage
   --              criteria), '-' (null coverage), '!' (partial coverage) or
   --              '.' (no code for this line).
   --  * OBJ_COVERAGE: can be either '+' (covered), '>' (branch taken),
   --              'V' (branch fallthrough) and '-' (not covered).
   --  * TEXT: any text into quotes. Mostly used for source lines.
   --  * ADDRESS: an hexademical number, C convention. e.g. 0xdeadbeef.
   --  * NUM: a decimal number.
   --
   --
   --  Index :
   --  -------
   --
   --
   --  Description :
   --  ............
   --
   --  The index file contains one root element:
```

```
--
-- <coverage_report>: it contains the following attributes:
--
--    coverage_level: COVERAGE_KIND; type of coverage operation that has
--                    been recorded in this report.
--
-- <coverage_report> has the following child elements:
--
--    <coverage_info>: information related to the coverage operation
--       (e.g. list of trace files).
--       This should contain a list of child elements <xi:include>
--       with the following attributes:
--
--       parse : set to "xml"
--       href  : path to the file that contains a trace info file.
--
--
--    <sources>: List of annotated source files. This should contain a list
--       of child elements <xi:include> with the following attributes:
--
--
--       parse : set to "xml"
--       href  : path to the file that contains an annotated source
--               report.
--
-- Example:
-- ........
--
-- Consider a program hello.adb that contains a package
-- pack.adb.  Suppose that two runs have been done for this program,
-- generating two trace files trace1/hello.trace and
-- trace2/hello.trace. Its branch coverage report would look like:
--
-- <?xml version="1.0" ?>
-- <document xmlns:xi="http://www.w3.org/2001/XInclude">
--  <coverage_report coverage_level="stmt">
--
--   <coverage_info>
--     <xi:include parse="xml" href="trace.xml"/>
--   </coverage_info>
--
--   <sources>
--     <xi:include parse="xml" href="hello.adb.xml"/>
--     <xi:include parse="xml" href="pack.adb.xml"/>
--   </sources>
--
--  </coverage_report>
--
-- </document>
--
--
-- Trace info :
-- ------------
--
--
-- Description:
-- ............
--
```

```
--  The trace info contains one root element:
--
--  <traces>: it represents the list of trace files given to the coverage
--     tool. It should contain a list of the following child elements:
--
--     <trace>: represents a given trace file. It shall have the following
--        attributes:
--
--        filename : name of the trace file on the host file system.
--        program  : name of the executable program on the host file system.
--        date     : date of the run that generated the trace file.
--        tag      : trace file tag.
--
--  Example:
--  ........
--
--  <?xml version="1.0" ?>
--  <traces>
--    <trace filename="explore1.trace"
--           program="explore"
--           date="2009-06-18 18:19:17"
--           tag="first run"/>
--
--    <trace filename="explore2.trace"
--           program="explore"
--           date="2009-06-18 18:22:32"
--           tag="second run"/>
--  </traces>
--
--
--  Annotated compilation unit :
--  ----------------------------
--
--  Some preliminary discussion first. A priori, there are two ways to
--  organize the coverage information in an annotated source:
--  * source-based view: iterating on lines; for each line, coverage
--    items (instruction/statement/decision...) are included.
--  * coverage-based view: iterating on coverage items; for each item, line
--    information is given.
--
--  Both approaches have their utility; the source-based view makes it easy
--  to generate source-based html reports (similar to the one generated by
--  --annotate=html+); the coverage-based view, closer to what the SCOs
--  provide, can more easily express the structure of decisions (the
--  condition that they contain, and which values they have taken).
--  The limitation of one approach is actually the asset of the other: a
--  coverage-centric report would make it hard for an external to rebuild
--  the source out of it; at the contrary, a source-centric report would
--  make it painful to aggregates informations about a particular decision.
--
--  The xml format proposed here tries to take the advantages of both
--  worlds.  Instead of starting from lines or from coverage item and
--  trying to make one a child of the other, this format is based on
--  an element that pairs the two together. That is to say, instead of
--  having:
--
--  [...]
--  <line num="1" src="      A := 1;">
```

