GNAT DAS Manuals

Version 27.0w (20260427)

AdaCore

Apr 28, 2026

CONTENTS

I GNATcoverage User's Guide 1

1 Getting Started 3
- 1.1 General notions . . . 3
- 1.2 Workflow overview . . . 4
- 1.3 A simple example . . . 4
  - 1.3.1 Sample project . . . 4
  - 1.3.2 Setup, Instrument, Build, Execute, Analyze . . . 7
- 1.4 Going Further . . . 9
- 1.5 Conventions used in the rest of this manual . . . 9

2 Specifying *Units Of Interest* 11
- 2.1 Using project files (`-P`, `--projects`, `--units`) . . . 11
  - 2.1.1 Conveying *projects* of interest . . . 12
  - 2.1.2 Conveying *units* of interest within projects . . . 14
  - 2.1.3 Conveying *subprograms* of interest (experimental) . . . 15
  - 2.1.4 Other switches or attributes of importance . . . 15
- 2.2 Conveying source files to ignore / handling Ada subunits . . . 16
- 2.3 Compilation unit vs source file names . . . 18
- 2.4 Providing coverage obligation files (`--sid`) . . . 19

3 Producing source traces with `gnatcov instrument` 21
- 3.1 Setting up the coverage runtime library . . . 22
  - 3.1.1 Multiple runtimes in the same prefix . . . 22
  - 3.1.2 Language restrictions . . . 23
  - 3.1.3 Default source trace dump options . . . 23
  - 3.1.4 Coverage runtime setup for configurations with no Ada runtime . . . 24
- 3.2 Instrumenting programs . . . 25
  - 3.2.1 **`gnatcov instrument`** command line . . . 25
  - 3.2.2 Output strategies for coverage information . . . 26
  - 3.2.3 Resetting coverage information during the execution . . . 27
  - 3.2.4 Controlling trace file names . . . 28
  - 3.2.5 Support for preprocessing . . . 29
  - 3.2.6 **`gnatcov instrument`** limitations . . . 29
- 3.3 Optimizing the execution of instrumented programs . . . 33
- 3.4 Building instrumented components . . . 33
- 3.5 Extracting a trace from standard output . . . 34
- 3.6 Composed instrumentation . . . 34
- 3.7 Instrumentation of a multi-languages project . . . 35
- 3.8 Instrumentation and coverage of SPARK code . . . 35
  - 3.8.1 Inhibiting SPARK related pragmas . . . 35

3.8.2 Coverage obligations for SPARK code . . . 36
3.9 Example use cases . . . 37
3.9.1 Whole program instrumented at once, cross configuration, base64 output . . . 37
3.9.2 Library instrumented separately, native configuration, trace output . . . 39
3.10 Coverage runtime customization . . . 41
3.10.1 Basics . . . 41
3.10.2 Customize the `base64-stdout` dump channel . . . 42
3.10.3 When the customized runtime needs elaboration . . . 43
3.11 Building instrumented programs with CCG . . . 44
3.11.1 Customizing the runtime . . . 45
3.11.2 Building an instrumented program with CCG . . . 45
3.12 Common instrumentation errors . . . 46
3.12.1 Linker error when building instrumented application . . . 46

4 Producing source traces with integrated instrumentation (experimental) 47
4.1 Windows specifics . . . 47
4.2 Specifying instrumentation switches . . . 48
4.3 Building an instrumented executable . . . 48
4.4 Generating a coverage report . . . 48
4.5 A simple Makefile example . . . 48
4.6 A simple CMake example . . . 49

5 Source coverage analysis with `gnatcov coverage` 51
5.1 **`gnatcov coverage`** command line . . . 51
5.2 Output report formats (`--annotate`) . . . 54
5.2.1 Cobertura report (`=cobertura`) . . . 55
5.2.2 Annotated sources, html (`=html`) . . . 55
5.2.3 SARIF report (`=sarif`) . . . 55
5.2.4 Violations summary, text (`=report`) . . . 56
5.2.5 Annotated sources, text (`=xcov[+]`) . . . 58
5.2.6 XML report, xml (`=xml`) . . . 59
5.3 Statement Coverage analysis . . . 59
5.3.1 Core notions and Reporting (`--level=stmt`) . . . 59
5.3.2 Example program and assessments . . . 60
5.4 Decision Coverage analysis . . . 63
5.4.1 Core notions and Reporting (`--level=stmt+decision`) . . . 63
5.4.2 Example program and assessments . . . 64
5.5 Modified Condition/Decision Coverage analysis . . . 66
5.5.1 Core notions and Reporting (`--level=stmt+mcdc`) . . . 66
5.5.2 Example decisions and conditions . . . 67
5.5.3 Handling non short-circuit operators . . . 68
5.5.4 MCDC variants . . . 69
5.5.5 Example program and assessments . . . 70
5.6 Assertion True Coverage (ATC) analysis (experimental) . . . 72
5.6.1 Core notions and Reporting (`--level=...+atc`) . . . 72
5.7 Assertion True Condition Coverage (ATCC) analysis (experimental) . . . 73
5.7.1 Core notions and Reporting (`--level=...+atcc`) . . . 73
5.8 Function and Call Coverage (FUN_CALL) analysis (experimental) . . . 73
5.8.1 Core notions and Reporting (`--level=...+fun_call`) . . . 74
5.9 Guarded Expression Coverage (GEXPR) analysis (experimental) . . . 74
5.9.1 Core notions and Reporting (`--level=...+gexpr`) . . . 74
5.10 Synthetic metrics . . . 75
5.11 Handling source relocation for annotated sources output formats . . . 77
5.12 Inlining & Ada generic units . . . 78

5.12.1 Combined coverage on generics . . . 79
5.13 Handling Ada assertions and contracts . . . 80
5.14 Processing of C macros . . . 80
5.15 Optimization and non-coverable items . . . 82

6 Rust coverage analysis with `gnatcov coverage` 85
6.1 Rust coverage workflow . . . 85
6.1.1 Instrument and build the project . . . 85
6.1.2 Run the executed binary . . . 86
6.1.3 Produce a coverage report . . . 86
6.2 Rust coverage limitations . . . 86
6.2.1 MC/DC false-negatives related to instrumentation level . . . 86
6.2.2 Approximative source location ranges . . . 86
6.2.3 Unsupported constructs . . . 87

7 Coverage Consolidation 89
7.1 Consolidation from trace files . . . 89
7.1.1 Single unit tested by different programs . . . 89
7.1.2 Disjoint units tested by different programs . . . 92
7.1.3 Multiple executions of a single program . . . 93
7.2 Consolidation from coverage checkpoints . . . 96
7.2.1 Efficient storage of test results . . . 97
7.2.2 Incremental coverage analysis . . . 98
7.2.3 Handle incidental coverage effects . . . 98
7.2.4 Compatibility considerations . . . 99
7.3 Consolidating Coverage for Different Project Build Configurations . . . 100
7.3.1 Use SID files to use all source traces at once . . . 102
7.3.2 Use project loading options to load source traces separately . . . 104

8 Coverage Exemptions 105
8.1 Defining Exemption Regions . . . 105
8.2 Reporting about coverage exemptions . . . 106
8.2.1 Undetermined Coverage state and Exemptions . . . 108
8.3 Object coverage exemptions . . . 108

9 Disable coverage analysis 109
9.1 Defining Disabled Coverage Regions . . . 109
9.2 Consolidating traces with varying disabled coverage regions . . . 110
9.3 Reporting about disabled coverage regions . . . 110

10 External source annotations 113
10.1 Generating external annotations . . . 113
10.1.1 Semantics of buffer manipulation annotations . . . 115
10.1.2 File relocation considerations . . . 116
10.2 Deleting a pre-existing annotation . . . 117
10.3 Displaying the annotations contained in annotation files . . . 117
10.4 Annotation stability through file modifications . . . 118

11 Using project files 119
11.1 General considerations . . . 119
11.2 Specifying `gnatcov` command Switches . . . 119
11.3 Specifying the Target and language Runtime . . . 120

12 Using GNATcoverage from GNAT Studio 121
12.1 Basic scenario . . . 121

12.2 Viewing a coverage report produced outside of GNAT Studio . . . 124
12.3 Using GNATtest's coverage integration makefile . . . 125
12.4 GNATtest scenario . . . 125

13 Glossary of terms & concepts **127**

14 Appendices **129**
14.1 Sample html index . . . 129
14.2 Sample html annotated source . . . 130

II GNATtest User's Guide 131

1 Running `gnattest` **135**
1.1 Framework Generation Mode . . . 135
1.2 Test Execution Mode . . . 136

2 Switches for `gnattest` in framework generation mode **137**

3 Switches for `gnattest` in test execution mode **141**

4 Project Attributes for `gnattest` **143**

5 Simple Example **145**

6 Setting Up and Tearing Down the Testing Environment **147**

7 Regenerating Tests **149**

8 Default Test Behavior **151**

9 Testing Primitive Operations of Tagged Types **153**

10 Testing Inheritance **155**

11 Tagged Type Substitutability Testing **157**

12 Testing with Contracts **159**

13 Additional Tests **161**

14 Individual Test Drivers **163**

15 Stubbing **165**

16 Integration with GNATcoverage **167**

17 Putting Tests under Version Control **169**

18 Aggregate project handling **171**
18.1 Artifact directories and aggregate projects . . . 171

19 Current Limitations **173**

20 Automatically generating test cases (experimental) **175**
20.1 Setting up the test generation runtime . . . 175
20.2 Generating test inputs . . . 176

III GNATfuzz User's Guide 177

1 Introduction 179
1.1 Fuzz-testing in a nutshell . . . 179
1.2 AFL++ basics . . . 180
1.3 GNATfuzz overview . . . 182

2 GNATfuzz command line interface 185
2.1 GNATfuzz *"analyze"* mode command line interface . . . 185
2.2 GNATfuzz *"generate"* mode command line interface . . . 186
2.3 GNATfuzz *"generate-corpus"* mode command line interface . . . 186
2.4 GNATfuzz *"minimize-corpus"* mode command line interface . . . 187
2.5 GNATfuzz *"fuzz"* mode command line interface . . . 187

3 GNATfuzz by example 189
3.1 Prerequisites and possible configurations . . . 189
3.1.1 AFL++ requirements . . . 189
3.1.2 GNATfuzz requirements . . . 189
3.1.3 GNATfuzz configurations . . . 190
3.1.4 Accommodating linker flags required by the user project . . . 190
3.1.5 GNATfuzz and the AFL++ execution modes . . . 190
3.2 GNATfuzz automation capabilities and limitations . . . 193
3.3 Fuzzing a simple Ada subprogram . . . 194
3.3.1 Using *analyze* mode . . . 194
3.3.2 Using *generate* mode . . . 195
3.3.3 Using **generate-corpus** mode . . . 196
3.3.4 Using *minimize-corpus* mode . . . 196
3.3.5 Decoding an auto-generated test case . . . 197
3.3.6 Using *fuzz* mode . . . 197
3.4 Fuzzing with non-automatically generated starting corpus . . . 202
3.4.1 Fuzzing with user-provided seeds . . . 202
3.4.2 Fuzzing with a dummy starting corpus . . . 205
3.5 Fuzzing instantiations of generic subprograms . . . 206
3.6 Fuzzing global and private state . . . 211
3.7 Fuzzing Access Types . . . 215
3.8 Filtering exceptions . . . 217
3.9 Fuzzing C functions through Ada bindings . . . 220
3.10 Resetting dynamically allocated memory when fuzz-testing with the AFL++ `persistent` mode . . . 224
3.11 CMPLOG (aka RedQueen) support . . . 228

4 The "User_Configuration" package 229

5 GNATfuzz stopping criteria 231

6 GNATfuzz integration in GNAT Studio 233
6.1 The GNATfuzz menu . . . 233
6.2 Viewing analysis results and generating a fuzz-testing harness . . . 233
6.3 Running a fuzz session . . . 234
6.4 Viewing and debugging fuzz crashes . . . 234

7 FAQ for GNATfuzz 239

8 GNATfuzz diagnostics and issue reporting 241
8.1 GNATfuzz Error Codes and Exit Statuses . . . 242
8.1.1 Error_Unknown . . . 242

8.1.2 Invalid_Command_Line_Options . . . 242
8.1.3 Invalid_GNATfuzz_Configuration_Options . . . 242
8.1.4 Invalid_Fuzzable_Subprogram . . . 242
8.1.5 Error_Initializing_Fuzzing_Campaigns . . . 243
8.1.6 Error_Launching_Fuzzing_Campaigns . . . 243
8.1.7 Error_During_Fuzzing_Campaigns . . . 243
8.1.8 Error_Terminating_Campaigns . . . 243
8.1.9 Error_Initializing_Supporting_Tool . . . 243
8.1.10 Error_Executing_Supporting_Tool . . . 243
8.1.11 Error_Generating_Starting_Corpus . . . 244

9 GNATfuzz Beta feature 245
9.1 Incremental fuzzing with TGen test vectors . . . 245
9.1.1 GNATfuzz *"import-tgen-vectors"* mode command line interface . . . 245
9.1.2 Incremental fuzzing workflow . . . 246
9.1.3 Beta experimental TGen flags for *"fuzz"* mode . . . 246
9.2 GNATfuzz and Symbolic Execution . . . 247
9.2.1 Enabling SymCC . . . 248
9.2.2 Current Limitations . . . 248
9.3 GNATfuzz Adaptive - "On-target" fuzzing (experimental) . . . 248
9.3.1 What is it? . . . 248
9.3.2 How to use? . . . 249
9.3.3 Setup the development environment . . . 249
9.3.4 Configure the fuzzer . . . 249
9.3.5 Understanding the results . . . 250
9.3.6 Example usage . . . 250
9.3.7 Known limitations . . . 252
9.4 Introducing the hot-spot workflow . . . 252
9.4.1 When to use the hot-spot workflow . . . 253
9.4.2 Prerequisites . . . 253
9.4.3 Hot-spot command line interface . . . 253
9.4.4 End-to-end example . . . 254
9.4.5 Workflow steps . . . 257
9.4.6 Terminal output . . . 258
9.4.7 Crash correlation and confidence scoring . . . 261
9.4.8 Augmented SARIF output . . . 262
9.4.9 Machine-readable summary report . . . 265
9.4.10 Coverage reports . . . 268
9.4.11 TGen test vector integration . . . 269
9.4.12 Output directory layout . . . 271
9.5 Introducing the fuzz-everything workflow . . . 272
9.5.1 Accessing the fuzz-everything workflow . . . 272
9.5.2 Fuzz-everything *"inspect"* mode command line interface . . . 273
9.5.3 Fuzz-everything *"test"* mode command line interface . . . 274
9.5.4 Fuzz-everything *"inspect"* mode usage example . . . 275
9.5.5 Fuzz-everything *"test"* mode usage example . . . 276
9.5.6 Utilizing fuzz-everything-generated test cases with GNATtest . . . 283
9.6 Libfuzzer . . . 284
9.6.1 Libfuzzer basics . . . 284
9.6.2 Libfuzzer requirements . . . 284
9.6.3 GNATfuzz and the libfuzzer execution configuration . . . 284

IV Using GNATtest with GNATcoverage 287

1 Using the Makefile generated by GNATtest 291
- 1.1 Special considerations for cross environment targets . . . 291
- 1.2 Usage example . . . 292
- 1.3 Instrumenting test harnesses for a SPARK project . . . 297

2 Developing a custom solution 299
- 2.1 Single test driver . . . 299
- 2.2 Separate test drivers . . . 299

V Using GNATtest with GNATfuzz (experimental) 301

1 Setting up the environment 305

2 Using GNATStudio to perform the integration 307

VI Appendix A. Using GNATcoverage Binary Traces 309

1 Introduction 311
- 1.1 Definition and main characteristics . . . 311
- 1.2 Workflow overview . . . 312
- 1.3 A simple example . . . 313
- 1.4 Going Further . . . 314

2 Producing binary traces with `gnatcov run` 315
- 2.1 Compilation prerequisites . . . 315
  - 2.1.1 General rules . . . 315
  - 2.1.2 Consolidation related rules . . . 315
- 2.2 **`gnatcov run`** command line . . . 316
- 2.3 Execution environment . . . 317
- 2.4 Program selection and Arguments passing . . . 317
- 2.5 Trace file contents . . . 319

3 Object coverage analysis with `gnatcov coverage` 321
- 3.1 **`gnatcov coverage`** command line . . . 321
- 3.2 Output report formats (`--annotate`) . . . 323
  - 3.2.1 Machine level reports (`=asm`) . . . 323
  - 3.2.2 Annotated sources, text (`=xcov[+]`) . . . 324
  - 3.2.3 Annotated sources, html (`=html`) . . . 325
- 3.3 Object Instruction Coverage analysis (`--level=insn`) . . . 325
- 3.4 Object Branch Coverage analysis (`--level=branch`) . . . 326
- 3.5 Focusing on subprograms of interest (`--routines`) . . . 327
- 3.6 Inlining & Ada Generic units . . . 328
- 3.7 Full object coverage considerations . . . 331

4 Object/Source level metrics considerations 333
- 4.1 Main differences exemplified . . . 333
- 4.2 Full branch coverage vs MCDC . . . 334

5 Using GNATcoverage from GNAT Studio 337

6 Importing binary traces with `gnatcov convert` 339
- 6.1 **`gnatcov convert`** command line . . . 339

6.2 Generating Coverage Information from Nexus Traces on MPC5634M . . . 340
6.2.1 Nexus Program Trace Data: the Hardware Perspective . . . 340
6.2.2 Configuring the winIDEA Analyzer . . . 341
6.2.3 Exporting the Trace Data . . . 341
6.3 Generating Coverage Information from Lauterbach Trace32 . . . 342
6.3.1 Enabling Trace Data . . . 342
6.3.2 Running the executable . . . 342
6.3.3 Exporting the Trace Data . . . 342
6.3.4 Automation . . . 343

7 Target specific points of note with Binary traces **345**

8 Known Limitations with Binary traces **347**
8.1 Incomplete statement coverage on lines with multiple statements . . . 347
8.2 MCDC inaccuracies with interrupts and multi-threaded applications . . . 347

Index **349**

Part I

GNATcoverage User's Guide

CHAPTER

ONE

GETTING STARTED

1.1 General notions

GNATcoverage is a code coverage analysis tool offering support for a range of coverage metrics and output formats associated with powerful *consolidation* features letting users assess the combined coverage achievements of multiple program executions. It supports Ada, C and C++.

The set of compilation units for which a user needs to assess coverage is commonly designated as the set of *units of interest*. This is a central notion to many of the tool's operations, conveyed by the user through command line switches or GPR project file attributes. Units of interest typically include the code under test in a campaign, as opposed to the sources of the test harness infrastructure.

After one or several program executions, the tool computes *coverage metrics* for a given set of units out of *coverage trace* data produced by the executions with the assistance of an *instrumentation* mechanism. The primary mode offered by GNATcoverage performs *source instrumentation*, where the tool produces a modified version of the program sources to keep track of coverage relevant facts along with the program control flow, and output the coverage data when the program terminates. We call *source traces* the coverage traces produced by programs instrumented this way.

The set of metrics that the tool can assess from source traces corresponds to the set defined by the **DO-178C** certification standard for civil avionics, that is:

- *Statement Coverage*, where the tools assesses the coverage status (executed or not) of source statements such as a variable assignment or a subprogram call;
- *Decision Coverage*, which, in addition to statement coverage, evaluates whether Boolean expressions (*decisions* in **DO-178C** parlance) have been exercised both True and False, then
- *Modified Condition/Decision Coverage*, commonly known as *MCDC*, which requires testing particular variations of individual Boolean operands (*conditions* in **DO-178C** parlance) within decisions.

All these metrics are defined with respect to source level entities (statements, decisions, conditions), and we commonly refer to such assessments as *source coverage analysis*. Individual statements, decisions, or conditions within the scope of an analysis are referred to as *Source Coverage Obligations* or *SCOs*.

1.2 Workflow overview

Figure Fig. 1 depicts the workflow involved in computing coverage based on source instrumentation.

A setup step is first needed to install a tiny runtime library used by the instrumented sources to register coverage facts and output them eventually. This runtime is parameterized for the project at hand, depending on the set of languages involved (Ada, C, C++) and on the kind of target for which the code will be compiled (native, cross with OS, or bareboard). Once the setup is done, users proceed with instrument / build / execute / analyze cycles to produce traces and compute coverage results.

The setup, instrumentation and analysis steps are all driven by a **gnatcov** command line tool which exposes a subcommand for each kind of operation; hence **gnatcov setup** for the setup step, **gnatcov instrument** to instrument and **gnatcov coverage** to compute metrics from traces.

GPR project files and associated command line switches are used to let the tool know about the project sources and to convey the units of interest. Consolidation can be performed by aggregating multiple traces directly, or internal representations of partial coverage results stored in files that we call *Coverage Checkpoints*.

gnatcov also provides an alternate workflow that consists in integrating seamlessly into an existing build process possibly not involving the use of gprbuild. This is provided as an experimental feature and is available for C/C++, compiling with gcc/g++ on a linux host. The figure Fig. 2 depicts this workflow that is more thoroughly detailed in the section *Producing source traces with integrated instrumentation (experimental)*.

1.3 A simple example

Here we first introduce a very simple example software project structure then demonstrate one basic analysis workflow for this project with actual commands.

The examples provided in most of this manual assume a native configuration and the process is similar cross environments. Typical variations would touch switches controlling where and when coverage trace data is output by the instrumented programs, and default values for such switches are selected by the tool based on `--target` and `--RTS` arguments at setup time.

1.3.1 Sample project

The sources for this examples are provided under the *share/examples/gnatcoverage/doc/getting_started/* directory of the GNAT DAS distribution.

We consider an Ada package providing a set of elementary operations over `Integer` objects, with a spec and body in source files named `ops.ads` and `ops.adb`:

```
-- ops.ads
package Ops is
  type Op_Kind is (Increment, Decrement);

  procedure Apply (Op : Op_Kind; X : in out Integer);
end;
```

```
-- ops.adb
package body Ops is
  procedure Apply (Op : Op_Kind; X : in out Integer) is
  begin
     case Op is
```

(continues on next page)

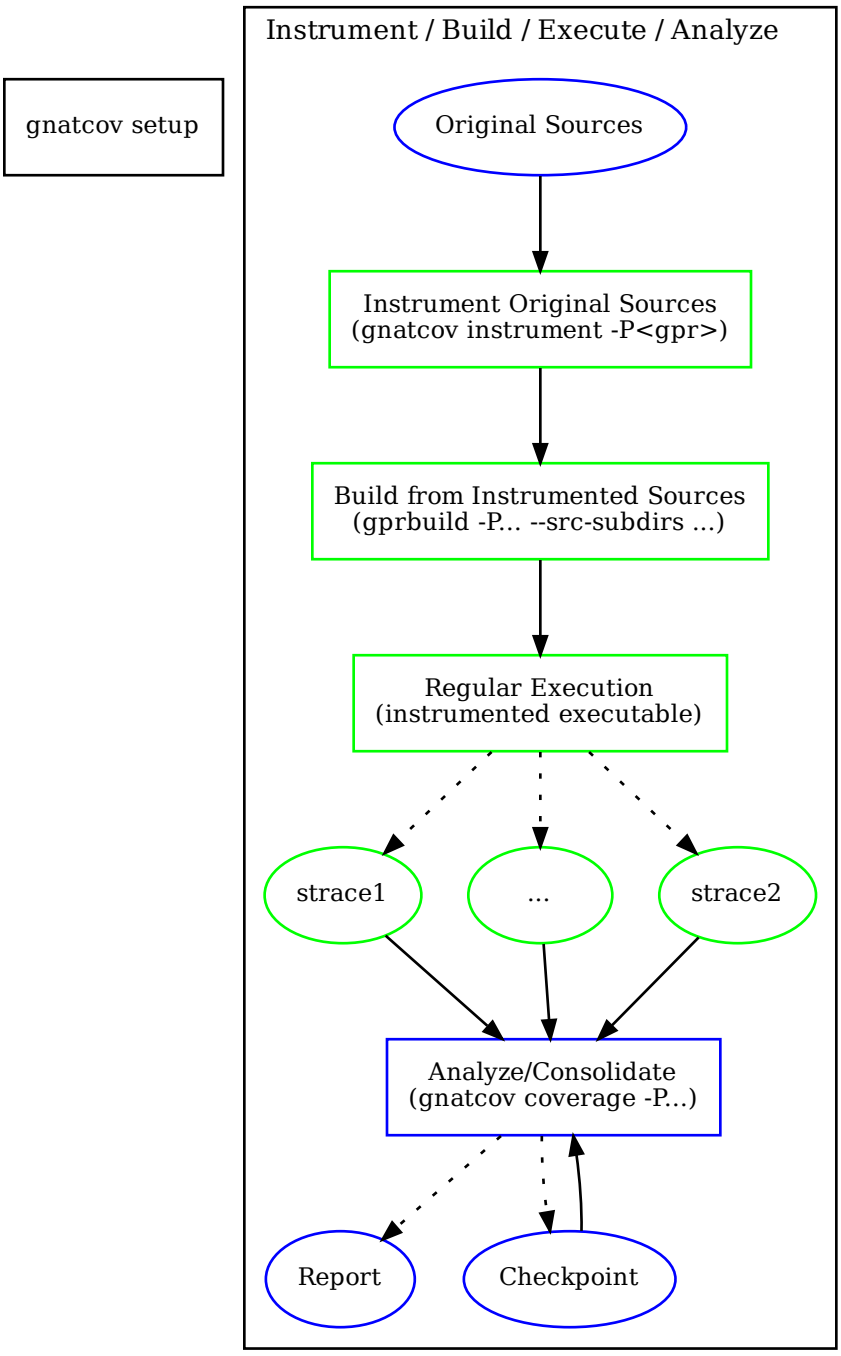


Fig. 1: Source instrumentation based Coverage analysis overview

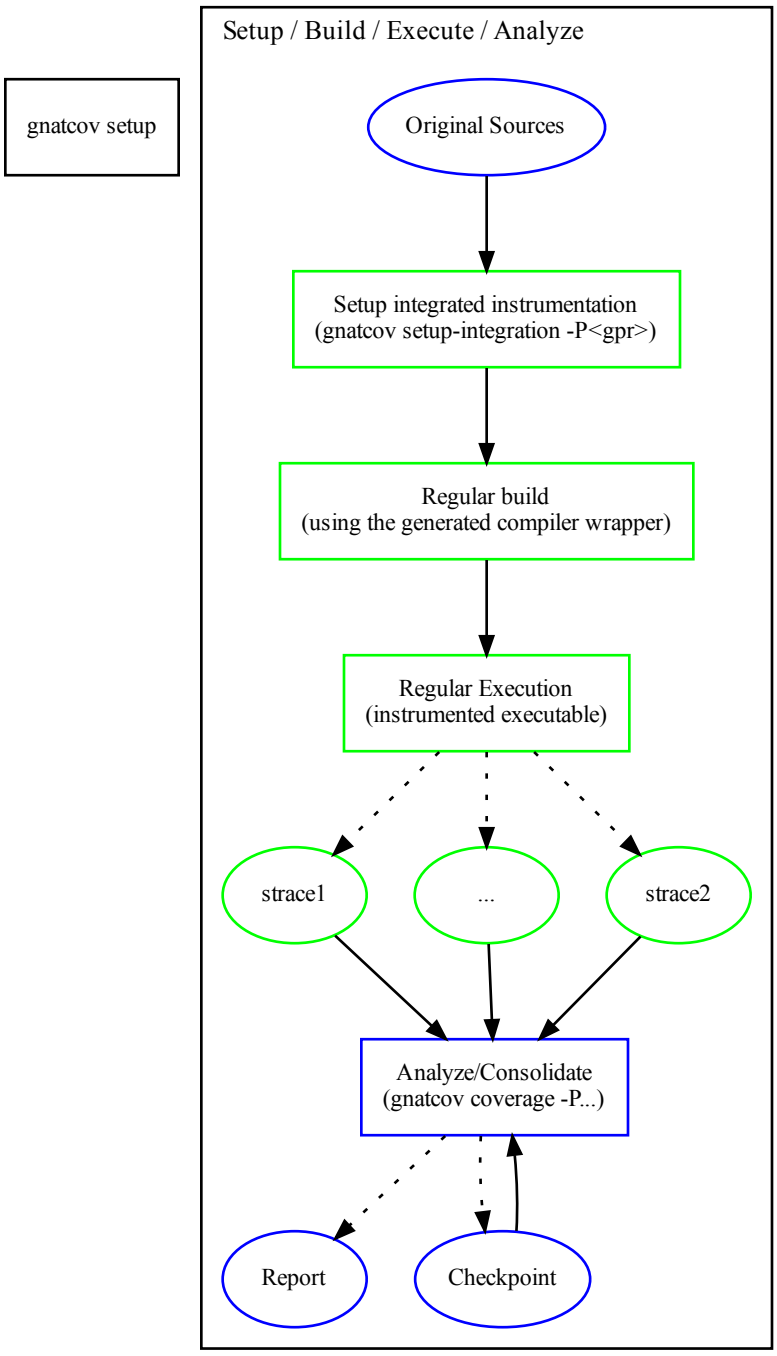


Fig. 2: Integrated instrumentation based Coverage analysis overview

(continued from previous page)

```
        when Increment => X := X + 1;
        when Decrement => X := X - 1;
     end case;
  end;
end;
```

We will analyze the coverage achieved by the sample unit *test driver* below, in `test_inc.adb`, which exercises the `Increment` operation only:

```
-- test_inc.adb
with Ops;
procedure Test_Inc is
  X : Integer := 4;
begin
  Ops.Apply (Ops.Increment, X);
  pragma Assert (X = 5);
end;
```

Assuming a working directory, with the *ops* sources in an `opslib` subdirectory and the *test* sources in a `tests` subdirectory, we will use a couple of project files in the common working directory:

```
-- code.gpr
project Code is
  for Source_Dirs use ("opslib");
  for Object_Dir use "obj-" & Project'Name;
end Code;
```

```
-- tests.gpr
with "code.gpr";

project Tests is
  for Source_Dirs use ("tests");
  for Object_Dir use "obj-" & Project'Name;

  for Main use ("test_inc.adb");
end Tests;
```

1.3.2 Setup, Instrument, Build, Execute, Analyze

The instrumentation step that follows assumes that the original program is well formed. A simple way to verify this is to build the non instrumented version first. For our example, this would be:

```
gprbuild -f -p -Ptests.gpr
```

We then first set up the instrumentation context, providing a local *prefix* location where the runtime and default parameters for future commands are going to be installed:

```
gnatcov setup --prefix=/path/to/gnatcov-rts
```

Letting further commands know about the *prefix* location is achieved by adding `<prefix>/share/gpr` to the `GPR_PROJECT_PATH` variable. In a Unix like environment, this would be:

```
export GPR_PROJECT_PATH=$GPR_PROJECT_PATH:/path/to/gnatcov-rts/share/gpr
```

This will both let the `gprbuild` command below locate the `gnatcov_rts.gpr` project file, and the **`gnatcov instrument`** command find default parameter values.

In addition, when using shared libraries, it is necessary to let the environment know about the coverage runtime's own shared libraries. The following command achieves this in a Unix like environment:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/gnatcov-rts/lib
```

The following command performs the corresponding action on Windows systems:

```
set PATH=%PATH%;C:\path\to\gnatcov-rts\bin\
```

Instrumentation is performed by a simple **`gnatcov instrument`** command:

```
gnatcov instrument -Ptests.gpr --level=stmt
```

The use of `tests.gpr`, not `code.gpr`, at this step is important as it lets the instrumenter know about the main subprogram, which needs to be processed specially to dump coverage data.

Building the instrumented program then goes like:

```
gprbuild -f -p -Ptests.gpr \
   --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts.gpr
```

This is the same command as for the regular build, with a couple of additional switches to:

- Instruct the builder to search for the instrumented versions of the sources (`--src-subdirs`). Note that, due to the way `--src-subdirs` works in `gprbuild`, even though **`gnatcov instrument`** creates directories with names derived from `*-gnatcov-instr`, the argument to pass to GPRbuild must always be exactly `--src-subdirs=gnatcov-instr`.
- Provide visibility to the builder over the coverage runtime referenced by the instrumented sources (`--implicit-with`).

Executing the test program in its native environment, as in:

```
obj-tests/test_inc
```

then produces a `test_inc-<stamp>.srctrace` source trace file in the current directory. The `-<stamp>` suffix is intended to prevent clashes in case of concurrent executions of the program in the same directory. It can be controlled in a variety of ways from the instrumentation command line, documented in the *Controlling trace file names* section of this manual.

Analysis of the coverage achieved by previous executions is done with **`gnatcov coverage`** commands. For our example use case, this could for instance be:

```
gnatcov coverage --level=stmt --annotate=xcov test_inc*.srctrace -Ptests.gpr
```

Here, we request:

- A source *statement coverage* assessment with `--level=stmt`,
- An annotated source report in text format with `--annotate=xcov`,
- For the complete set of units involved in the executable with `-Ptests.gpr`.

This produces annotated sources in the project's object directory, with `ops.adb.xcov` quoted below:

```
examples/starter/src/ops.adb:
67% of 3 lines covered
Coverage level: stmt
 1 .: package body Ops is
 2 .:    procedure Apply (Op : Op_Kind; X : in out Integer) is
 3 .:    begin
 4 +:       case Op is
 5 +:          when Increment => X := X + 1;
 6 -:          when Decrement => X := X - 1;
 7 .:       end case;
 8 .:    end Apply;
 9 .: end Ops;
```

The analysis results are visible as + / - annotations on source lines, next to the line numbers. The results we have here indicate proper coverage of all the statements except the one dealing with a `Decrement` operation, indeed never exercised by our driver.

The command actually also produces reports for `ops.ads` and `test_inc.adb`, even though the latter is not really relevant. Focus on specific units can be achieved by providing a more precise set of units of interest at this step, for example by adding `--projects=code.gpr` to the command line, or setting dedicated attributes in the project files themselves. See the *Specifying Units Of Interest* chapter for details on this aspect of the procedure.

1.4 Going Further

Each of the steps involved in the process overview presented previously is described in detail in a specific chapter of this manual. The most important ones are:

- *Producing source traces with gnatcov instrument*
- *Specifying Units Of Interest*
- *Source coverage analysis with gnatcov coverage*
- *Coverage Consolidation*

Coverage Exemptions is also worth noting here, a mechanism allowing users to define code regions for which coverage violations are expected and legitimate.

The known limitations of the tool are outlined in section *gnatcov instrument limitations*.

1.5 Conventions used in the rest of this manual

- A number of example commands include a `--level=<>` switch, which conveys a target coverage criterion when needed. <> is a placeholder for an actual level supported by the tool in this case, such as `stmt`, `stmt+decision`, or `stmt+mcdc` for source criteria.
- Example command lines might also include as `<units-of-interest>` placeholder, which represents a set of switches conveying the set of units for interest for source coverage assessments. GPR project files provide the most elaborate mechanisms for this purpose and the *Specifying Units Of Interest* chapter describes all the available options.

CHAPTER

TWO

SPECIFYING *UNITS OF INTEREST*

This chapter describes the means to convey the set of units on which coverage should be assessed, which we commonly refer to as the *units of interest*.

The first opportunity to do so is at **gnatcov instrument** time, with GPR project file oriented command line switches possibly associated with coverage specific attributes within project files. This step controls which units are instrumented to track coverage to begin with. It also needs visibility over the main subprogram(s) as they require special processing to trigger the calls dumping coverage data when a program terminates.

The use of such GPR based mechanisms is also allowed at **gnatcov coverage** time to further refine the focus of reports or coverage checkpoints. Another mechanism is available at this point, with lower level command line switches letting users provide lists of files holding SCO definitions for the units of interest, where one such file is produced per unit by the **gnatcov instrument** command. When both obligation files and project file options are on the command line, the former prevail and the project files are only used for switches or the determination of the target and runtime configuration.

Regardless of how units of interest were requested, the actual list of units for which a report is produced can be displayed with the `--dump-units-to` option of the **gnatcov coverage** command. This also displays the list of individually excluded source files for each unit, controlled by the `--excluded-source-files` switch.

2.1 Using project files (`-P`, `--projects`, `--units`)

The simplest possible form of units of interest specification with GPR facilities is a lone:

```
-P<myproject>.gpr
```

provided to both **gnatcov instrument** and **gnatcov coverage**. In the absence of coverage related attributes within the project file(s), this requests considering *of interest* all the units of *<myproject>* and its project dependency closure.

For **gnatcov instrument**, the source files containing main subprograms need to be encompassed by `myproject.gpr` and specified either by a `Main` project file attribute or provided on the command line, as for `gprbuild` commands.

Finer grain control is possible with additional switches and attributes, letting users first specify the set of *projects of interest* where the units of interest reside, then may filter the *units* of interest therein.

Please note that **non-library aggregate project files are currently not supported** and will be rejected by **gnatcov**. The aggregated projects will have to be processed individually and coverage reports consolidated *through checkpoints*, when this makes sense.

2.1.1 Conveying *projects* of interest

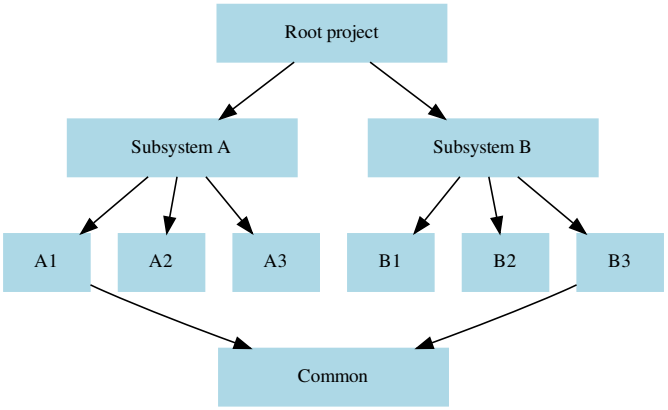
The set of projects of interest is computed by the following rules:

- A set of *base* projects is first selected from the recursive dependency closure of a root project;
- A set of *candidate* projects of interest is established, as the union of the dependency closures of all the base projects by default, or as the mere set of base projects alone if the `--no-subprojects` switch is used;
- The actual projects of interest are the candidate ones minus those with an `Externally_Built` attribute set to `"True"`.

For the determination of the base projects set, a single *root project* must first be specified using the `-P` option. The set may then be refined according to the following rules with an optional list of `--projects` switches naming projects within the dependency closure of the root:

- Without `--projects`, the base projects set is the root project designated by `-P` alone, or the project designated by the `Origin_Project` attribute therein if there is such an attribute;
- With `--projects` options, the listed projects are taken as the base and the root project needs to be listed as well to be included.

Let us illustrate the effect of various combinations, assuming an example project tree depicted below:



Assuming none of the projects is flagged `Externally_Built`:

- *-Proot --no-subprojects* restricts the analysis to units in the root project only (Fig. 1);
- *-Proot --projects=subsystem_a --no-subprojects* focuses on Subsystem A alone (Fig. 2);
- If the root project is also of interest, it must be listed explicitly, as in *-Proot --projects=root --projects=ss_a --no-subprojects* (Fig. 3);
- Removing `--no-subprojects` as in *-Proot --projects=subsystem_a*, lets you consider all the projects transitively imported by the base ones (Fig. 4);

Projects with an `Externally_Built` attribute set to `"True"` are just removed from the set of interest at the end, without influencing the processing of dependency closures. In the last example above, if project `A1` had the attribute set to `"True"`, `Common` would remain of interest to the assessment even though it was dragged as a dependency of `A1`.

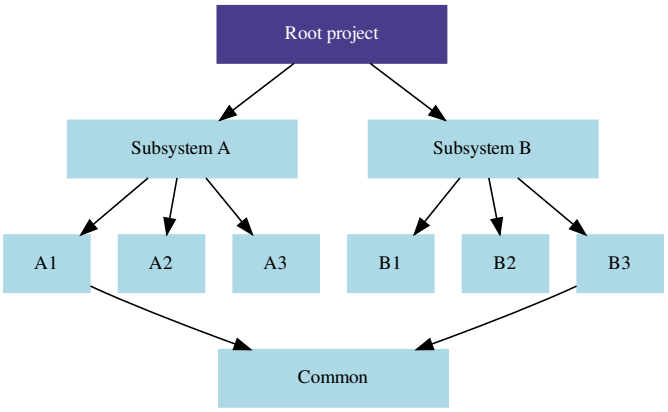


Fig. 1: `-Proot --no-subprojects`

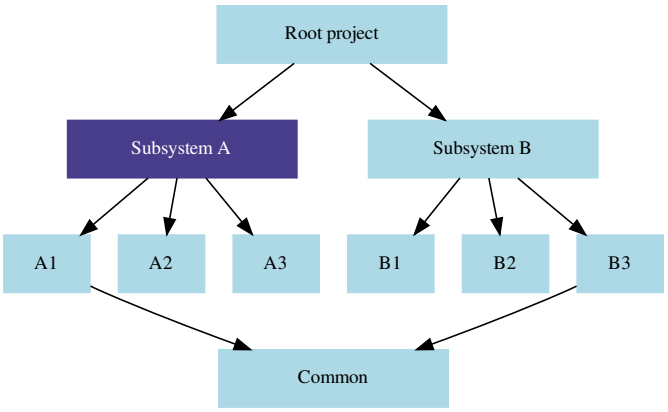


Fig. 2: `-Proot --projects=subsystem_a --no-subprojects`

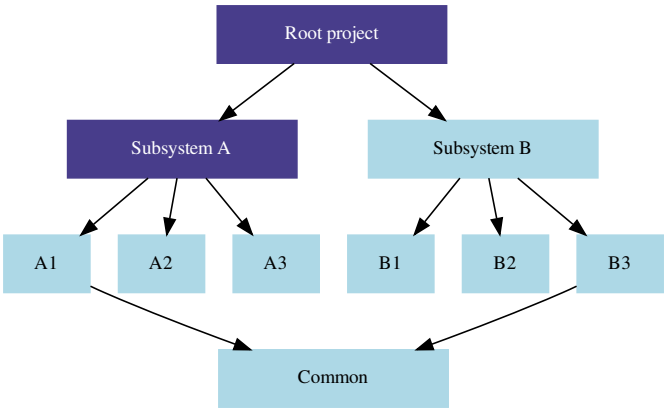


Fig. 3: `-Proot --projects=root --projects=ss_a --no-subprojects`

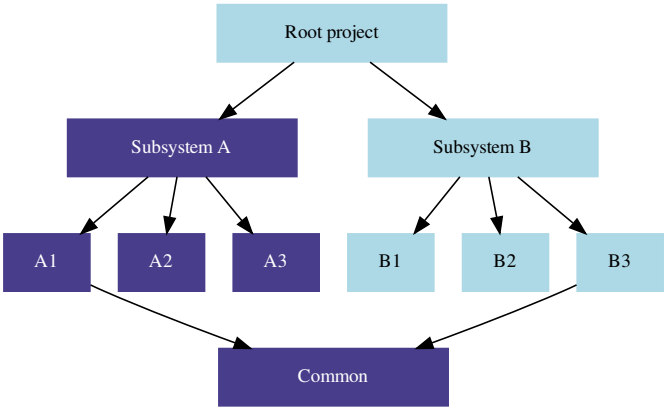


Fig. 4: `-Proot --projects=subsystem_a`

2.1.2 Conveying *units* of interest within projects

By default, all the units encompassed by a project of interest are considered of interest. This can be tailored first with specific attributes in package `Coverage` of project files.

Four attributes are available to control the set of units to be considered of interest within a project: `Units`, `Units_List`, `Excluded_Units`, and `Excluded_Units_List`.

`Units` and `Units_List` are used to construct an initial set of units for which coverage analysis should be performed. For example, given a project with three packages `Pak1`, `Pak2`, and `Pak3`, if you want to do coverage analysis only for `Pak1` and `Pak3` you can specify:

```
package Coverage is
  for Units use ("pak1", "pak3"); -- pak1 and pak3 are of interest
end Coverage;
```

Similarily to `Sources` and `Sources_List`, the `Units` attribute specifies a set of units and `Units_List` specifies the name of a text file containing a list of units. See the *Compilation unit vs source file names* section for details how individual units should be denoted depending on the source language.

`Excluded_Units` and `Excluded_Units_List` operate like `Units` and `Units_List` but for units that should never be considered of interest for coverage. Back to our example, the same result as above is obtained by specifying:

```
package Coverage is
   for Excluded_Units use ("pak2");  -- all units except pak2 are of interest
end Coverage;
```

When the exclude/include sets overlap, the excluding attributes prevail over the including ones. The exact rules for computation of the units to be considered of interest within a project are as follows:

- An initial set is determined using the `Units` and `Units_List` attributes in the project's `Coverage` package; By default, if no such attribute is found, the initial set comprises all the units of the project,
- Units determined using the `Excluded_Units` and `Excluded_Units_List` attributes are removed from the initial set to yield the set to consider.

Finally, the list of units of interest for a given execution of **`gnatcov`** can also be overridden from the command line using the `--units` switch. When this option is used, the project files attributes are ignored.

The switch may appear multiple times. Each occurrence indicates one unit to focus on, or with the @ syntax the name of a text file containing a list of units to focus on, one per line. The effect of multiple switches accumulate.

The effect of the example attributes provided previously could then first be achieved with:

```
gnatcov <command> -P... --units=pak1 --units=pak3
```

or by creating a `units.list` file with:

```
pak1
pak3
```

and then executing:

```
gnatcov <command> --units=@units.list
```

2.1.3 Conveying *subprograms* of interest (experimental)

gnatcov enables even finer grain control through the use of `--subprograms` switch, which restricts coverage analysis to the specified list of subprograms of interest.

The `--subprograms` switch expects a `<filename>:<line>` argument, where `<filename>` is a source file name path, absolute or relative to the current directory, pointing to the subprogram source and `line` is the first line of the subprogram specification in Ada, and the first line of the function definition in C/C++.

Every coverage obligation from non subprograms of interest is discarded and reported as *no code* in the various output formats, and the associated coverage data is ignored. This means that checkpoints will hold coverage data for subprograms of interest only.

The `--subprograms` switch acts only on subprograms within units of interest. If a unit was ignored for coverage analysis through a project attribute (e.g. `Excluded_Units`) or through a command line switch (e.g. `--units`), the designated subprogram will be ignored for coverage analysis as well.

2.1.4 Other switches or attributes of importance

Independently from coverage considerations, project files offer a significant range of possibilities to compose a software system. To get an accurate view of the set of projects and units from which those of interest to coverage should be taken, GNATcoverage needs to operate on the same base information as the builder.

To this effect, all the switches you would pass to gprbuild to control the build configuration should also be passed to GNATcoverage commands when conveying units of interest through project facilities, in particular `--target`, `--RTS`, `--config` switches very common in cross configurations, as well as the `-X` series setting scenarii variables.

In some cases, such as `--target` or `--RTS`, the effect of the command line switch can be achieved with a project file attribute, which GNATcoverage knows how to interpret as well.

2.2 Conveying source files to ignore / handling Ada subunits

Two attributes in the `Coverage` package make it possible to specify specific source file names for which the tool should not generate a report eventually, even if the these sources are within units of interest. This is intended for situations where the source files for a unit of interest encompass some of its testing sources, for example when parts of the testing code is implemented with `separate` subunits in Ada.

The sources for the following example can be found under the *share/examples/gnatcoverage/doc/subunits/* directory of the GNAT DAS distribution.

The dummy example below shows a possible organization of this kind, with a `Data_Processing` package to be tested which contains a `Test` procedure declared as a `separate` entity:

```
-- spec and body of a package to test

package Data_Processing is
   procedure Process (X : Integer);

   procedure Test;
end;

package body Data_Processing is

   Internal_Data : Integer := 0;

   procedure Process (X : Integer) is
   begin
      ...
   end;

   procedure Test is separate; -- subunit declaration here
end;
```

We can have different implementations of the `Test` subprogram body in different source files and a project file based mechanism to select one or the other based on a scenario variable:

```
-- data_processing-test1.adb; test variation #1

separate (Data_Processing)
procedure Test is
begin
   Process (X => 12);
   pragma Assert (Internal_Data > 0);
end;

-- data_processing-test2.adb; test variation #2

separate (Data_Processing)
procedure Test is
begin
   Process (X => -8);
   pragma Assert (Internal_Data < 0);
end;
```

(continues on next page)

(continued from previous page)

```
-- run_all.adb; main file to run the test

with Data_Processing;
procedure Run_All is
begin
   Data_Processing.Test;
end;

-- Project file with a Body source file name selection in a
-- Naming project package:

project P is

  for Object_Dir use "obj";
  for Source_Dirs use ("src");
  for Main use ("run_all.adb");

  TEST := external ("TEST");
  package Naming is
    for Body ("data_processing.test") use "data_processing-" & TEST & ".adb";
  end Naming;
end P;
```

Then we can build one variant or the other with:

```
$ gprbuild -Pp.gpr -XTEST=test1
$ gprbuild -Pp.gpr -XTEST=test2
...
```

As any testing code, such subunits usually need to be excluded from the coverage analysis scope. However, even though implemented in separate source files, subunits are technically not units on their own, so could not be excluded alone by the unit-based mechanisms presented in previous sections.

The two attributes introduced here allow the specification of file names to be ignored as a list of globbing patterns akin to those allowed in Unix shells. All source files whose *base* name matches any of the patterns are excluded from the analysis and from the output report. Since only base names are matched, the provided patterns to ignore should not include any path or directory component.

The first attribute, `Excluded_Source_Files`, expects a direct list of patterns. Even though intended for subunits, the attribute allows file names corresponding to regular units as well. For our dummy example, this could be:

```
package Coverage is
  for Excluded_Source_Files use ("*-test*.adb", "run_all.adb");
end Coverage;
```

The second one, `Excluded_Source_Files_List`, expects the name of a text file which contains the list of globbing patterns to ignore, one line per pattern.

To achieve the same effect as with the first attribute for our example, we could create a text file named `ignore.list` which would contain:

```
*-test.adb
run_all.adb
```

And then have:

```
package Coverage is
  for Excluded_Source_Files_List use "ignore.list";
end Coverage;
```

As a possible alternative to the project file attributes, the **gnatcov coverage** and **gnatcov instrument** commands accept a `--excluded-source-files` switch on the command line.

This option can appear multiple times on the command line. Each occurrence expects a single argument which is either a globbing pattern for the name of source files to ignore (as for a `Excluded_Source_Files` attribute), or a *@listfile argument* that contains a list of such patterns (as for a `Excluded_Source_Files_List` attribute), and the effects of all the options accumulate.

The example attributes provided previously would become:

```
gnatcov <command> --excluded-source-files=*-test.adb --excluded-source-files=run_all.adb
```

or:

```
gnatcov <command> --excluded-source-files=@ignore.list
```

When `--excluded-source-files` is provided on the command line, all the `Excluded_Source_Files` and `Excluded_Source_Files_List` attributes are ignored.

2.3 Compilation unit vs source file names

For Ada, explicit *compilation unit* names are given to library level packages or subprograms, case insensitive. This is what must be used in project file attributes or `--units` arguments to elaborate the set of *units of interest*, not source file names.

This offers a simple and consistent naming basis to users, orthogonal to the unit/source name mapping. Consider, for example, a project file with the set of declarations below, which parametrizes the source file name to use for the body of a `Logger` package depending on the kind of build performed:

```
type Build_Mode_Type is ("Production", "Debug");
Build_Mode : Build_Mode_Type := external ("BUILD_MODE", "Debug");

package Naming is
   case Build_Mode is
      when "Production" =>
         for Implementation ("Logger") use "production-logger.adb";
      when "Debug" =>
         for Implementation ("Logger") use "debug-logger.adb";
   end case;
end Naming;
```

Regardless of the build mode, restricting the analysis to the `Logger` package would be achieved with `-P<project> --units=logger` or with a `Units` attribute such as:

```
package Coverage is
   for Units use ("Logger"); -- compilation unit name here
end Coverage;
```

Source file names are used in the output reports, still, either in source location references as part of the `=report` outputs, or as the base filename of annotated source files for other formats. For our `Logger` case above, the analysis with, for

example, `--annotate=xcov` of a program built in Debug mode would yield a `debug-logger.adb.xcov` annotated source result.

For C, the notion of *translation unit* resolves to the set of tokens that the compiler gets to work on, after the pre-processing expansion of macros, #include directives and the like. This doesn't have an explicit name and *units of interest* must be designated by the toplevel source file names from which object files are produced.

Typically, from a sample `foo.c` source like:

```
#include "foo.h"

static int bar (void)
{ ... }

...
void foo (int x)
{ ... }
```

excluding `foo.c` from the analysis scope can be achieved with:

```
package Coverage is
   for Excluded_Units use ("foo.c"); /* source file name here  */
end Coverage;
```

2.4 Providing coverage obligation files (`--sid`)

With the `--sid` command line option, users can convey the set of units of interest by directly providing the set of files which contain the coverage obligations for those units.

One such file is produced for each unit instrumented by the **`gnatcov instrument`** command, next to the object file for a unit, with a *.sid* extension which stands for *Source Instrumentation Data*.

Each occurrence of `--sid` on the command line expects a single argument which specifies a set of units of interest. Multiple occurrences are allowed and the sets accumulate. The argument might be either the name of a single *.sid* file for a unit, or a *@listfile arguments* expected to contain a list of such file names.

For example, focusing on Ada units `u1`, `u2` and `u3` can be achieved with either `--sid=u1.sid --sid=u2.sid --sid=u3.sid`, with `--sid=u3.sid --sid=@lst12` where `lst12` is a text file containing the first two SID file names, or with other combinations alike.

CHAPTER

THREE

PRODUCING SOURCE TRACES WITH `GNATCOV INSTRUMENT`

The production of *source traces* is performed by an instrumented version of the original program running in its regular execution environment. A **`gnatcov instrument`** command produces alternative sources for units of interest, with additional data structures and code statements solely aimed at tracking coverage related information. Coverage data is output in accordance with a user selectable policy, for example upon exit of the program, for which the program main unit gets instrumented as well.

To gather coverage data while the program is running and to output this data in the form of a source trace file eventually, the instrumented code relies on common data types and subprograms provided by a *coverage runtime* library. The first thing to do for a given project is then to build and install the coverage runtime so it becomes available to the instrumented sources afterwards, as we will describe in the *Setting up the coverage runtime library* section of this chapter.

Once the coverage runtime is setup, the simplest possible use of the instrumentation is on entire programs at once, with their main unit(s) included. The process essentially goes like:

1. Instrument, both units-of-interest and main units;
2. Build the instrumented code;
3. Execute the program to produce a trace.

GNATcoverage also allows partial instrumentations to be combined together, which we will describe in the *Composed instrumentation* part.

The instrumenter is generally careful not to introduce constructs specific to a version of Ada more recent than the original one, only assuming that this is at least Ada 95. In order to support Pure or Preelaborable units, we currently assume that compilations are performed with a GNAT Pro toolchain. Parts of the effects reach into the coverage runtime, so this also concerns application programs which do not have so categorized units.

GNATcoverage supports two direct ways of outputting coverage data from the program:

- Produce a trace file directly, or
- Output a base64 encoded version of the data on standard output, using `Ada.Text_IO`. Coverage data is emitted between markers and a trace file can be produced offline from a captured output thanks to the **`gnatcov extract-base64-trace`** command.

The **`gnatcov instrument`** command allows instrumenting main units according to a specified mode, with a choice of possible insertion points. It also features a *manual* mode so users can implement a custom strategy, which only requires that the output format abides by one of the two variants we support.

Thread safety on accesses to coverage data structures is achieved by writing at most one status Boolean at a time in data structures voluntarily *not* packed.

3.1 Setting up the coverage runtime library

GNATcoverage is shipped with the sources of the coverage runtime and with a companion project file. The **gnatcov setup** command helps automating the build and installation of this project.

Just like **gprbuild**, **gnatcov setup** accept the `--RTS`, `--config` and `--target` command line options: you need to build the coverage runtime with the same toolchain and runtime as the ones used to build the application code. It also supports a `--gargs` option to pass arguments to the **gprbuild** invocation. For instance:

```
# For a native project
gnatcov setup

# For a project that targets ARM ELF with the light-stm32f4 runtime
gnatcov setup --target=arm-eabi --RTS=light-stm32f4
```

By default, **gnatcov setup** installs the coverage runtime in the same prefix as the selected toolchain (just like **gprinstall**). In order to install it in another location, pass the `--prefix` option:

```
gnatcov setup --prefix=/usr/custom
```

When installing the coverage runtime to a non-default location, tools that process project files must be instructed how to find the coverage runtime project file that is installed. To achieve this, update the `GPR_PROJECT_PATH` environment variable:

```
export GPR_PROJECT_PATH=$GPR_PROJECT_PATH:/usr/custom/share/gpr
```

In addition, when using shared libraries, the environment needs to be updated so that instrumented programs can load the coverage runtime's own shared libraries. The following command achieves this in a Unix like environment:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/gnatcov-rts/lib
```

The following command performs the corresponding action on Windows systems:

```
set PATH=%PATH%;C:\path\to\gnatcov-rts\bin\
```

Note: When the target supports different kinds of libraries (`static`, `static-pic` or `relocatable`), **gnatcov setup** will build and install all variants. Use the external variables `GNATCOV_RTS_LIBRARY_TYPE` or `LIBRARY_TYPE` to indicate the appropriate variant to link against. The former only influences the build type of `gnatcov_rts` while the later may influence other libraries as well. The default value for the library type is `static`.

3.1.1 Multiple runtimes in the same prefix

It is sometimes convenient to install multiple times the coverage runtime in the same prefix, for instance when working on a project that runs both on the native platform and on an embedded target while the two toolchains are also installed in the same prefix. It is possible to make **gnatcov setup** install the coverage runtime under different project names: the default is `gnatcov_rts`, and using the `--install-name` option changes it.

```
# Install the coverage runtime both for native projects (gnatcov_rts_native)
# and for ARM ELF/light-stm32f4 projects (gnatcov_rts_stm32f4).
gnatcov setup --install-name=gnatcov_rts_native
gnatcov setup --target=arm-eabi --RTS=light-stm32f4 \
  --install-name=gnatcov_rts_stm32f4
```

When building instrumented projects later with **gprbuild**, you then have to select the appropriate coverage runtime with the `--implicit-with` switch:

```
# Build the instrumented project with the native toolchain and the
# corresponding coverage runtime.
gprbuild -Pmy_project \
  --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts_native

# Now do the same, but for ARM ELF/light-stm32f4
gprbuild -Pmy_project --target=arm-eabi --RTS=light-stm32f4 \
  --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts_stm32f4
```

3.1.2 Language restrictions

By default, **gnatcov setup** builds and installs the coverage runtime with Ada and C units. When using GNATcoverage with a C-only toolchain, it is necessary to build the coverage runtime without its Ada units. The `--restricted-to-languages` option allows that:

```
# Build and install the coverage runtime with C units only
gnatcov setup --restricted-to-languages C

# Build and install the coverage runtime with both Ada and C units (the
# default).
gnatcov setup --restricted-to-languages Ada,C
```

Note that for now, only Ada and C are supported. The core runtime is implemented in C, so the C language must be enabled in all cases.

3.1.3 Default source trace dump options

Source trace dump settings (`--dump-trigger`, `--dump-channel` and other related `--dump-*` command line options) are used to make **gnatcov instrument** produce instrumented mains that automatically dump source trace files at the end of the program execution, or to leave this to manually written code (see the full documentation in *later sections*).

Without any `--dump-*` option, **gnatcov setup** selects defaults for the target platform, which **gnatcov instrument** can then reuse automatically. There are two ways to override these defaults: pass `--dump-*` options directly to **gnatcov instrument**, or to **gnatcov setup**. If `--dump-*` options are passed to both **gnatcov setup** then to **gnatcov instrument**, the options passed to the latter take precedence. For instance:

```
# Use what gnatcov thinks the most sensible defaults for the target platform
# are.
gnatcov setup
gnatcov instrument # ...

# Setup to use atexit/bin-file by default, then use these defaults for
# instrumentation.
gnatcov setup --dump-trigger=atexit --dump-channel=bin-file
gnatcov instrument # ...

# Make atexit/bin-file the default at setup time, but actually use
# main-end/base64-stdout for source instrumentation.
gnatcov setup --dump-trigger=atexit \
```

(continues on next page)

(continued from previous page)

```
              --dump-channel=bin-file
gnatcov instrument --dump-trigger=main-end \
                   --dump-channel=base64-stdout # ...

# Make atexit/bin-file the default at setup time, but actually use
# main-end for source instrumentation (thus still using the default
# bin-file).
gnatcov setup --dump-trigger=atexit \
              --dump-channel=bin-file
gnatcov instrument --dump-trigger=main-end # ...
```

Note that the defaults that **gnatcov setup** uses for each target platform may change between versions of GNATcoverage.

3.1.4 Coverage runtime setup for configurations with no Ada runtime

If the application is not linked against the Ada runtime library (but contains Ada code), the coverage runtime needs to be setup accordingly, so as not to rely on any features from the Ada runtime.

This can be achieved with the `--no-stdlib` **gnatcov setup** command line switch. The resulting coverage runtime only supports a `manual` or `main-end` dump trigger, and will use a special implementation of the `base64-stdout` dump channel, relying on a user-provided function to output the coverage data.

The output function expected by the coverage runtime should have the same signature as the stdlib's `putchar` function:

```
extern int gnatcov_rts_putchar(int __c);
```

Note that the coverage runtime will ignore the return value.

In all cases the coverage runtime still requires an implementation of `memset` and `memcpy` to be provided, either from the C standard library, or provided by the user.

Note: This method of providing a function name upon which the coverage library depends, without adding this information in the project file means that gprbuild will produce a link command with the assumption that the coverage runtime does not depend on the instrumented project. This may result in the executable link failing due to an undefined reference to the output function symbol name, if the object file is not already pulled in the executable in a non-instrumented build.

To work around this, either ensure the output function is defined in a compilation unit that is part of the executable in a non-coverage build, or use an extending project to add your own sources to provide the `gnatcov_rts_putchar` symbol, as detailed in the *section dedicated to coverage runtime customization*.

Note: While the coverage runtime does not explicitly depend on any object or function from the Ada runtime, runtime checks added by the compiler will produce a reference to the `__gnat_last_chance_handler` symbol on configurations that do not support exception propagation. This in turn means that for configurations that provide a default implementation of the last chance handler, such as light runtimes, the runtime checks will implicitly depend on some Ada runtime symbols.

If this dependency is a problem, it can be avoided by either providing a user defined last chance handler, as detailed in the GNAT's users guide supplement for cross platforms, or by disabling runtime checks during the coverage runtime setup:

```
gnatcov setup --no-stdlib -gargs -gnatp
```

3.2 Instrumenting programs

Instrumentation is performed upfront for an intended strictest coverage criterion on a given set of *units of interest*. The production of a coverage report afterwards might restrict the report to a subset of those units, or lower to a less strict criterion. Instrumenting programs is achieved with **gnatcov instrument** commands, which might involve two distinct kinds of operations:

- Modify the code in units-of-interest so the program records, while it is running, facts of relevance to the coverage metrics to be assessed,
- Modify the main unit(s) to output the so gathered coverage data to an externally readable channel, typically either a source trace file or some communication port.

3.2.1 `gnatcov instrument` command line

As for other commands, help on the command line interface is displayed by `gnatcov instrument --help`. The general synopsis is as follows:

```
gnatcov instrument --level=<> <units-of-interest> [OPTIONS]
```

`--level` states the strictest criterion that this instrumentation will allow assessing afterwards and the `<units-of-interest>` switches specify the set of units for which such assessment will be possible. The latter may only resort to project file facilities, as described in the *Using project files (-P, --projects, --units)* section of this manual. Projects marked `Externally_Built` in the closure are not instrumented or otherwise modified.

A few `[OPTIONS]` allow controlling the instrumentation of main units, if any are designated by the root project:

`--dump-trigger`
: selects the execution point at which the output of coverage data should be injected in main units. The default is inferred from the installed instrumentation runtime. The possible choices are `manual`, `atexit`, `main-end` and `ravenscar-task-termination`.

`--dump-channel`
: selects the mechanism used to output coverage data at the selected triggering point, if any. The possible choices are `bin-file`, to create a source trace file, or `base64-stdout` to emit a base64 encoded version of the data through `GNAT.IO`. `bin-file` is the default.

`--externally-built-projects`
: instructs the instrumenter to look into projects marked as externally built when computing the list of units of interest (they are ignored by default), for the sole purpose of instrumenting mains.

In addition, for trace files produced automatically from a `bin-file` dump-channel, the `--dump-filename-<>` family of switches provides control over the name of trace files. See *Controlling trace file names* for more details on the default behavior and possibilities to alter it.

The instrumentation process can be parallelized using the `-j` (shorthand for `--jobs`) switch. `-j0` can be used for maximal parallelism, and `-jN` with N > 0 to specify the desired level of concurrency.

gnatcov instrument may emit warnings in situations where an anomaly occurs, when that anomaly does not technically prevents the instrumentation process: in that case, **gnatcov instrument** prints a warning message and resumes its operations. Passing `--warnings-as-errors` makes it treat warnings as errors, i.e. exit with a non-zero status code.

3.2.2 Output strategies for coverage information

The choice of a `--dump-trigger`/`--dump-channel` pair for coverage information dump depends on the runtime environment available to your program.

For a native program with access to a full Ada runtime and the associated coverage runtime, `--dump-channel=bin-file` is the recommended choice as it produces a trace in the most direct manner and separates the trace data from the regular output. `--dump-trigger=atexit` is a natural triggering choice in this case, as it takes care of outputting the data automatically at a point where we know the program is not going to execute more, regardless of how or why the program exits.

The `--dump-trigger=main-end` alternative simply inserts the calls at the end of the main subprogram bodies, which may be bypassed if the program exits abruptly, or miss data if the program has tasks not terminated when execution of the main subprogram/thread reaches its end.

For Ravenscar programs, another option is to use `--dump-trigger=ravenscar-task-termination`, which triggers the creation of a source trace whenever an Ada task terminates.

For more restricted environments where, say, there is limited file IO available to the program, a `--dump-channel=base64-stdout` kind of output is needed in association with the restricted coverage runtime.

If none of the available automatic triggering option work out well, full control is offered by the `--dump-trigger=manual` policy where the instrumenter doesn't actually add any code to main units for emitting the collected coverage data. You will have to indicate the point at which you wish to emit this data by inserting:

- a `pragma Annotate (Xcov, Dump_Buffers);` pragma statement in Ada code;
- a `/* GNATCOV_DUMP_BUFFERS */` comment on its own line in C and C++ code

where necessary in your code. During instrumentation, **gnatcov** will replace them with a call to the procedure responsible for dumping the coverage buffers, at which point the source traces will be created during the execution of the program. Therefore, the pragma or comment should be placed at a location at which such a function call would be appropriate.

By default, gnatcov will look into all the files of the project. Alternatively, the user can specify the list of files containing manual indications using `--manual-dump-files=@FILES.list` or `--manual-dump-files=FILE1,FILE2` where the files are specified as full / relative names. Note that for C / C++ files, the user shall specify the files where the `/* GNATCOV_DUMP_BUFFERS */` indication is ultimately expanded by the preprocessor. Note that the `--manual-dump-files` option requires `--dump-trigger=manual`, and that the option may be given more than once, appending to the list of files to look at.

A dump procedure is only able to dump the buffers of the project tree which root is the project it is called from. A dump procedure call done in a subproject will result in a trace containing all code of projects higher in the project tree marked as not covered.

One source trace is dumped per call to the dump buffers procedure. For the `bin-file` dump configuration, each trace is written in a file which name depends on the selected trace file naming scheme. For the `base64-stdout` dump configuration the traces are sequentially dumped in the same output file, from which the **gnatcov** command `extrace-base64-trace` will be able to produce a source trace file.

It is also possible to configure the prefix of the trace filename (see *Controlling trace file names* for more details) through these indications:

- `pragma Annotate (Xcov, Dump_Buffers, Prefix);` in Ada code, where `Prefix` resolves to type `String`;
- `/* GNATCOV_DUMP_BUFFERS (Prefix) */` in C or C++ code, where `Prefix` resolves to a null-terminated `char *` type.

No type checking will be done during instrumentation, the values passed as prefix indications are used as parameters to the buffer dump procedure call. The Ada procedure call expects a `String` parameter, and the C or C++ functions

expect a null-terminated `const char *`, so in both cases specifying a string literal is allowed. If the prefix indication does not resolve to the correct type, the instrumented code will fail to build.

This can be used in case multiple traces are dumped in rapid succession, to distinguish between each trace:

```
procedure Dump is
  Prefix : constant String := "variable";
begin
  --  Ok, Prefix is of type String
  --  Trace filename will be "variable-<hex_part>.srctrace"

  pragma Annotate (Xcov, Dump_Buffers, Prefix);

  --  Ok, String literal
  --  Trace filename will be "literal-<hex_part>.srctrace"

  pragma Annotate (Xcov, Dump_Buffers, "literal");

  --  Not ok, Undefined does not resolve to a variable of type String
  --  This will fail to compile.

  pragma Annotate (Xcov, Dump_Buffers, Undefined);
end Dump;
```

```
void dump(void){

  /* Ok, prefix if type char * and null-terminated.
     Trace filename will be "variable-<hex_part>.srctrace"  */
  char *prefix = "variable";
  /* GNATCOV_DUMP_BUFFERS (prefix) */

  /* Ok, string literal.
     Trace filename will be "literal-<hex_part>.srctrace"  */
  /* GNATCOV_DUMP_BUFFERS ("literal") */

  /* Not ok, Undefined does not resolve to a null terminated string.
     This will not compile.  */
  /* GNATCOV_DUMP_BUFFERS (Undefined) */
}
```

3.2.3 Resetting coverage information during the execution

gnatcov instrument offers the possibility to indicate, in the program sources, one or multiple locations at which the buffers used to track the execution of the program should be cleared. This can be used in various situations where a finer grain control of the traces generated by the instrumented program is required. For instance, this can be used to ignore some initialization phase in the coverage report, or to emit multiple coverage traces from a single execution of the instrumented program, each trace containing the coverage information for a specific part of the execution, such as in a unit test driver.

This fine grain control of the coverage buffers mandates the use of a manual dump of the coverage buffers, thus requiring passing the `--dump-trigger=manual` command line option to **gnatcov instrument** invocations, as described in *Output strategies for coverage information*.

The buffer reset location indications follow a similar convention as the manual dump of the traces. Indicating the point

at which coverage buffers must be cleared is done by inserting:

- a `pragma Annotate (Xcov, Reset_Buffers);` pragma statement in Ada code;
- a `/* GNATCOV_RESET_BUFFERS */` comment on its own line in C and C++ code

where necessary. During instrumentation, **gnatcov** will replace them with a call to the procedure responsible for clearing the coverage buffers. Therefore, the pragmas or comments should be placed at locations at which such a procedure call would be legal and appropriate.

For performance reasons, no synchronization mechanisms are used to protect the buffer clear procedure call against concurrent writes in the coverage buffers (by regular program execution). **It is up to the user to ensure that no other section of the instrumented code runs concurrently to the buffer clear procedure, otherwise the correctness of the corresponding coverage report cannot be guaranteed.**

Both buffer dump indications and buffer reset indications can be specified in external annotation files, see *External source annotations* for more information.

3.2.4 Controlling trace file names

When an instrumented program produces a trace file through a `bin-file` dump-channel, the file is by default created in the current working directory at the data output point (for example, at exit time for an `atexit` dump-trigger), and named as `<ename>-<istamp>-<pid>-<estamp>.srctrace`, where:

- `<ename>` is the executable name,
- `<istamp>` is the instrumentation time stamp, representing the time at which the instrumentation took place,
- `<pid>` is the execution process identifier,
- `<estamp>` is an execution time stamp, representing the time at which coverage data was written out to the file.

The `<estamp>` and `<pid>` components are intended to ensure that parallel executions of the program from the same working directory write out to different files. The `<istamp>` component allows distinguishing traces issued from different versions of the program. These three components are expressed as hexadecimal integers to limit the growth of file name lengths.

This default behavior can be influenced in several manners. First:

- The `--dump-filename-prefix` switch to **gnatcov instrument** requests replacing the `<ename>` component by the switch argument;
- The `--dump-filename-simple` switch requests the removal of the variable components (stamps and pid), so only the `<ename>` component remains or the replacement provided by `--dump-filename-prefix` if that switch is also used.

The use of a specific location for the file, or of a specific file name can be requested at run time by setting the `GNATCOV_TRACE_FILE` variable in the program's environment.

If the variable value ends with a `/` or `\` character, this value is interpreted as the name of a directory where the trace file is to be produced, following the rules we have just described for the file base name. The directory reference may be a full or a relative path, resolved at the trace file creation point and expected to exist at that time.

If the variable value does *not* end with a `/` or `\` character, the value is used directly as the name of the file to create. This name may hold a path specification, full or relative, also resolved at the trace file creation point and the directories involved are expected to exist at that time.

For specific needs of programs wishing to output to different places from within the same environment, the variable name for a program can actually be tailored by passing a `--dump-filename-env-var` switch to **gnatcov instrument**, providing the variable name to use.

3.2.5 Support for preprocessing

gnatcov instrument automatically detects preprocessor configuration from the compiler switches present in project files (`-gnatep` and `-gnateD` for Ada sources, `-D` and the like for C/C++ sources). It then runs preprocessing on the source code *before* the instrumentation itself happens. This allows gnatcov to compute the code coverage only for code that is left "enabled" by preprocessing directives: disabled code (for instance what follows `#if Debug then` in Ada when the preprocessing symbol `Debug` is set to `False`) is ignored and thus creates no coverage obligation. Note that consolidation will not help including code from all "preprocessing branches" in coverage reports, as gnatcov requires (and checks) that coverage obligations are the same for two units to consolidate.

Ada pecularities

The coverage obligations for code that comes from symbol expansion (for example, `$Foo = 42` expanded into `My_Variable = 42` with `-Dfoo=My_Variable`) designate expanded code. Even though line numbers are preserved during preprocessing, column numbers may be different between the original code and the preprocessed code and thus the coverage report.

C/C++ pecularities

As a wide variety of compiler switches impacts the preprocessing configuration, gnatcov supports passing compiler switches to the compiler preprocessor invocation through the `-c-opts` and `-c++-opts`. Generally speaking, any switch that could impact the preprocessing of the source (e.g. macro definitions / undefinitions, explicit include paths ...) must be added to the **gnatcov instrument** command line.

Coverage obligations resulting from macro expansions are designated as such in detailed (e.g. HTML and XML) coverage reports and their expansions chain is reported. Note that contrarily to Ada, line and column numbers are preserved in the coverage report.

3.2.6 gnatcov instrument limitations

This section details limitations of the source based instrumentation tool. These are split up by language and/or coverage level.

Ada limitations

There are situations and code patterns not handled correctly by **gnatcov instrument**. Below are listed the limitations associated with general Ada sources. Coverage of SPARK sources require additional considerations, detailed in section *Instrumentation and coverage of SPARK code*.

gnatcov instrument limitations are inherently bound to the version of the Ada language used. The version is indicated to the tool via the `--ada` command line switch, which is set to `2012` by default. Additionally, **gnatcov instrument** can also retrieve the langage version for a specific source file which contains a `pragma Ada_nnnn;` directive. The pragma prevails on the command line argument.

Unsupported source constructs

There are a few language constructs that **gnatcov instrument** doesn't support. The tool emits a warning when it encounters such cases and the corresponding code is not instrumented. Source coverage obligations are still emitted, and the unsupported constructs will be reported in a separate `Undetermined_Coverage` category, to differentiate them from actual coverage violations.

The list of unsupported constructs is as follows:

- Generic null procedures,
- Protected bodies entry guards when the `Simple_Barriers` restriction or the `Pure_Barriers` one apply.
- Decisions in default expressions of subprogram formals (for all decision-related coverage criteria).

Additionally, if the Ada language version in use, is less or equal to Ada 2012, the following constructs are also unsupported:

- Generic expression functions,
- Recursive expression functions which are primitives of some tagged type,
- Expression functions which are primitives of their return type, when it is a tagged type.
- Decisions in default expressions of record component declarations (for MC/DC coverage criteria).

The simplest way to work around the limitation concerning expression functions is to turn them into regular functions, by giving them a proper body, containing a single return statement with the original expression. Otherwise it is possible to exempt those constructs (see *Coverage Exemptions*) and/or perform a manual coverage analysis for these special cases.

Other Ada source-traces limitations

In Ada, variable or type declarations at the package level can yield elaboration code. Such code constructs are thus considered to have corresponding coverage obligations.

In the case where a `pragma Preelaborate` restriction affects the instrumented unit, variable and type declarations at the package level are not considered as coverage obligations, although some elaboration code may still be emitted in rare instances. Note that declarations within a unit constrained by a `No_Elaboration_Code` pragma don't produce coverage obligation either, which is always correct as no executable code can be emitted by the compiler for them.

There are also a few limitations concerning the source trace workflow as a whole:

- Separate analysis of generic package instances is not supported.

Additionally, the detected Ada language version can also alter the way some constructs are instrumented, to use new language features and make the instrumentation more robust.

This limitation happens on the decision coverage level, which makes use of if-expressions to instrument `elsif` decisions when the language version is 2012 or above.

Ada toolchain-specific limitations

With GNAT versions from 7.1 to 7.3, compiling with optimization will result in coverage violations on all statement obligations associated with expression functions. Explicitly disabling optimization (with `-O0` for instance) will resolve this issue.

C/C++ limitations

The instrumentation process relies on function calls that have side effects. These functions are thus not `constexpr` and cannot be inserted in any such context. As such, **`gnatcov instrument`** will not instrument functions or constructors declared as `constexpr`, and they will not generate any coverage obligations.

The instrumentation process yields preprocessed versions of the sources. Thus, it is required to remove any `-include` switch that is passed to the compiler invocation, by having a dedicated scenario variable for a coverage build for instance.

To support annotations such as *exemptions* or *buffer output indications*, the **`gnatcov instrument`** attempts to preserve comments when preprocessing the sources. If this is not possible (e.g. due to illegal concatenation of a comment with another token), the tool will emit a warning and ignore any in-source annotation for the mentioned file, as well as any other transitively included file.

Modified condition decision coverage limitations

The MC/DC instrumentation of decisions with many conditions may require more memory than available (during instrumentation and/or at run-time) to enumerate the possible paths through the decision. To avoid this, **`gnatcov`** will not instrument such decisions for MC/DC, emitting a warning in the process, and the MC/DC coverage for each decision will be reported as `Undetermined_Coverage` state. Should the default limit not be satisfactory, it can be tuned with the option `--path-count-limit`.

Function and call coverage limitations

Function and call coverage supports all three Ada, C and C++ languages, with language-specific limitations.

For the Ada language, gnatcov is currently unable to provided coverage information for:

- Call expressions in general for pre-Ada 2012 codebases.
- Call expressions part of the prefix of a dotted name.

The coverage state of these cases will be reported as undetermined.

For the C++ language, gnatcov will currently not instrument nor provide coverage information for:

- Body-less constructor and destructor functions (example: `Foo() = default`).
- Constructor calls that use the "call-init" convention and whose type is "auto" (example: `auto a(Foo(...))`).

Assertion Coverage limitations (ATC and ATCC)

Assertion coverage is only supported for the Ada language. However there are language constructs for which gnatcov is unable to perform coverage analysis for assertions:

- Preconditions and postconditions expressed as pragmas;
- Contracts for types and functions expressed as aspects for versions of the language anterior to Ada 2022.

Guarded expression coverage limitations

Guarded expression coverage is only available for the Ada language. For implementation reasons, it is only available starting from version 2022 of the language.

Parallel instrumentation limitations

While it is possible to instrument source files in parallel for a given `gnatcov instrument` command (using the `--jobs=N`/`-jN` command line argument), running multiple `gnatcov instrument` commands in parallel is not supported when these commands would be concurrent. This limitation is very common in build systems, like `make` or `gprbuild` that must not be run in parallel when write races are possible.