```
--     <statement_start coverage="+"/>
--  </line>
--  [...]
--
--  or something like:
--
--  [...]
--  <statement line_begin="1" line_end="2" coverage="+" src="A := 1;"/>
--  [...]
--
--  we will have:
--
--  [...]
--  <src_mapping>
--    <src>
--      <line num="1" src="      A := 1;"/>
--    </src>
--
--    <statement coverage="+"/>
--  </src_mapping>
--  [...]
--
--  What we call here a "src mapping" is the relation between a set of
--  line in the source code and a tree of coverage items.
--
--  One property that we would then be able to inforce is: monotonic
--  variation of src lines. More clearly: if a src mapping has a child
--  element src that contains line 12 and 13, the src mapping before it
--  will contain line 11, the src mapping after it will contain line 14.
--  This will ease the  generation of a human-readable (say, HTML) report
--  based on source lines; remember, that was one of the good properties
--  of the line-based approach.
--
--  Now, let us have a look to the details...
--
--  Description :
--  .............
--
--  The annotated compilation unit contains one root element:
--
--  <source>: it contains the following attributes:
--
--     file           : TEXT; path to the source file.
--     coverage_level : COVERAGE_KIND; type of coverage operation that has
--                      been recorded in this report.
--
--  It may contain a list of the followind child elements:
--
--     <src_mapping>: node that associate a fraction of source code to
--        coverage item. It may have the following attribute:
--
--        coverage: aggregated coverage information for this fraction of
--                  source code.
--
--        It should contains the following mandatory child element...
--
--        <src>: node that contains a list of contiguous source lines of
--               code.
```

```
--          It contains a list of the following child elements:
--
--          <line/>: represents a line of source code. It shall have the
--                   following attributes:
--
--             num : NUM; line number in source code.
--             src : TEXT; copy of the line as it appears in the source
--                   code.
--
--
--
--      ...and <src_mapping> may also contain a list of child elements
--      that represents coverage items. These coverage items can be
--      instruction sets, statements or decision. Here are the
--      corresponding child elements:
--
--       <message/>: represents an error message or a warning attached to
--       this line. It can have the following attributes:
--
--          kind    : warning or error
--          SCO     : Id of the SCO to which this message is attached
--          message : actual content of the message
--
--       <instruction_set>: node that represents a set of instructions.
--       It should contain the following attribute:
--
--          coverage : COVERAGE; coverage information associated to this
--          instruction set.
--
--          The element <instruction_set> may also contain a list of the
--          following child elements:
--
--             <instruction_block>: coverage information associated to
--                contiguous instructions. It has the following attributes:
--
--                name     : TEXT; name of the symbol. e.g. "main",
--                           "_ada_p".
--                offset   : ADDRESS; offset from the symbol.
--                coverage : COVERAGE; how this instruction block
--                           is covered.
--
--                The element <instruction_block> may contain a list of the
--                following child elements:
--
--                   <instruction/>: coverage information associated to
--                      a given instruction. it contains the following
--                      attributes:
--
--                      address  : ADDRESS;
--                      coverage : OBJ_COVERAGE; how this instruction has
--                         been covered.
--                      assembly : TEXT; assembly code for this
--                         instruction.
--
--
--       <statement>: represents a statement. It may contain the
--          following attributes:
--
```