For instance, `gnatcov instrument -P p1.gpr` and `gnatcov instrument -P p2.gpr` can be run in parallel if there is no common project in the closures of `p1.gpr` and `p2.gpr`. However, they should be run in sequence if closures for `p1.gpr` and `p2.gpr` have projects in common, for example:

- `p2.gpr` contains `with "p1";`, or `p1.gpr` contains `with "p2";`,
- both `p1.gpr` and `p2.gpr` contain `with "lib";`, with `lib.gpr` being a library project file.

Note that this limitation does not apply to externally built projects (i.e. projects considered by GPR tools as read-only), as precisely GPR tools will not try to write to their object/library/executable directories. See the *Composed instrumentation* section for an example of how externally built projects can be used to achieve this: the instrumentation of each test can be run in parallel because the library (a project that is in the closure of all tests) is externally built.

Using the `--subdirs` argument (accepted by both `gprbuild` and `gnatcov`) with different subdirectory names for each concurrent invocation is another possible workaround to this limitation: in that case, the common sources will be instrumented once per `gnatcov instrument` invocation. Like in `gprbuild`, passing `--src-subdirs=mysubdir` to `gnatcov instrument` will make gnatcov produce instrumentation artifacts in subdirectories of the projects object and library directories. For instance:

```
library project Mylib is
   for Library_Name use "mylib";
   for Object_Dir use "obj";
   for Library_Dir use "lib";
end Mylib;
```

```
# Instrumented source files will go to obj/mylib-gnatcov-instr/ and SID
# files will go to obj/ and lib/.
gnatcov instrument -Pmylib --level=stmt
gprbuild -Pmylib --implicit-with=gnatcov_rts --src-subdirs=gnatcov-instr

# Instrumented source files will go to obj/s1/mylib-gnatcov-instr/ and SID
# files will go to obj/s1/ and lib/s1/.
gnatcov instrument -Pmylib --level=stmt --subdirs=s1
gprbuild -Pmylib --implicit-with=gnatcov_rts --src-subdirs=gnatcov-instr \
  --subdirs=s1
```

Yet another workaround is to use the GPR out-of-tree build switches (`--relocate-build-tree`, `--root-dir`) in both `gnatcov instrument` and `gprbuild` to arrange for object, library and executable directories to be disjoint across parallel instrumentation commands. Please refer to the GPRbuild documentation for a description of how `--relocate-build-tree` and `--root-dir` switches must be used.

3.3 Optimizing the execution of instrumented programs

To monitor the execution of statement obligations, gnatcov instruments the code which results in an execution overhead.

To mitigate the performance impact, one can use the `--instrument-block switch`. Instead of instrumenting for the execution of one statement, gnatcov instrument the code for the execution of a block of statements, greatly reducing the execution overhead. The coverage report is left unchanged, only the way coverage obligation are discarded is.

It is important to note that this could result in *false negatives* (i.e. covered code reported as uncovered) when exceptions are used as a control flow mechanism, and do not necessarily signal an abnormal termination of the program. The tool will never report false positives (uncovered code reported as covered).

3.4 Building instrumented components

Compared to a regular build, the intermediate instrumentation process requires two specific actions:

- For the units which have been instrumented (as main units or declared of-interest to coverage instrumentation time), arrange to use the instrumented sources instead of the original ones; and
- Provide the instrumented code with access to the coverage runtime support.

Since release 20, our GPRbuild builder incorporates features allowing a direct reuse of a project hierarchy without replication of the directory structure, not even modification of the project files.

For each project in the closure of-interest, the instrumentation generates the alternative sources in the `PROJECT-gnatcov-instr` subdirectory of the project's object directory (replace `PROJECT` with the lower cased actual name of the project: e.g. `mylib-gnatcov-instr` for `mylib.gpr`). Giving priority to this subdir when it exists is achieved by passing a `--src-subdirs` switch to **gprbuild**, naming that particular relative subdirectory.

Then **gprbuild** now supports a `--implicit-with` option which requests processing every project as if it started with a `with` statement for a given project, which we can use to designate the coverage runtime project file so all the compiled code gets access to the support packages.

The build of instrumented components then proceeds almost exactly as a regular one, only adding `--src-subdirs=gnatcov-instr` and `--implicit-with=<gnatcov_rts_gpr>` to the build options, where `<gnatcov_rts_gpr>` would be the coverage runtime project file setup beforehand for the project, as described previously in this chapter. This project file could be referred to with a full path specification, or with a simple basename if the `GPR_PROJECT_PATH` environment variable is updated to designate the directory where the project file is located, which would be the `share/gpr` subdirectory of the runtime installation tree.

While the scheme relies on the use of GNAT project files, it does not absolutely require **gprbuild** to build the instrumented programs, even though we have augmented that builder with a few features to make that process very efficient and straightforward.

Note: The variety of supported compilers/compiler versions/optional warnings makes it an irrealistic goal for **gnatcov instrument** (and source code generators in general) to produce warning-free code. For instance, a hypothetical compiler is perfectly allowed to warn when functions have more than X statements; yet it would not be reasonable for **gnatcov instrument** to accomodate this restriction.

It is important to note that these are warnings (notes that the compiler can emit on code deemed suspicious, but which do not abort compilation), as opposed to errors, which do abort compilation.

For these reasons, the most reasonable thing to do with automatically generated code is to disable "warnings-as-error" (`-gnatwn` for GNAT, `-Wno-error` for GCC) when building instrumented code.

3.5 Extracting a trace from standard output

With the `base64-stdout` channel, coverage data is emitted with `GNAT.IO` on the program's standard output stream. The actual base64 encoded data is framed by start/end-of-coverage-data markers and GNATcoverage provides the **`gnatcov extract-base64-trace`** command to extract this data from a captured output and create a trace file offline (outside of the program's execution context). The extraction command line simply is:

```
gnatcov extract-base64-trace <captured-output> <output-trace-file>
```

The captured output may be used directly, there is no need to first extract the trace data section.

3.6 Composed instrumentation

To prevent unnecessary re-instrumentation and re-build of components which don't change, GNATcoverage allows partial instrumentations to be combined together. A common use case would be the testing of library components, where the library doesn't change and its coverage needs to be assessed incrementally as new tests get developed.

In such situations, the process would become something like:

1. Setup or reuse a separate project file for the library, which normally wouldn't have any main unit;
2. Instrument the library using this project as the root project;
3. Build the instrumented library;

Then for each new test:

1. Setup or reuse a separate project file for the test, designating the main unit if you wish to leverage the instrumenter's ability to insert the coverage coverage data output code. Setup a dependency from this project on the library project, with an `Externally_Built` attribute set to `"True"`;
2. Instrument the testing code main unit alone;
3. Build a program combining the library (instrumented for coverage measurement) and the testing code (instrumented to output the gathered coverage data);
4. Execute the program to produce a trace.

The section *Example use cases* illustrates such a use case.

3.7 Instrumentation of a multi-languages project

The GNATcoverage instrumentation mode supports Ada, C and C++.

To change the set of languages to be instrumented, pass the `--restricted-to-languages` option to **gnatcov instrument**. For instance, to instrument only Ada units:

```
gnatcov instrument --restricted-to-languages=Ada # ...
```

And to instrument Ada, C and C++ units:

```
gnatcov instrument --restricted-to-languages=Ada,C,C++ # ...
```

3.8 Instrumentation and coverage of SPARK code

The instrumentation of a SPARK project requires an additional step in order to make the compiler accept the instrumented code. Additionally, some parts of SPARK sources are not processed by the instrumenter, and thus will not have any coverage obligation attached to them.

3.8.1 Inhibiting SPARK related pragmas

SPARK introduces a certain number of pragmas and aspects to aid the developer in writing program contracts and guiding the automatic provers. These are only useful for static proof purposes, and are not used when assessing the coverage of a project during testing. As such, the instrumenter ignores those pragmas/aspects, and the compiler must be instructed to disregard them when processing instrumented sources.

To do so, the simplest option is to pass a configuration pragma file which inhibits each of the above pragmas when building the project. Such a configuration pragma file can be found at `examples/support/instrument-spark.adc` in the installation tree of **gnatcov**. Its contents are:

```
--  This is the list of global restrictions to be used when building
--  instrumented spark code.
--
--  We need to enforce such restrictions, as gnatcov instrumentation generates
--  constructs incompatible with a SPARK_Mode compilation.

pragma Ignore_Pragma (SPARK_Mode);
pragma Ignore_Pragma (Refined_State);
pragma Ignore_Pragma (Abstract_State);
pragma Ignore_Pragma (Global);
pragma Ignore_Pragma (Depends);
pragma Ignore_Pragma (Part_Of);
pragma Ignore_Pragma (Initializes);
pragma Ignore_Pragma (Refined_Global);
pragma Ignore_Pragma (Refined_Depends);
```

The source for the following example can be found under the *share/examples/gnatcoverage/spark_instrumentation/* directory of the GNAT DAS distribution.

The configuration pragma file can be passed to the compiler either by specifying it on the gprbuild command line with the `-gnatec` switch:

```
gprbuild -Pproject --src-subdirs=gnatcov-instr --implicit-with=<path-to-runtime> -
↪cargs:Ada -gnatec=instrument-spark.adc
```

or by way of a `Global_Configuration_Pragmas` project file attribute, possibly controlled by a scenario variable as in:

```
type mode is ("prod", "coverage");
BUILD_MODE : mode := external ("BUILD_MODE", "prod")

package Builder is
  case BUILD_MODE is
    when "coverage" => for Global_Configuration_Pragmas use "instrument-spark.adc";
    when "prod"     => null;
  end case;
end Compiler;
```

and then building with:

```
gprbuild -Pproject --src-subdirs=gnatcov-instr --implicit-with=<path-to-runtime> -XBUILD_
↪MODE=coverage
```

For SPARK projects for which unit testing is performed through GNATtest, see *Instrumenting test harnesses for a SPARK project* for instructions on how to pass the configuration pragma file when building the test harness.

3.8.2 Coverage obligations for SPARK code

By default, GNATcoverage does not instrument ghost code and contracts in SPARK, as they are mainly used to aid the proof of the program. Those regions will thus not have any coverage obligation attached to them in the coverage reports, unless explicitly requested by enabling the coverage of assertions, or by enabling ghost code instrumentation using the `--instrument-ghost` switch.

The concerned pieces of code are notably:

- any entity that is `Ghost`
- any contract (`Pre`/`Post`/`Contract_Cases`/`Loop_Invariant`)

Note that since no coverage obligations are emitted for such source constructs, they will not appear in the coverage reports even if assertions are enabled and the assertion policy enables the compilation of ghost code.

When using assertion coverage levels, as described in *Assertion True Coverage (ATC) analysis (experimental)* and *Assertion True Condition Coverage (ATCC) analysis (experimental)*, gnatcov shall instrument and report coverage results over contract code.

When enabling ghost code instrumentation with the `--instrument-ghost` switch, gnatcov shall instrument and report coverage results over ghost code.

3.9 Example use cases

3.9.1 Whole program instrumented at once, cross configuration, base64 output

Here we will consider examining the coverage achieved by the execution of the very basic sample program below, assuming the existence of a `Sensors` source unit providing access to some sensor values.

The sources for this example can be found under the *share/examples/gnatcoverage/doc/cross_example* directory of the GNAT DAS disctribution.

```
with Sensors; use Sensors;
with Ada.Text_IO; use Ada.Text_IO;

procedure Monitor is
   Sensor_Value : Integer;
begin
   for Sensor_Index in Sensor_Index_Range loop
      Sensor_Value := Sensors.Value (Sensor_Index);
      Put ("Sensor(" & Sensor_Index'Img & ") = " & Sensor_Value'Img & " ");
      if (Sensor_Value > 1000) then
         Put_Line ("!Alarm!");
      else
         Put_Line ("!Ok!");
      end if;
   end loop;
end;
```

We will consider a cross target environment, say PowerPC-VxWorks, using Real Time Processes hence an `rtp` Ada runtime library. We will assume we don't have a filesystem at hand, so will rely on the base64 encoded output of trace data to standard output.

Setting up the coverage runtime

As seen in the *Setting up the coverage runtime library* section, we use the `gnatcov setup` command to build and install the *coverage runtime*.

For our intended target environment, this would be something like:

```
gnatcov setup --target=powerpc-wrs-vxworks7r2 --RTS=rtp \
  --prefix=<gnatcov_rts-ppc-install-dir>

# Allow references to the coverage runtime project from other project files:
export GPR_PROJECT_PATH=<gnatcov_rts-ppc-install-dir>/share/gpr
```

Instrument and build

We setup a `monitor.gpr` project file for our program, where we

- Provide the main unit name, so it can be instrumented automatically, and...
- State the target configuration name and Ada runtime library so we won't have to pass explicit `--target` and `--RTS` on every command line involving project files afterwards.

For example:

```
project Monitor is
  for Target use "powerpc-wrs-vxworks7r2";
  for Runtime ("Ada") use "rtp";

  for Object_Dir use "obj-" & Project'Runtime("Ada");
  for Main use ("monitor.adb");
end Monitor;
```

We can now instrument with:

```
gnatcov instrument -Pmonitor.gpr --level=stmt+decision
  --dump-trigger=main-end --dump-channel=base64-stdout
```

This is VxWorks where we don't necessarily have an `atexit` service. Our program doesn't have tasks so `main-end` is a suitable alternative. The *stmt+decision* instrumentation will let us assess either *statement* coverage alone or *statement* and *decision* coverage afterwards.

Building the instrumented version of the program is then achieved with:

```
gprbuild -p -Pmonitor.gpr
  --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts.gpr
```

Execute, extract a trace and report

The steps required to execute are very environment specific. Symbolically, we do something like:

```
run-cross obj-rtp/monitor.vxe > monitor.stdout
```

In our case, we have stubbed 4 sensors and obtain an output such as:

```
Sensor( 1) =  1 !Ok!
Sensor( 2) =  5 !Ok!
Sensor( 3) =  3 !Ok!
Sensor( 4) =  7 !Ok!

== GNATcoverage source trace file ==
R05BVGNvdiBzb3VyY2UgdHJhY2UgZmlsZQAAAAAAAAAAAAAABAEAAAAAAAEAAAAHbW9
uaXRvcgAAAAACAAAACAAAAAAAAAAAAAAAAwAAAAAAAAAAAAAAAAAAAAcAAAAHAAAAAg
AAAAAAALNVLgQbmnY19sbrMoReNvzLLN1DAABtb25pdG9yAF8
== End ==
```

From which we can extract a source trace file like so:

```
gnatcov extract-base64-trace monitor.stdout mon.srctrace
```

And finally produce a report, with a **gnatcov coverage** command such as:

```
gnatcov coverage --level=stmt+decision --annotate=xcov mon.srctrace -Pmonitor.gpr
```

3.9.2 Library instrumented separately, native configuration, trace output

For this case we will consider a sample native software system with two source directories: one `code` directory with the sources of a library to test, and a `tests` directory with main programs verifying that the library services and operate as intended.

For the sake of the example, we will consider that

- The library source code is not going to change, and
- We will be adding tests and assess the achieved coverage by each new test individually or for the current set of tests at a given point in time.

The sources for this example can be found under the *share/examples/gnatcoverage/doc/library_instr/* directory of the GNAT DAS distribution.

Setting up the coverage runtime

As seen in the *Setting up the coverage runtime library* section, we use the `gnatcov setup` command to build and install the *coverage runtime*:

```
gnatcov setup --prefix=<gnatcov_rts-install-dir>

# Allow references to the coverage runtime project from other project files:
export GPR_PROJECT_PATH=<gnatcov_rts-install-dir>/share/gpr
```

Project file architecture

A possible straightforward way to handle code + tests system when all the code is available upfront is to setup a single project file designating the two source dirs and the main units within the *tests* component.

When part of the code, as the set of tests in our case, is being developed and the other is frozen, best is to isolate the frozen part as a separate project and declare it `Externally_Built` once the instrumented version has been built.

This would normally be achieved by **gprinstall** after the build, except the support for instrumentation artifacts (`--src-subdirs` option) may not be available.

One solution consists in setting up a separate library project file for the library `code` part, build the library, use the build tree in-place as the installation prefix, and switch the `Externally_Built` attribute to `"True"` before proceeding with separate steps for the tests, instrumenting main units in particular.

Using an scenario variable to influence the `Externally_Built` status, we could have something like the following project file for the library:

```
--  code.gpr
library project Code is

  for Library_Name use "code";
  for Library_Kind use "static";
  for Library_Dir use "lib-" & Project'Name;
```

(continues on next page)

(continued from previous page)

```
  for Object_Dir use "obj-" & Project'Name;

  for Source_Dirs use ("code");

  type Mode is ("build", "instrument", "use");
  LIB_MODE : Mode := external ("CODE_LIBMODE", "use");

  case LIB_MODE is
     when "build"      => for Externally_Built use "False";
     when "instrument" => for Externally_Built use "False";
     when "use"        => for Externally_Built use "True";
  end case;

end Code;
```

And for the tests, a separate project file where we can list the main units and state that none of the test units are of interest to the coverage metrics:

```
--  tests.gpr
with "code.gpr";

project Tests is
  for Source_Dirs use ("tests");
  for Object_Dir use "obj-" & Project'Name;

  for Main use ("test_inc.adb");

  package Coverage is
    for Units use ();
  end Coverage;
end Tests;
```

Instrument and build the library

We would first instrument and build the instrumented library with commands such as:

```
gnatcov instrument -Pcode.gpr -XCODE_LIBMODE=instrument --level=stmt+decision

gprbuild -f -Pcode.gpr -XCODE_LIBMODE=build -p
  --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts.gpr
```

Both commands proceed with `Externally_Built "False"`. There is no main unit attached to the library per se, so no need for `--dump-trigger` or `--dump-channel` at instrumentation time.

Then we can go on with the tests using the default `CODE_LIBMODE` value, implicitly switching the attribute to `"True"`.

Instrument, build and run the tests to produce traces

Here the only point of the instrumentation phase is to instrument the main units, in our case to dump trace files when the test programs exit:

```
gnatcov instrument -Ptests.gpr --level=stmt+decision
  --dump-trigger=atexit [--dump-method=bin-file] --externally-built-projects
```

The `--externally-built-projects` option is required to consider units from the library code project as contributing to the set of units of interest, for the purpose of instrumenting mains, that is, so the instrumentation of main considers coverage data from those units when producing the trace file.

The build of instrumented tests then proceeds as follows:

```
gprbuild -Ptests.gpr -p
  --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts.gpr
```

And a regular execution in the host environment would produce a source trace in addition to performing the original functional operations.

3.10 Coverage runtime customization

3.10.1 Basics

Some configurations have constraints that make the default coverage runtime inadequate. For instance, targeting a board that has no serial port, making the default implementation for `--dump-channel=base64-stdout` ineffective. In order to accomodate for such special needs, it is possible to extend the coverage runtime project and add/override some of its sources.

First, build and install the default coverage runtime (`gnatcov_rts`):

```
# Add --target / --RTS / --no-stdlib if needed according to the toolchain to use
gnatcov setup
```

Then create the project extension. The integration with the **`gnatcov setup`** workflow requires the project file and the sources to be in a dedicated directory:

```
mkdir my_rts
cat > my_rts/my_rts.gpr <<EOF
project My_RTS extends "gnatcov_rts" is
   for Object_Dir use "obj." & GNATcov_RTS.Library_Type;
   for Library_Dir use "lib." & GNATcov_RTS.Library_Type;
end My_RTS;
EOF
```

It is then possible to add source files to the `my_rts` directory to be included in this custom coverage runtime. Like in non-customized cases, **`gnatcov setup`** can then be used to buid and install the coverage runtime:

```
# Add --target and --RTS if needed according to the toolchain to use
# Make sure not to run this in my_rts.gpr directory
gnatcov setup my_rts/my_rts.gpr
```

While it is technically possible with this mechanism to modify all aspects of the default coverage runtime library, in practice only one use case is supported currently: changing the behavior of the `--dump-channel=base64-stdout` instrumentation option.

3.10.2 Customize the `base64-stdout` dump channel

To achieve this, the coverage runtime defines a single routine, in charge of writing bytes to "the output stream":

- When Ada support is enabled, in the `gnatcov_rts-base_io.adb` source file, using the Ada runtime package `GNAT.IO` to write to the serial port (bareboard targets) or to the standard output (native targets).
- When Ada support is disabled, in the `gnatcov_rts-base_io.c` source file, using the libc's `fwrite` function on `stdout`.

Overriding the source file for the relevant configuration is enough to customize the `base64-stdout` dump channel.

When Ada support is enabled, override the `gnatcov_rts-base_io.adb` source file so that it can still be built with the (unmodified) `gnatcov_rts-base_io.ads` source file.

```
--  gnatcov_rts-base_io.ads

with GNATcov_RTS.Strings; use GNATcov_RTS.Strings;

package GNATcov_RTS.Base_IO is
   pragma Preelaborate;
   procedure Put (S : GNATcov_RTS_String);
   pragma Export (C, Entity => Put, External_Name => "gnatcov_rts_put_string");
end GNATcov_RTS.Base_IO;

--  gnatcov_rts-strings.ads

with Interfaces.C; use Interfaces.C;

package GNATcov_RTS.Strings is
   pragma Pure;
   type GNATcov_RTS_String is record
      Str    : System.Address;
      Length : size_t;
   end record;
   pragma Convention (C_Pass_By_Copy, GNATcov_RTS_String);
end GNATcov_RTS.Strings;
```

When Ada support is disabled, override the `gnatcov_rts_c-base_io.c` source file so that it implements the interface described in the (unmodified) `gnatcov_rts_c-base_io.h` source file.

```
/* gnatcov_rts_c-base_io.h */

#include "gnatcov_rts_c_strings.h"
extern int gnatcov_rts_put_string (struct gnatcov_rts_string str);

/* gnatcov_rts_c_strings.h */

#include <stddef.h>
struct gnatcov_rts_string
{
```

(continues on next page)

(continued from previous page)

```
  const char *str;
  size_t length;
};
```

3.10.3 When the customized runtime needs elaboration

Ada units in the coverage runtime are all required to be either pure or preelaborated (`pragma Pure;` or `pragma Preelaborate;`). This requirement is essential for the instrumenter to generate compilable code for manual reset/dump of coverage buffers in instrumented Ada units that are themselves preelaborated: these units will need to have the coverage runtime in their closure, and it is illegal for a preelaborated unit to have in its closure a unit that is not either preelaborated or pure.

This requirement is however impractical in some cases: for instance, on some targets, coverage buffers may need to be dumped on a custom stream that needs to be initialized at elaboration time: running initialization code during the elaboration of the coverage runtime would be handy, but it unfortunately violates the preelaborated requirement mentionned above.

Ultimately, it is necessary for this case to work around compiler-enforced elaboration constraints using interfacing pragmas (`pragma Import`/`pragma Export`). Assuming that the IO handling code is self-contained and is already present in the closure of the project to analyze, the following example demonstrates how to handle this case. The sources can be found under the *share/examples/gnatcoverage/custom_rt/* directory of the GNAT DAS distribution.

First, move the IO handling code to an independent library project:

```
-- stateful_io.gpr
library project Stateful_IO is
   type Any_Library_Type is ("static", "relocatable", "static-pic");
   Library_Type : Any_Library_Type := external ("LIBRARY_TYPE", "static");

   for Library_Name use "stateful_io";
   for Library_Kind use Library_Type;
   for Library_Dir use "lib." & Library_Type;
   for Object_Dir use "obj." & Library_Type;
end Stateful_IO;

-- stateful_io.ads
package Stateful_IO is
   procedure Put (S : String);
   pragma Export (Ada, Put, "stateful_io_put");
end Stateful_IO;

-- stateful_io.adb
with GNAT.IO;

package body Stateful_IO is

   --  Since we are short-circuiting the Ada compiler's elaboration checks,
   --  preserve a manual guard to detect attempts to send data to the IO
   --  channel before elaboration has occurred.

   Initialized : Boolean := False;
```

(continues on next page)

(continued from previous page)

```
   procedure Put (S : String) is
   begin
      if not Initialized then
         raise Program_Error with
           "attempt to call Stateful_IO.Put before elaboration";
      end if;

      --  Replace the following with the actual code to send data to the IO
      --  stream used by coverage data.

      GNAT.IO.Put (S);
   end Put;

begin
   --  Here, do whatever necessary to initialize the IO stream

   Initialized := True;
end Stateful_IO;
```

Then, bind this code (non preelaborated) to the customized coverage runtime (`my_rts.gpr` as in the previous section, preelaborated) using the interfacing pragmas:

```
-- gnatcov_rts-base_io.adb
package body GNATcov_RTS.Base_IO is

   procedure Stateful_IO_Put (S : String);
   pragma Import (Ada, Stateful_IO_Put, "stateful_io_put");

   procedure Put (S : GNATcov_RTS_String) is
      Str : String (1 .. Integer (S.Length));
      for Str'Address use S.Str;
      pragma Import (Ada, Str);
   begin
      Stateful_IO_Put (Str);
   end Put;

end GNATcov_RTS.Base_IO;
```

Finally, make sure that the project file for the instrumented codebase contains a `with "stateful_io";` clause and that at least one of its unit has a `with Stateful_IO;` clause, so that the `Stateful_IO` library gets included during the build of the instrumented project.

3.11 Building instrumented programs with CCG

Programs instrumented with **`gnatcov instrument`** can be built using the GNAT Pro Common Code Generator provided a few modifications are made to the coverage runtime, and respecting some limitations in terms of dump-trigger and dump-channel choice.

3.11.1 Customizing the runtime

Given the workflow associated with CCG, it is not possible to use the **gnatcov setup** command to setup the coverage runtime. Instead, it must be prepared manually.

The coverage runtime contains both Ada and C sources. When using CCG through GPRbuild, projects containing C sources are not well handled and some steps of the build procedure won't be executed. There thus is an external variable to remove all C sources from the project. This means that C sources must be manually managed during the executable compilation later on.

The first step is to copy the runtime sources in a dedicated directory. For the remainder of this section, it will be denoted by `ccg_gnatcov_rts`.

```
mkdir ccg_gnatcov_rts
cp -r <gnatdas_install_dir>/share/gnatcoverage/gnatcov_rts ccg_gnatcov_rts
```

Then, C sources not relevant to the coverage runtime for the CCG configuration must be deleted:

```
rm ccg_gnatcov_rts/gnatcov_rts_c-base-io.c
rm ccg_gnatcov_rts/gnatcov_rts_c-trace-output-files.*
rm ccg_gnatcov_rts/gnatcov_rts_c-os_interface.*
```

Finally, it is necessary to modify the contents of `ccg_gnatcov_rts/gnatcov_rts-base_io.adb` to use an alternate medium on which the execution trace will be output. By default this relies on `GNAT.IO`, which is not available in the CCG runtime. One way to modify it is to implement it on top of the `putchar` libc function:

```
declare
   function Putchar (C : Integer) return Integer;
   pragma Import (C, Putchar);
   Ignored : Integer;
begin
   for C of Str loop
      Ignored := Putchar (Character'Pos (C));
   end loop;
end;
```

Otherwise, see *Customize the base64-stdout dump channel* for more details on the expected interface to dump the coverage trace information.

3.11.2 Building an instrumented program with CCG

Building an instrumented program with CCG is done in the same manner as with any other target, with the exception that GPRbuild must be instructed to use the custom coverage runtime prepared in the previous step, and by setting an external variable to configure the coverage runtime project for a build with CCG:

```
gprbuild -P <user_project.gpr> --src-subdirs=gnatcov-instr \
  --implicit-with=ccg_gnatcov_rts/gnatcov_rts.gpr \
  -XGNATCOV_RTS_FOR_CCG=true \
  <relevant target cargs>
```

After the generation of C sources is complete, all the coverage runtime C sources must be included in the build process of the executable. These can be found under `ccg_gnatcov_rts/` for the sources already present before the gprbuild invocation, and under `ccg_gnatcov_rts/obj-gnatcov_rts.static` for the sources generated by CCG. The C sources generated from the instrumented Ada sources will be available in the object directory of their corresponding project.

3.12 Common instrumentation errors

3.12.1 Linker error when building instrumented application

If the build of the instrumented programs fails with an undefined reference to a gnatcov defined symbol (e.g. “xcov_buf_stmt_<slug>”), it usually means that the list of coverage buffers that contain coverage data is not in the closure of the main. This is typically the case when the main was not instrumented as such, or when using the manual dump-trigger and inserting a dump buffer annotation in a source that is not in the main closure. Refer to *Output strategies for coverage information* for more information.

CHAPTER

FOUR

PRODUCING SOURCE TRACES WITH INTEGRATED INSTRUMENTATION (EXPERIMENTAL)

gnatcov provides an alternate way of instrumenting sources, available for C/C++ when using gcc or g++ as a compiler. Theoretically, any build system should be supported, but it has only been tested for the Makefile / CMake build systems so far.

As with the workflow involving a separate instrumentation, a *coverage runtime* to be used by the instrumented code needs to be setup as a required prerequisite. Refer to the *Setting up the coverage runtime library* section of this manual for a description of this step. The installed coverage runtime must be visible on the `GPR_PROJECT_PATH`. Note that, by default, **gnatcov setup** sets up the coverage runtime for Ada and C/C++. If no Ada toolchain is available, its support should be explicitly disabled. For instance:

```
gnatcov setup --restricted-to-languages=C,C++ --prefix=/usr/custom
export GPR_PROJECT_PATH=$GPR_PROJECT_PATH:/usr/custom/share/gpr
```

Then the process essentially goes like:

1. Setup the integrated instrumentation process, specifying the instrumentation parameters (units of interest, coverage level etc.) and the compilers of use.
2. Build the instrumented code using the generated compiler wrapper(s);
3. Execute the program to produce a trace.

The compiler wrapper embeds knowledge through the `gnatcov_config.json` configuration file that is generated alongside it. It is thus necessary to re-run the setup step it if the closure of sources of interest, or if the build environment (e.g. the path for the original compiler) changes.

4.1 Windows specifics

GNATcoverage uses `ldd` to investigate the dynamic dependencies of the compiled binaries. It should thus be installed on the Windows machine.

4.2 Specifying instrumentation switches

The user specifies the switches through the **gnatcov setup-integration** command, which accepts any option accepted by **gnatcov instrument**, except the ones using project mechanisms to filter out units of interest: `--units` and `--projects`. Refer to *Producing source traces with gnatcov instrument* for more information regarding the source instrumentation specific switches.

The same limitations apply as those specific to C and C++ coverage, see the *C/C++ limitations* paragraph for the full details.

As there is no project acting as a units of interest provider, every file that is not a system header is considered as a file of interest. The user can explicitly specify files of interest through the `--files` switch, which expects a list of full filenames that can be passed through the command line, or using a response file.

To generate a compiler wrapper, use the `--compilers` switch. The list of supported compilers is `gcc` and `g++`.

This will generate in the current directory, or in the directory specified by the `--output-dir` switch an executable compiler wrapper and a `gnatcov_config.json` file along. Note that this configuration file is used to configure various instrumentation options: it is thus very important that it resides in the same directory as the compiler wrapper. Note that all of the instrumentation artifacts, notably SID files that must be passed later to **gnatcov coverage**, are also generated in the current or specified output directory.

4.3 Building an instrumented executable

The build process can be run unchanged after it has been configured to use the generated compiler wrapper instead of the original compiler.

4.4 Generating a coverage report

The coverage generation report is as documented under the section *Source coverage analysis with gnatcov coverage*. The coverage obligations are specified through the `--sid` by passing the list of files with a sid extension generated by the build of the instrumented executable, under the output directory that was specified or implicitly assumed, when running the **gnatcov setup-integration** command.

4.5 A simple Makefile example

The sources for the following example can be found under the `share/examples/gnatcoverage/doc/integrated` directory of the GNAT DAS distribution.

The following considers that the coverage runtime was previously installed with **gnatcov setup**, and that the `GPR_PROJECT_PATH` variable contains its installed location. See *Setting up the coverage runtime library*.

Let's consider a simple `main.cpp` file:

```
#include <iostream>

int main(int argc, char **argv){
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

and the following Makefile:

```
CC=g++
OBJ = main.o

%.o: %.c
        $(CC) -c -o $@ $<

test: $(OBJ)
        $(CC) -o $@ $^
```

We start by configuring the instrumentation process:

```
cd <my-project>
gnatcov setup-integration --files=<my_project>/main.cpp --compilers=g++
```

Then, we launch the build processed unchanged, with the compiler wrapper first on the path:

```
export PATH=<my-project>:$PATH
make
```

This will produce an instrumented executable, that will produce a source trace when run, that can be analyzed with **gnatcov setup**.

4.6 A simple CMake example

The sources for the following example can be found under the `share/examples/gnatcoverage/doc/integrated` directory of the GNAT DAS distribution. To ensure that the Makefile provided with the example sources uses CMake as a build system, specify `BUILD_SYSTEM=CMake` on the *make* invocation.

The following considers that the coverage runtime was installed through the use of **gnatcov setup**.

Let's consider a simple `main.cpp` file

```
#include <iostream>

int main(int argc, char **argv){
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

The CMakeLists.txt file to be used to compile the main.cpp file is :

```
cmake_minimum_required(VERSION 3.5)
project(HelloWorld)

add_executable(hello_world main.cpp)
```

We start by creating the build directory, and configuring the instrumentation process there:

```
cd <my-project>
mkdir build
cd build
gnatcov setup-integration --files=<my_project>/main.cpp --compilers=g++
```

This creates a g++ compiler wrapper in the build directory, along with a `gnatcov_config.json` file that we intend to use as a proxy for compilation. To do that, we have to configure the CMake build process accordingly, using the `CMAKE_CXX_COMPILER` variable. We run the configuration command in the build directory:

```
cmake .. -DCMAKE_CXX_COMPILER=<my_project>/build/g++
```

The default generator for CMake is "Unix Makefiles", so we can then run the build process with `make`, and our executable which will produce a source trace that can be analyzed by **`gnatcov coverage`**.

CHAPTER

FIVE

SOURCE COVERAGE ANALYSIS WITH GNATCOV COVERAGE

Source coverage analysis computes metrics focused on source programming language entities such as high level *statements* or *decisions* (**DO-178C** parlance for boolean expressions), which translate as *Source Coverage Obligations*, or SCOs, in GNATcoverage terms.

Once you have produced coverage traces, actual analysis is performed with **gnatcov coverage** to generate coverage reports. *Source* coverage is queried by passing a specific `--level` argument to **gnatcov coverage**.

The general structure of this command line is always like:

```
gnatcov coverage --level=<criterion> --annotate=<format>
                 <units-of-interest> ... <traces-or-checkpoints>
```

The next sections describe the *available report formats*, then provide more details regarding *Statement Coverage analysis*, *Decision Coverage analysis*, and *Modified Condition/Decision Coverage analysis*.

The *Specifying Units Of Interest* chapter describes how application units to be considered for coverage assessment are to be specified.

5.1 gnatcov coverage command line

Coverage analysis with GNATcoverage is performed by invoking **gnatcov coverage** for a set of criteria queried via the `--level` command line option. The general interface synopsis is available from **gnatcov** `--help`:

```
gnatcov coverage OPTIONS TRACE_FILES
```

The available options are as follows:

-c, --level ***(mandatory)*****:**
Tell the set of coverage criteria to be assessed. The possible values for source coverage analysis are `stmt`, `stmt+decision`, `stmt+mcdc`, and `stmt+uc_mcdc`, all explained later in this chapter.

-a, --annotate ***(mandatory)*****:**
Request one or more output report formats. All the criteria support `cobertura`, `html,`, `report`, `xcov[+]`, `xml` formats, with interpretations that vary depending on the assessed criteria. See the corresponding documentation later in this chapter for more details. This option accepts comma separated values and/or can be specified multiple times on the command line, in which case there will be one report produced for each of the requested formats.

-o :
Request that the synthetic report produced by `--annotate=report` be output in the provided filename instead of standard output by default. This is just ignored for other output formats.

--timezone :
Select the timezone to use when displaying trace creation dates in reports, which could be either `local` or `utc`. The local timezone is used by default.

--output-dir :
Request that the report files (index and annotated sources for the `xcov` and `html` output formats) be output in the provided directory. If not specified, the default is the root project's object directory if using projects, and the current directory if not. If more than one of the above annotation formats is requested, then each report will be placed in a subdirectory named accordingly.

--report-title :
Request that generated HTML documents (index for the `html` output format) are assigned a customized title. For instance, passing `--report-title="Project ABC"` will yield titles such as: `Project ABC - GNATcoverage Report`. If passed multiple times, passing an empty string last will restore the default behavior. This option is ignored is the selected output format does not support titles.

-T, --trace *(mandatory)*, possibly repeated and accepting *@listfile arguments* :
Provide the set of execution traces for which a report is to be produced. When multiple traces are provided, **gnatcov** produces a consolidated result, as described in detail in the *Coverage Consolidation* chapter of this manual.

--exec:
Override executable from traces. Trace files contain an indication of the executable used to generate them. This option causes the named executable to be loaded for coverage analysis, and to override the indication contained in any trace specified after it on the command line. An empty executable name may be specified to restore the default behavior of using the indication contained in each trace file. Note that `--exec` may appear last on the command line, in which case it applies to no trace file, but still causes the indicated executable to be included in the coverage analysis. This ensures that any code in that executable that is not exercised by some trace file will be reported as not covered.

--warnings-as-errors:
Treat warnings as errors, i.e. exit with a non-zero status code if a warning is emitted.

-t, --target :
State the target toolchain configuration used to build the analyzed programs, as provided to `gprbuild`. For cross or 32bit native configurations, this switch together with its possible `--RTS` companion is required for all commands using project files unless the root project provides the information with `Target`/`Runtime` attributes.

--non-coverable:
Report about language statements for which no object code could be found in the surrounding subprogram (typically out of optimization).

-P:
Use the indicated project file as the root project to select the units of interest for the analysis and find default options. Default options are taken only from this project. In absence of `--projects` and of `--no-subprojects`, the units of interest are those designated by this project and all it's transitive dependencies, minus those advertised as externally-built. The target/runtime/scenario contextual options that would be passed to build the project should also be passed for proper interpretation of the project files. See the *Specifying Units Of Interest* chapter of this manual for more details.

--projects, possibly repeated and accepting *@listfile arguments*:
When using `-P`, use the provided projects to select units of interest, together with their transitive dependencies unless `--no-subprojects` is also provided. The projects designated by this option must all be part of the import transitive closure reachable from the root project designated by `-P`.

--no-subprojects:
Consider only the projects (and units) encompassed by the `-P` / `--projects` options as of-interest. Don't include any project imported from there.

`--units`, possibly repeated and accepting *@listfile arguments*:
When using project files, override the list of units of interest for source coverage with those provided.

`--subdirs`:
When using project files, look for *Library Information files* in the indicated subdirectory of each project's object directory.

`--scos`, possibly repeated and accepting *@listfile arguments*:
Provide the set of *Library Information files* from which Source Coverage Obligations (SCOs) should be loaded to process **binary** traces. This low-level switch effectively overrides the project based units of interest selection. It only has effect if used with binary traces.

`--sid`, possibly repeated and accepting *@listfile arguments*:
Provide the set of *Source Instrumentation Data files* from which Source Coverage Obligations (SCOs) should be loaded to process **source** traces. This low level switch effectively overrides the project based units of interest selection. It only has effect if used with source traces.

`--excluded-source-files`, possibly repeated and accepting *@listfile arguments*:
Provide a list of globbing patterns (as in Unix shells) of source files to be excluded from the analysis and from the output report. See the *Conveying source files to ignore / handling Ada subunits* section for more information.

`--dump-units-to`:
For source coverage analysis specifically, output the names of units that are considered of-interest to the requested assessment, that is, for which a report or checkpoint is going to be produced. This also outputs, for each unit of interest, the list of files that were individually ignored using the `Excluded_Source_Files` project attribute or corresponding command-line option. Excluded source files listed with `--dump-units-to` will be either marked as `always ignored`, if they were ignored in all the inputs (traces or checkpoints) that were used to produce this report, or as `sometimes ignored`, if the source files were ignored in at least one of the inputs of this report, but not all of them. The argument may be either the name of a file, clobbered it if it already exists, or '-' to request displaying the list on standard output. In the latter case, when a `report` output is also requested, the list of units is displayed as an additional report section.

`--save-checkpoint`:
Save the resulting coverage analysis to the named checkpoint file.

`--checkpoint`, possibly repeated and accepting *@listfile arguments*:
Load previously saved coverage analysis checkpoint(s), and continue coverage analysis from that initial state.

`--source-rebase`, possibly repeated and accepting *@listfile arguments*:
Specify alternate absolute path prefixes to locate source files when producing annotated source outputs. See *Handling source relocation for annotated sources output formats*

`--source-search`, possibly repeated and accepting *@listfile arguments*:
Specify a directory in which missing source files will be searched when producing annotated source outputs. See *Handling source relocation for annotated sources output formats*

A lot of options are available to control the set of units for which coverage is to be assessed. They may be combined in multiple ways and attributed within the project files are available to refine the set of units to include or exclude from each designated project. See *Specifying Units Of Interest* for extra details, and *Using project files* for a general overview of how the project facilities operate.

Saving coverage analysis state checkpoints allows the production of consolidated results from successive runs of the `coverage` command. In particular this allows coverage results to be computed incrementally, and allows consolidation with different sets of units of interest, in order to avoid incidental coverage. See *Consolidation from coverage checkpoints* for a discussion of these use cases.

Positional arguments on the command line (not tied to a particular option) are considered as trace file arguments. At least one trace file is required for the `coverage` command to operate, which may but need not be introduced with `-T` or `--trace`. Here are a few examples of valid command lines to illustrate. Other examples will be exposed along the course of the following sections:

```
gnatcov coverage --level=stmt -P ut.gpr --annotate=report --trace=prog.srctrace
#                      (a)       (b)            (c)             (d)
# (a) Request Statement coverage assessment,
# (b) for units belonging to the "ut.gpr" project,
# (c) producing a synthetic text report on standard output (no -o option),
# (d) out of a single execution trace "prog.trace".

gnatcov coverage --level=stmt+decision -P ut.gpr --annotate=html t1 t2
# Statement and Decision coverage assessments for two traces "t1" and "t2",
# producing an html report in the current directory.

gnatcov coverage --level=stmt+decision -P ut.gpr --annotate=html @mytraces
# Same report, with t1 and t2 listed in the "mytraces" text file

gnatcov coverage --level=stmt -Papp.gpr --annotate=html @mytraces
# Same kind of report, for Statement coverage only, on source units owned
# by "app.gpr" and its transitive closure of project dependencies.

gnatcov coverage --level=stmt -Papp.gpr --no-subprojects --annotate=html @mytraces
# Likewise, considering only the units owned by app.gpr
```

5.2 Output report formats (`--annotate`)

Source coverage reports may be produced in various formats, as requested with the `--annotate` command line argument of **`gnatcov coverage`**. The `xcov` and `html` formats produce a set of annotated source files, in the directory where **`gnatcov`** is launched unless overridden with a `--output-dir` option. The `report` output consists in a synthetic text report of *coverage violations* with respect to the requested criteria, produced on standard output by default or in the file specified by the `-o` command line option. The `cobertura` produce a single xml file containing a coverage report in the Cobertura format. Its name is by default *cobertura.xml* and can be altered using the `-o` command line option.

Later in this chapter we name output formats by the text to add to `--annotate` on the command line. For example, we use "the `=report` outputs" to mean "the coverage reports produced with `--annotate=report`".

We will illustrate the various formats with samples extracted from outputs obtained by performing coverage analysis of the following example Ada application unit:

```
function Between (X1, X2, V : Integer) return Boolean;
--  Is V between X1 and X2, inclusive and regardless of their ordering?

function Between (X1, X2, V : Integer) return Boolean is
begin
   if X1 < X2 then
      return V >= X1 and then V <= X2;
   else
      return V >= X2 and then V <= X1;
   end if;
end Between;
```

5.2.1 Cobertura report (=cobertura)

`--annotate=cobertura` produces a coverage report in the Cobertura format, as specified per the cobertura.dtd (document type description) that is generated alongside.

This format specification is not maintained by gnatcov, and it does not thus provide all of the information that other report formats do, notably MC/DC coverage information, violation messages. Decision violations are output using the "branch" terminology of Cobertura (one decision being two branches, and either 0, 1 or 2 of those branches are covered).

It is mainly provided for integration with external tools, such as continuous integration systems, e.g. gitlab, which supports integration of coverage reports into merge requests using this format.

File names in this coverage report are absolute. Depending on the use context of this coverage report, it can also make sense to strip a given prefix from the absolute paths to make them relative to, e.g. a project root. The `--source-root` command line option accepts a string prefix that will be removed from absolute path references in the report.

5.2.2 Annotated sources, html (=html)

For source coverage criteria, **gnatcov coverage** `--annotate=html` produces an index-based report under the HTML format (the other command names are aliases for backward compatibility).

To navigate the report, open the index.html file using the browser of your choice. This index file contains a summary of the assessment context (assessed criteria, trace files involved, ...) and of the coverage results for all the units, with links to their annotated sources. If the `-P` was used to designate the source units of interest, sources are indexed per-project.

Note that some dynamic filtering / sorting can be applied:

- Filter by kind of coverage obligations: either reporting on lines, or on statement / decision / MCDC (one or several) obligations, depending on the coverage level. See *Synthetic metrics* for more information.
- Sort indexes by clicking on column headers, allowing for example sorts keyed on unit names or on relative coverage achievement.

See our *sample html index* appendix for an example index page, which embeds a self-description of all the items it contains.

The user can browse through an annotated version of the sources from the index. Each annotated source page contains a summary of the assessment results. This summary can be expanded to print subprogram metrics: the user can click on a subprogram's metric to access it in the annotated source immediately. This summary is followed by the original source lines, all numbered and marked with a coverage annotation as in the `--annotate=xcov` case. Lines with obligations are colorized in green, orange or red for `+`, `!` or `-` coverage respectively.

See the *sample annotated source* appendix for a sample of html annotated source.

5.2.3 SARIF report (=sarif)

For source coverage criteria, **gnatcov coverage** `--annotate=sarif` produces a JSON-based report under the Static Analysis Results Interchange Format (SARIF).

The generation of SARIF reports is mainly provided to allow viewing gnatcov coverage reports with external IDEs that supports viewing them (for example, Visual Studio Code with Microsoft's SARIF Viewer extension). To navigate this report, open the `coverage.sarif` file using an integrated development environment which supports viewing SARIF results.

The reports contain information on the violations observed by gnatcov as "Results" and the enabled coverage levels as "Rules". Each violation is registered as "error". Exempted violations are reported as "note" and undetermined coverage results as "warning".

5.2.4 Violations summary, text (=report)

For source coverage criteria, **gnatcov coverage** `--annotate=report` produces a summary that lists all the *coverage violations* (failure to satisfy some aspect of a coverage criterion) relevant to the set of assessed criteria.

The report features explicit start/end of report notifications and at least three sections in between: Assessment Context, Coverage Violations, and Analysis Summary. Should **gnatcov** be unable to determine the coverage state of some coverage obligations, those will be reported in a dedicated Undetermined Coverage Items section, with a description of why the tool was unable to determine the coverage state for each obligation. A few variations are introduced when *exemption regions* are in scope. See the *Coverage Exemptions* section for more details on their use and effect on the output reports.

If `--dump-units-to -` is also on the command line, a *UNITS OF INTEREST* section is produced, which contains the list of units considered of-interest for the reported assessment, as well as the list of source files individually ignored with the `Excluded_Source_Files` project attribute and corresponding command-line option.

Assessment Context

The *Assessment Context* report section exposes the following information items:

- Date & time when the report was produced
- Command line and Version of GNATcoverage that produced the report. The set of units that the report is about is conveyed by the command line switches summarized there (`--projects`, `--units`, `--scos`).
- Coverage level requested to be analyzed
- Details on the input trace files, such as the path to the binary program exercised (as recorded in the trace header) or the trace production time stamp.

Here is a example excerpt:

```
==========================
== 1. ASSESSMENT CONTEXT ==
==========================

Date and time of execution: 2011-11-24 16:33:44.00
Tool version: GNATcoverage 1.0.0w (20111119)

Command line:

gnatcov coverage -Pmytest.gpr --level=stmt+mcdc --annotate=report test_x1x2.trace

Coverage level: stmt+mcdc

Trace files:

test_x1x2.trace
  program: obj/test_x1x2
  date   : 2011-11-24 15:33:44
  tag    : sample run
```

Coverage Violations

The *Coverage Violations* report section lists and counts the coverage violations that relate to source lines not part of an exemption region. The violations are grouped in subsections, one per assessed criterion according to the `--level` option:

`--level=`	Assessed criteria / Report subsections
stmt	Statement Coverage
stmt+decision	Statement and Decision Coverage
stmt+mcdc	Statement, Decision and MCDC Coverage

All the violations are reported using a consistent format, as follows:

```
ranges.adb:7:10: statement not executed
  source  :sloc: violation description
```

source and *sloc* are the source file basename and the precise `line:column` location within that source where the violation was detected.

The following table summarizes the list of violation items that might be emitted together for each criterion:

Criterion	Possible violations
Statement Coverage	`statement not executed`
Decision Coverage	`decision outcome TRUE not covered`
	`decision outcome FALSE not covered`
	`one decision outcome not covered`
MCDC Coverage	all the decision coverage items, plus ...
	`condition has no independent influence pair`

When multiple violations apply someplace, only the most basic diagnostic is emitted, not the more precise ones corresponding to stricter criteria. For instance, if an Ada statement like `X := A and then B;` is not covered at all, a `statement not executed` violation is always emitted alone, even when assessing `--level=stmt+mcdc` and we also have improper decision and conditions coverage.

Here is an output excerpt for our example with `--level=stmt+mcdc`, producing one subsection for each of the three criteria requested at that level:

```
============================
== 2. COVERAGE VIOLATIONS ==
============================

2.1. STMT COVERAGE
------------------

ranges.adb:7:10: statement not executed

1 violation.

2.2. DECISION COVERAGE
----------------------

ranges.adb:4:10: decision outcome FALSE never exercised
```

(continues on next page)

(continued from previous page)

```
1 violation.

2.3. MCDC COVERAGE
------------------

ranges.adb:5:17: condition has no independent influence pair, MC/DC not achieved

1 violation.
```

Analysis Summary

The *Analysis Summary* report section summarizes just the counts reported in each of the previous sections. For our example report so far, this would be:

```
==========================
== 3. ANALYSIS SUMMARY ==
==========================

1 non-exempted STMT violation.
1 non-exempted DECISION violation.
1 non-exempted MCDC violations.
```

This section provides a quick way to determine whether the requested coverage level is fully satisfied, with details available from the per criterion sections that precede.

5.2.5 Annotated sources, text (=xcov[+])

For source coverage criteria, **gnatcov coverage** `--annotate=xcov` produces an annotated version of each source file, in text format, named after the original source with an extra `.xcov` extension at the end (`x.ext.xcov` for a source named `x.ext`).

Each annotated source contains a global summary of the assessment results followed by the original source lines, all numbered and marked with a coverage annotation next to the line number. The annotation on a line always consists in a single character, which may be one of the following:

Annotation	Meaning
.	No coverage obligation is attached to the line
-	Coverage obligations attached to the line, none satisfied
!	Coverage obligations attached to the line, some satisfied
?	Coverage obligations attached to the line, undetermined coverage state (*)
+	Coverage obligations attached to the line, all satisfied

(*) The Undetermined Coverage state (?) is only shown on the line in the absence of other known violations for that same line.

Here is, to illustrate, the full statement coverage report produced for our example unit when the `Between` function was called so that the `if` control evaluated True only. The function is actually part of an Ada package, called Ranges, with an original body source file named `ranges.adb`:

```
examples/src/ranges.adb:
67% of 3 lines covered
Coverage level: stmt
  1 .: package body Ranges is
  2 .:    function Between (X1, X2, V : Integer) return Boolean is
  3 .:    begin
  4 +:       if X1 < X2 then
  5 +:          return V >= X1 and then V <= X2;
  6 .:       else
  7 -:          return V >= X2 and then V <= X1;
  8 .:       end if;
  9 .:    end;
 10 .: end;
```

`--annotate=xcov+` (with a trailing +) works the same, only providing extra details below lines with improperly satisfied obligations. The available details consists in the list of *coverage violations* diagnosed for the line, which depends on the coverage criteria involved. Here is an excerpt for our previous example, where the only improperly satisfied obligation is an uncovered statement on line 7:

```
7 -:          return V >= X2 and then V <= X1;
STATEMENT "return V ..." at 7:10 not executed
```

5.2.6 XML report, xml (=xml)

`--annotate=xml` produces a coverage report in the XML format, as specified per the `gnatcov-xml-report.xsd` (XML schema description) that is generated alongside.

This report format is on par with the HTML report in terms of features, and it is the preferred choice for programmatically accessing the coverage results.

5.3 Statement Coverage analysis

5.3.1 Core notions and Reporting (--level=stmt)

gnatcov performs Statement Coverage assessments with the `--level=stmt` command line option. The assessment determines the status of statement coverage obligations out of the tests execution, considering that:

- A statement is *covered*, and the obligation *discharged*, as soon as the control flow has reached the statement.
- A statement is *uncovered* otherwise.

There is no notion of partial coverage for statements, even when a statement execution is interrupted somehow, for example by an exception occurrence. For instance, the statement below will be reported as covered as soon as the statement is reached, even if the expression evaluation never really terminates:

```
X := Function_That_Raises_Exception (Y) + Z;
```

In synthetic `=report` outputs, unexecuted source statements are listed as Statement Coverage violations in the report section dedicated to these.

In annotated source outputs, the coverage annotations convey the following indications:

Annotation	Meaning
-	At least one statement on the line, none covered
!	More than one statement on the line, some covered
?	At least one statement on this line, some with undetermined coverage state (*)
+	At least one statement on the line, all covered

(*) The Undetermined Coverage state (?) is only shown on the line in the absence of other known violations for that same line.

When a single statement spans multiple lines, the coverage annotation is present on all the lines, as the two + signs for the single assignment in the following excerpt:

```
3 .:  -- A single assignment spanning two lines:
4 +:  Result :=
5 +:     Input1 * Input2;
```

For compound statements, the coverage status of the compound construct per se is reported only on the parts that embed flow control expressions. For an Ada *if* statement, for example, coverage is reported on the `if` or `elsif` lines only, not on the `else`, or `end if;` lines, and not on lines where inner statements reside. The lines where inner statements reside are annotated in accordance with the nature and coverage status of those statements only. For example, see the `.` annotations on lines 4 and 6 in:

```
2 +:  if This_Might_Not_Be_True then
3 -:     Result := -1;
4 .:  else
5 +:     Result := 12;
6 .:  end if;
```

Declarations are generally considered as statements, so are reported covered/uncovered when they have initialization code associated with them.

Note that if no executable code for a given unit can be found in any of the executables submitted to gnatcov, then all statements in the unit will be conservatively reported as uncovered. This ensures that if tests for an entire unit have been omitted from a test campaign, a violation will be properly reported. Such violations can be suppressed either using exemptions, or by removing the unit from the list of units of interest.

5.3.2 Example program and assessments

Let us consider the example functional unit below, with the spec and body stored in source files named `div_with_check.ads` and `div_with_check.adb`:

```
function Div_With_Check (X, Y : Integer) return Integer;
--  return X / Y if Y /= 0. Raise Program_Error otherwise

function Div_With_Check (X, Y : Integer) return Integer is
begin
   if Y = 0 then
      raise Program_Error;
   else
      return X / Y;
   end if;
end;
```

We first exercise the function for Y = 1 only, using the following *test driver* in `test_div1.adb`:

```
procedure Test_Div1  is
   X : constant Integer := 4;
begin
   Assert (Div_With_Check (X, 1) = X);
end;
```

From a `test_div1.srctrace` obtained by building the instrumented source produced with **gnatcov instrument**, we analyze for the Statement Coverage criterion using the following **gnatcov coverage** invocation:

```
gnatcov coverage --level=stmt -P ut.gpr --annotate=xcov test_div1.srctrace
```

We get an =xcov annotated source result in text format for the functional unit on which the analysis is focused, in `div_with_check.adb.xcov`:

```
examples/src/div_with_check.adb:
67% of 3 lines covered
Coverage level: stmt
  1 .: function Div_With_Check (X, Y : Integer) return Integer is
  2 .: begin
  3 +:    if Y = 0 then
  4 -:       raise Program_Error;
  5 .:    else
  6 +:       return X / Y;
  7 .:    end if;
  8 .: end;
```

We can observe that:

- Only the `if` line of the compound *if* statement is annotated, as covered since the function was called.
- The inner `raise` and `return` statements are marked uncovered and covered respectively, as expected since the function was only called with arguments for which the `if` controlling decision evaluates False.

As a second experiment, we exercise the function for Y = 0 only, using:

```
procedure Test_Div0  is
   Result : Integer
     := Div_With_Check (4, 0);
begin
   Put_Line ("R = " & Integer'Image (Result));
end;
```

We request results on the test driver as well this time, as it features constructs of relevance for our purposes:

```
gnatcov coverage --level=stmt -Pmytest.gpr --annotate=xcov test_div0.trace
```

The =xcov outputs follow. First, for the functional unit, with the `if` statement coverage reversed compared to the previous testcase:

```
1 .: function Div_With_Check (X, Y : Integer) return Integer is
2 .: begin
3 +:    if Y = 0 then
4 +:       raise Program_Error;
5 .:    else
```

(continues on next page)

(continued from previous page)

```
6 -:       return X / Y;
7 .:    end if;
8 .: end;
9 .:
```

Then, for the test driver where we can note that

- The two lines of the local `Result` definition are annotated,
- This definition is marked covered even though it was evaluated only once with an initialization expression that raised an exception, and
- The driver body is reported uncovered, as expected since an exception triggered during the elaboration of the subprogram declarative part.

```
67% of 3 lines covered
Coverage level: stmt
   1 .: with Div_With_Check, Ada.Text_IO; use Ada.Text_IO;
   2 .:
   3 .: procedure Test_Div0  is
   4 +:    Result : Integer
   5 +:      := Div_With_Check (4, 0);
   6 .: begin
   7 -:    Put_Line ("R = " & Integer'Image (Result));
   8 .: end;
```

The corresponding synthetic report is simply obtained by running **gnatcov coverage** again with `--annotate=report` instead of `--annotate=xcov`:

```
===========================
== 1. ASSESSMENT CONTEXT ==
===========================
...

============================
== 2. COVERAGE VIOLATIONS ==
============================

2.1. STMT COVERAGE
------------------
div_with_check.adb:6:7: statement not executed
test_div0.adb:7:4: statement not executed

2 violations.

========================
== 3. ANALYSIS SUMMARY ==
========================

2 STMT violations.
```

We can see here that the two lines marked `-` in the `=xcov` outputs are properly reported as violations in the `STMT COVERAGE` section of this report, and that this section is the only one presented in the `COVERAGE VIOLATIONS` part, as only this criterion was to be analyzed per the `--level=stmt` argument.

5.4 Decision Coverage analysis

5.4.1 Core notions and Reporting (`--level=stmt+decision`)

With the `--level=stmt+decision` command line option, **`gnatcov`** performs Statement and Decision Coverage assessments combined together. *Decisions* in this context are defined as:

- Any Boolean expression used to influence the control flow via explicit constructs in the source program, such as `if` statements or `while` loops;
- The controlling predicate of Ada if-expressions or C conditional-expressions, in any kind of context except Ada contracts.
- The iteration predicate of Ada quantified expressions, in the same contexts as for if-expressions.
- The iterator filter predicate optionally present in a loop parameter specification or in an iterator specification, in the same context as if-expressions.

The expression may be of essentially any type in C. In Ada, this may be the standard Boolean type, or subtypes of it, or derived types thereof. Here are a few examples show-casing decisions in a variety of contexts:

```
while (z++) ...  /* "z++", controlling a while-loop.  */

if X > 0 then ...  -- "X > 0", controlling an if-statement

q = z-- ? x : y;  /* "z++", conditional-expression predicate.  */

Q := (X if Z < 2 else Y); -- "Z < 2", if-expression predicate

Z := (for all X of Container => P(X));  -- "P(X)", quantified-expression predicate
T := (for some X of Container => P(X));

for I in 1 .. 10 when (I mod 2) = 0 loop ... -- "(I mod 2) = 0", iterator filter␣
↪predicate
F := (for Elt of Container when Pred (Elt) => Elt); -- "Pred (Elt)", iterator filter␣
↪predicate
```

The coverage status of a decision obligation is determined as follows:

- A decision is said *fully covered*, or just *covered*, and the obligation discharged, as soon as the decision has been evaluated at least once True and once False during program execution.
- A decision is said *uncovered* when none of the possible outcomes was exercised, either because the enclosing statement was not executed at all or when all the attempted evaluations were interrupted e.g. because of exceptions.
- If only one of the two possible outcomes was exercised, the decision is said *partially covered*, and the obligation only partially discharged.

When a decision is part of a statement and the statement is not executed at all, only the statement level violation is reported. The nested decision level violations are implicit in this case and diagnosing them as well would only add redundancy.

The `=report` synthetic output lists the statement and decision coverage violations in the `STMT` and `DECISION` coverage report section respectively. For the `=xcov` and `=html` annotated-source oriented formats, the single annotation produced on each source line combines the statement and decision coverage indications. The following table summarizes the meaning of the possible annotations:

Annotation	Meaning
-	At least one statement on the line, none executed.
!	Unless multiple statements are involved, decision partially covered on the line.
?	At least one statement or decision on the line with undetermined coverage state. (*)
+	All the statements and decisions on the line are covered.

(*) The Undetermined Coverage state (?) is only shown on the line in the absence of other known violations for that same line.

When a trailing + is added to the format passed to `--annotate` (`=xcov+`), a precise description of the actual violations is available for each line in addition to the annotation. The `=html` provides it by default.

5.4.2 Example program and assessments

Consider the example functional Ada unit below, with the spec and body stored in source files named `divmod.ads` and `divmod.adb`:

```
procedure Divmod
  (X, Y : Integer; Value : out Integer;
   Divides : out Boolean; Tell : Boolean);
--  Compute X / Y into VALUE and set DIVIDES to indicate whether
--  Y divides X. Output a note to this effect when requested to TELL.
```

```
procedure Divmod
  (X, Y : Integer; Value : out Integer;
   Divides : out Boolean; Tell : Boolean) is
begin
   if X mod Y = 0 then
      Divides := True;
      if Tell then
         Put_Line (Integer'Image(Y) & " divides " & Integer'Image(X));
      end if;
   else
      Divides := False;
   end if;

   Value := X / Y;
end Divmod;
```

We first experiment with the following test driver:

```
procedure Test_Divmod2  is
   Value : Integer;
   Divides : Boolean;
begin
   Divmod (X => 5, Y => 2, Value => Value,
           Divides => Divides, Tell => True);
   Assert (Divides = False);

   Divmod (X => 6, Y => 2, Value => Value,
           Divides => Divides, Tell => True);
```

(continues on next page)

(continued from previous page)

```
   Assert (Divides = True);
end Test_Divmod2;
```

This exercises the `Divmod` function twice. The outer `if` construct executes both ways and the `if Tell then` test runs once only for `Tell` True. As a result, the only `stmt+decision` violation by our driver is the `Tell` decision coverage, only partially achieved since we have only exercised the True case. This is confirmed by `=report` excerpt below, where we find the two violations sections in accordance with the requested set of criteria:

```
2.1. STMT COVERAGE
------------------

No violation.

2.2. DECISION COVERAGE
----------------------

divmod.adb:14:10: decision outcome FALSE never exercised

1 violation.
```

For `--annotate=xcov`, this translates as a single partial coverage annotation on the inner `if` control line:

```
 8 .: procedure Divmod
 9 .:    (X, Y : Integer; Value : out Integer;
10 .:     Divides : out Boolean; Tell : Boolean) is
11 .: begin
12 +:     if X mod Y = 0 then
13 +:        Divides := True;
14 !:        if Tell then
15 +:           Put_Line (Integer'Image (Y) & " divides " & Integer'Image (X));
16 .:        end if;
17 .:     else
18 +:        Divides := False;
19 .:     end if;
20 .:
21 +:     Value := X / Y;
22 .: end Divmod;
```

Now we exercise with another test driver:

```
procedure Test_Divmod0  is
   Value : Integer;
   Divides : Boolean;
begin
   Divmod (X => 5, Y => 0, Value => Value,
           Divides => Divides, Tell => True);
end Test_Divmod0;
```

Here we issue a single call passing 0 for the Y argument, which triggers a check failure for the `mod` operation.

This results in the following `=xcov` output:

```
 8 .: procedure Divmod
 9 .:   (X, Y : Integer; Value : out Integer;
10 .:    Divides : out Boolean; Tell : Boolean) is
11 .: begin
12 !:    if X mod Y = 0 then
13 -:       Divides := True;
14 -:       if Tell then
15 -:          Put_Line (Integer'Image (Y) & " divides " & Integer'Image (X));
16 .:       end if;
17 .:    else
18 -:       Divides := False;
19 .:    end if;
20 .:
21 -:    Value := X / Y;
22 .: end Divmod;
```

We have an interesting situation here, where

- The outer `if` statement is reached and covered (as a statement),
- No evaluation of the `X mod Y = 0` decision terminates, because the only attempted computation is interrupted by an exception, so none of the other statements is ever reached.

This gets all confirmed by the `=report` output below, on which we also notice that the only diagnostic emitted for the uncovered inner `if` on line 14 is the statement coverage violation:

```
2.1. STMT COVERAGE
------------------
divmod.adb:13:7: statement not executed
divmod.adb:14:7: statement not executed
divmod.adb:15:10: statement not executed
divmod.adb:18:7: statement not executed
divmod.adb:21:4: statement not executed

5 violations.

2.2. DECISION COVERAGE
----------------------
divmod.adb:12:7: decision never evaluated

1 violation.
```

5.5 Modified Condition/Decision Coverage analysis

5.5.1 Core notions and Reporting (`--level=stmt+mcdc`)

Combined Statement and Modified Condition/Decision Coverage (MCDC) analysis is performed by passing the `--level=stmt+mcdc` option to **`gnatcov coverage`** commands. *Decisions* in this context are defined as:

- All the expressions considered as decisions for decision coverage, as well as:
- All the Boolean expressions which combine operands with *short-circuit* logical operators ("&&" and "`||`" in C, "`and then`" and "`or else`" by default in Ada), in any kind of context except Ada contracts.

Then for all the decisions in the sources of interest:

- Testing shall exercise both the True and False outcomes of the expression as a whole and demonstrate the *independent influence* of *conditions* in **DO-178C** parlance, where ...
- Separate conditions in a decision are identified as the operands of *short-circuit* operators.

Regarding coverage status definitions:

- A condition is *covered*, and the obligation discharged, when the independent effect on the enclosing decision was demonstrated by the tests,
- A condition is said *uncovered* otherwise.

The *Example decisions and conditions* section that follows provides a few examples to illustrate the identification of decisions and conditions. *Handling non short-circuit operators* focuses on the handling of computational Boolean operators, then *MCDC variants* expands on the notion of *independent influence* and on possible variations of the MCDC criterion definition.

Output-wise, the source annotations for the `=xcov` or `=html` formats are the same as for decision coverage, with condition specific cases marked with a `!` as well:

Annotation	Meaning
-	At least one statement associated with this line, none executed.
!	For a single statement line, decision partially covered or condition not covered on the line.
?	At least one statment or decision on the line with undetermined coverage state. (*)
+	All the statements, decisions and conditions on the line are covered.

(*) The Undetermined Coverage state (`?`) is only shown on the line in the absence of other known violations for that same line.

The `=report` outputs feature an extra MCDC section in the Coverage Violations segment, which holds:

- The condition specific diagnostics (`independent influence not demonstrated`), as well as
- Decision level diagnostics (such as `decision outcome True not covered` messages) for the Complex Boolean Expressions not directing a control-flow oriented statement and which we treat as decisions nevertheless.

There again, condition or decision related messages are only emitted when no more general diagnostic applies on the associated entity. Condition specific diagnostics, for example, are only produced in absence of enclosing statement or decision level violation.

See the *Example program and assessments* section of this chapter for a few illustrations of these points.

5.5.2 Example decisions and conditions

As described in the previous section, the set of expressions considered as *decisions* for MCDC is wider than for decision coverage analysis, which encompasses all the expressions used as conditionals in explicit control-flow construct, for example:

```
if Check(X) then -- 1 decision (if-stmt control)
```

```
while (x > 12) { // 1 decision (while loop control)
```

MCDC also includes the conditionals of Ada *if-expressions* or they C *conditional-expression* counterpart, as in:

```
Val := (if X > 0 then X else -X); -- 1 decision (if-expr control)
```

```
return (x > y) ? 1 : -1; // 1 decision (cond-expr control)
```

Then we have logical expressions combining operands with short-circuit operators in other contexts, such as:

```
X := A and then B; -- 1 decision (short-circuit and-then)
Check (C or else D); -- 1 decision (short-circuit or-else)
```

```
return (x < low || y > high); // 1 decision (short-circuit ||)
```

This can raise lead to various cases of multiple decisions within a single expression, for example:

```
if Check (Arg1 => A and then B, Arg2 => C or else D) then
-- 3 decisions here:
-- * A and then B (short circuit and-then)
-- * C or else D (short circuit or-else)
-- * Check (...)  (if-stmt control)
```

In C as in Ada, logical negation is allowed anywhere and influences neither decision boundaries nor the split into conditions:

```
if (x && !y) { // 1 decision (if-stmt control)
               // 2 conditions (short-circuit &&)
```

```
if not (A or else B) then -- 1 decision (if-stmt control)
                          -- 2 conditions (short-circuit or-else)
```

5.5.3 Handling non short-circuit operators

Short-circuit operators are key to instating a decision at all in some contexts, and to direct the split of decisions into conditions in all contexts. By default, non short-circuit binary operators are taken as regular computational devices:

```
if A or B then -- 1 decision (if-statement control)
               -- single condition (no short-circuit by default)

if A or else B then -- 1 decision (if-statement control)
                    -- 2 conditions (short-circuit or-else)

X := A and B;  -- no decision (no short-circuit by default)

X := A and then B;  -- 1 decision (short-circuit and-then)
                    -- 2 conditions (short-circuit and-then)
```

This computational view for non short-circuit operators in Ada is similar to the C model where bit-wise operations on non purely Boolean types, yet used as a Boolean result, are extremely common as in:

```
return (x & 0x3) && (y & 0x3);  // 1 decision (short-circuit &&)
                                // 2 conditions, each a & on int
```

It can also result in splits introducing the possibility of multiple decisions in arguments, as in:

```
if ((A and then not B) == (C or else (D and then E))) then
-- 3 decisions here:
```

(continues on next page)

(continued from previous page)

```
-- * The top-level expression as a whole (... == ...), if-stmt control,
--   single condition.
-- * (A and then not B), short-circuit and-then in the first operand of ==,
--   2 conditions.
-- * (C or else (D and then E)), short-circuit tree in the second operand,
--   3 conditions.
```

GNAT compilers offer two devices to mitigate possible issues caused by this behavior on non short-circuit Boolean operators for Ada:

- To make sure operators of this kind are not used, e.g. as a coding standard rule, a `No_Direct_Boolean_Operator` restriction pragma can be setup in the development environment to trigger compilation errors on such uses; Alternatively:
- The `Short_Circuit_And_Or` configuration pragma requests that the non-short circuit `and` and `or` operators on standard Booleans are compiled as as their short-circuit counterparts, then processed as such for coverage analysis.

There is no equivalent in C, where the allowed operand types are much more varied and where the restriction would make the language really much harder to use.

5.5.4 MCDC variants

Compared to Decision Coverage, achieving MCDC requires tests that demonstrate the independent influence of conditions in decisions. Several variants of the criterion exist.

Unique Cause MCDC is the original criterion described in the **DO-178C** reference guidelines, where independent influence of a specific condition must be demonstrated by a pair of tests where only that condition changes and the decision value toggles.

Consider the following table which exposes the 4 possible condition/decision vectors for the `A and then B` expression, where T stands for True, F stands for False, and the italics indicate that the condition evaluation is short-circuited:

#	A	B	A and then B
1	T	T	T
2	T	F	F
3	F	*T*	F
4	F	*F*	F

Each line in such a table is called an *evaluation vector*, and the pairs that demonstrate the independent effect of conditions are known as *independence pairs*.

Evaluations 1 + 3 constitute a Unique Cause independence pair for A, where A changes, B does not, and the expression value toggles. 1 + 2 constitutes a pair for B.

The closest criterion supported by GNATcoverage is a very minor variation where conditions that are not evaluated due to short-circuit semantics are allowed to differ as well in a pair. Indeed, their value change cannot possibly have influenced the decision toggle (since they are not even considered in the computation), so they can never invalidate the effect of another condition.

We call this variation *Unique Cause + Short-Circuit MCDC*, activated with `--level=stmt+uc_mcdc` on the command line. From the `A and then B` table just introduced, 4 + 1 becomes another valid independence pair for A, as *B* is not evaluated at all when *A* is False so the change on *B* is irrelevant in the decision switch.

`--level=stmt+mcdc` actually implements another variant, known as *Masking MCDC*, accepted as a sound alternative and offering improved support for coupled conditions.

Masking MCDC allows even further flexibility in the possible variations of conditions in an independence pair. Indeed, as soon as only short-circuit operators are involved, all the conditions that appear on the left of a given condition in the expression text are allowed to change without invalidating the said condition influence demonstration by a pair.

5.5.5 Example program and assessments

We reuse one of our previous examples to illustrate, with a simple functional unit to exercise:

```
function Between (X1, X2, V : Integer) return Boolean;
--  Whether V is between X1 and X2, inclusive and however they are ordered

function Between (X1, X2, V : Integer) return Boolean is
begin
   if X1 < X2 then
      return V >= X1 and then V <= X2;
   else
      return V >= X2 and then V <= X1;
   end if;
end Between;
```

First consider the following test driver, which exercises only a single case where X1 < V < X2:

```
procedure Test_X1VX2 is
begin
   Assert (Between (X1 => 2, X2 => 5, V => 3)); -- X1 < V < X2
end Test_X1VX2;
```

After instrumentation for MCDC and execution, we can then request, say, an `=xcov+` report to get a first set of results in the `ranges.adb.xcov` annotated source:

```
gnatcov coverage --level=stmt+mcdc -Pmytest.gpr --annotate=xcov+ test_x1vx2.srctrace

...
   8 .:    function Between (X1, X2, V : Integer) return Boolean is
   9 .:    begin
  10 !:       if X1 < X2 then
DECISION "X1 < X2" at 10:10 outcome FALSE never exercised
  11 !:          return V >= X1 and then V <= X2;
DECISION "V >= X1 a..." at 11:17 outcome FALSE never exercised
  12 .:       else
  13 -:          return V >= X2 and then V <= X1;
STATEMENT "return V ..." at 13:10 not executed
  14 .:       end if;
  15 .:    end Between;
```

This is all as expected from what the driver does, with a few points of note:

- The diagnostic on line 11 confirms that Complex Boolean Expression are treated as decisions even when not used to direct a conditional control-flow statement. The expression is indeed used here as a straight, unconditional `return` statement value;

- Only the decision level violations are emitted for lines 10 and 11. The independent influence of the conditions is not demonstrated but this is implicit from the decision partial coverage so is not notified;
- Similarly, only the statement level violation is emitted for line 13, even though there are decision and condition level violations as well.

Another aspect of interest is that we have partial decision coverage on two kinds of decisions (one control-flow decision controlling the *if*, and another one used a straight return value), and this distinction places the two `decision outcome FALSE never exercised` violations in distinct sections of the `=report` output:

```
2.1. STMT COVERAGE
------------------
ranges.adb:13:10: statement not executed

2.2. DECISION COVERAGE
----------------------
ranges.adb:10:10: decision outcome FALSE never exercised

2.3. MCDC COVERAGE
------------------
ranges.adb:11:17: decision outcome FALSE never exercised
```

Now running another test driver which exercises two cases where X1 < X2:

```
procedure Test_X1VX2V is
begin
   Assert (Between (X1 => 2, X2 => 5, V => 3)); -- X1 < V < X2
   Assert (not Between (X1 => 2, X2 => 5, V => 8)); -- X1 < X2 < V
end;
```

The first return expression is valued both ways so we get an example of *condition* specific diagnostic on line 11:

```
   8 .:    function Between (X1, X2, V : Integer) return Boolean is
   9 .:    begin
  10 !:       if X1 < X2 then
DECISION "X1 < X2" at 10:10 outcome FALSE never exercised
  11 !:          return V >= X1 and then V <= X2;
CONDITION "V >= X1" at 11:17 has no independent influence pair, MC/DC not achieved
  ...
```

Indeed, looking at an evaluation table for the first return decision:

#	A: V >= X1	B: V <= X2	A and then B	Case
1	T	T	T	X1 < V < X2
2	T	F	F	X1 < X2 < V
3	F	*T*	F	
4	F	*F*	F	

We observe that our driver exercises vectors 1 and 2 only, where:

- The two evaluations toggle the decision and the second condition only, so achieve decision coverage and demonstrate that condition's independent influence;
- The first condition (V >= X1) never varies so the independent influence of this condition isn't demonstrated.

As we mentioned in the discussion on MCDC variants, adding vector 3 would achieve MCDC for this decision. Just looking at the table, adding vector 4 instead would achieve MCDC as well since the second condition is short-circuited so its value change is not relevant. The condition expressions are such that running vector 4 is not possible, however, since we can't have V both < X1 (condition 1 False) and V > X2 (condition 2 False) at the same time when X1 < X2.

5.6 Assertion True Coverage (ATC) analysis (experimental)

Along with the previous coverage levels, **gnatcov** can provide different strictness levels of assertion coverage for source traces for Ada.

Assertion coverage warrants defining separate criteria to perform coverage analysis since contrary to all other regular decisions, those contained in assertions are expected to never be evaluated to False. Assertion decisions and conditions are defined the same way as for Decision and MCDC but, the coverage is computed differently.

If such coverage analysis is needed, it should always be activated along one of the previously described coverage levels, as an "addition". In the following sections on assertion coverage, their associated command line options will be written as `--level=...+<assertion coverage level>` where `...` is one of `stmt`, `stmt+decision`, `stmt+mcdc` and `stmt+uc_mcdc`.

Using assertion coverage levels allows to compute the coverage of the pragma statements `Assert`, `Assert_And_Cut`, `Assume`, `Check`, `Loop_Invariant`, and the aspects `Type_Invariant`, `Precondition` and `Postcondition`.

The limitations of this coverage level are detailed in the *gnatcov instrument limitations* section.

5.6.1 Core notions and Reporting (`--level=...+atc`)

gnatcov performs Assertion True Coverage assessments with the `--level=...+atc` command line option. The assessment determines the status of assertion true coverage obligations out of the tests execution, considering that:

- An assertion is *covered*, and the obligation *discharged*, as soon as the assertion's decision has been evaluated once to True.
- An assertion is *uncovered* otherwise.

In synthetic `=report` outputs, uncovered source assertions are listed as ATC Coverage violations in the report section dedicated to these.

In annotated source outputs, the coverage annotations convey the following indications:

Annotation	Meaning
-	At least one assertion on the line, none covered
!	More than one assertion on the line, some covered
?	At least one assertion on this line, some with undetermined coverage state (*)
+	At least one assertion on the line, all covered

(*) The Undetermined Coverage state (?) is only shown on the line in the absence of other known violations for that same line.

5.7 Assertion True Condition Coverage (ATCC) analysis (experimental)

5.7.1 Core notions and Reporting (`--level=...+atcc`)

The coverage status of an "ATCC" obligation is determined as follows:

- An assertion is said *fully covered*, or just *covered*, and the obligation discharged, as soon as all conditions have been evaluated to True or False at least once across all evaluations to True of the whole decision.
- An assertion is said *uncovered* when the decision was never evaluated to True, either because the enclosing assertion statement was not executed at all or when all the attempted evaluations were interrupted e.g. because of exceptions.
- An assertion is said *partially covered* when at least one of the conditions of the assertion's decision was never evaluated to either True or False in the context of an evaluation to True of the whole decision. In this case, the obligation is partially discharged.