```
--          coverage : COVERAGE; coverage information associated to a
--             statement.
--          id : NUM; identifier of the associated source coverage
--             obligation
--          text : TEXT; short extract of code used that can be used to
--             identify the corresponding source entity.
--
--
--          The element <statement> may contain one child element:
--
--             <src>: source information associated to this statement. If
--                no src node is given, then the src of the upper node is
--                "inherited".
--                Same thing for conditions, decisions, statements...
--
--             The element <src> may contain a list of the following child
--             elements:
--
--                <line/>: represents a line of source code. It may have
--                   the following attributes:
--
--                   num          : NUM; line number in source code.
--                   column_begin : NUM; column number for the beginning
--                                  of the coverage item we are
--                                  considering.
--                   column_end   : NUM; column number for the end of the
--                                  coverage item we are considering.
--                   src          : TEXT; copy of the line as it appears
--                                  in the source code.
--
--       <decision>: represents a decision. It may contain the following
--          attributes:
--
--          coverage : COVERAGE; coverage information associated to a
--             statement.
--          id : NUM; identifier of the associated source coverage
--             obligation
--          text : TEXT; short extract of code used that can be used to
--             identify the corresponding source entity.
--
--          The element <decision> may also contain the following child
--          elements:
--
--             <src>: same as its homonym in <statement>; see above.
--
--             <condition>: represents a condition. It may contains the
--             following attributes:
--
--                coverage : COVERAGE; coverage information associated to a
--                   statement.
--                id : NUM; identifier of the associated source coverage
--                   obligation
--                text : TEXT; short extract of code used that can be
--                   used to identify the corresponding source entity.
--
--             ...and the following child elements:
--
--                <src>: same as its homonym in <statement>; see above.
```

```
--
--  Example:
--  ........
--
--  Consider the following Ada function, defined in a file named test.adb:
--
--  --  file test.adb
--
--  with Pack;
--
--  function Test
--    (A : Boolean;
--     B : Boolean;
--     C : Boolean;
--     D : Boolean) return Integer is
--  begin
--     if A and then (B or else F (C
--                                 and then D))
--        return 12;
--     end if;
--     Pack.Func; return 13;
--  end Test;
--
--
--  This coverage of this file can be represented by the report shown below.
--  Notice in particular:
--  * how the two statements at line 14 can be represented;
--  * how the coverage of the two decisions on line 11-12 are represented.
--
--  <?xml version="1.0" ?>
--  <source file="test.adb" coverage_level="stmt+mcdc">
--     <src_mapping coverage=".">
--        <src>
--           <line num="1" src="--  file test.adb"/>
--           <line num="2" src=""/>
--           <line num="3" src="with Pack;"/>
--           <line num="4" src=""/>
--           <line num="5" src="function Test"/>
--           <line num="6" src="  (A : Boolean;"/>
--           <line num="7" src="   B : Boolean;"/>
--           <line num="8" src="   C : Boolean;"/>
--           <line num="9" src="   D : Boolean) return Integer is"/>
--           <line num="10" src="begin"/>
--        </src>
--     <src_mapping>
--
--     <src_mapping coverage="!">
--        <src>
--           <line num="11" src="   if A and then (B or else F (C"/>
--           # This src_mapping could also contain the line that follows;
--           # after all, the two decisions that it contains end on line
--           # 12. It does not matter much at this point. The important
--           # property is that every coverage entity that starts on line
--           # 11 is defined in this src_mapping.
--        </src>
--
--        <decision id="1" text="A and th..." coverage="!">
--           <src>
```

```
--             <line num="11" src="   if A and then (B or else F (C"/>
--             <line num="12"
--                   src="                               and then D))"/>
--           </src>
--
--
--           <condition id="2" text="A" coverage="+">
--              <src>
--                 <line num="11"
--                       column_begin="6"
--                       column_end="7"
--                       src="A"/>
--              </src>
--           </condition>
--
--           <condition id="3" text="B" coverage="-">
--              <src>
--                 <line num="11"
--                       column_begin="18"
--                       column_end="19"
--                       src="B"/>
--              </src>
--
--           </condition>
--
--           <condition id="4" text="F (C..." coverage="-">
--              <src>
--                 <line num="11"
--                       column_begin="28"
--                       src="F (C"/>
--                 <line num="12"
--                       src="                            and then D"/>
--              </src>
--
--           </condition>
--        </decision>
--
--
--        <decision id="5" text="C..." coverage="-">
--           <src>
--              <line num="11"
--                    column_begin="31"
--                    src="C"/>
--              <line num="12"
--                    column_end="41"
--                    src="                            and then D"/>
--           </src>
--
--           <condition id="6" text="C" coverage="-">
--              <src>
--                 <line num="11"
--                       column_begin="31"
--                       column_end="32"
--                       src="C"/>
--              </src>
--
--           </condition>
--
```