The `=report` synthetic output lists the ATC and ATCC coverage violations in the `ATC` and `ATCC` coverage report section respectively. For the `=xcov` and `=html` annotated-source oriented formats, the single annotation produced on each source line combines the statement and ATC coverage indications. The following table summarizes the meaning of the possible annotations:

Annotation	Meaning
-	At least one statement on the line, none executed.
!	Unless multiple statements are involved, assertion partially covered on the line.
?	At least one statement or assertion on the line with undetermined coverage state. (*)
+	All the statements and assertions on the line are covered.

(*) The Undetermined Coverage state (`?`) is only shown on the line in the absence of other known violations for that same line.

When a trailing `+` is added to the format passed to `--annotate` (`=xcov+`), a precise description of the actual violations is available for each line in addition to the annotation. The `=html` provides it by default.

The limitations of this coverage level are detailed in the *gnatcov instrument limitations* section.

5.8 Function and Call Coverage (FUN_CALL) analysis (experimental)

gnatcov can also provide an analysis of the coverage of subprograms and calls for Ada, C, and C++ source traces.

If such coverage analysis is needed, it should always be activated along one of the non-assertion coverage levels previously described. In this section on function and call coverage, the associated command line option will be written as `--level=...+fun_call` where `...` is one of `stmt`, `stmt+decision`, `stmt+mcdc` and `stmt+uc_mcdc`. Assertion coverage can also be activated at the same time.

5.8.1 Core notions and Reporting (`--level=...+fun_call`)

gnatcov performs Function and Call Coverage assessments with the `--level=...+fun_call` command line option. The assessment determines the status of subprograms and call coverage obligations out of the tests execution, considering that:

- A subprogram is *covered*, and the obligation *discharged*, as soon as the execution as entered it at least once.
- A subprogram is *uncovered* otherwise.
- A call is *covered*, and the obligation *discharged*, if it has been executed at least once.
- A call is *uncovered* otherwise.

In synthetic `=report` outputs, uncovered source subprograms and calls are listed as FUN_CALL Coverage violations in the dedicated report section.

It is important to note that for an *uncovered* statement which happens to be a call statement, only a statement violation will be emitted in coverage reports.

The limitations of this coverage level are detailed in the *gnatcov instrument limitations* section.

5.9 Guarded Expression Coverage (GEXPR) analysis (experimental)

For the Ada language, **gnatcov** can provide an analysis on coverage of guarded expressions.

5.9.1 Core notions and Reporting (`--level=...+gexpr`)

gnatcov considers the following syntaxes to be *guarded expressions*:

- Then and Else dependent expressions of *if expressions*:

```
procedure Foo (A : Boolean) is
   Var : constant String :=
     (if A
      then "True expression"
      else "False expression");
begin
   null;
end Foo;
```

- Dependent expressions of *case expressions*:

```
type Animal is (Cat, Dog, Cow);

procedure Foo (A : Animal) is
   Var : constant String :=
     (case A
      when Cat    => "Expression Cat",
      when Dog    => "Expression Dog",
      when others => "Expression other");
begin
   null;
end Foo;
```

- Predicate expression of *quantified expressions*:

```
function Prime_In_Range (L, R : Natural) return Boolean is
begin
   --  Is_Prime (I) is the child expression that will be analyzed.

   return (for some I in (L .. R) => Is_Prime (I));
end Foo;
```

For each of these, we consider the expression to be *covered* and the obligation to be *discharged* if the execution flow evaluated it at least once.

The limitations of this coverage level are detailed in the *gnatcov instrument limitations* section.

5.10 Synthetic metrics

A number of output formats feature *synthetic metrics* (essentially, coverage percentages) for each source unit, and for the entire analysis when an index summary is also produced. We first introduce the metrics definition for the `=xcov` format then describe how they are presented in other outputs.

For the `=xcov` format, the synthetic metrics are summary numbers at the top the annotated source for each unit. For example:

```
.../metrics/main.adb:
50% of 4 lines covered
75% statement coverage (3 out of 4)
0% decision coverage (0 out of 1)

Coverage level: stmt+decision
 1 .: with Ada.Text_IO; use Ada.Text_IO;
 2 .: with Types; use Types;
 3 .:
 4 .: procedure main is
 5 +:    X : My_Int := 12 with Volatile;
 6 .: begin
 7 !:    if X < 2 then
 8 -:       Put_Line ("X LT 2");
 9 .:    else
10 +:       Put_Line ("X GE 2");
11 .:    end if;
12 .: end;
```

The metric displayed first is:

```
50% of 4 lines covered
```

This is *line count* based, where the count includes all the lines to which is attached at least one coverage obligation of the asserted level (here, `stmt+decision`). In the provided example, these are lines 5, 7, 8 and 10, for a total of 4 lines, where line 7 encompasses two obligations (one statement and one decision).

The metric conveys the total number of lines and the percentage of such lines for which all the obligations are fully covered, that is, those marked with a '+' annotation.

The two other metrics in this example are:

```
75% statement coverage (3 out of 4)
0% decision coverage (0 out of 1)
```

These are *coverage obligation count* based, one for each obligation kind involved in the assessed level (here, for `--level=stmt+decision`). As for lines, each metric displays the total number of obligations of the given kind and the ratio of such obligations which have been fully discharged. The `xcov` obligation metrics don't distinguish partially covered from uncovered items. This information is available from the `html` report.

For the `html` output, synthetic metrics for all the units are first displayed on the index page, together with metrics for the entire analysis and for sets of units grouped by GPR project.

Initially, the line count based metrics are displayed, as illustrated by Fig. 1. The main procedure source used in our `xcov` report example is the `main.adb` source here, single source encompassed by the `main.gpr` project:

Switching to coverage obligation count based metrics is simply achieved by clicking the button(s) at the top of the page labeled with obligation kind names ("Stmt", "Decision" or "Mcdc").

Each kind of obligation can be selected alone. Selecting multiple kinds is also allowed and just sums the individual counts. Fig. 2 illustrates the results we get with the `Stmt` and `Decision` kinds selected together for our previous example:

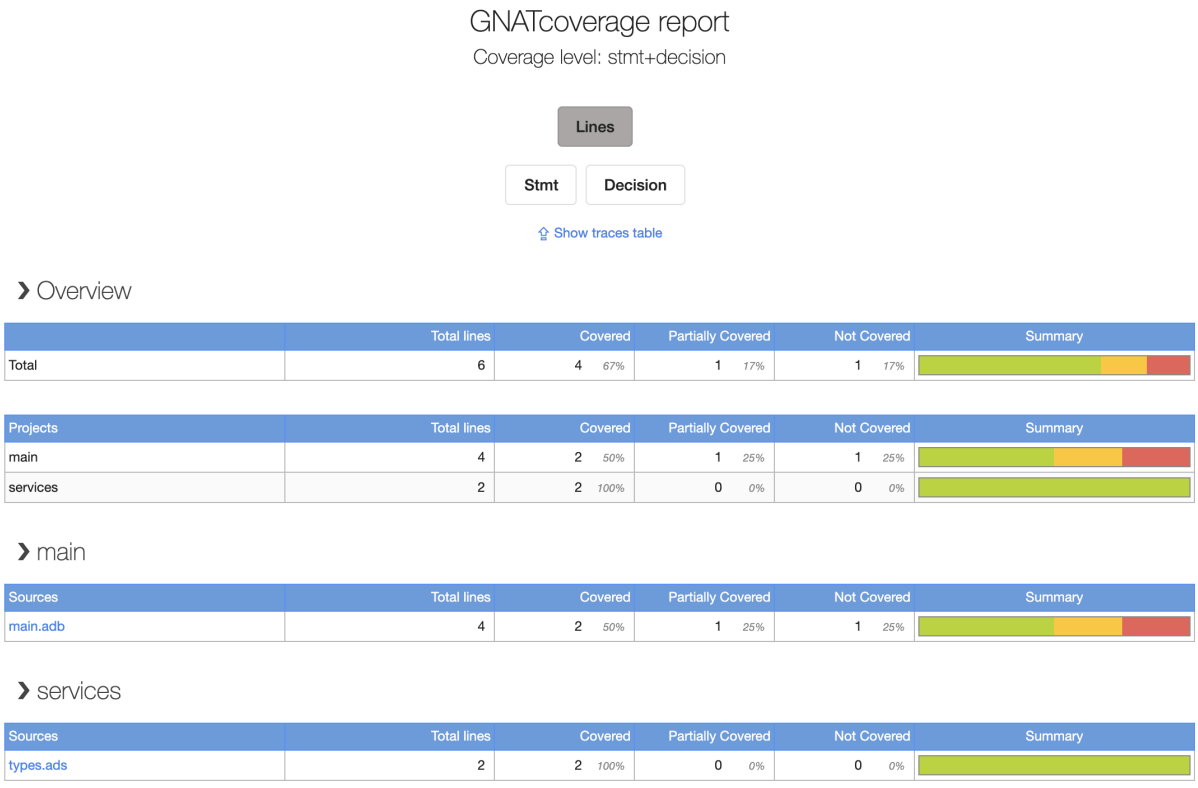


Fig. 1: HTML index with line count based synthetic metrics

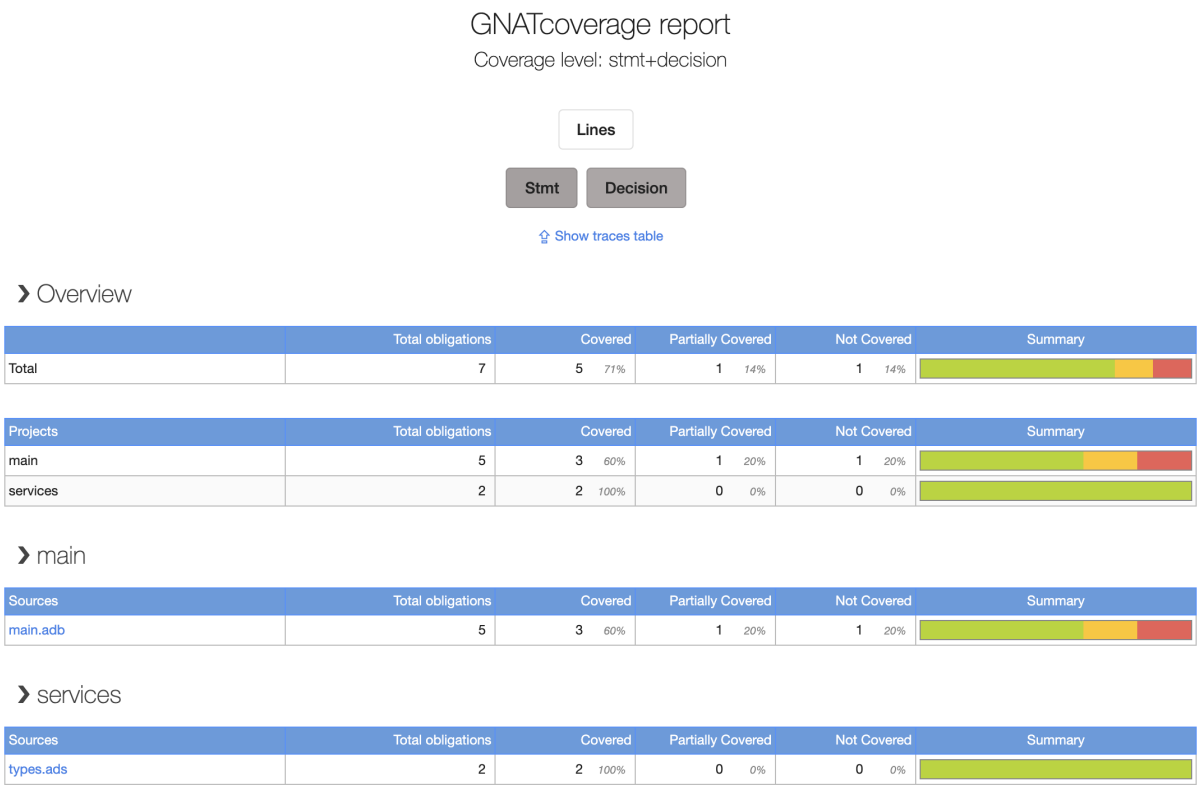


Fig. 2: HTML index with obligations count based synthetic metrics (stmt+decision)

5.11 Handling source relocation for annotated sources output formats

For all annotated sources output formats (`--annotate=xcov[+]|html`), **gnatcov** needs access to the sources of the *units of interest* to generate the output. The tool searches for sources in some default locations which are usually correct when the whole coverage analysis process is done on the same machine.

When coverage reports are produced on a different machine than the one where the coverage run was done, or when sources are moved in the interim, **gnatcov** will likely not be able to automatically find the source files. The tool offers two options to help in those situations:

- The option `--source-rebase=<old_prefix>=<new_prefix>`

  This option allows the specification of alternate path prefixes for source files.

  If the location of a source file, when built on a first machine, was `/work/build_1234/src/main.adb` but on the machine where the report is to be generated, the same file is located at `/some/path/project/src/main.adb`, then in order for **gnatcov** to find this file, the option `--source-rebase=/work/build_1234/=/some/path/project/` can be passed. Note that the prefixes apply to absolute paths.

  This option supports globbing expressions for the `old_prefix` part of the option, so in the example above, if the build number varies from run to run, the option passed can be `--source-rebase=/work/build_*=/some/path/project/`.

  This option also allows for response-files (`--source-rebase=@rebase_file`) in which each line must be of the form `old_prefix=new_prefix`.

  This option can be passed multiple times on the command line, and **gnatcov** will try each pair in the order in

which they were passed until the source file is found.

- The option `--source-search=directory_path`

  This option allows the specification of directories in which to search for missing source files.

  If the location of a source file, when built on a first machine, was `/work/build_1234/src/main.adb` but its location on the machine where the report is generated is `/some/path/project/src/main.adb` then passing `--source-search=/some/path/project/src` will enable **gnatcov** to find the missing source file. This option also accepts response files.

  This option can appear multiple times on the command line. Note that **gnatcov** will first try the `--source-rebase` prefix pairs, if any, and if the source file was not found, it will then look for the file in all the directories passed with `--source-search` in the order in which they appear on the command line.

These two options perform very similar functions and can sometimes be used interchangeably, however the second option is less selective and can lead to unexpected results in some circumstances. Since `--source-search` only specifies an additional directory in which to search for missing sources, it does not guarantee that if a file is found then it is the correct one, but only that the file found has the same basename than the searched file. For Ada projects this should not be an issue, since all files should have different names, but for C projects where there can be homonyms, this can be problematic:

If a project is built with the files `/work/build_1234/src1/foo.h` and `/work/build_1234/src2/foo.h` among its sources, and `--source-search=/some/path/src1 --source-search=/some/path/src2` is passed to **gnatcov** to add the two source directories to the list of locations where sources can be found, then the resulting location for both `foo.h` files will be `/some/path/src1/foo.h`. In this case, passing `--source-rebase=/work/build=/some/path/` removes the ambiguity.

`--source-rebase` is also more convenient when the project has multiple source directories, provided that their structure is the same when building/instrumenting the project and generating the coverage report. In the example above, it is necessary to pass `--source-search` twice to include all the source directories in the search path, but only one instance of `--source-rebase` is needed. For very large projects, since the number of required instances of `--source-search` is equal to the number of source directories, it is clearly more advantageous to use `--source-rebase` which only needs to be passed once.

5.12 Inlining & Ada generic units

In the vast majority of situations, inlining is just transparent to source coverage metrics: calls are treated as regular statements, and coverage of the inlined bodies is reported on the corresponding sources regardless of their actual inlining status. See the *Optimization and non-coverable items* section for a description of effects that might show up on rare occasions.

Generic units have a different status, where each instantiation produces a distinct, potentially unique instance of the generic code working on a different types, objects, with different helper subprograms or packages.

GNATcoverage offers two main kinds of coverage strategies for generic units: one where the generic source is considered as the entity of interest, to which all the instances contribute, and one where each instance is considered as a separate entity of interest.

5.12.1 Combined coverage on generics

Ada generic units are also uniformly treated as single source entities, with the coverage achieved by all the instances combined and reported against the generic source only, not for each individual instance.

Consider the following functional Ada generic unit for example. It provides a simple vector type abstraction on which two operations are available; `Inc` adds some amount to each element of a vector, and `Mult` multiplies each element by some amount. The exposed type is of fixed size, provided as a parameter:

```
generic                                 -- vops.ads
   Size : in Integer;
package Vops is
   type Vector_Type is array (1 .. Size) of Integer;

   procedure Inc (V : in out Vector_Type; Amount : Integer);
   procedure Mult (V : in out Vector_Type; Amount : Integer);
end;

package body Vops is                    -- vops.adb

   procedure Inc (V : in out Vector_Type; Amount : Integer) is
   begin
      for I in V'Range loop
         V(I) := V(I) + Amount;
      end loop;
   end;

   procedure Mult (V : in out Vector_Type; Amount : Integer) is
   begin
      for I in V'Range loop
         V(I) := V(I) * Amount;
      end loop;
   end;
end;
```

Now consider this test, checking operations on vectors of different sizes, from two instances of the `Vops` unit:

```
with Vops;                              -- v5.ads
package V5 is new Vops (Size => 5);

with Vops;                              -- v8.ads
package V8 is new Vops (Size => 8);

with V5, V8;                            -- test_5inc_8mult.adb
procedure Test_5inc_8mult is
   V5o : V5.Vector_Type := (others => 1);
   V8o : V8.Vector_Type := (others => 2);
begin
   V5.Inc (V5o, 3);
   V8.Mult (V8o, 2);
end;
```

Only the `Inc` subprogram is called through the V5 instance and only the `Mult` subprogram is called through the V8 instance. Both suprograms are nevertheless called overall, so the `Vops` package body is claimed fully covered by default:

```
gnatcov coverage -Pvops.gpr --level=stmt --annotate=xcov test_5inc_8mult.trace
...
100% of 4 lines covered
Coverage level: stmt
  1 .: package body Vops is
  2 .:
  3 .:    procedure Inc (V : in out Vector_Type; Amount : Integer) is
  4 .:    begin
  5 +:       for I in V'Range loop
  6 +:          V(I) := V(I) + Amount;
  7 .:       end loop;
  8 .:    end;
  9 .:
 10 .:    procedure Mult (V : in out Vector_Type; Amount : Integer) is
 11 .:    begin
 12 +:       for I in V'Range loop
 13 +:          V(I) := V(I) * Amount;
 14 .:       end loop;
 15 .:    end;
 16 .: end;
```

5.13 Handling Ada assertions and contracts

Ada 2012 introduced sophisticated *programming by contract* mechanisms which generalize the base `pragma Assert` construct originally offered by the language. All these facilities allow asserting various properties of the program state at specific points of its execution, for example as type invariants or as Pre/Post-conditions on subprograms.

Each property is expressed a Boolean expression expected to hold True, triggering an exception otherwise. When the current assertion policy activates a given assertion construct, the associated Boolean expression is treated by **gnatcov** as a decision for MCDC purposes.

As assertions are by construction designed never to evaluate False, reaching proper coverage for them is non-trivial, if not entirely meaningless, for post-conditions in particular.

The simple way out consists in disabling the relevant constructs in builds intended for coverage analysis, by setting the corresponding `Assertion_Policy` to `Disable` with GNAT Pro toolchains.

5.14 Processing of C macros

For source coverage purposes, Source Coverage Obligations for C are produced after the preprocessing of sources, with two consequences of note:

- Macro expansions leading to code with conditionals will trigger coverage violations, and multiple "calls" to the same macro just multiply these as they yield distinct expansions.
- The source locations output by **gnatcov** for coverage violations within macro expansions designate preprocessed tokens at the macro expansion site, typically on the line of the macro invocation but with column numbers unrelated to what is visible in the source on this line.

Consider this C code for example:

```
1  #define COND_INC(cond,x,y) \
2    do {                     \
3      if (cond)              \
4        (x)++;               \
5      else                   \
6        (y)++;               \
7    } while(0)
8
9  int main ()
10 {
11   volatile x = 0, y = 0;
12
13   COND_INC(x == 0, x, y);
14   COND_INC(x == 0, x, y);
15 }
```

The two macro invocations actually expand as:

```
13     do { if (x == 0) (x)++; else (y)++; } while(0);
14     do { if (x == 0) (x)++; else (y)++; } while(0);
```

The expanded version is the basis of SCO identification process, so we have one decision and two conditioned statements on line 13, likewise on line 14. Only one of each is exercised at execution time, and a `stmt+decision` analysis on this program yields:

```
2.1. STMT COVERAGE
------------------

t.c:13:32: statement not executed
t.c:14:20: statement not executed

2 violations.

2.2. DECISION COVERAGE
----------------------

t.c:13:12: decision outcome FALSE never exercised
t.c:14:12: decision outcome TRUE never exercised

2 violations.
```

We do see one statement and one decision coverage violation per invocation, different in the two cases since the `x == 0` test is True on the first call and False on the second one. We also observe column numbers unrelated to what the original source lines contain on line 13 and 14.

5.15 Optimization and non-coverable items

GNATcoverage essentially operates by relating execution traces to source entities of interest thanks to debug information mapping machine code addresses to source locations. With optimization enabled, there sometimes is no machine code attached to a given statement, for example when the statement is determined to be redundant or when the machine code for it can be factorized with the machine code for another statement. When the coverage status of a code-less statement cannot be be inferred from that of other statements around, GNATcoverage categorizes the statement as *non-coverable*.

By default, nothing is said about non-coverable statements in the `=report` outputs and the corresponding lines are marked with a '.' in annotated sources, as for any other line to which no machine code is attached. Below is an example source annotated for statement coverage, where absence of code for a couple of Ada statements was triggered by constant propagation and inlining. The local `Pos` function is called only once, with a constant argument such that only one alternative of the `if` statement is taken. With `-O1 -gnatn`, the compiler sees that the `else` part can never be entered and no code is emitted at all for this alternative:

```
 4 .: procedure Test_Pos1 is
 5 .:    function Pos (X : Integer) return Boolean;
 6 .:    pragma Inline (Pos);
 7 .:
 8 .:    function Pos (X : Integer) return Boolean is
 9 .:    begin
10 +:       if X > 0 then
11 +:          Put_Line ("X is positive");
12 +:          return True;
13 .:       else
14 .:          Put_Line ("X is not positive");
15 .:          return False;
16 .:       end if;
17 .:    end Pos;
18 .: begin
19 +:    Assert (Pos (1) = True);
20 .: end Test_Pos1;
```

A common similar case is that of debugging code inhibited on purpose for regular operation, for example with constructs like:

```
Debug_Mode : constant Boolean := False;    or     #define DEBUG_MODE 0
...                                               ...
if Debug_Mode then                                #if DEBUG_MODE
```

Another way to get this in Ada is with generic instantiations where constant parameters turn what appears to be conditional in the source into a constant value in some instances.

Back to our `Test_Pos1` example, no code is emitted for the test on line 10 either. **gnatcov** is however able to infer the `if` coverage status by looking at the status of statements controlled by the decision, and the Decision coverage report remains accurate:

```
   8 .:    function Pos (X : Integer) return Boolean is
   9 .:    begin
  10 !:       if X > 0 then
decision "X > 0" at 10:10 outcome FALSE never exercised
  11 +:          Put_Line ("X is positive");
  12 +:          return True;
```

gnatcov coverage features the `--non-coverable` command line option to expose the non-coverable statements if

needed. They are listed in an additional "NON COVERABLE ITEMS" section of the =report outputs and the corresponding lines are flagged with a '0' mark in annotated sources, as well as a specific color in the HTML formats. For our example, this yields:

```
10 !:       if X > 0 then
11 +:          Put_Line ("X is positive");
12 +:          return True;
13 .:       else
14 0:          Put_Line ("X is not positive");
15 0:          return False;
16 .:       end if;
```

CHAPTER

SIX

RUST COVERAGE ANALYSIS WITH GNATCOV COVERAGE

Source coverage analysis for the Rust language is an unstable feature of **gnatcov**. It relies on the internal coverage instrumentation of the rustc compiler, which is itself leaning on the code coverage framework of the LLVM project.

Note: This feature is made to be compatible with the **GNATPro For Rust** tool suite, compatibility with other rust toolchains is not guaranteed.

6.1 Rust coverage workflow

The workflow consists in 3 steps. First, build the program while instrumenting it for coverage analysis. The instrumentation is delegated to `rustc`. Then, run the program, which produces trace files. Finally, use **gnatcov coverage** to generate a coverage report.

An example project demonstrating Rust instrumentation can be found at `share/examples/gnatcoverage/doc/rust_cov`.

6.1.1 Instrument and build the project

Unlike source coverage analysis for Ada, C and C++ which rely on **gnatcov** to instrument the source code *before* compiling it, the Rust workflow delegates instrumentation to the `rustc` compiler.

To instrument the code, pass the `-Cinstrument-coverage` argument to the compiler. If instrumenting for the `stmt+decision` or `stmt+mcdc` coverage levels, pass the `-Ccoverage-options=mcdc` additionally.

Decision coverage as defined by **gnatcov** does not exist in the rust compiler, thus **gnatcov** will infer the decision coverage from MC/DC coverage.

The example showcases using the `Cargo` build system to instrument a project for `stmt+mcdc` analysis analysis.

```
RUSTFLAGS="-Cinstrument-coverage -Ccoverage-options=mcdc" cargo build
```

6.1.2 Run the executed binary

Once again, the example showcases running the compiled program through `Cargo`. At the end of execution, a trace file ending with `.profraw` is created.

The name of the file can be customized with the `LLVM_PROFILE_FILE` environment variable. See LLVM_PROFILE_FILE format.

```
LLVM_PROFILE_FILE="trace-%p.profraw" cargo run -- ARGS...
```

The program may be run several times with different inputs to improve final coverage.

6.1.3 Produce a coverage report

To produce the coverage analysis from the obtained `.profraw` files, proceed as follow:

```
gnatcov coverage --level=stmt+mcdc -a xcov --exec EXEC_BIN PROFRAW_FILES...
```

The important part is that the program binary needs to be passed with the `--exec` argument. In a `Cargo` project, the binary usually lives under `target/debug` or `target/release`.

Note that aside of producing a report, the rust coverage report can be saved as a *checkpoint* and reused for consolidation.

6.2 Rust coverage limitations

This feature being relatively new, there are known limitations that we shall mention hereafter.

6.2.1 MC/DC false-negatives related to instrumentation level

While the **`gnatcov instrument`** workflow controls and tracks the instrumentation level of artifacts, making it possible to raise an error when trying to perform MC/DC analysis on an artifact that was not instrumented for MC/DC, the Rust workflow does not have such a safety feature yet.

It means that if the project is instrumented without MC/DC support, and the analysis is done with `--level=stmt+mcdc`, no MC/DC violation will be raised, even though there could have been some if the program had been instrumented adequately.

6.2.2 Approximative source location ranges

Currently, you may witness that some SCOs have a source location range that can overlap or merge with its neighbors. This is due to bugs in the compiler's instrumentation logic and will be fixed in coming releases.

6.2.3 Unsupported constructs

Some constructs are not yet supported:

- MC/DC instrumentation of pattern matching. Pattern matching can be viewed as a decision with multiple conditions. However, it can lead to more than just 2 outcomes.
- Rust `macros by example` and procedural macros are not yet instrumented.

CHAPTER

SEVEN

COVERAGE CONSOLIDATION

Coverage consolidation designates the general facility allowing the computation of the overall coverage achieved by a set of executions. When different executions get through common *units of interest*, consolidation computes the combined coverage of all the executions on these units.

A typical case where consolidation is useful is when some part of an application depends on external inputs and several executions are required to exercise different scenarios in the application program. The execution traces to consolidate are obtained from the same executable in this case. Another common situation is when execution of different executables is needed to achieve the required coverage for a software, either because distinct software modules are tested independently (e.g. the different units of a library), or because different aspects of the behavior of modules are tested separately (e.g. the different subprograms of a library unit or different scenarios of a given subprogram).

The production of consolidated coverage reports can proceed either directly from a set of traces produced by the executions, or from a set of pre-computed partial results latched in what we refer to as *coverage checkpoints*, which offer a lot of advanced capabilities. The following sections illustrate various combinations of the possibilities.

7.1 Consolidation from trace files

Computing a consolidated coverage result from multiple executions is simply achieved by providing the set of trace files resulting for these executions as inputs to a single **gnatcov coverage** command.

The set of traces involved in a computation, with some details about each trace (file name, timestamp, tag), is visible in the index page of HTML reports and in the *Assessment Context* section of `=report` outputs, where the command line is quoted.

The following subsections provide examples of possible use cases of this facility.

7.1.1 Single unit tested by different programs

The sources for the following example can be found under the *share/examples/gnatcoverage/doc/cons_traces* directory of the GNAT DAS distribution.

We will first consider achieving statement coverage of the following example Ada unit, which implements part of a robot controller able to send actuator commands depending on what a front sensor perceives is ahead of the robot:

```
package Commands is

   type Command is (Step, Hold);
   type Perceived is (Ground, Rock, Pit);

   function Safe (Cmd : Command; Front : Perceived) return Boolean;
```

(continues on next page)

(continued from previous page)

```
   --  Whether executing CMD is safe with FRONT perceived ahead

   N_Safe, N_Unsafe : Integer := 0;
   --  Count the number of safe/unsafe cases we have evaluated
end Commands;

package body Commands is

   procedure Stat (Safe : Boolean) is
   begin
      if Safe then
         N_Safe := N_Safe + 1;
      else
         N_Unsafe := N_Unsafe + 1;
      end if;
   end Stat;

   function Safe (Cmd : Command; Front : Perceived) return Boolean is

      --  Standing straight is always safe, and any other action is
      --  safe as soon as there is just solid ground ahead:
      Result : constant Boolean := Cmd = Hold or else Front = Ground;
   begin
      Stat (Result);
      return Result;
   end Safe;
end Commands;
```

We exercise the Commands body by combining two sorts of tests. The first one exercises safe commands only:

```
procedure Test_Cmd_Safe is
begin
   --  Remaining still is always safe, as is stepping with room ahead:
   Assert (Safe (Cmd => Hold, Front => Rock));
   Assert (Safe (Cmd => Hold, Front => Pit));
   Assert (Safe (Cmd => Step, Front => Ground));
end Test_Cmd_Safe;
```

The second one exercises unsafe commands only:

```
procedure Test_Cmd_Unsafe is
begin
   --  Stepping forward without room ahead is always unsafe
   Assert (not Safe (Cmd => Step, Front => Rock));
   Assert (not Safe (Cmd => Step, Front => Pit));
end Test_Cmd_Unsafe;
```

If we were to produce a coverage report for each individual test, we would observe partial coverage of the Commands body regardless of the kind of trace used.

Indeed, an =xcov report from the first test only would typically display:

```
 6 .:    procedure Stat (Safe : Boolean) is
```

(continues on next page)

(continued from previous page)

```
 7 .:    begin
 8 +:       if Safe then
 9 +:          N_Safe := N_Safe + 1;
10 .:       else
11 -:          N_Unsafe := N_Unsafe + 1;
12 .:       end if;
13 .:    end Stat;
```

Where, in accordance with the testcase strategy, everything is statement covered except the code specific to *unsafe* situations, here the counter update on line 11.

Conversely, an `=xcov` report from the second test only would yield the symmetric results:

```
 6 .:    procedure Stat (Safe : Boolean) is
 7 .:    begin
 8 +:       if Safe then
 9 -:          N_Safe := N_Safe + 1;
10 .:       else
11 +:          N_Unsafe := N_Unsafe + 1;
12 .:       end if;
13 .:    end Stat;
```

Then a report obtained by combining traces will show achievement of complete statement coverage like so:

```
 6 .:    procedure Stat (Safe : Boolean) is
 7 .:    begin
 8 +:       if Safe then
 9 +:          N_Safe := N_Safe + 1;
10 .:       else
11 +:          N_Unsafe := N_Unsafe + 1;
12 .:       end if;
13 .:    end Stat;
```

Assuming you have obtained one trace for the execution of each test, the command to produce the combined report would be something like:

```
gnatcov coverage --level=stmt <units-of-interest> --annotate=xcov
  test_cmd_safe.srctrace test_cmd_unsafe.srctrace
```

The means to obtain the traces and the `<units-of-interest>` switches would depend on how the functional and testing code has been organized.

One possibility would be to have all the code hosted by a single project where we could explicitly state which units are of interest, for example:

```
project Example1 is
  for Source_Dirs use ("code", "tests");

  -- Test drivers
  for Main use ("test_cmd_safe.adb", "test_cmd_unsafe.adb");

  -- State subset of units of interest to coverage analysis
  package Coverage is
    for Units use ("commands");
```

(continues on next page)

(continued from previous page)

```
  end Coverage;
end Example1;
```

Another possibility would be to have a separate project for each family of units (*code* or *tests*), possibly a library project for the *code* part, or whatever else suits your build and testing environment best.

In this example, consolidation involved different programs with partial code overlap, as depicted on the following representation:

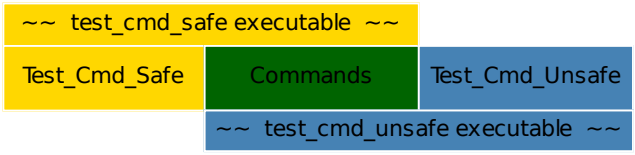


Fig. 1: Overlapping executables

Consolidation actually doesn't *require* overlapping: users might well, for example, consolidate results from different programs testing entirely disjoint sets of units. A typical situation where this would happen is when testing independent units of a library, as illustrated by the following example.

7.1.2 Disjoint units tested by different programs

Let us consider an example library composed of the following two Ada procedures, implemented in separate source files `inc.adb` and `mult.adb`:

```
procedure Inc (X : in out Integer; Amount : Integer) is   -- inc.adb
begin
   X := X + Amount;
end;

procedure Mult (X : in out Integer; Amount : Integer) is  -- mult.adb
begin
   X := X * Amount;
end;
```

We write two different programs to test the code from inc.adb on the one hand and the code from mult.adb on the other hand:

```
with Inc, Assert;      -- test_inc.adb
procedure Test_Inc is
   X : Integer := 0;
begin
   Inc (X, 1);
   Assert (X = 1);
end;

with Mult, Assert;     -- test_mult.adb
procedure Test_Mult is
   X : Integer := 2;
begin
```

(continues on next page)

(continued from previous page)

```
   Mult (X, 2);
   Assert (X = 4);
end;
```

Here as well, assuming you have obtained one trace for the execution of each test, assessing the library statement coverage achieved by `test_inc` alone, as a violations report, would go as:

```
gnatcov coverage --level=stmt --annotate=report
   <units-of-interest> test_inc.trace
```

There is no reference to the `mult` unit at all in the test and all the associated statements are marked uncovered in this case, this would yield:

```
2.1. STMT COVERAGE
------------------

mult.adb:3:4: statement not executed

1 violation.
```

Proper coverage of the library units is achieved by the two unit tests, which we can see by requesting the consolidated coverage achieved by the two executions:

```
gnatcov coverage --level=stmt --annotate=report
   <units-of-interest> test_inc.trace test_mult.trace

 ...
 2.1. STMT COVERAGE
 ------------------

 No violation.
```

7.1.3 Multiple executions of a single program

Consider the example C program below, offering a simple command line interface to perform very basic math operations. This is split in two main source files: `process.c` doing the computation and displaying the result, and `main.c` for the main entry point and basic usage control:

```
#include <stdio.h>        /* main.c */
#include <assert.h>
#include "process.h"

void usage ()
{
  printf ("calc <int1> <int2> <op>, print result of <int1> <op> <int2>\n");
}

int main (int argc, const char * argv[])
{
  if (argc != 4)
    {
```

(continues on next page)

(continued from previous page)

```
      usage ();
      exit (1);
    }

  process (argv);
  return 0;
}
```

```
#include <stdio.h>        /* process.c */
#include <assert.h>
#include "process.h"

void process (const char * argv[])
{
  int x = atoi (argv[1]), y = atoi (argv[2]);
  char opcode = argv[3][0];

  int result;

  switch (opcode)
    {
    case '*':
      result = x * y;
      break;
    case '+':
      result = x + y;
      break;
    default:
      printf ("unsupported opcode %c\n", opcode);
      return;
    }

  printf ("%d %c %d = %d\n", x, opcode, y, result);
}
```

```
#ifndef __PROCESS_H__     /* process.h */
#define __PROCESS_H__
extern void process (const char * argv[]);
#endif
```

Assuming an instrumented version of the program was built, here is a sequence of executions for various use cases, producing source traces on a native system and controlling the trace name by way of our dedicated environment variable:

```
GNATCOV_TRACE_FILE=mult.srctrace ./calc 6 5 '*'
GNATCOV_TRACE_FILE=plus.srctrace ./calc 2 3 '+'
GNATCOV_TRACE_FILE=div.srctrace  ./calc 2 3 '/'
GNATCOV_TRACE_FILE=misuse.srctrace ./calc
```

Now we can use **gnatcov coverage** to assess the coverage achieved by arbitrary combinations of the executions, just by passing the corresponding traces. For example, combining the two executions exercising the * and + computations for statement coverage can be achieved with:

```
gnatcov coverage --scos=main.c.gli --scos=process.c.gli \
   --annotate=xcov --level=stmt mult.srctrace plus.srctrace
```

And this yields reports in `main.c.xcov` and `process.c.xcov` like:

```
 ...
 5 .: void usage ()
 6 .: {
 7 -:   printf ("calc <i1> <i2> <op>, print result of <i1> <op> <i2>\n");
 8 .: }
 9 .:
10 .: int main (int argc, const char * argv[])
11 .: {
12 +:   if (argc != 4)
13 .:     {
14 -:       usage ();
15 -:       exit (1);
16 .:     }
17 .:
18 +:   process (argv);
19 +:   return 0;
20 .: }

 ...
 5 .: void process (const char * argv[])
 6 .: {
 7 +:   int x = atoi (argv[1]), y = atoi (argv[2]);
 8 +:   char opcode = argv[3][0];
 9 .:
10 +:   int result;
11 .:
12 +:   switch (opcode)
13 .:     {
14 .:     case '*':
15 +:       result = x * y;
16 +:       break;
17 .:     case '+':
18 +:       result = x + y;
19 +:       break;
20 .:     default:
21 -:       printf ("unsupported opcode %c\n", opcode);
22 -:       return;
23 .:     }
24 .:
25 +:   printf ("%d %c %d = %d\n", x, opcode, y, result);
26 .: }
```

We observe a reported absence of coverage for statements corresponding to the treatment of two kinds of usage error: wrong number of command line arguments, visible on lines 7, 14, and 15 of `main.c`, and attempt to compute an unsupported operation, visible on lines 21 and 22 of `process.c`. These two scenarios, exercised through `div.srctrace` and `misuse.srctrace` were indeed not included in the consolidation scope.

7.2 Consolidation from coverage checkpoints

A *coverage checkpoint* is a file containing a condensed version of a coverage assessment computed by **gnatcov coverage**, which the tool can reload and combine with other assessments very efficiently. This may be viewed as an internal representation of a coverage report which `--annotate` would output in a user readable form.

As reports, checkpoints are initially produced from traces, thanks to a `--save-checkpoint` option to **gnatcov coverage**. The switch expects the name of the checkpoint file to create as an argument, as in:

```
gnatcov coverage --level=<> <units> <traces> --save-checkpoint=<filename>
```

`--checkpoint` options then allow consolidating sets of checkpoints together to produce a user level report and/or another checkpoint. The example command below shows how to reuse the previous checkpoint to produce a user readable report:

```
gnatcov coverage --level=<> --checkpoint=<filename> --annotate=<>
```

The second `--level` option may differ from the first, as long as it does not request a stricter criterion. Assessing `stmt` or `stmt+decision` coverage from checkpoints containing `stmt+mcdc` data is allowed, not the other way around.

In user readable reports, the set of traces eventually contributing to the coverage result is available from the index page for HTML formats and in the `Assessment Context` header section of `=report` outputs. In the latter case, if the contribution comes from an intermediate checkpoint, the command which produced the checkpoint is displayed. For example, a sequence of commands such as:

```
obj/pgm1  # producing trace1
obj/pgm2  # producing trace2

gnatcov coverage --level=<> trace1 <units> --save-checkpoint=t1.ckpt
gnatcov coverage --level=<> trace2 <units> --checkpoint=t1.ckpt --annotate=report
```

Would produce a report with this kind of information in the header:

```
Trace files:

trace2
kind     : source
program  : obj/pgm2
date     : 2020-09-22 16:32:52
tag      :

trace1
kind     : source
program  : obj/pgm1
date     : 2020-09-22 16:32:47
tag      :
processed: gnatcov coverage <...> --save-checkpoint=t1.ckpt @ 2020-09-22 16:35:06.61
```

An important point of note: the specification of units of interest applies to *traces* only, never to input checkpoints, regardless of the checkpoints use.

The `<units>` in the second command of our previous example serve to scope the coverage achieved by the `trace2` argument and has no influence, for the output report, on the units held by `t1.ckpt`.

Suppose we had used different sets of units for the first and second commands, say:

```
gnatcov coverage --level=<> trace1 <a b c> --save-checkpoint=t1.ckpt
gnatcov coverage --level=<> trace2 <b c> --checkpoint=t1.ckpt --annotate=report
```

The output report produced by the second command would convey the coverage achieved for unit "a" out of `trace1` (as held by `t1.ckpt`), together with the coverage for units "b" and "c" out of both `trace1` (as held by t1.ckpt) and `trace2` (direct input). The possible coverage for unit "a" by `trace2` is filtered out (not in the set of units of interest), but the coverage for that unit held by the checkpoint input is included in the report.

We will now describe a few example situations of possible checkpoint uses, then discuss compatibility issues regarding consolidation.

7.2.1 Efficient storage of test results

Trace files can be large and often need to be kept together with the executable files that were used to produce the traces, for traceability reasons for example, or for specific processing operations in some circumstances. Preserving collections of traces and executables to allow consolidation can be problematic, space-wise.

In some situations (depending on the overall size of programs, number of units of interest, assessed coverage criteria, number and structure of decisions, ...) a checkpoint obtained from a trace can be a lot smaller than the trace + executable size combination.

When that is the case for most executions in a testing campaign, a possibility to improve storage efficiency consists in producing a checkpoint for each trace and preserve just that to consolidate afterwards, as in:

```
gnatcov coverage --level=<> <units> <trace1> --save-checkpoint=ckpt1
(possibly, remove <trace1> as well as the corresponding executable)
...
gnatcov coverage --level=<> <units> <traceN> --save-checkpoint=ckptN
```

Then consolidation can be achieved with:

```
gnatcov coverage --level=<> --annotate=<> --checkpoint=ckpt1 ... --checkpoint=ckptN
```

or, using a response file with the list of checkpoint names:

```
gnatcov coverage --level=<> --annotate=<> --checkpoint=@<ckpts.list>
```

As checkpoints contain high level coverage information, they are a lot more efficient to aggregate, which is all the more beneficial if that processing is performed repeatedly.

For situations where individual checkpoints are more often larger than their respective trace and executable, a checkpoint can still be of interest to store consolidated results for a group of tests, after all the traces are produced:

```
gnatcov coverage --level=<> <units> @<traces.list> --save-checkpoint=<filename>
```

Another possibility consists in accumulating results in a common checkpoint as each test gets processed. This is referred to as *Incremental coverage analysis*, described in the following section:

7.2.2 Incremental coverage analysis

Incremental Coverage Analysis consists in constructing a consolidated coverage result incrementally, using a single checkpoint file to accumulate new info about discharged coverage obligations as tests execute in sequence.

When providing execution traces as *the only* input to **gnatcov coverage**, coverage analysis starts with an empty coverage state where none of the coverage obligations in units of interest are discharged; that is, the units are all considered entirely not covered. Providing a checkpoint on input to **gnatcov coverage** simply instructs the tool to start from the previously computed coverage state stored in the given checkpoint. When traces are also provided, the tool consolidates the coverage achieved by the traces with that of the initial state and the result can be output as a report and/or saved in a checkpoint again.

Reusing an input checkpoint file as the output is allowed, and the computation of a cumulative coverage result by an entire testsuite is then possible with a sequence of commands such as:

```
# Process test1.trace, saving the resulting coverage state in a newly
# created checkpoint:
gnatcov coverage --level=<> -P<project> test1.srctrace \
                 --save-checkpoint=testsuite.ckpt

# Process subsequent test traces test2.srctrace .. testN.srctrace, each time
# starting with the coverage state reached at the previous iteration,
# and saving the resulting coverage state in the same checkpoint file
# (overwriting it):
gnatcov coverage --level=<> -P<project> test2.srctrace \
                 --checkpoint=testsuite.ckpt --save-checkpoint=testsuite.ckpt
...
gnatcov coverage --level=<> -P<project> testN.srctrace \
                 --checkpoint=testsuite.ckpt --save-checkpoint=testsuite.ckpt

# Now produce a report from the cumulated results:
gnatcov coverage --level=<> --checkpoint=testsuite.ckpt --annotate=<>
```

The big advantage of this approach is that it stores everything in a single checkpoint, which will occupy much less space than when using a checkpoint per trace and might then also be more efficient than preserving traces and executables.

This is very useful for situations where a given set of units is exercised by such a large testsuite that it is impractical to preserve the executables and execution traces, or individual checkpoints for all tests.

The individual coverage results are lost, however, and the (heavier) trace processing steps cannot be done in parallel since they all read and write a common checkpoint file which **gnatcov** does not lock.

7.2.3 Handle incidental coverage effects

In this scenario, we are assuming that two units A and B are being tested, that contain calls to each other. Each unit has its own testsuite, based on its specific requirements: testsuite A covers the requirements for unit A, and testsuite B covers the requirements for unit B. Running the two testsuites produces two sets of trace files, one set for testsuite A and one set for testsuite B.

Now suppose that you want to assess the global coverage for a system comprising both unit A and unit B. If the two sets of trace files are consolidated using a single execution of **gnatcov coverage** as in:

```
gnatcov coverage --level=stmt --units=A --units=B --annotate=report \
                 testA1.srctrace ... testAN.srctrace \
                 testB1.srctrace ... testBN.srctrace
```

then calls to B made by A while running testsuite A will contribute to discharging coverage obligations for unit B, and the other way round. This "incidental coverage" may be undesirable, as testsuite A is meant to exercise the requirements of unit A only (not unit B) and so should not contribute to the coverage of unit B.

This can be resolved using checkpointed coverage state, because each separate run of **gnatcov coverage** can consider a different set of units of interest – traces processed in each run will only contribute to the coverage of the units of interest for that run.

A consolidated coverage report can thus be constructed using a two pass analysis:

```
# Discharge the coverage obligations for unit A (--units=A) using
# trace files from testsuite A.
gnatcov coverage --level=stmt --units=A \
                 testA1.srctrace ... testAN.srctrace \
                 --save-checkpoint=testsuiteA.ckpt

# Discharge the coverage obligations for unit B (--units=B) using
# trace files from testsuite B, consolidate with previous results from
# testsuite A (--checkpoint), and produce a report (--annotate).
gnatcov coverage --level=stmt --units=B \
                 testB1.srctrace ... testBN.srctrace --checkpoint=testsuiteA.ckpt \
                 --annotate=report
```

In a consolidated report produced following this procedure, each set of trace files contributes only to the coverage of the units of interest specified for the execution of **gnatcov coverage** in which it is processed.

7.2.4 Compatibility considerations

Allowed coverage criteria combinations

A bit like source traces which hold results for some pre-stated strictest possible criterion (at instrumentation time), checkpoints hold results computed for a given coverage criterion, provided to the **gnatcov coverage** command used to produce the checkpoint. Computing results (e.g. a report) from such checkpoints may not request a criterion stricter than the least strict of the criteria held by the checkpoints. For instance, from a set of checkpoints established for *stmt+decision* and *stmt+mcdc*, one may request the production of a report for at most *stmt+decision*. Requesting *stmt* alone would be fine as well, and *stmt+mcdc* would be rejected because one of the checkpoints doesn't contain precise enough information.

Checkpoint format versions

The format of checkpoint files sometimes needs to evolve to support new functionality and each format is identified by a version number stored within the checkpoints. Maintaining degraded modes for old formats in a given version of GNATcoverage proved very intricate and error prone so a GNATcoverage designed for checkpoint format *N* now just rejects attempts at processing checkpoints of a different version. Not every new release of GNATcoverage incurs a change of format though, and we hope that incompatibilities will only rarely turn out annoying in practice.

7.3 Consolidating Coverage for Different Project Build Configurations

Some projects need to provide alternative implementations depending on the context. For instance, when using a GPR file, an external variable may configure a target-specific implementation of an API at build-time (you can learn more about external variables in the GPR Tools User's Guide, section 2.9.6.3 The function External).

Here is a simple example: the `Logging` package a means to emit errors or warnings. A single source file (`logging.ads`) defines the API to emit these messages, but two source files (`logging__text_io.adb` and `logging__gnat_io.adb`) implement this API: one relies on the `Ada.Text_IO` runtime unit, and the other on the `GNAT.IO` unit.

```
-- example.gpr
project Example is
   type Logging_Backend_Type is ("text_io", "gnat_io");
   Logging_Backend : Logging_Backend_Type := external ("LOGGING_BACKEND");

   for Main use ("main.adb");
   for Object_Dir use "obj-" & Logging_Backend;

   package Naming is
      for Body ("logging") use "logging__" & Logging_Backend & ".adb";
   end Naming;
end Example;

-- logging.ads
package Logging is
   procedure Warn (Message : String);
   procedure Error (Message : String);
end Logging;

-- logging__gnat_io.adb
with GNAT.IO; use GNAT.IO;
package body Logging is

   procedure Write (Prefix, Message : String) is
   begin
      Put (Prefix);
      Put (Message);
      New_Line;
   end Write;

   procedure Warn (Message : String) is
   begin
      Write ("warning: ", Message);
   end Warn;

   procedure Error (Message : String) is
   begin
      Write ("error: ", Message);
   end Error;
end Logging;

-- logging__text_io.adb
with Ada.Text_IO; use Ada.Text_IO;
```

(continues on next page)

(continued from previous page)

```
package body Logging is

   procedure Write (Prefix, Message : String) is
   begin
      Put (Prefix);
      Put_Line (Message);
   end Write;

   procedure Warn (Message : String) is
   begin
      Write ("warning: ", Message);
   end Warn;

   procedure Error (Message : String) is
   begin
      Write ("error: ", Message);
   end Error;
end Logging;
with Logging;

-- main.adb
with Logging;
procedure Main is
begin
   Logging.Warn ("message");
end Main;
```

gnatcov allows to compute code coverage for both implementations. They must be instrumented separately so that all source files have a chance of being instrumented:

```
gnatcov instrument -Pexample --level=stmt -XLOGGING_BACKEND=text_io
gnatcov instrument -Pexample --level=stmt -XLOGGING_BACKEND=gnat_io
```

Building and running the programs in order to generated source traces is next:

```
gprbuild -Pexample -XLOGGING_BACKEND=text_io \
   --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts
gprbuild -Pexample -XLOGGING_BACKEND=gnat_io \
   --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts

GNATCOV_TRACE_FILE=main_text_io.srctrace obj-text_io/main
GNATCOV_TRACE_FILE=main_gnat_io.srctrace obj-gnat_io/main
```

As described in *Specifying Units Of Interest*, coverage obligations for all the source files that need to appear in the coverage report must be conveyed to `gnatcov coverage` eventually. Since a given set of GPR file loading options (`-P`, `-X`) deals with one implementation at a time, special care must be taken when generating the coverage report. The two methods of consolidation can be used here:

1. Manually pass the relevant SID files (`--sid` switch) to use all source traces at once (i.e. *Consolidation from trace files*).
2. Keep using project loading options to load source traces separately and produce checkpoint files (i.e. *Consolidation from coverage checkpoints*).

In the following sections, we will use the above example to demonstrate consolidation for two variant programs running

on the same host, but you can expect the same kind of coverage consolidation even if traces are obtained from execution on different targets. For instance, for a project that targets a bareboard ARM system, with a variant running on a native host for testing purposes:

```
# Instrument and run the program to run on the ARM system
gnatcov instrument --level=stmt \
  -P example.gpr --target=arm-eabi --RTS=light-stm32f4
gprbuild --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts \
  -P example.gpr --target=arm-eabi --RTS=light-stm32f4
# [execute on target and produce onboard.srctrace]

# Instrument and run the program to run on the host system
gnatcov instrument --level=stmt \
  -P example.gpr
gprbuild --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts \
  -P example.gpr
GNATCOV_TRACE_FILE=native.srctrace obj/example

# It is then possible to produce a coverage report using either of the two
# methods presented below from onboard.srctrace and native.srctrace.
```

7.3.1 Use SID files to use all source traces at once

In order to make **gnatcov coverage** aware of all source files referenced in source traces, pass all the relevant SID files (*Providing coverage obligation files (--sid)*):

```
gnatcov coverage \
  --sid obj-text_io/logging__text_io.sid \
  --sid obj-gnat_io/logging__gnat_io.sid \
  main*.srctrace \
  --level=stmt --annotate=xcov --output-dir=xcov
```

This yields the expected coverage report, displaying the coverage for the two alternative implementations of `Logging`:

```
cat xcov/logging.ads.xcov
[...]/logging.ads:
no code
no code
Coverage level: stmt
   1 .:    package Logging is
   2 .:       procedure Warn (Message : String);
   3 .:       procedure Error (Message : String);
   4 .:    end Logging;

cat xcov/logging__text_io.adb.xcov
[...]/logging__text_io.adb:
75% of 4 lines covered
75% statement coverage (3 out of 4)

Coverage level: stmt
   1 .:    with Ada.Text_IO; use Ada.Text_IO;
   2 .:    package body Logging is
```

(continues on next page)

(continued from previous page)

```
   3 .:
   4 .:       procedure Write (Prefix, Message : String) is
   5 .:       begin
   6 +:          Put (Prefix);
   7 +:          Put_Line (Message);
   8 .:       end Write;
   9 .:
  10 .:       procedure Warn (Message : String) is
  11 .:       begin
  12 +:          Write ("warning: ", Message);
  13 .:       end Warn;
  14 .:
  15 .:       procedure Error (Message : String) is
  16 .:       begin
  17 -:          Write ("error: ", Message);
  18 .:       end Error;
  19 .:    end Logging;

cat xcov/logging__gnat_io.adb.xcov
[...]/logging__gnat_io.adb:
80% of 5 lines covered
80% statement coverage (4 out of 5)

Coverage level: stmt
   1 .:    with GNAT.IO; use GNAT.IO;
   2 .:    package body Logging is
   3 .:
   4 .:       procedure Write (Prefix, Message : String) is
   5 .:       begin
   6 +:          Put (Prefix);
   7 +:          Put (Message);
   8 +:          New_Line;
   9 .:       end Write;
  10 .:
  11 .:       procedure Warn (Message : String) is
  12 .:       begin
  13 +:          Write ("warning: ", Message);
  14 .:       end Warn;
  15 .:
  16 .:       procedure Error (Message : String) is
  17 .:       begin
  18 -:          Write ("error: ", Message);
  19 .:       end Error;
  20 .:    end Logging;
```

7.3.2 Use project loading options to load source traces separately

gnatcov coverage must load each source trace with the corresponding set of project loading options (`-P`, `-X`): each time, we create a checkpoint to carry the corresponding coverage information:

```
gnatcov coverage -Pexample -XLOGGING_BACKEND=text_io \
  main_text_io.srctrace \
  --level=stmt --save-checkpoint=text_io.ckpt

gnatcov coverage -Pexample -XLOGGING_BACKEND=gnat_io \
  main_gnat_io.srctrace \
  --level=stmt --save-checkpoint=gnat_io.ckpt
```

Now that each source trace was loaded into a checkpoint, it is possible to generate the coverage report from the checkpoints:

```
gnatcov coverage \
  --checkpoint=text_io.ckpt \
  --checkpoint=gnat_io.ckpt \
  --level=stmt --annotate=xcov --output-dir=xcov
```

The same as the first method, this yields the expected coverage report, displaying coverage for the two alternative implementations of `Logging`: see the coverage report in the previous section.

CHAPTER

EIGHT

COVERAGE EXEMPTIONS

In some circumstances, there are good and well understood reasons why proper coverage of some source construct is not achievable. The GNATcoverage *exemptions* facility was designed to allow abstracting these coverage violations away from the genuine defects of a testing campaign.

8.1 Defining Exemption Regions

Exemption regions are lexical sections of sources in which coverage violations are expected and can be justified.

For Ada with the GNAT compilers, regions are defined by the insertion of dedicated pragmas in the sources:

- `pragma Annotate (Xcov, Exempt_On, "justification text");` starts a region, providing some justification text that will be recalled in coverage reports.
- `pragma Annotate (Xcov, Exempt_Off);` closes the current exemption region.

There may be no overlap between exemption regions.

To illustrate, let us consider a common assertion control procedure in Ada:

```
procedure Eassert (T : Boolean) is
begin
   pragma Annotate (Xcov, Exempt_On, "assert condition never to be False");
   if not T then
      raise Program_Error;
   end if;
   pragma Annotate (Xcov, Exempt_Off);
end Eassert;
```

We expect never to reach here with `T` False, so we declare an exemption region to state that coverage violations across this region are expected and not a testing campaign deficiency.

For C code, exemptions are only supported with *source traces* and are defined using comment markers to delimit the exempted regions:

- Any comment containing the string `GNATCOV_EXEMPT_ON` followed by a string in double quotes starts a region, the string within the double quotes being used as justification text that will be recalled in coverage reports.
- Any comment containing the string `GNATCOV_EXEMPT_OFF` closes the current exemption region.

The following assert function illustrates the definition of an exemption block:

```
void
assert (bool x){
```

(continues on next page)

(continued from previous page)

```
 // GNATCOV_EXEMPT_ON "assert condition never to be False"
  if (!x)
    abort();
 // GNATCOV_EXEMPT_OFF
}
```

As in the first example, we never expect to reach this function with x false, so an exemption region is declared to state that all coverage violations within the region are expected.

An exemption comment marker may not intersect any coverage obligation, such as a statement or a decision, as in the following example attempting to exempt a decision only partially:

```
if(a && /*GNATCOV_EXEMPT_ON "justification"*/ b /*GNATCOV_EXEMPT_OFF*/){
  ...
}
```

Such markers are ignored by **gnatcov instrument**, after emitting a warning.

It is also possible to define exemption regions trough external annotation files, see *External source annotations* for more information.

8.2 Reporting about coverage exemptions

Exempted regions are reported as blocks in both the annotated source and the synthetic text reports, for both source and object coverage metrics. In annotated source reports, a # or * character annotates all the exempted lines, depending on whether 0 or at least 1 violation was exempted over the whole section, respectively. For our `Eassert` example above, a typical `=xcov` output for `stmt+decision` coverage for would be:

```
 6 .: procedure Eassert (T : Boolean) is
 7 .: begin
 8 *:    pragma Annotate (Xcov, Exempt_On, "assert condition never to be False");
 9 *:    if not T then
10 *:       raise Program_Error;
11 *:    end if;
12 *:    pragma Annotate (Xcov, Exempt_Off);
13 .: end Eassert;
```

The whole block is marked with * annotations to indicate that some violations were actually exempted; 2 in this case: the statement coverage violation for the `raise` and the decision coverage violation for the `if` control.

In synthetic text reports, a single indication is emitted for each exempted region as a whole, and the indications for all the regions are grouped in a separate *Exempted Regions* report section, only present if there are exemption regions in the analysis scope. This section lists the exempted regions, displaying for each the source location span, the number of actually exempted violations in the region, the exemption justification text and the observed exempted violations per regions. It also includes a total count of the number of exempted regions and another for exempted violations across all sections at the end.

The corresponding `=report` excerpt below illustrates this for the `Eassert` example:

```
...
========================
== 3. EXEMPTED REGIONS ==
========================
```

(continues on next page)

(continued from previous page)

```
eassert.adb:8:4-12:4: 2 exempted violations, justification:
"assert condition never to be False"

Exempted violations:
eassert.adb:8:4: decision outcome TRUE never exercised
eassert.adb:9:8: statement not executed

1 exempted region, 2 exempted violations.

=========================
== 4. ANALYSIS SUMMARY ==
=========================

No non-exempted STMT violation.
No non-exempted DECISION violation.
1 exempted region, 2 exempted violations.
```

The *Coverage Violations* section is renamed to convey that it contains "NON-EXEMPTED" violations only, and the *Analysis Summary* counters are adjusted in a similar manner. The number of exempted regions is added to the list of counters in this section.

If the executed tests actually trigger an assertion failure, there is no coverage violation to be exempted any more and this translates as visible differences in the reports:

In annotated sources, the region is annotated with # signs instead of *, as in:

```
 6 .: procedure Eassert (T : Boolean) is
 7 .: begin
 8 #:    pragma Annotate (Xcov, Exempt_On, "assert condition never to be False");
 9 #:    if not T then
10 #:       raise Program_Error;
11 #:    end if;
12 #:    pragma Annotate (Xcov, Exempt_Off);
13 .: end Eassert;
```

In synthetic reports, the count of exempted violations is 0, like:

```
=========================
== 3. EXEMPTED REGIONS ==
=========================

eassert.adb:8:4-12:4: 0 exempted violation, justification:
assert condition never to be False

1 exempted region.
```

8.2.1 Undetermined Coverage state and Exemptions

For each exemption region in which there are obligations with undetermined coverage state, the synthetic text report will indicate the count of undetermined coverage obligations in the region, in addition to the number of violations.

In the annotated sources, exemption regions with *only* undetermined coverage items are annotated with `@` signs instead of `*`. If there are both undetermined coverage obligations as well as violations in the exemption region, the corresponding lines is still annotated with `#`.

8.3 Object coverage exemptions

Exemption regions specified via annotations in source files actually apply to both source and object level criteria analyzed over the annotated regions.

In the previous example, we would have used similar exemption annotations to deal with expected object instruction and branch coverage failures in Eassert, as the conditional branch used to implement the `if` statement is expected to remain partially covered, as well as the sequence of machine instructions triggering the Ada exception raise.

As for Source Coverage Obligations for source level criteria, information about the declared exemption regions is located in the *Library Information files* produced by the compiler for every compilation unit. The mechanisms described in chapter *Specifying Units Of Interest* can then also be used to designate units for which exemptions regions should be accounted for.

CHAPTER

NINE

DISABLE COVERAGE ANALYSIS

It is possible to fully disable coverage analysis over a specific region. In contrast to exemption regions, gnatcov skips over disabled coverage region and treats them similarly as a no code region, e.g. with no source coverage obligations.

Such a mechanism can typically be used to disable coverage for a code region that may or may not expand according to the build configuration at use.

This mechanism only works when using source traces.

9.1 Defining Disabled Coverage Regions

Disabled Coverage Regions are lexical sections of sources in which coverage analysis is fully disabled.

For Ada with the GNAT compilers, regions are defined by the insertion of dedicated pragmas in the sources:

- `pragma Annotate (Xcov, Cov_Off, "justification text");` starts a region, providing some justification text that will be recalled in coverage reports.
- `pragma Annotate (Xcov, Cov_On);` closes the current disabled coverage region.

There may be no overlap between disabled coverage regions.

To illustrate, let us consider a common assertion control procedure in Ada:

```
procedure Eassert (T : Boolean) is
begin
   pragma Annotate (Xcov, Cov_Off, "my justification");
   if not T then
      raise Program_Error;
   end if;
   pragma Annotate (Xcov, Cov_On);
end Eassert;
```

For C code, disabled coverage regions are defined using comment markers to delimit the region

- Any comment containing the string `GNATCOV_COV_OFF` followed by a string in double quotes starts a region, the string within the double quotes being used as justification text that will be recalled in coverage reports.
- Any comment containing the string `GNATCOV_COV_ON` closes the current region.

The following assert function illustrates the definition of a disabled coverage region:

```
void
assert (bool x){
 // GNATCOV_COV_OFF "my justification"
```

(continues on next page)

(continued from previous page)

```
  if (!x)
    abort();
 // GNATCOV_COV_ON
}
```

It is also possible to define disabled coverage regions from external annotation files, see *External source annotations* for more information.

9.2 Consolidating traces with varying disabled coverage regions

As stated, there can be a varying set of disabled coverage regions according to the build configuration and when considering C/C++ preprocessing. If you lie in that case, read thoroughly the information below.

It is not allowed to consolidate traces originating from instrumentations with mismatching disabled coverage regions. If doing so, gnatcov will warn with `traces for <unit_name> (from <filename>) are inconsistent with the corresponding Source Instrumentation Data`.

To circumvent this, it is mandatory to produce checkpoints using the instrumentation artifacts the source trace originates from. GNATcov then allows consolidation of checkpoints with mismatching disabled coverage regions. See *Coverage Consolidation* for more information.

9.3 Reporting about disabled coverage regions

Exempted regions are reported as blocks in both the annotated source and the synthetic text reports, for both source and object coverage metrics. In annotated source reports, a `D` annotates all the lines of a disabled coverage region.

For our `Eassert` above, a typical `=xcov` output for `stmt+decision` coverage for would be:

```
 6 .: procedure Eassert (T : Boolean) is
 7 .: begin
 8 D:    pragma Annotate (Xcov, Cov_Off, "my justification");
 9 D:    if not T then
10 D:       raise Program_Error;
11 D:    end if;
12 D:    pragma Annotate (Xcov, Cov_On);
13 .: end Eassert;
```

In synthetic text reports, a single indication is emitted for each disabled coverage region as a whole, and the indications for all the regions are grouped in a separate *Disabled Coverage Regions* report section, only present if there are disabled coverage regions in the analysis scope.

The corresponding `=report` excerpt below illustrates this for the `Eassert` example:

```
...
================================
== 3. DISABLED COVERAGE REGIONS ==
================================

eassert.adb:8:4-12:4: justification:
"my justification"
```

(continues on next page)

(continued from previous page)

```
1 region with disabled coverage.

=========================
== 4. ANALYSIS SUMMARY ==
=========================

1 region with disabled coverage.
```

CHAPTER

TEN

EXTERNAL SOURCE ANNOTATIONS

In case modifying the sources is not possible or desirable, it is possible to generate annotations in separate files, which can be passed to **gnatcov instrument** or **gnatcov coverage** to modify the behavior of the instrumenter or of the coverage analysis.

Annotations loaded from an external file can be used along in-source annotations, however in case of conflicts, the annotations defined in the sources will always be prioritized.

External annotations are stored in TOML files, which can be manipulated through three **gnatcov** commands, **gnatcov add-annotation**, **gnatcov delete-annotation** and **gnatcov show-annotations**, to respectively add a new annotations, delete an existing annotation from the files, or show the annotations that are stored in the files.

Once generated, annotation files should be passed to the **gnatcov instrument** or **gnatcov coverage** commands with the `--external-annotations` switch for them to be taken into account by **gnatcov**.

10.1 Generating external annotations

The **gnatcov add-annotation** command can be used to create an annotation, tied to a specific source location.

The help section for the **gnatcov add-annotation** command can be displayed by running `gnatcov add-annotation --help`. Its synopsis is:

```
gnatcov add-annotation --kind=KIND [--external-annotations=FILENAME] --output=OUTPUT_
↪FILENAME [OPTIONS] FILENAME
```

Some notable command line options are:

`--output`
: Name of the file to the newly created annotation will be written. If there already is a file, it will be overwritten.

`--external-annotations`, possibly repeated and accepting *@listfile arguments*
: Loads pre-existing annotations from *FILENAME*. They are used to check that the new annotation does not conflict with any pre-existing one. The loaded annotations are all written to the output file specified through `--output`.

`FILENAME`, positional
: Filename to which the new annotation should apply. There are special considerations to keep in mind when specifying the name of the file to be annotated, see *File relocation considerations*

`--annotation-id=IDENTIFIER`, optional
: Unique identifier for the new annotation. If not specified, **gnatcov** will generate one based on the kind of annotation and the designated location.

  This identifier must be unique across all external annotation files passed to any **gnatcov** invocation, and is used in diagnostics, or in the other annotation manipulation commands, **gnatcov delete-annotation** and **gnatcov show-annotations** to uniquely designate an annotation.

--force, optional
Force overwriting of a pre-existing annotation for the same location, or with the same identifier. If not specified, gnatcov will emit an error and abort the annotation generation. The output file will not be modified.

The required command line switches depend on the value of the `--kind`, conveying the kind annotation to be generated, which correspond to the annotations kinds supported in `pragma Annotate (Xcov, KIND, ..)`. The required switches are detailed in the help text for the **gnatcov add-annotation** command, and are detailed bellow. A switch in brackets signifies that the switch is optional, otherwise the switch is required and **gnatcov add-annotation** will emit an error if not found on the command line.

- **--kind=Exempt_On**
  Generate an annotation symbolizing the beginning of an *exempted region*.

  --location=LINE:COL
  Source location for the beginning of the exempted region.

  --justification=MESSAGE
  Justification message to be displayed in the coverage reports for the exempted region.

- **--kind=Exempt_Off**
  Generate an annotation symbolizing the end of an exempted region.

  --location=LINE:COL
  Source location for the end of the exempted region.

- **--kind=Exempt_Region**
  Generate an annotation symbolizing an entire exempted region.

  --start-location=LINE:COL
  Source location for the beginning of the exempted region.

  --end-location=LINE:COL
  Source location for the end of the exempted region.

  --justification=MESSAGE
  Justification message to be displayed in the coverage reports for the exempted region.

- **--kind=Cov_Off**
  Generate an annotation symbolizing the beginning of a *disabled coverage region*.

  --location=LINE:COL
  Source location for the beginning of the disabled coverage region.

  --justification=MESSAGE
  Justification message for the disabled coverage region, to be displayed in the coverage reports.

- **--kind=Cov_On**
  Generate an annotation symbolizing the end of a disabled coverage region.

  --location=LINE:COL
  Location for the end of the disabled coverage region.

- **--kind=Dump_Buffers**
  Generate an annotation instructing **gnatcov** to insert a *buffer dump procedure call* at the specified location. This is only taken into account when the selected dump trigger is `manual`, see *Output strategies for coverage information* for more information concerning the dump triggers.

  --location=LINE:COL
  Source location at which the buffer dump procedure call should be inserted.

  [--dump-filename-prefix=TEXT]
  Optional trace filename prefix to be passed to the buffer dump procedure call. This will be textually passed as argument to the buffer dump, and must be an expression

evaluating to a null-terminated `char *`. As such, if the prefix to be used is a literal string, the argument passed to `--dump-filename-prefix` must contain quotes (e.g. `--dump-filename-prefix='"my_trace"'`).

[--annotate-after]
If specified, instruct **gnatcov** to insert the buffer dump procedure **after** the statement designated by the annotation. See *Semantics of buffer manipulation annotations* for more details on the meaning of this option.

- **--kind=Reset_Buffers**
  Generate an annotation instructing gnatcov to insert a *coverage buffer reset procedure call* at the specified location. This is only taken into account when the selected dump trigger is `manual`, see *Output strategies for coverage information* for more information concerning the dump triggers.

  --location=LINE:COL
  Location at which the buffer reset procedure call should be inserted.

  [--annotate-after]
  If specified, instruct **gnatcov** to insert the buffer reset procedure **after** the statement designated by the annotation. See *Semantics of buffer manipulation annotations* for more details on the meaning of this option.

10.1.1 Semantics of buffer manipulation annotations

Due to the differences in instrumentation technology used by **gnatcov** for C/C++ and Ada, the external annotations concerning buffer dump/reset have different semantics that need to be taken into account when first annotation sources.

For C and C++ sources, **gnatcov** will insert the buffer dump/reset call at the exact location designated by the annotation, without validating if the resulting code is legal. It is thus recommended to choose a location corresponding to a whitespace character, immediately before or after a statement.

For instance, starting from the following source file:

```
int main(){
  // Execute the core program
  do_stuff();

  // Cleanup temp files
  cleanup();
}
```

Creating an annotation as follows:

```
gnatcov add-annotation --kind=Dump_Buffers -o annotations.toml --location=6:3 main.c
```

would result in the following invalid code to be generated:

```
int main(){
  //Execute the core program
  do_stuff();

  // Cleanup temp files
  cgnatcov_dump_buffers();leanup();
}
```

Instead, it is better to target any whitespace character before the statement, as in `--location=6:2`.

For Ada sources, **gnatcov** will locate the inner-most statement list that encloses the designated location, and insert the procedure call immediately **before** this statement by default. The `--annotate-after` switch can be used to instruct gnatcov to instead insert the procedure call **after** the designated statement. This in particular is necessary to add a buffer dump annotation after the last statement of a list.

If gnatcov cannot locate a statement list enclosing the designated location, a warning will be emitted and the annotations will be ignored.

For instance, starting from the following source file:

```
procedure Main is
begin
   --  Run the actual program

   Do_Processing;

   --  Cleanup temp files

   Do_Cleanup;
end Main;
```

Generating an annotation with:

```
gnatcov add-annotation --kind=Dump_Buffers -o annotations.toml --location=9:15 main.adb
```

results in the following source, despite the source location pointing at the end of the Do_Cleanup procedure call:

```
procedure Main is
begin
   --  Run the actual program

   Do_Processing;

   --  Cleanup temp files

   GNATCov_RTS_Dump_Buffers; Do_Cleanup;
end Main;
```

To ensure the buffer dump procedure is inserted after the Do_Cleanup call, it is necessary to pass the `--annotate-after` command line switch.

10.1.2 File relocation considerations

The external file annotation mechanism stores the filename passed to the **gnatcov add-annotation** command in the generated annotation file. When the annotations are loaded by a **gnatcov instrument** or **gnatcov coverage** command invocation, to determine if an annotation is relevant for any of the processed files, **gnatcov** checks whether the full filename of the file being processed ends with the annotation target filename. It is thus important to only store in the annotation the part of the filename that will not change between the different **gnatcov** command invocations.

This means that relative paths components (e.g. `./` or `../`), and absolute paths are likely to not be properly recognized.

The **gnatcov add-annotation** command accepts a `--source-root=PREFIX` option that will strip `PREFIX` from the target filename when generating the annotations. As such, it is possible to generate an annotation for a file located in a parent directory, while ensuring the generated annotation will correctly be taken into account in subsequent **gnatcov** invocations with the following command line:

```
gnatcov add-annotation [OPTIONS] --source-root="../" ../src/file.adb
```

gnatcov can also automatically deduce the appropriate prefix to be stripped from the filename if a project file is passed to **gnatcov add-annotation** with the `-P` option. Note that this only works if the file is unique in the project tree, or if the file is located in a sub-directory of its project root directory.

10.2 Deleting a pre-existing annotation

The **gnatcov delete-annotation** command can be used to remove a pre-existing annotation from an external annotation file.

The help section for the **gnatcov add-annotation** command can be displayed by running `gnatcov delete-annotation --help`. Its synopsis is:

```
gnatcov delete-annotation --external-annotations=FILENAME --output=OUTPUT_FILENAME --
↪annotation-id=IDENTIFIER
```

The semantics of each command line switch is:

--annotation-id=IDENTIFIER:
Unique IDENTIFIER of the annotation to be deleted.``

--external-annotations=FILENAME, possibly repeated and accepting *@listfile arguments*:
External annotation files from which the annotation will be loaded. If multiple files are passed to **gnatcov**, the annotations will be consolidated together and all written to the output file.

--output=OUTPUT_FILENAME:
Name of the file where the annotations will be written back after deletion of the designated annotation. This will overwrite any pre-existing file with the same OUTPUT_FILENAME.

10.3 Displaying the annotations contained in annotation files

The command **gnatcov show-annotations** can be used to display the annotations contained in annotation files in a more user-friendly manner.

The help section for the **gnatcov add-annotation** command can be displayed by running `gnatcov show-annotations --help`. Its synopsis is:

```
gnatcov show-annotations --external-annotations=FILENAME [--kind=KIND] [-P PROJECT]␣
↪[FILENAMES]
```

The semantics of the command line switches are as follow:

--external-annotations=FILENAME, possibly repeated and accepting *@listfile arguments*:
External annotation files from which annotations will be loaded

--kind=KIND, optional:
Only display the annotations of kind KIND.

-P PROJECT, optional:
Show all annotations applicable to all source files of the project tree rooted at PROJECT.

FILENAMES, positional:
Only show the annotations applicable to the listed files.

Either the `-P` command line option or positional filenames must be specified.

The output format is as follows:

```
BASENAME_1:
- START_LOCATION - END_LOCATION; id: IDENTIFIER; kind: KIND; [EXTRA_FIELDS]
- ...

BASENAME_2:
- ...
```

`BASENAME_i` corresponds to the basename of each file for which there is an annotation. The each annotation is displayed on each line, starting by the location range for the annotation. If the annotation only concerns a single location, the `END_LOCATION` field will be identical to the `START_LOCATION`. The unique identifier of the annotation is then displayed in place of `IDENTIFIER`, and the annotation kind is displayed in place of `KIND`. The `EXTRA_FIELDS` concerns options specific to each annotation kind, and are displayed as a semi-column separated list. See *Generating external annotations* for more details on the extra fields that each annotation kind supports.

10.4 Annotation stability through file modifications

The external annotations generated by the **gnatcov add-annotation** command embed varying levels of information so that the source location designated on the command line option can be remapped when possible, or invalidated otherwise.

This depends mainly on the language of the file to be annotated:

- For Ada sources, the annotation is tied to the inner-most enclosing named construct, such as a subprogram or a package. If the file is modified outside of that construct the annotation will be remapped properly. If the enclosing construct is modified, the annotation will be invalidated.
- For Cor C++ sources, the annotations are tied to the inner-most enclosing named declaration, such as a function declaration for C, or any of a namespace declaration, a class declaration or function/method declaration for C++.

Note that in both cases, if no enclosing named construct can be found, the **gnatcov add-annotation** command will emit a warning and fall back to an absolute annotation, which is invalidated as soon as the file is modified.

If an annotation is invalidated gnatcov will emit a warning stating that the annotation was ignored, along with its unique identifier.

The output of the **gnatcov show-annotations** command will also display stale annotations, the format for those annotations will be:

```
- STALE ANNOTATION; id: IDENTIFIER; kind: KIND; [EXTRA_FIELDS]; diagnostic: DIAGNOSTIC
```

where `DIAGNOSTIC` will contain a short explanation of why the entry is stale.

To fix this, simply replace the entry with an invocation of **gnatcov add-annotation**, specifying the annotation identifier to be replaced, and forcing the replacement:

```
gnatcov add-annotation --annotation-id=IDENTIFIER --force [OPTIONS]
```

CHAPTER

ELEVEN

USING PROJECT FILES

11.1 General considerations

GNAT project files are a useful device to describe the structure of larger applications, for the benefit of the tools intervening at various stages of development, from the IDE to build and analysis tools.

GNATcoverage takes full advantage of GNAT projects for various aspects of the coverage analysis activity, in particular:

1. Select units of interest and retrieve Source Coverage Obligations for source coverage analysis,
2. Retrieve exemption regions for source and object coverage analysis,
3. Specify default switches for the various **gnatcov** commands,
4. Specify the target architecture and runtime profile for which the analyzed program was built.

The project selection and units-of-interest designation facilities are described in the *Specifying Units Of Interest* chapter of this manual, with a focus on source coverage perspectives. In the root project, a `Coverage` package may be used to specify the arguments for **gnatcov** commands and the target/runtime profile may be set using toplevel attributes.

Please note that **non-library aggregate project files are currently not supported** and will be rejected by **gnatcov**. The aggregated projects will have to be processed individually and coverage reports consolidated *through checkpoints*, when this makes sense.