```
--          <condition id="7" text="D" coverage="-">
--             <src>
--                <line num="12"
--                      column_begin="40"
--                      column_end="41"
--                      src="D"/>
--             </src>
--          </condition>
--       </decision>
--
--       <message kind="warning"
--                SCO="SCO #3: CONDITION"
--                message="failed to show independent influence"/>
--       <message kind="warning"
--                SCO="SCO #4: CONDITION"
--                message="failed to show independent influence"/>
--       <message kind="error"
--                SCO="SCO #5: DECISION"
--                message="statement not covered"/>
--
--    </src_mapping>
--
--    <src_mapping coverage=".">
--       # As said previously, this line could have been included in the
--       # previous src_mapping.
--       <src>
--          <line num="12"
--                src="                              and then D))"/>
--       </src>
--    </src_mapping>
--
--    <src_mapping coverage="+">
--       <src>
--          <line num="13" src="      return 12;"/>
--       </src>
--
--       <statement id="8" text="return 1..." coverage="+"/>
--    </src_mapping>
--
--    <src_mapping>
--       <src>
--          <line num="13" src="   end if;"/>
--       </src>
--    </src_mapping>
--
--    <src_mapping coverage="+">
--       <src>
--          <line num="14" src="   Pack.Func; return 13;"/>
--       </src>
--
--       <statement id="9" text="Pack.Fun..." coverage="+">
--          <src>
--             <line num="14"
--                   column_begin="3"
--                   column_end="12"
--                   src="Pack.Func;"/>
--          </src>
--       </statement>
```

```
   --
   --       <statement id="9" text="return 1..." coverage="+">
   --          <src>
   --             <line num="14"
   --                   column_begin="14"
   --                   column_end="23"
   --                   src="return 13;"/>
   --          </src>
   --       </statement>
   --    </src_mapping>
   --
   --  </source>

   function To_Xml_String (S : String) return String;
   --  Return the string S with '>', '<' and '&' replaced by XML entities

   procedure Generate_Report;

end Annotations.Xml;
```

4.5 `--annotate=report` output format - source coverage

This section describes the format of the synthetic text report produced by the `--annotate=report` mode of GNATcoverage for source coverage criteria. We use a generated report as an example, filtering the elements relevant to the code excerpt below:

```
[...]
59    procedure Notify_Error_On (Q : in Queue) is
60    begin
61       raise Program_Error;
62    end Notify_Error_On;
63
64    procedure Push (Item : Data_Type; Q : in out Queue) is
65    begin
66       pragma Annotate (Xcov, Exempt_On, "we never overflow a Queue");
67       if Full (Q) then
68          Notify_Error_On (Q);
69       end if;
70       pragma Annotate (Xcov, Exempt_Off);
[...]
```

This code excerpt is extracted from a sample Queues data type abstraction. Queue overflows are expected never to happen, so an exemption section is in place for the code performing the corresponding check at the beginning of a Push operation. Below is a copy of the report produced for a sample run where the general `Notify_Error_On` subprogram is not called otherwise:

```
COVERAGE REPORT

1. OVERVIEW

Date and time of execution: 2010-11-10 18:16:59.00
Tool version: XCOV 1.0.0w (20081119)
Command line:
xcov coverage --scos=queues.ali --level=stmt+decision --annotate=report xplr.trace

Coverage level: stmt+decision

trace files:

 xplr.trace
  program: obj/powerpc-elf/explore
  date: 2010-11-10 17:16:47
  tag: some exemption test

2. NON-EXEMPTED VIOLATIONS

queues.adb:61:7: statement not executed
1 violation

3. EXEMPTED VIOLATIONS

queues.adb:66:7-70:7: 2 exempted violations, justification:
we never overflow a Queue

1 exempted region.