11.2 Specifying `gnatcov` command Switches

`Switches` attributes in `Coverage` package of the root project file are treated as lists of command line switches for **gnatcov** commands. Each attribute specification requires an index indicating what **gnatcov** operation the switches apply to. Each attribute value is expected to be a list of options for this operation. Here is a first basic example:

```
package Coverage is
   level := "--level=stmt"; -- to be reused in different contexts

   for Switches ("run") use (level);
   -- This will apply to "gnatcov run"

   for Switches ("coverage") use (level, "--annotate=report");
   -- This will apply to "gnatcov coverage"
end Coverage;
```

For switches applicable to all the commands you are planning to use, the special `"*"` index is available to denote *any* command. If you are going to use only `run` and `coverage`, for instance, the example above might be re-written as:

```
package Coverage is
   for Switches ("*") use ("--level=stmt");
   for Switches ("coverage") use ("--annotate=report");
end Coverage;
```

The * arguments are always inserted first with respect to the final command line interpretation. In the example above, `--level` from the * list cumulates before `--annotate` for **gnatcov coverage**. Similarly, switches from the project file are always processed as if appearing before the others on the command line.

For switches like `--level` that don't accumulate to produce sets, the last occurrence on the command line prevails. The project file values act as defaults that can be overriden by an explicit value on the command line, wherever they are placed (before or after `-P`).

For switches such as `--units` which have cumulative effect, later occurrences on the command line add up with, rather than replace, those specified in the project file.

11.3 Specifying the Target and language Runtime

Similarly to other tools, **gnatcov** uses any existing `Target` or `Runtime` attribute in the root project file to detect what target architecture and associated language runtime profile to consider. This can be done instead of providing the `--target` and `--RTS` options both for correct processing of project files and to run the appropriate execution environment in **gnatcov run**. Here is a simple example:

```
project My_Program is
   for Languages use ("Ada");
   for Main use ("my_program.adb");

   for Target use "powerpc-elf";
   for Runtime use "zfp-mpc8641";
end My_Program;
```

When a `--target` or a `--RTS` option is provided on the command line, the option takes precedence over the corresponding attribute.

CHAPTER

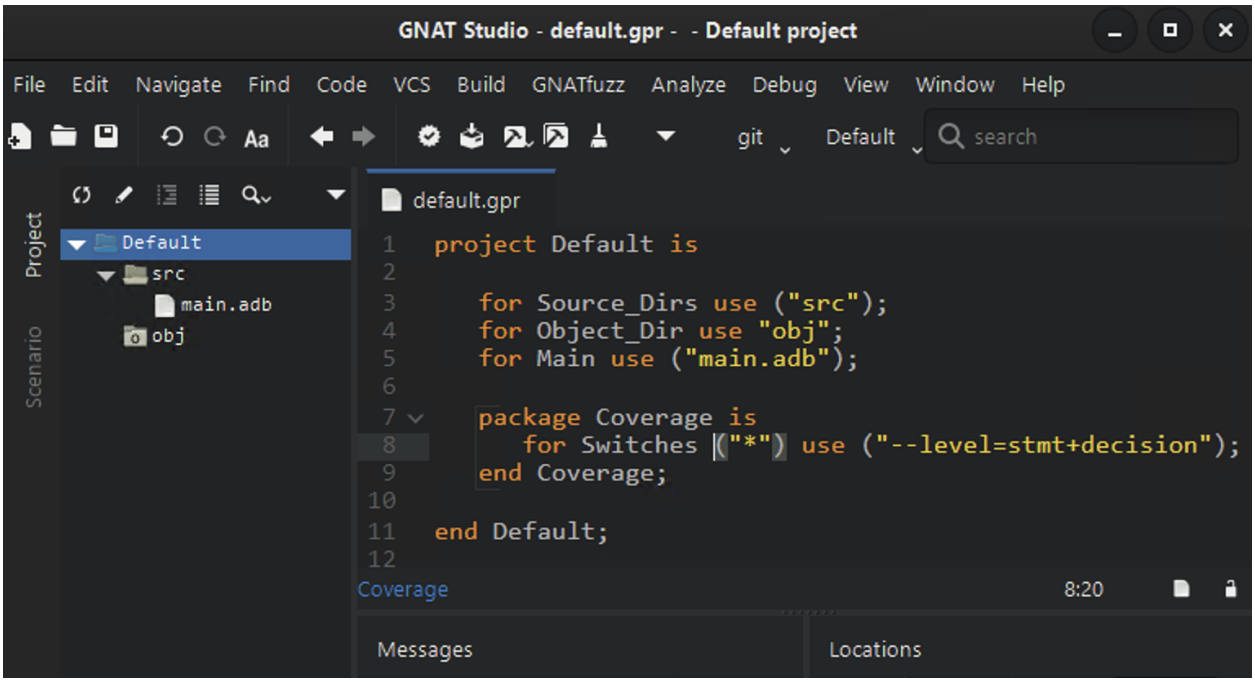
TWELVE

USING GNATCOVERAGE FROM GNAT STUDIO

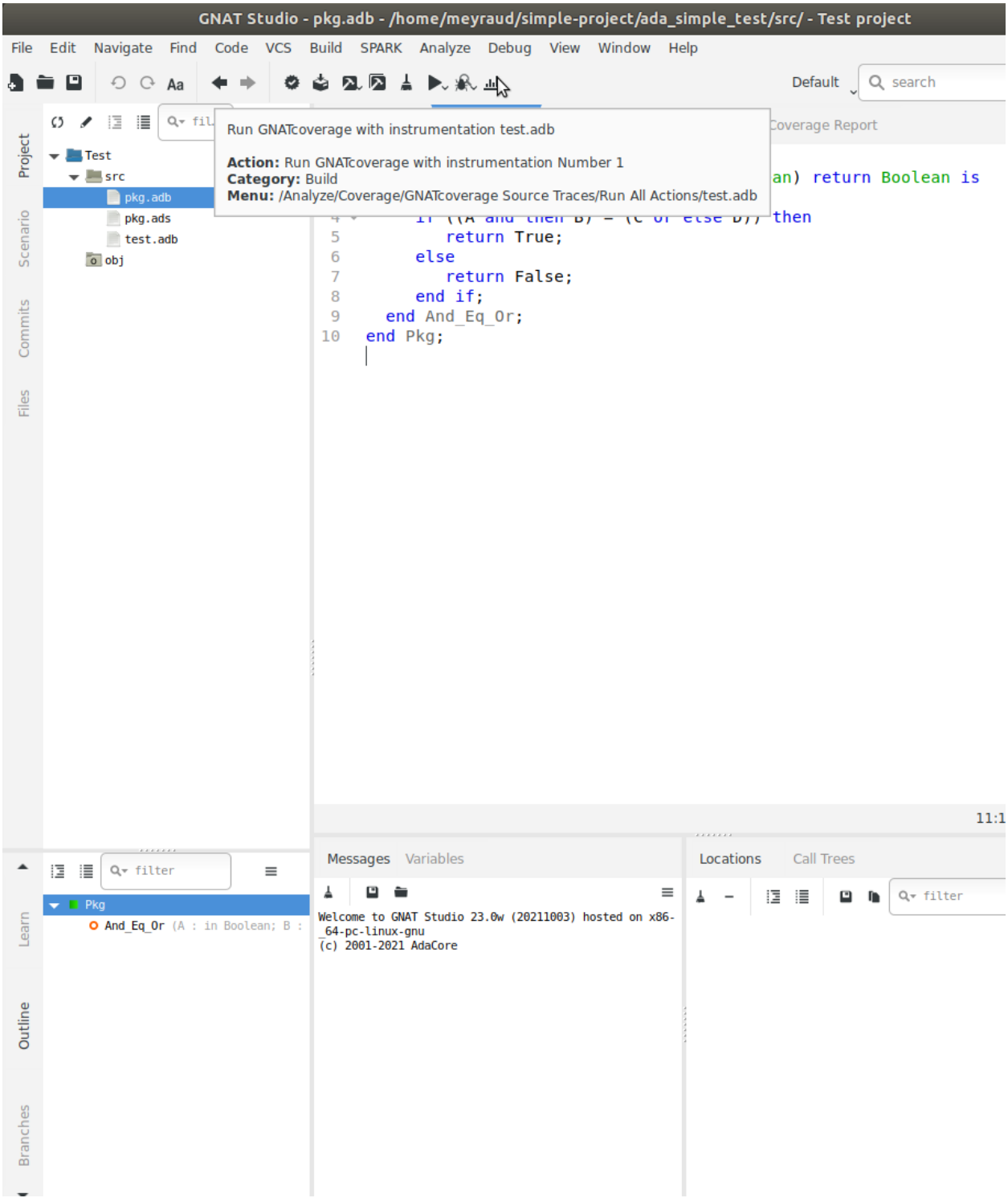
GNATcoverage is integrated in GNAT Studio so that it is possible to analyze code coverage without leaving your favorite editor.

12.1 Basic scenario

As a first step, indicate which coverage criterion GNATcoverage should use through the *project file*. If no coverage level is set in the project file, gnatcov will default to `stmt`, and emit a warning indicating that the coverage level is not set. You can specify the coverage level by by editing directly the project file: GNAT Studio provides completion, tooltips, outline and other common IDE features for project files through LSP and the Ada Language Server, helping you to customize your project more easily.



This is all for the setup part. Then you just need to click on the *Run GNATcoverage* toolbar button as shown in the following screenshot.

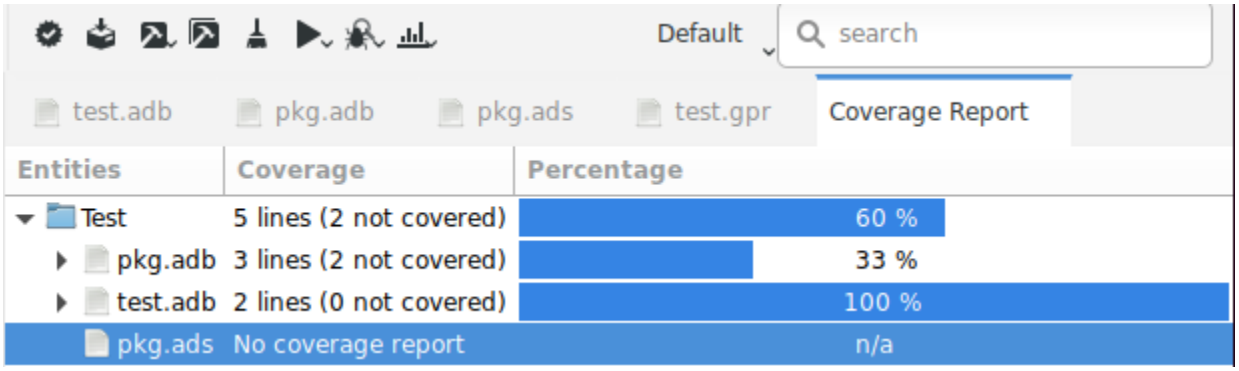


This button uses GNATcoverage instrumentation capabilities by default. It will perform each needed step: build the GNATcoverage runtime, instrument your project, build the instrumented version of it, run the instrumented program, and display a *Coverage Report* view which displays the ratio of source lines that are covered.

Note that each step can be performed individually by clicking on their respective menu items under the *Analyze* → *Coverage* → *GNATcoverage Source Traces*.

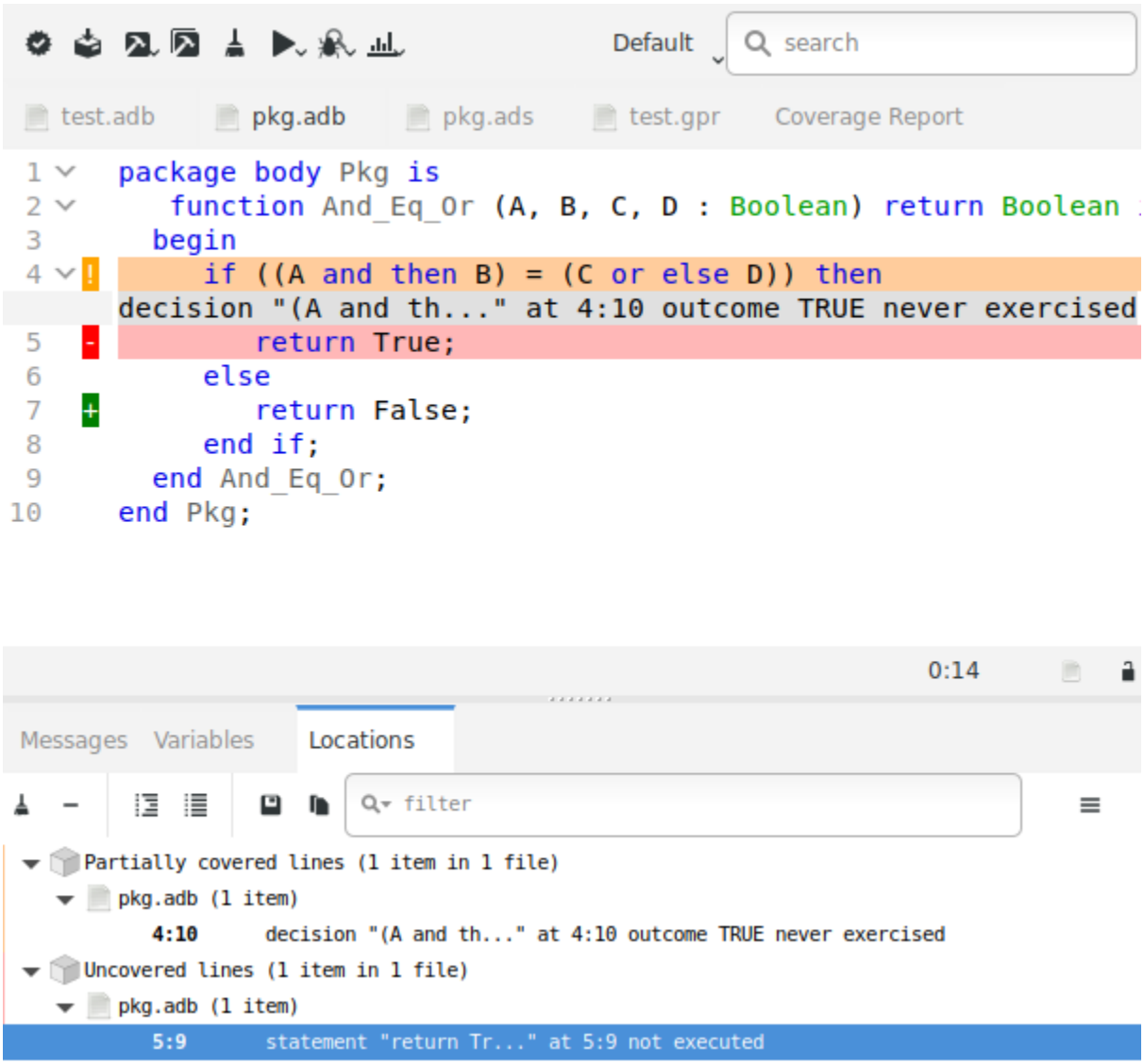
Also Note that you can select a prebuilt gnatcov runtime under the same menu. This will skip the "build the gnatcov

runtime" step, and will be especially useful if you bring modifications to this runtime. By default, GNAT Studio will rebuild the runtime that is shipped with GNATcoverage and reinstall it in a temporary directory every time.



In order to read the detail of which line is covered and which line is not, double-click on the file you want to inspect. This will open an editor on the file with annotations that represent the coverage state:

- red annotations for uncovered ones;
- orange annotations for partially covered ones;
- green annotations for covered ones.



Note that the Locations view reports all non-covered lines. It makes navigation easy as you only have to click on a diagnostic to go to the corresponding line.

12.2 Viewing a coverage report produced outside of GNAT Studio

If the workflow is not adapted to your process, GNAT Studio also supports displaying a coverage report produced externally, by first loading the coverage data using any of the actionable *Load Data* item under *Analyze → Coverage*.

GNAT Studio expects a coverage report under the xcov+ format, under the object directory of the project (<prj_obj>), or under <prj_obj>/xcov+.

Then you can visualize the report by clicking on *Analyze → Coverage → Show coverage report*.

12.3 Using GNATtest's coverage integration makefile

When using individual test drivers, it is not possible to use the regular GNATcoverage integration. This is true when GNATtest is called with `--separate-drivers` or `--stub` (as it implies `--separate-drivers`), as described in *Individual Test Drivers*. In this case, the integration of GNATcoverage is done through the use of the generated integration makefile. For details on how to use this makefile, see *Integration with GNATcoverage*.

Producing coverage reports through GNATtest's coverage integration makefile generates the coverage reports under the test driver's object director by default. However, these reports contain coverage information not for the test harness project, but for the project being tested with gnattest.

If the harness project is opened in GNAT Studio, the coverage data should be loaded using the actionable *Load Data for All Projects* item.

Visualization of the reports is then done as previously described.

12.4 GNATtest scenario

Using GNATcoverage GNATtest-based testsuite in GNAT Studio comes naturally as the combination of both. Open the project you want to test and then generate the unit test setup for it. Using the menu: *Analyze* → *GNATtest* → *Generate Unit Test Setup*.

Write your testcases, and when you are ready to run your testsuite, switch to the `gnatcov` build mode, select the desired coverage criteria in the project properties. Now, click on the *Run GNATcoverage* toolbar button.

Depending on your version of GNATcoverage, logs in the Messages view might include warnings which mention the various helper projects generated and used by the GNATtest setup, for instance AUnit. These should not influence coverage analysis results for the units of your project.

CHAPTER

THIRTEEN

GLOSSARY OF TERMS & CONCEPTS

Binary Trace
: File containing low level information about the blocks of machine code executed by a program, from which GNATcoverage can infer source level coverage assessments. Such traces are produced by an instrumented execution environment for programs built from their original sources.

Complex Boolean Expression
: Boolean expression with at least two operands, where all the binary operators are defined to enforce short-circuit semantics.

Coverage Violation
: Failure to satisfy a particular requirement of a coverage criterion. For example, not executing a given statement is a failure to satisfy part of the Statement Coverage criterion requirements, triggering a "statement not executed" violation of the criterion.

Condition
: Leaf operand, possible unary operator included, in a Decision expression. For example, `A`, `not B` and `C` in `if A or else (not B and then C) then`.

Coverage Checkpoint
: File a condensed version of a coverage assessment computed by **`gnatcov coverage`**, which the tool can reload and combine with other assessments very efficiently. This may be viewed as an internal representation of a coverage report which `--annotate` would output in a user readable form. See *Consolidation from coverage checkpoints* for more information.

Coverage Runtime
: Library of services used by instrumented sources to manipulate coverage data while the program is running and produce a source trace file from such data at user selectable execution points.

Decision
: Any Simple or Complex Boolean expression used as the control value in a source level control-flow oriented statement (*if*, *while*, *for*). For MCDC analysis purposes, we also treat as decisions Complex Boolean Expressions anywhere they appear, for example also on the right hand side of an assignment.

Evaluation Vector
: A set of Boolean values assigned to each *condition* of a *decision* together with the resulting expression value. For example A=True, B=False, Expression=False for `A and then B`.

Exemption Region
: Regions of source for which *coverage violations* are expected, and explicitly marked as such with a mandatory justification text.

Incremental Coverage Analysis
: The process of constructing a consolidated coverage result incrementally, using a single checkpoint file to accumulate new info about discharged coverage obligations as tests execute in sequence. See *Incremental coverage analysis*.

Independence Pair
: A pair of decision evaluation vectors that demonstrate the independent influence of a specific condition on the decision for MCDC analysis purposes. The set of valid pairs for a condition depends on the particular MCDC variant driving the analysis.

Library Information file
: A text file produced by the compiler as it processes a source unit, containing information about the entities present in the unit; for example: dependencies w.r.t. other units, cross-reference information, Source Coverage Obligations, annotation pragmas for exemption regions, ... These are `.ali` files for Ada and `.gli` files for C, produced at the same place as the object file for a given compilation.

Simple Boolean Expression
: Boolean expression with a single operand, possibly negated. No binary operator.

Source Coverage Obligation
: Item that designates a source program entity which must be subject to coverage checks for a set of criteria. This conveys an entity kind, for example *statement*, *decision* or *condition*, together with a specific unit:line:column source location. Also known as SCO.

Source Instrumentation Data file
: A binary file produced by the gnatcov instrumenter as it processes a source unit, containing information about the entities present in the unit; for example: Source Coverage Obligations, annotation pragmas for exempted regions, ... These are `.sid` files, produced at the same place object files would be created during the compilation of the unit.

Source Trace
: File(s) containing high level information about the source coverage achievements resulting from a program execution. Such traces are produced by an instrumented version of the program running in its regular execution environment.

Test Driver
: Part of a program that is used to exercise a set of application units in a specific manner, to test conformance of these units to particular requirements. Coverage objectives are typically associated with the applicative part only.

Units of Interest
: Set of units on which coverage should be assessed. See *Specifying Units Of Interest* for more information.

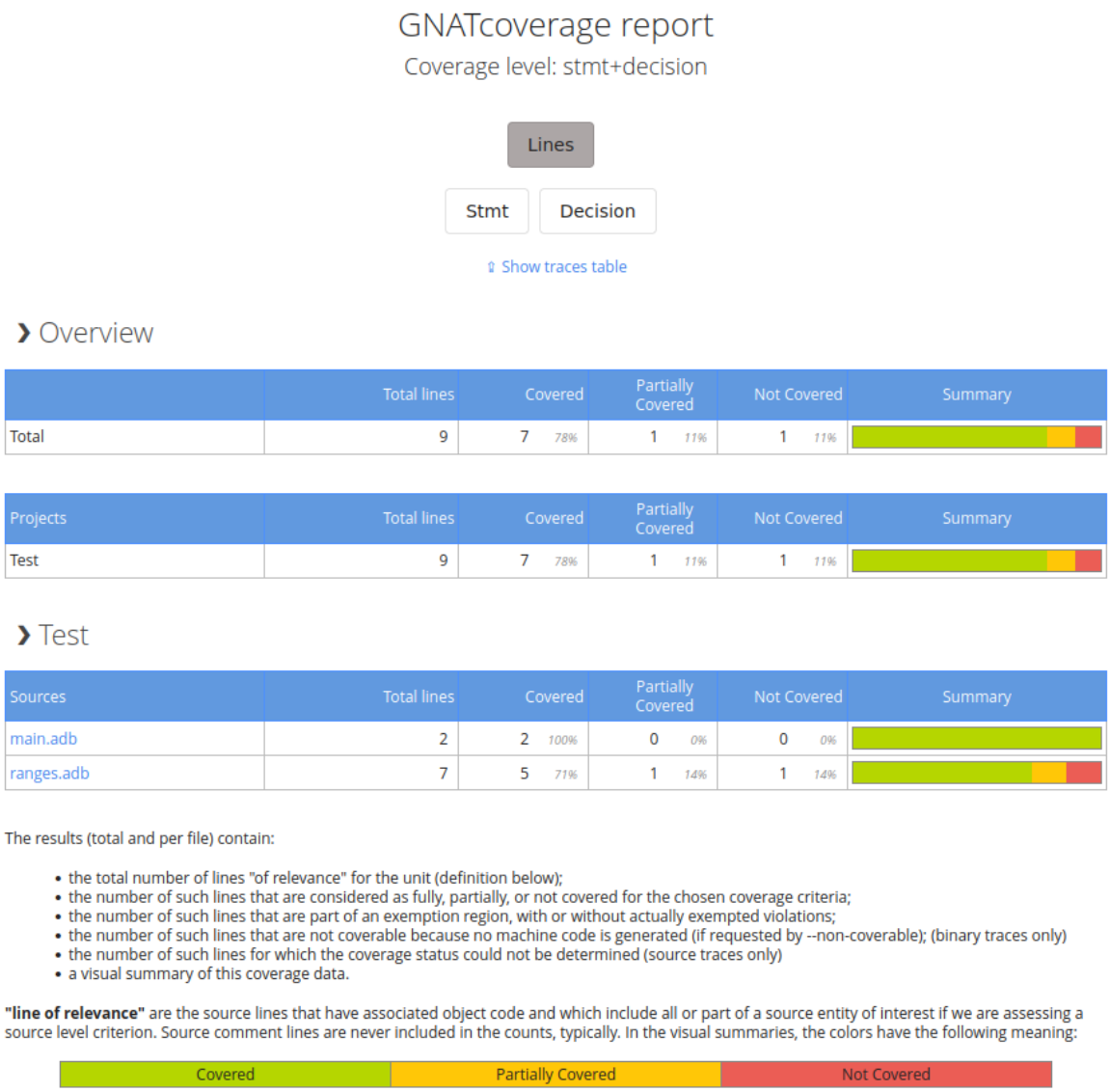
@listfile argument
: Part of a command line that designates a text file which contains a list of items, one per line in the file. The expected nature of each item depends on the context.

CHAPTER

FOURTEEN

APPENDICES

14.1 Sample html index



GNATcoverage report

Coverage level: stmt+decision

Lines

Stmt | Decision

⇳ Show traces table

› Overview

	Total lines	Covered	Partially Covered	Not Covered	Summary
Total	9	7 78%	1 11%	1 11%	

Projects	Total lines	Covered	Partially Covered	Not Covered	Summary
Test	9	7 78%	1 11%	1 11%	

› Test

Sources	Total lines	Covered	Partially Covered	Not Covered	Summary
main.adb	2	2 100%	0 0%	0 0%	
ranges.adb	7	5 71%	1 14%	1 14%	

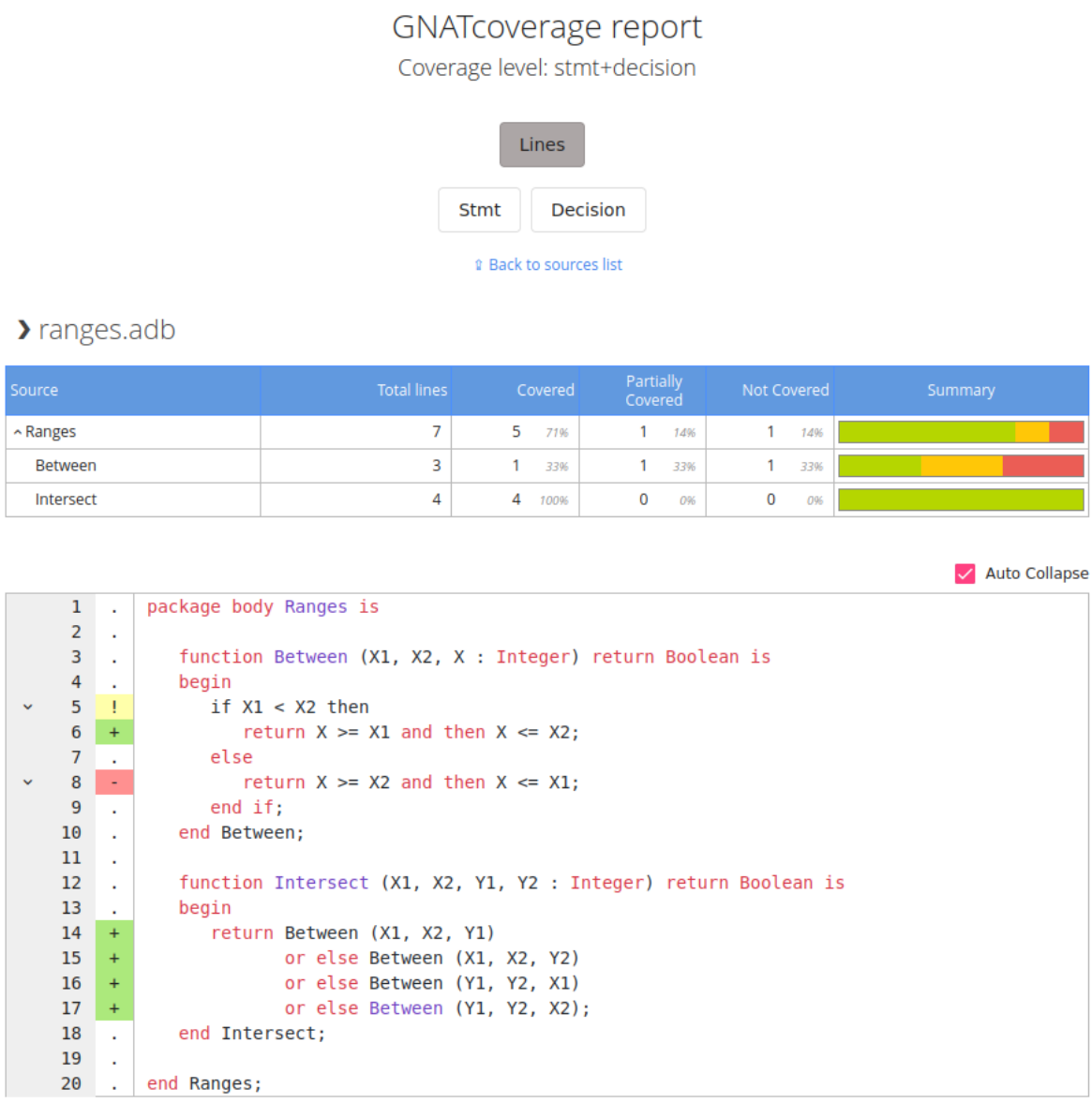
The results (total and per file) contain:

- the total number of lines "of relevance" for the unit (definition below);
- the number of such lines that are considered as fully, partially, or not covered for the chosen coverage criteria;
- the number of such lines that are part of an exemption region, with or without actually exempted violations;
- the number of such lines that are not coverable because no machine code is generated (if requested by --non-coverable); (binary traces only)
- the number of such lines for which the coverage status could not be determined (source traces only)
- a visual summary of this coverage data.

"line of relevance" are the source lines that have associated object code and which include all or part of a source entity of interest if we are assessing a source level criterion. Source comment lines are never included in the counts, typically. In the visual summaries, the colors have the following meaning:

Covered | Partially Covered | Not Covered

14.2 Sample html annotated source



GNATcoverage report

Coverage level: stmt+decision

Lines

Stmt Decision

⇧ Back to sources list

› ranges.adb

Source	Total lines	Covered	Partially Covered	Not Covered	Summary
^ Ranges	7	5 71%	1 14%	1 14%	
Between	3	1 33%	1 33%	1 33%	
Intersect	4	4 100%	0 0%	0 0%	

Auto Collapse

```
    1  .  package body Ranges is
    2  .
    3  .     function Between (X1, X2, X : Integer) return Boolean is
    4  .     begin
    5  !        if X1 < X2 then
    6  +           return X >= X1 and then X <= X2;
    7  .        else
    8  -           return X >= X2 and then X <= X1;
    9  .        end if;
   10  .     end Between;
   11  .
   12  .     function Intersect (X1, X2, Y1, Y2 : Integer) return Boolean is
   13  .     begin
   14  +        return Between (X1, X2, Y1)
   15  +               or else Between (X1, X2, Y2)
   16  +               or else Between (Y1, Y2, X1)
   17  +               or else Between (Y1, Y2, X2);
   18  .     end Intersect;
   19  .
   20  .  end Ranges;
```

Part II

GNATtest User's Guide

`gnattest` tool is a utility that creates unit-test skeletons as well as a test driver infrastructure (harness). `gnattest` creates a skeleton for each visible subprogram in the packages under consideration when they do not exist already.

`gnattest` is a project-aware tool. A project file is mandatory for test driver generation. The project file package that can specify `gnattest` switches is named `gnattest`.

The user can choose to generate a single test driver that will run all individual tests, or separate test drivers for each test. The second option allows much greater flexibility in test execution environment, allows to benefit from parallel tests execution to increase performance, and provides stubbing support.

`gnattest` also has a mode of operation where it acts as the test aggregator when multiple test executables must be run, in particular when the separate test drivers were generated. In this mode it handles individual tests execution and upon completion reports the summary results of the test run.

In order to process source files from a project, `gnattest` has to semantically analyze the sources. Therefore, test skeletons can only be generated for legal Ada units. If a unit is dependent on other units, those units should be among the source files of the project or of other projects imported by this one. Note that it is no longer necessary to specify the Ada language version; `gnattest` can process Ada source code written in any version from Ada 83 onward without specifying any language version switch.

Generated skeletons and harnesses are based on the AUnit testing framework. AUnit is an Ada adaptation of the xxxUnit testing frameworks, similar to JUnit for Java or CppUnit for C++. While it is advised that gnattest users read the AUnit manual, deep knowledge of AUnit is not necessary for using `gnattest`. For correct operation of `gnattest`, AUnit should be installed and aunit.gpr must be on the project path. Except for some special circumstances (e.g. a custom run-time is used), this should normally be the case out of the box.

CHAPTER

ONE

RUNNING GNATTEST

There are two ways of running `gnattest`.

1.1 Framework Generation Mode

In this mode `gnattest` has the following command-line interface:

```
$ gnattest -Pprojname [ switches ] [ filename ]
```

where

- ***-Pprojname***
  specifies the project defining the location of source files. When no file names are provided on the command line, all sources in the project are used as input. This switch is required.

  For the semantics of aggregate project processing by gnattest, see the *Aggregate project handling* section.
- ***filename***
  is the name of the source file containing the library unit package *declaration* (the package "spec") for which a test package will be created. The file name may be given with a path.
- ***switches***
  is an optional sequence of switches as described below.

`gnattest` results can be found in two different places.

- ***automatic harness*:**
  This is the harness code, which is located by default in "gnattest/harness" directory created in the object directory of the main project file. All of this code is generated completely automatically and can be destroyed and regenerated at will, with the exception of the file *gnattest_common.gpr*, which is created if absent, but never overwritten. It is not recommended to modify other files manually, since these modifications will be lost if `gnattest` is re-run. The entry point in the harness code is the project file named *test_driver.gpr*. Tests can be compiled and run using a command such as:

  ```
  $ gprbuild -P<harness-dir>/test_driver
  ```

  Note that if you need to adjust any options used to compile the harness, you can do so by editing the file *gnattest_common.gpr*.
- ***actual unit test skeletons*:**
  A test skeleton for each visible subprogram is created in a separate file, if it doesn't exist already. By default, those separate test files are located in a "gnattest/tests" directory that is created in the object directory of corresponding project file. For example, if a source file my_unit.ads in directory src contains a visible subprogram Proc, then the corresponding unit test will be found in file src/tests/my_unit-test_data-tests.adb

and will be called Test_Proc_<code>. <code> is a signature encoding used to differentiate test names in case of overloading.

Note that if the project already has both my_unit.ads and my_unit-test_data.ads, this will cause a name conflict with the generated test package.

1.2 Test Execution Mode

In this mode `gnattest` has a the following command-line interface:

```
$ gnattest test_drivers.list [ switches ]
```

where

- ***`test_drivers.list`***
  is the name of the text file containing the list of executables to treat as test drivers. This file is automatically generated by gnattest, but can be hand-edited to add or remove tests. This switch is required.
- ***`switches`***
  is an optional sequence of switches as described below.

CHAPTER

TWO

SWITCHES FOR GNATTEST IN FRAMEWORK GENERATION MODE

--strict
Return error exit code if there are any compilation errors.

-q
Quiet mode: suppresses noncritical output messages.

-v
Verbose mode: produces additional output about the execution of the tool. When specified alone on the command line, prints tool version and exits.

-r, -U
Recursively considers all sources from all projects.

-U *source_file*
Process only those source files for units in the closure of the Ada source contained in `source_file`. Note that this option expects the source file name but not the Ada unit name as its parameter.

--no-subprojects
Process only source files from the root project.

-X*name*=*val*
Indicates that the external variable `name` in the project has the value `val`.

-files=*filename*
Take as arguments the files listed in text file `file`. Text file `file` may contain empty lines that are ignored. Each nonempty line should contain the name of an existing file. Several such switches may be specified simultaneously.

--ignore=*filename*
Do not process the sources listed in a specified file.

--RTS=*rts-path*
Specifies the default location of the runtime library. For restricted profiles, `gnattest` takes into account the run-time limitations when generating the harness.

--additional-tests=*projname*
Sources described in `projname` are considered potential additional manual tests to be added to the test suite.

--harness-only
When this option is given, `gnattest` creates a harness for all sources, treating them as test packages. This option is not compatible with closure computation done by -U main.

--separate-drivers[=*val*]
Generates a separate test driver for each test or unit under test, rather than a single executable incorporating all tests. `val` can be “unit” or “test”, or may be omitted, which defaults to “unit”.

`--stub`
Generates the testing framework that uses subsystem stubbing to isolate the code under test.

`--recursive-stub`
Used along –stub, indicates gnattest to generate stubs for all the packages that are withed by the stubbed units, recursively.

`--harness-dir=`*`dirname`*
Specifies the directory that will hold the harness packages and project file for the test driver. If the `dirname` is a relative path, it is considered relative to the object directory of the project file.

`--tests-dir=`*`dirname`*
All test packages are placed in the `dirname` directory. If the `dirname` is a relative path, it is considered relative to the object directory of the project file. When all sources from all projects are taken recursively from all projects, `dirname` directories are created for each project in their object directories and test packages are placed accordingly.

`--subdirs=`*`dirname`*
Test packages are placed in a subdirectory of the corresponding source directory, with the name `dirname`. Thus, each set of unit tests is located in a subdirectory of the code under test. If the sources are in separate directories, each source directory has a test subdirectory named `dirname`.

`--tests-root=`*`dirname`*
The hierarchy of source directories, if any, is recreated in the `dirname` directory, with test packages placed in directories corresponding to those of the sources. If the `dirname` is a relative path, it is considered relative to the object directory of the project file. When projects are considered recursively, directory hierarchies of tested sources are recreated for each project in their object directories and test packages are placed accordingly.

`--stubs-dir=`*`dirname`*
The hierarchy of directories containing stubbed units is recreated in the `dirname` directory, with stubs placed in directories corresponding to projects they are derived from. If the `dirname` is a relative path, it is considered relative to the object directory of the project file. When projects are considered recursively, directory hierarchies of stubs are recreated for each project in their object directories and test packages are placed accordingly.

`--exclude-from-stubbing=`*`filename`*
Disables stubbing of units listed in `filename`. The file should contain corresponding spec files, one per line.

`--exclude-from-stubbing:`*`spec`*`=`*`filename`*
Same as above, but corresponding units will not be stubbed only when testing unit whose specification is declared in specified `spec` file.

`--validate-type-extensions`
Enables substitution check: run all tests from all parents in order to check substitutability in accordance with the Liskov substitution principle (LSP).

`--inheritance-check`
Enables inheritance check: run inherited tests against descendants.

`--no-inheritance-check`
Disables inheritance check.

`--test-case-only`
Generates test skeletons only for subprograms that have at least one associated pragma or aspect Test_Case.

`--skeleton-default=`*`val`*
Specifies the default behavior of generated skeletons. `val` can be either “fail” or “pass”, “fail” being the default.

`--passed-tests=`*`val`*
Specifies whether or not passed tests should be shown. `val` can be either “show” or “hide”, “show” being the default.

--exit-status=*val*
Specifies whether or not generated test driver should return failure exit status if at least one test fails or crashes. `val` can be either "on" or "off", "off" being the default. If `--exit-status=on` is used to generate the test harness, it should also be used if running the test drivers via the `gnattest test_drivers.list` command.

--omit-sloc
Suppresses comment line containing file name and line number of corresponding subprograms in test skeletons.

--no-command-line
Don't add command line support to test driver. Note that regardless of this switch, `gnattest` will automatically refrain from adding command line support if it detects that the selected run-time doesn't provide this capability.

--test-duration
Adds time measurements for each test in generated test driver.

--reporter=*val*
Use specified reporter in the test driver. `val` is expected to be a name of child package of AUnit.Reporter. Test drivers generated with non-default reporter specified cannot be properly processed by test execution mode of `gnattest`.

`--tests-root`, `--subdirs` and `--tests-dir` switches are mutually exclusive.

CHAPTER

THREE

SWITCHES FOR GNATTEST IN TEST EXECUTION MODE

--passed-tests=*val*
Specifies whether or not passed tests should be shown. `val` can be either "show" or "hide", "show" being the default.

--exit-status=*val*
Specifies whether or not generated test driver should return failure exit status if at least one test fails or crashes. `val` can be either "on" or "off", "off" being the default. The switch `--exit-status=on` should be used both when generating the test harness and when running the test drivers via the `gnattest test_drivers.list` command.

--queues=*n*, -j*n*
Runs `n` tests in parallel (default is 1).

--copy-environment=*dir*
Contents of `dir` directory will be copied to temporary directories created by gnattest in which individual test drivers are spawned.

--subdirs=*dirname*
Test driver executables from `test_drivers.list` are searched in `dirname` subdirectories of specified locations.