END OF REPORT
```

The start and end of report are explicit, and the report body features three sections: Overview, Non-exempted violations and Exempted violations.

The **Overview** section exposes elements about the report production context:

- Date & time when the report was produced
- Command line and Version of GNATcoverage that produced the report
- Coverage criterion assessed
- Details on the input trace files: path to binary program exercised (as provided on the command line), production time stamp, –tag argument to `xcov run` when the trace was produced

The **Exempted violations** section lists and counts the exempted regions, displaying for each the source location span, the number of actually exempted violations in the region, and the exemption justification text.

The **Non-exempted violations** section lists and counts the coverage violations (with respect to the assessed criteria) that relate to source lines not part of an exemption region. All the non-exempted violations are reported using a consistent format, as follows:

```
queues.adb:1641:17: statement not executed
 (source) : (loc) : (vfamily)  (details)
```

source and *loc* are the basename of the source file and the precise `line:column` location within that source where the violation was detected. *vfamily* identifies the family of coverage violation reported in this particular case, and *details* provides additional information. Below is the list of family/detail items that might be emitted together with the `--level` argument from which each may appear:

--level	**family**	**detail**
`=stmt`	`statement`	`not executed`
`=stmt+decision`	`decision`	`outcome TRUE not covered`
		`outcome FALSE not covered`
		`one outcome not covered`
`=stmt+mcdc`	`condition`	`has no independent influence pair`

Violations for one level may be issued while assessing stricter levels as well. For example, `"statement not executed"` or `"decision outcome TRUE not covered"` violations might be emitted in the course of a `stmt+mcdc` assessement.

When multiple violations apply someplace, the largest grain diagnostic is emitted alone. For instance, if an Ada statement like `"X := A and then B;"` is not covered at all, a `"statement not executed"` violation is emitted alone, even if we're assessing for, say, `--level=stmt+decision` ; `gnatcov` emits no decision oriented violation in this case.

5 Bibliography

[gcc] GCC: The GNU Compiler Collection. http://gcc.gnu.org

[qemu] QEMU, a Fast and Portable Dynamic Translator. Fabrice Bellard. Proceedings of the “USENIX 2005 Annual Technical Conference, FREENIX Track”, pp 41-46. http://bellard.org/qemu/

[mctut] A Practical Tutorial on Modified Condition/Decision Coverage. John J. Chilenski et al. NASA/TM-2001-210876, 2001.

[mcapp] Applicability of Modified Condition/Decision Coverage to Software Testing. John J. Chilenski and S. Miller. IEEE Software Engineering Journal, volume 9, issue 5, September 2004.

[cast6] Rationale for accepting Masking MCDC in certification projects. CAST, Certification Authorities Software Team. Position Paper #6, August 2001.

[cast10] What is a Decision in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC) ? CAST, Certification Authorities Software Team. Position Paper #10, June 2002.

[ar0118] An Investigation of Three Forms of the Modified Condition/Decision Coverage (MCDC) Criterion. John J. Chilenski. DOT/FAA/AR-01/18, April 2001.

[ar0654] Software Verification Tools Assessment Study. FAA, Federal Aviation Administration. DOT/FAA/AR-06/54, June 2007.

[ar0720] Object Oriented Technology Verification Phase 3 Report - Structural Coverage at the Source Code and Object Code Levels. John J. Chilenski and John L. Kurtz. DOT/FAA/AR-07/20, June 2007.

[obc-mcdc] Technical Report on OBC/MCDC properties. Jrme Guitton, Yannick Moy and Thomas Quinot. Couverture project, October 2010.

[rcdc] From MC/DC to RC/DC: Formalization and Analysis of Control-Flow Testing Criteria. S. Vilkomir and J. Bowen. ZB2002: Formal Specification and Development in Z and B, Springer LNCS 2272, 2002.

6 Index