CHAPTER

FOUR

PROJECT ATTRIBUTES FOR GNATTEST

Most of the command-line options can also be passed to the tool by adding special attributes to the project file. Those attributes should be put in package `Gnattest`. Here is the list of attributes:

- **Tests_Root**
  is used to select the same output mode as with the `--tests-root` option. This attribute cannot be used together with `Subdir` or `Tests_Dir`.
- **Subdir**
  is used to select the same output mode as with the `--subdirs` option. This attribute cannot be used together with `Tests_Root` or `Tests_Dir`.
- **Tests_Dir**
  is used to select the same output mode as with the `--tests-dir` option. This attribute cannot be used together with `Subdir` or `Tests_Root`.
- **Stubs_Dir**
  is used to select the same output mode as with the `--stubs-dir` option.
- **Harness_Dir**
  is used to specify the directory in which to place harness packages and project file for the test driver, otherwise specified by `--harness-dir`.
- **Additional_Tests**
  is used to specify the project file, otherwise given by `--additional-tests` switch.
- **Skeletons_Default**
  is used to specify the default behaviour of test skeletons, otherwise specified by `--skeleton-default` option. The value of this attribute should be either `pass` or `fail`.
- **Default_Stub_Exclusion_List**
  is used to specify the file with list of units whose bodies should not be stubbed, otherwise specified by `--exclude-from-stubbing=filename`.
- **Stub_Exclusion_List ("spec")**
  is used to specify the file with list of units whose bodies should not be stubbed when testing "spec", otherwise specified by `--exclude-from-stubbing:spec=filename`.

Each of those attributes can be overridden from the command line if needed. Other `gnattest` switches can also be passed via the project file as an attribute list called `Gnattest_Switches`.

CHAPTER

FIVE

SIMPLE EXAMPLE

Let's take a very simple example using the first `gnattest` example located in:

```
<install_prefix>/share/examples/gnattest/simple
```

This project contains a simple package containing one subprogram. By running `gnattest`:

```
$ gnattest --harness-dir=driver -Psimple.gpr
```

a test driver is created in directory `driver`. It can be compiled and run:

```
$ cd obj/driver
$ gprbuild -Ptest_driver
$ test_runner
```

One failed test with the diagnosis "test not implemented" is reported. Since no special output option was specified, the test package `Simple.Tests` is located in:

```
<install_prefix>/share/examples/gnattest/simple/obj/gnattest/tests
```

For each package containing visible subprograms, a child test package is generated. It contains one test routine per tested subprogram. Each declaration of a test subprogram has a comment specifying which tested subprogram it corresponds to. Bodies of test routines are placed in test package bodies and are surrounded by special comment sections. Those comment sections should not be removed or modified in order for gnattest to be able to regenerate test packages and keep already written tests in place. The test routine `Test_Inc_4f8b9f` located at `simple-test_data-tests.adb` contains a single statement: a call to procedure `Assert`. It has two arguments: the Boolean expression we want to check and the diagnosis message to display if the condition is false.

That is where actual testing code should be written after a proper setup. An actual check can be performed by replacing the `Assert` call with:

```
Assert (Inc (1) = 2, "wrong incrementation");
```

After recompiling and running the test driver, one successfully passed test is reported.

CHAPTER

SIX

SETTING UP AND TEARING DOWN THE TESTING ENVIRONMENT

Besides test routines themselves, each test package has a parent package `Test_Data` that has two procedures: `Set_Up` and `Tear_Down`. This package is never overwritten by the tool. `Set_Up` is called before each test routine of the package, and `Tear_Down` is called after each test routine. Those two procedures can be used to perform necessary initialization and finalization, memory allocation, etc. Test type declared in `Test_Data` package is parent type for the test type of test package and can have user-defined components whose values can be set by `Set_Up` routine and used in test routines afterwards.

CHAPTER

SEVEN

REGENERATING TESTS

Bodies of test routines and `Test_Data` packages are never overridden after they have been created once. As long as the name of the subprogram, full expanded Ada names and order of its parameters are the same, and comment sections are intact, the old test routine will fit in its place and no test skeleton will be generated for the subprogram.

This can be demonstrated with the previous example. By uncommenting declaration and body of function Dec in `simple.ads` and `simple.adb`, running `gnattest` on the project, and then running the test driver:

```
$ gnattest --harness-dir=driver -Psimple.gpr
$ cd obj/driver
$ gprbuild -Ptest_driver
$ test_runner
```

The old test is not replaced with a stub, nor is it lost, but a new test skeleton is created for function `Dec`.

The only way of regenerating tests skeletons is to remove the previously created tests together with corresponding comment sections.

CHAPTER

EIGHT

DEFAULT TEST BEHAVIOR

The generated test driver can treat unimplemented tests in two ways: either count them all as failed (this is useful to see which tests are still left to implement) or as passed (to sort out unimplemented ones from those actually failing).

The test driver accepts a switch to specify this behavior: `--skeleton-default=`*`val`*, where `val` is either `pass` or `fail` (exactly as for `gnattest`).

The default behavior of the test driver is set with the same switch as passed to `gnattest` when generating the test driver.

Passing it to the driver generated on the first example:

```
$ test_runner --skeleton-default=pass
```

makes both tests pass, even the unimplemented one.

CHAPTER

NINE

TESTING PRIMITIVE OPERATIONS OF TAGGED TYPES

Creation of test skeletons for primitive operations of tagged types entails a number of features. Test routines for all primitives of a given tagged type are placed in a separate child package named according to the tagged type. For example, if you have tagged type `T` in package `P`, all tests for primitives of `T` will be in `P.T_Test_Data.T_Tests`.

Consider running `gnattest` on the second example (note: actual tests for this example already exist, so there's no need to worry if the tool reports that no new stubs were generated):

```
$ cd <install_prefix>/share/examples/gnattest/tagged_rec
$ gnattest --harness-dir=driver -Ptagged_rec.gpr
```

Taking a closer look at the test type declared in the test package *Speed1.Controller_Test_Data* is necessary. It is declared in:

```
<install_prefix>/share/examples/gnattest/tagged_rec/obj/gnattest/tests
```

Test types are direct or indirect descendants of *AUnit.Test_Fixtures.Test_Fixture* type. In the case of non-primitive tested subprograms, the user doesn't need to be concerned with them. However, when generating test packages for primitive operations, there are some things the user needs to know.

Type `Test_Controller` has components that allow assignment of various derivations of type `Controller`. And if you look at the specification of package *Speed2.Auto_Controller*, you will see that `Test_Auto_Controller` actually derives from `Test_Controller` rather than AUnit type `Test_Fixture`. Thus, test types mirror the hierarchy of tested types.

The `Set_Up` procedure of `Test_Data` package corresponding to a test package of primitive operations of type `T` assigns to `Fixture` a reference to an object of that exact type `T`. Note, however, that if the tagged type has discriminants, the `Set_Up` only has a commented template for setting up the fixture, since filling the discriminant with actual value is up to the user.

The knowledge of the structure of test types allows additional testing without additional effort. Those possibilities are described below.

CHAPTER

TEN

TESTING INHERITANCE

Since the test type hierarchy mimics the hierarchy of tested types, the inheritance of tests takes place. An example of such inheritance can be seen by running the test driver generated for the second example. As previously mentioned, actual tests are already written for this example.

```
$ cd obj/driver
$ gprbuild -Ptest_driver
$ test_runner
```

There are 6 passed tests while there are only 5 testable subprograms. The test routine for function Speed has been inherited and run against objects of the derived type.

CHAPTER

ELEVEN

TAGGED TYPE SUBSTITUTABILITY TESTING

Tagged Type Substitutability Testing is a way of verifying the global type consistency by testing. Global type consistency is a principle stating that if `S` is a subtype of `T` (in Ada, `S` is a derived type of tagged type `T`), then objects of type `T` may be replaced with objects of type `S` (that is, objects of type `S` may be substituted for objects of type `T`), without altering any of the desirable properties of the program. When the properties of the program are expressed in the form of subprogram preconditions and postconditions (let's call them pre and post), the principle is formulated as relations between the pre and post of primitive operations and the pre and post of their derived operations. The pre of a derived operation should not be stronger than the original pre, and the post of the derived operation should not be weaker than the original post. Those relations ensure that verifying if a dispatching call is safe can be done just by using the pre and post of the root operation.

Verifying global type consistency by testing consists of running all the unit tests associated with the primitives of a given tagged type with objects of its derived types.

In the example used in the previous section, there was clearly a violation of type consistency. The overriding primitive `Adjust_Speed` in package `Speed2` removes the functionality of the overridden primitive and thus doesn't respect the consistency principle. `gnattest` has a special option to run overridden parent tests against objects of the type which have overriding primitives:

```
$ gnattest --harness-dir=driver --validate-type-extensions -Ptagged_rec.gpr
$ cd obj/driver
$ gprbuild -Ptest_driver
$ test_runner
```

While all the tests pass by themselves, the parent test for `Adjust_Speed` fails against objects of the derived type.

Non-overridden tests are already inherited for derived test types, so the `--validate-type-extensions` enables the application of overridden tests to objects of derived types.

CHAPTER

TWELVE

TESTING WITH CONTRACTS

`gnattest` supports pragmas `Pre`, `Post`, and `Test_Case`, as well as the corresponding Ada 2012 aspects. Test routines are generated, one per each `Test_Case` associated with a tested subprogram. Those test routines have special wrappers for tested functions that have composition of pre- and postcondition of the subprogram with "requires" and "ensures" of the `Test_Case` (depending on the mode, pre and post either count for `Nominal` mode or do *not* count for `Robustness` mode).

The third example demonstrates how this works:

```
$ cd <install_prefix>/share/examples/gnattest/contracts
$ gnattest --harness-dir=driver -Pcontracts.gpr
```

Putting actual checks within the range of the contract does not cause any error reports. For example, for the test routine which corresponds to test case 1:

```
Assert (Sqrt (9.0) = 3.0, "wrong sqrt");
```

and for the test routine corresponding to test case 2:

```
Assert (Sqrt (-5.0) = -1.0, "wrong error indication");
```

are acceptable:

```
$ cd obj/driver
$ gprbuild -Ptest_driver
$ test_runner
```

However, by changing 9.0 to 25.0 and 3.0 to 5.0, for example, you can get a precondition violation for test case one. Also, by using any otherwise correct but positive pair of numbers in the second test routine, you can also get a precondition violation. Postconditions are checked and reported the same way.

CHAPTER

THIRTEEN

ADDITIONAL TESTS

`gnattest` can add user-written tests to the main suite of the test driver. `gnattest` traverses the given packages and searches for test routines. All procedures with a single in out parameter of a type which is derived from *AUnit.Test_Fixtures.Test_Fixture* and that are declared in package specifications are added to the suites and are then executed by the test driver. (`Set_Up` and `Tear_Down` are filtered out.)

An example illustrates two ways of creating test harnesses for user-written tests. Directory `additional_tests` contains an AUnit-based test driver written by hand.

```
<install_prefix>/share/examples/gnattest/additional_tests/
```

To create a test driver for already-written tests, use the `--harness-only` option:

```
gnattest -Padditional/harness/harness.gpr --harness-dir=harness_only \\
  --harness-only
gprbuild -Pharness_only/test_driver.gpr
harness_only/test_runner
```

Additional tests can also be executed together with generated tests:

```
gnattest -Psimple.gpr --additional-tests=additional/harness/harness.gpr \\
  --harness-dir=mixing
gprbuild -Pmixing/test_driver.gpr
mixing/test_runner
```

CHAPTER

FOURTEEN

INDIVIDUAL TEST DRIVERS

By default, `gnattest` generates a monolithic test driver that aggregates the individual tests into a single executable. It is also possible to generate separate executables for each test or each unit under test, by passing the switch `--separate-drivers` with corresponding parameter. This approach scales better for large testing campaigns, especially involving target architectures with limited resources typical for embedded development. It can also provide a major performance benefit on multi-core systems by allowing simultaneous execution of multiple tests.

`gnattest` can take charge of executing the individual tests; for this, instead of passing a project file, a text file containing the list of executables can be passed. Such a file is automatically generated by gnattest under the name `test_drivers.list`, but it can be hand-edited to add or remove tests, or replaced. The individual tests can also be executed standalone, or from any user-defined scripted framework.

CHAPTER

FIFTEEN

STUBBING

Depending on the testing campaign, it is sometimes necessary to isolate the part of the algorithm under test from its dependencies. This is accomplished via *stubbing*, i.e. replacing the subprograms that are called from the subprogram under test by stand-in subprograms that match the profiles of the original ones, but simply return predetermined values required by the test scenario.

This mode of test harness generation is activated by the switch `--stub`.

The implementation approach chosen by `gnattest` is as follows. For each package under consideration all the packages it is directly depending on are stubbed, excluding the generic packages and package instantiations. The stubs are shared for each package under test. The specs of packages to stub remain intact, while their bodies are replaced, and hide the original bodies by means of extending projects. Also, for each stubbed package, a child package with setter routines for each subprogram declaration is created. These setters are meant to be used to set the behavior of stubbed subprograms from within test cases.

Note that subprograms belonging to the same package as the subprogram under test are not stubbed. This guarantees that the sources being tested are exactly the sources used for production, which is an important property for establishing the traceability between the testing campaign and production code.

Due to the nature of stubbing process, this mode implies the switch `--separate-drivers`, i.e. an individual test driver (with the corresponding hierarchy of extending projects) is generated for each unit under test.

Note: Developing a stubs-based testing campaign requires good understanding of the infrastructure created by `gnattest` for this purpose. We recommend following the two stubbing tutorials `simple_stubbing` and `advanced_stubbing` provided under `<install_prefix>/share/examples/gnattest` before attempting to use this powerful feature.

CHAPTER

SIXTEEN

INTEGRATION WITH GNATCOVERAGE

In addition to the harness, `gnattest` generates a Makefile. This Makefile provides targets for building the test drivers and also the targets for computing the coverage information using GNATcoverage framework when this coverage analysis tool is available. The target `coverage` fully automates the process: it will first build all test drivers, then run them under GNATcoverage, analyze individual trace files, and finally aggregate them:

```
make coverage
```

For more details about using GNATtest with GNATcoverage see *Using GNATtest with GNATcoverage*.

CHAPTER

SEVENTEEN

PUTTING TESTS UNDER VERSION CONTROL

As has been stated earlier, `gnattest` generates two different types of code, test skeletons and harness. The harness is generated completely automatically each time, does not require manual changes and therefore should not be put under version control. It makes sense to put under version control files containing test data packages, both specs and bodies, and files containing bodies of test packages. Note that test package specs are also generated automatically each time and should not be put under version control.

Additionally, if stubbing is enabled with `--stub`, it also makes sense to put the stubbed bodies, as well as the stub-data bodies under source control, as gnattest will preserve modifications made to these files.

Option `--omit-sloc` may be useful when putting test packages under version control.

CHAPTER

EIGHTEEN

AGGREGATE PROJECT HANDLING

If the project passed to `gnattest` with the `-P` switch is an aggregate project, the aggregated projects will be processed sequentially and independently. This will result in one harness directory being generated by default, in the object directories of each of the aggregated projects.

`gnattest` will not generate any project file or makefile to automate the build of the harnesses of each of the aggregated project.

18.1 Artifact directories and aggregate projects

By default, all artifacts generated by `gnattest` are located in subdirectories of the object directory of each of the aggregated projects. This in particular means that tests or stubs for a common dependency of two aggregated projects will be duplicated. In order to avoid this, options such as `--stubs-dir`., `--tests-dir` or `--subdirs` can be used, with **relative paths** so that the artifacts for the common dependencies are generated in the same location, and re-used across each test harness.

On the contrary, using `--harness-dir` or `--tests-dir` with an absolute path will result in the harness and/or files of a first aggregated project being overwritten by the generation of the test harness for subsequent aggregated projects, and should thus be avoided.

CHAPTER

NINETEEN

CURRENT LIMITATIONS

The tool currently has the following limitations:

- generic tests for nested generic packages and their instantiations are not supported;
- tests for protected subprograms and entries are not supported;
- pragma `No_Run_Time` is not supported;
- pragma `No_Secondary_Stack` is not supported;
- if pragmas for interfacing with foreign languages are used, manual adjustments might be necessary to make the test harness compilable;
- use of some constructs, such as elaboration-control pragmas, Type_Invariant aspects, and complex variable initializations that use Subprogram'Access, may result in elaboration circularities in the generated harness;
- heavy usage of preprocessor that affects constructs like subprogram profiles or tagged type hierarchies may result in improper test driver generation.

CHAPTER

TWENTY

AUTOMATICALLY GENERATING TEST CASES (EXPERIMENTAL)

Please note that all the features described bellow are experimental, and the interface is subject to change.

GNATtest has the capability to generate test inputs for subprograms under test. This test generation feature is also usable in conjunction with GNATfuzz, in order to use GNATtest harnesses (generated or manually written) as a starting corpus for a fuzzing session, and to integrate inputs of interest found by GNATfuzz back into the test harness. For more details, see section *Using GNATtest with GNATfuzz (experimental)*.

20.1 Setting up the test generation runtime

Generation of values for Ada cannot be fully done statically, as the bounds of some types may only be defined at runtime. As such, the test generation feature requires the compilation and installation of a runtime project.

The sources for that project are located at `<GNATdas_install_dir/share/tgen/tgen_rts`.

To build the runtime, simply copy the above directory to a location of you choice, build the project using `gprbuild`, install it using `gprinstall` and make it available to the tools by referencing it in the `GPR_PROJECT_PATH` environment variable:

```
# Clean previous source if present
rm -rf /tmp/tgen_rts_src

# Copy the sources
cp -r <GNATdas_install_dir>/share/tgen/tgen_rts /tmp/tgen_rts_src

# Build the project
cd /tmp/tgen_rts_src
gprbuild -P tgen_rts.gpr

# Install the project (removing the previous one if needed)
gprinstall --uninstall -P tgen_rts.gpr --prefix=/tmp/tgen_rts_install
gprinstall -p -P tgen_rts.gpr --prefix=/tmp/tgen_rts_install

# Make it available to other tools
export GPR_PROJECT_PATH=/tmp/tgen_rts_install/share/gpr:$GPR_PROJECT_PATH
```

20.2 Generating test inputs

`gnattest` provides a `--gen-test-vectors` switch that can be used to automatically generate test cases for all of the supported subprogram profiles. The number of generated test cases can be configured through the `--gen-test-num` switch.

`gnattest` can automatically generate test cases unless any of the following are true:

1. Any of the subprogram's "in" or "in out" mode parameters are of an Access type or contain a sub-component of an Access type.
2. Any of the subprogram's "in" or "in out" mode parameters are Subprogram Access Types.
3. Any of the subprogram's "in" or "in out" mode parameters are Limited types.
4. Any of the subprogram's "in" or "out" mode parameters are tagged types.
5. Any of the subprogram's "in" or "out" mode parameters is a private type of a nested package.

Input value generation currently follows a simple strategy for each input parameter of the subprogram under test. Parameters of scalar types, and scalar components of composite types have their values uniformly generated. For unconstrained array types, a length is randomly chosen between 0 and 10 elements, then the low bound is randomly chosen and the high bound computed accordingly to those two first points.

For record discriminants, different strategies are chosen depending on the use of the discriminant within the record: If the discriminant constraints a array component, then the array strategy described above is used. If the discriminant is used in a variant part, generation will be biased in order to generated all possible shapes of the record (i.e. explore all variants). Otherwise, these are generated as any other scalar component.

The generated test cases are then stored in a ad-hoc (and yet to be specified) JSON format, in files under the `<obj_dir>/gnattest/tests/JSON_Tests` directory. The generated JSON files are preserved through a `gnattest` rerun. The user is thus free to modify them, to e.g. fill in expected return values, though backward compatibility of the format is not guaranteed at this stage.

`gnattest` also generates Ada files to actually execute the test cases. Each test vector has its own AUnit test case, and all test cases for a specific subprogram are all stored in a dedicated file, namely `<unit_name>-test_data-test_<subp_name>_<subp_hash>.ad[bs]`, where `<unit_name>` is the name of the unit in which the subprogram is declared, `<subp_name>` is the name of the subprogram, and `<subp_hash>` is a hash based on the profile of the subprogram, in order to differentiate overloads.

The content of these files are re-generated each time `gnattest` is invoked, independently of the presence of the `--gen-test-vectors` switch on the command line. It is thus not necessary to re-invoke `gnattest` with that switch more than once, unless the goal is to generate additional test inputs.

Part III

GNATfuzz User's Guide

CHAPTER

ONE

INTRODUCTION

GNATfuzz provides fuzz-testing facilities for Ada projects. The main features it provides are:

- Identifying subprograms that are suitable for automated fuzz-testing (referred to as *automatically fuzzable* subprograms).
- Generating fuzz-test harnesses that receive binary encoded data, invoke the subprogram under test, and report run-time anomalies.
- Generating fuzz-test initial test cases.
- Building and executing fuzz-testing campaigns controlled by user-defined stopping criteria.
- Real-time test coverage analysis through integration with GNATcoverage.

These features are available both via the command line and through integration with the GNAT Studio IDE.

GNATfuzz supports two fuzzing engines: AFL++ and libfuzzer. Both are popular and highly configurable fuzzers. While AFL++ uses an instrumentation plugin shipped with the GNAT Pro toolchain, the libfuzzer backend leverages GNAT LLVM, an Ada compiler that connects the GNAT front-end to the LLVM code generator.

The support of the libfuzzer engine is currently in beta mode. Please refer to the *Libfuzzer* section.

The primary goal of GNATfuzz is to abstract a lot of the technical details needed to set up, run, and understand the outcome of a fuzzing session on an application.

1.1 Fuzz-testing in a nutshell

Note: In the context of GNATfuzz, we refer to the project or program that you're targeting with fuzz-testing as the "system under test".

Fuzzing (or fuzz-testing) in the context of software is an automated testing technique that, based on some initial input test cases (the corpus) for a program, will automatically and repeatedly run tests and generate new test cases at a very high frequency to detect faulty behavior of the system under test. Such erroneous behavior is captured by monitoring the system for triggered exceptions, failing built-in assertions, and signals such as SIGSEGV (which shouldn't occur in Ada programs unless they're erroneous).

Fuzz-testing is now widely associated with cybersecurity. It has proven to be an effective mechanism for finding corner-case code vulnerabilities that traditional human-driven verification mechanisms, such as unit and integration testing, can miss. Such code vulnerabilities can often lead to malicious exploitations.

Fuzz-testing comes in different flavors. `Black-box` fuzz-testing refers to the approach of randomly generating test cases without taking into account any information about the internal structure, functional behavior, or execution of the system under test. GNATfuzz's fuzzing engines, AFL++ and libFuzer, are considered `grey box` fuzzers because they

obtains code coverage by using code instrumentation. This information helps the mutation and generation phase of the fuzzer generate test cases that have a higher probability of increasing code coverage than randomly generated test cases would. This instrumentation is added automatically by GNATfuzz during the `fuzz` mode, explained in detail later in this guide.

1.2 AFL++ basics

This section contains a high-level overview of AFL++, a highly-configurable, instrumentation-guided genetic fuzzer. A GCC plugin, enabled via the **AFL-GCC-fast** compiler driver directive, generates efficient code instrumentation that enables AFL++ to process GCC-compiled code. Once the system under test is compiled with the AFL++ GCC plugin, you can use the "afl-fuzz" fuzzing driver to run a fuzzing session. The different basic elements and phases involved in an AFL++ fuzzing session are described in Fig. 1

Note: This document is not intended to be a substitute for the AFL++ documentation or that of its GCC plugin extensions, but rather to provide some information needed to understand AFL++ and how it works. In this document, we mostly focus on the added value of the GNATfuzz toolchain. For more information about AFL++, you can visit the AFL++ official site.

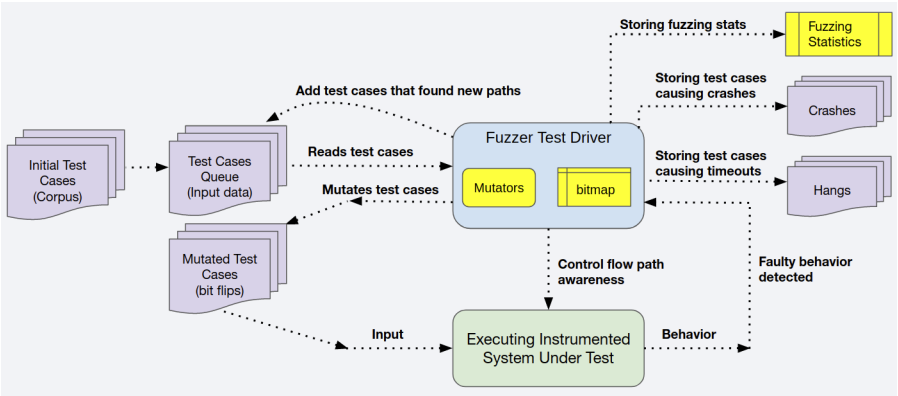


Fig. 1: Overview of an AFL++ fuzz-testing session

Prior to starting a fuzzing session, you must provide some initial test cases, called the **corpus**, each in a separate file. Starting a fuzzing session moves the corpus to what's called the test cases **queue**, which isn't really a queue, but instead a directory containing a list of files, each representing a test case. AFL++ adds tests to the queue, but never removes any from it.

The details are complex, but in general, for each test case in the queue, AFL++ uses one or more internal or external mutators to create multiple new test cases (it can create between hundreds and tens of thousands of new tests from each test in the queue), each of which it executes on the system under test. It monitors each execution for any faulty behavior, as describe above, and moves any test cases that cause such behavior to a directory called **crashes**. AFL++ provides a configuration parameter which sets a timeout for each test. If the test exceeds the timeout, it's killed and stored in the session's **hangs** directory. Saving these tests allows you to replay and investigate these problematic test cases.

While running each test, AFL++ collects all the instrumentation data and updates its **bitmap**, an array global to the session where each index represents an instrumentation point added to the system under test during its compilation by the GCC AFL++ plugin. Whenever an instrumentation point is executed, its corresponding index in the bitmap is incremented. The bitmap enables the fuzzing session to be path-coverage aware.

If a mutated test case executes a new path, it's considered a good candidate to further increase the coverage when mutated and is added to the queue so AFL++ will generate further mutations of that test. The fuzzing session **cycle**

counter is increased when all the test cases in the queue have been through the fuzzing process at least once. The cycle counter is part of the fuzzing statistics that GNATfuzz produces and stores during a session. An example file containing these statistics is shown below:

```
start_time        : 1646044557
last_update       : 1646044557
run_time          : 0
fuzzer_pid        : 305809
cycles_done       : 0
cycles_wo_finds   : 0
execs_done        : 33
execs_per_sec     : 33000.00
execs_ps_last_min : 0.00
corpus_count      : 4
corpus_favored    : 4
corpus_found      : 0
corpus_imported   : 0
corpus_variable   : 0
max_depth         : 1
cur_item          : 0
pending_favs      : 4
pending_total     : 4
stability         : 100.00%
bitmap_cvg        : 1.40%
saved_crashes     : 0
saved_hangs       : 0
last_find         : 0
last_crash        : 0
last_hang         : 0
execs_since_crash : 33
exec_timeout      : 20
slowest_exec_ms   : 0
peak_rss_mb       : 0
cpu_affinity      : 0
edges_found       : 919
total_edges       : 65536
var_byte_count    : 0
havoc_expansion   : 0
auto_dict_entries : 0
testcache_size    : 37
testcache_count   : 1
testcache_evict   : 0
afl_banner        : path_to_executable/fuzz_test_harness.afl_fuzz
afl_version       : ++4.00c
target_mode       : shmem_testcase default
command_line      : afl-fuzz -i/.../path_to/starting_corpus -o /.../path_to/fuzzer_output
                    -D -M gnatfuzz_1_master -- .../path_to/fuzz_test_harness.afl_fuzz
```

Fig. 2: Overview of an AFL++ **fuzzer_stats** file (produced by afl-fuzz++3.00a)

Some of these statistics are also included in the fuzzing session information terminal (which can be optionally shown during the execution of GNATfuzz):

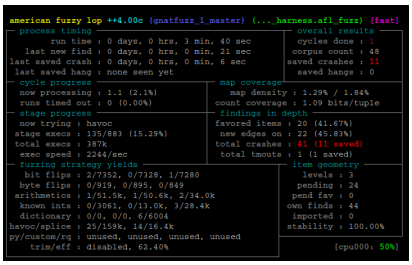


Fig. 3: Fuzzing session information terminal example. (produced by afl-fuzz++3.00a)

Some of the most important statistics are:

- start_time - start time of afl-fuzz (as a unix time)
- last_update - last update of this file (as a unix time)
- fuzzer_pid - PID of the fuzzer process
- cycles_done - queue cycles completed so far
- execs_done - number of execution calls to the system under test
- execs_per_sec - current number of executions per second

- corpus_count - total number of entries in the queue
- corpus_found - number of entries discovered through local fuzzing
- corpus_imported - number of entries imported from other instances
- max_depth - number of levels in the generated data set
- cur_item - entry number currently being processed
- pending_favs - number of favored entries still waiting to be fuzzed
- pending_total - number of all entries waiting to be fuzzed
- stability - percentage of bitmap bytes that behave consistently
- corpus_variable - number of test cases showing variable behavior
- unique_crashes - number of unique crashes recorded
- unique_hangs - number of unique hangs encountered

The complete list of all the statistics generated by AFL++ and the description of each can be found here. These statistics are meant to assist you in making informed decisions on when to stop a fuzzing session. Typically, based on the statistics, you can identify when a fuzzing session has plateaued, meaning that GNATfuzz has no further potential to explore new execution paths in the system under test.

Setting up, running, and stopping a fuzzing session can be challenging. You must build and execute a coverage campaign that allows you to understand the impact of fuzz-testing on your project. There are many steps to this. First, you must construct an adequate starting corpus. The quality of the starting corpus can significantly affect the fuzzing session's results. The more meaningful the starting corpus is as an input to the system under test, the more coverage GNATfuzz can achieve. After providing the corpus, you need to wrap the system under test in a suitable test harness. Next, you have to build the wrapped system with the GCC AFL++ plugin to add instrumentation to the code. Then you need to call the command that starts the fuzzing session with all the required configuration flags and actively monitor it to decide if it has performed enough fuzzing.

AFL++ fuzzing is primarily used to fuzz programs that accept files as inputs. This makes it more challenging to use AFL++ as a form of unit-level vulnerability testing. The current built-in mutators will mutate test cases at the binary level and are unaware of any structure within the test cases. This can be problematic when, for example, it mutates static data that should not be changed, such as an array's upper and lower bounds - their mutation can result in corruption of the test-case data resulting in a wasted mutation cycle.

1.3 GNATfuzz overview

GNATfuzz utilizes several AdaCore technologies to provide ease of use and enhanced fuzzing capability for Ada and SPARK programs. Fuzzing Ada and SPARK code has a significant advantage over fuzzing other traditional, less memory-safe languages, like C. Ada's extended runtime constraint checks can capture faulty behavior, such as overflows, which often go undetected in C. This provides a significant advantage because the software assurance and security confidence that can be achieved when fuzzing Ada and SPARK programs is higher than when fuzzing C programs.

In a nutshell, GNATfuzz abstracts away the complexity of the building, execution, and termination of fuzzing campaigns and enables unit-level-based fuzz-testing for Ada and SPARK programs by:

- Automating the detection of fuzzable subprograms.
- Automating the creation of the test-harness for a fuzzable subprogram.
- Automating the creation and minimization of the starting corpus.
- Allowing for simple specification of criteria for stopping the session and monitoring how those criteria are being met during the session.

- Leveraging the GNATcoverage tool to provide coverage information during or after a fuzzing session.
- Supporting multicore fuzzing.

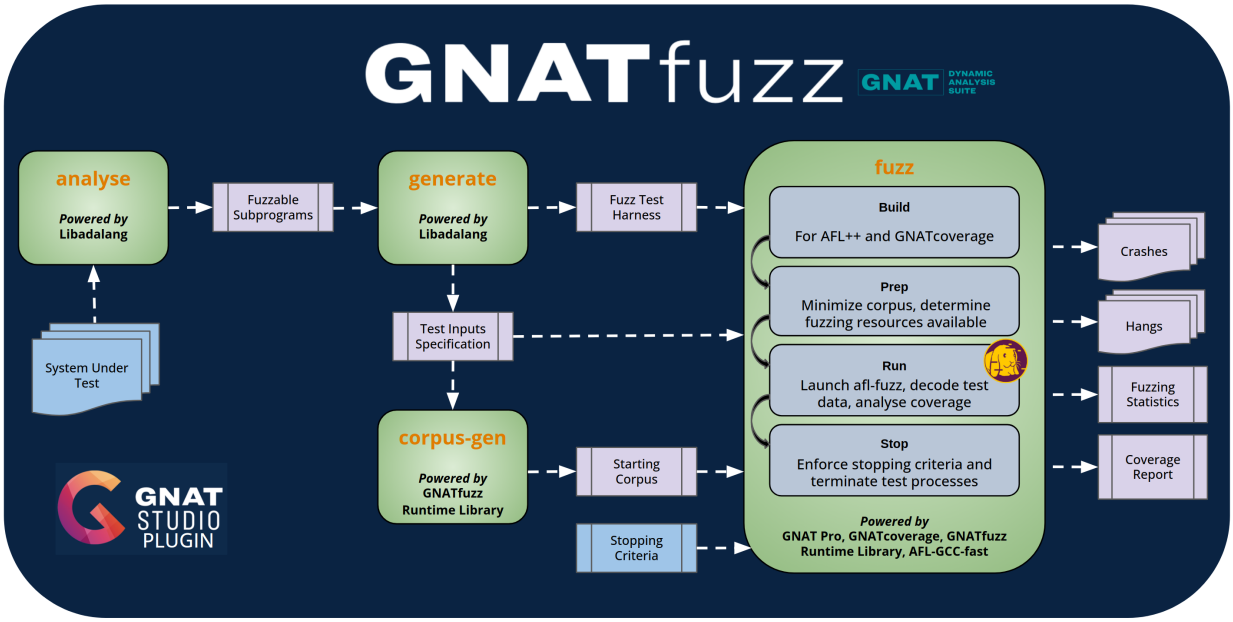


Fig. 4: GNATfuzz overview

Fig. 4 gives an overview of the GNATfuzz toolchain. GNATfuzz offers several modes which you can use to largely automate the manual activities needed for setting up and running a fuzz-test campaign. The green boxes in Fig. 4 represent the four main operating modes, of GNATfuzz: **analyze**, **generate**, **generate-corpus**, and **fuzz**.

The *gnatfuzz* command line interface provides complete access to these features and the IDE integration in GNAT Studio provides a full UI workflow to drive the command line interface.

We start by giving a very high-level description of the GNATfuzz workflow and then provide details about the information footprints of each GNATfuzz mode and how each works in later sections.

The workflow shown in Fig. 4 is intended for unit-level fuzz-testing. In this scenario, you first identify fuzzable subprograms within the system under test. You do this by specifying the project file for the system under test as an argument to GNATfuzz's **analyze** mode. This mode scans the project for all fuzzable subprograms and lists them in a JSON file, including the source file and line information for each. Subprograms considered to be fuzzable are those for which the GNATfuzz can automatically generate a test harness and a starting corpus. See section *GNATfuzz automation capabilities and limitations* for the auto-generation capabilities of the current version of GNATfuzz.

You then select one of these fuzzable subprograms and specify it as an argument to the **generate** mode of GNATfuzz. This mode creates a test harness for that subprogram, including all the execution scripts needed to run the fuzzing campaign and the input-data specifications for the subprogram. Based on these input-data specifications, you can automatically generate the starting corpus using the **generate-corpus** mode. After you run that mode, all the prerequisites needed to execute a fuzz-testing session are in place. The two last steps you might optionally want to perform are creating stopping criteria for the fuzzing campaign and manually providing some additional test cases for the starting corpus.

You can then call the GNATfuzz **fuzz** mode. This command accepts an optional --engines switch to specify the fuzzing engine(s) you want to use. By default, it uses `afl,cmplog`, but you can also specify libfuzzer or a combination like `afl,libfuzzer`.

If you use a configuration that includes AFL++, GNATfuzz will build the system under test with AFL++ code instrumentation, using the **AFL-GCC-fast** compiler plugin. The system under test is also built with support for GNATcov-

erage; see the GNATcoverage User's Guide for more information.

If you use a configuration that includes libfuzzer, GNATfuzz will build the system under test using the GNAT LLVM backend, which provides the necessary instrumentation via the -fsanitize=fuzzer switch.

After the system is built, the **fuzz** mode executes one or more fuzzing sessions for the specified engines. By default, GNATfuzz handles the multicore aspects of the fuzz testing campaign, such as allocating parallel fuzzing sessions of the system under test to available cores. It also periodically synchronizes the test-case queues and coverage reporting and monitors the stopping criteria between all the parallel fuzzing sessions. For details of the internal processes of a test fuzzing session, see Fig. 3.

Finally, all fuzzing activities will stop when the stopping criteria (either specified by you or by default) are met, at which point the final results of the fuzzer are ready for your inspection, including the set of crashing test cases, the set of test cases causing the system to timeout (the hangs), the set of test cases that increased coverage, the coverage report, and the fuzzer statistics.

In the following sections, we first provide detailed information on the various GNATfuzz modes, including their configuration, and then provide examples, first using the GNATfuzz command-line interface and then using the GNATstudio IDE GNATfuzz interface, to illustrate these modes.

CHAPTER

TWO

GNATFUZZ COMMAND LINE INTERFACE

This section describes the various possible parameters for each GNATfuzz mode. To see the detailed effect, the artifacts each switch can generate, and how these artifacts can be fed into the next mode of the GNATfuzz toolchain, see section *GNATfuzz by example*.

Note: For this documentation, the '$' sign denotes the system prompt.

2.1 GNATfuzz *"analyze"* mode command line interface

In this mode GNATfuzz has the following command line interface:

Listing 1: GNATfuzz fuzz-analyze-mode usage

```
 $ gnatfuzz analyze [--help|-h] [-P <PROJECT>] [-S <SPEC-FILE>] [-X <NAME=VALUE>] [--
↪disable-styled-output]
```

where

- **--help**
  Shows the usage information for the analyze mode
- **-P <PROJECT>**
  Specifies the project file to be analyzed Mandatory
- **-S <SPEC-FILE>**
  Targets the analysis at an Ada specification file within the given project
- **-X <NAME=VALUE>**
  Sets the external reference `NAME` in the system-under-test's GPR project to `VALUE`
- **--disable-styled-output**
  By default GNATfuzz may style some of its output using ANSI escape sequences. This switch can be used to disable this behavior.
- **--no-subprojects**
  By default GNATfuzz analyzes sources in all projects, to the exclusion of externally built projects. With this switch, GNATfuzz will only process sources in the root project.

2.2 GNATfuzz *"generate"* mode command line interface

In this mode GNATfuzz has the following command line interface:

Listing 2: GNATfuzz fuzz-generate-mode usage

```
$ gnatfuzz generate [--help|-h] [-P <PROJECT>] [-S <SOURCE-FILE>] [-L <LINE-NUMBER>]
                    [--analysis <ANALYZE-JSON>] [--subprogram-id <SUBPROGRAM-ID>]
                    [-o <OUTPUT-DIRECTORY>] [-X <NAME=VALUE>] [--disable-styled-output]
```

where

- **--help**
  Shows the usage information for the generate mode
- **-P <PROJECT>**
  Specifies the project file to be analyzed Mandatory
- **-S <SOURCE-FILE>**
  Specifies the source file of the subprogram under test Mandatory Requires: -L Mutually Exclusive: --analysis, --subprogram-id
- **-L**
  Line number of the subprogram under test (indexed at 1) Mandatory Requires: -S Mutually Exclusive: --analysis, --subprogram-id
- **--analysis**
  Path to the analysis json generated by the analyze mode Mandatory Requires: --subprogram-id Mutually Exclusive: -S, -L
- **--subprogram-id**
  Id of the target fuzzable subprogram found in the analysis file Mandatory Requires: --analysis Mutually Exclusive: -S, -L
- **-o**
  Path where the generated files are created Default Location: <PROJECT-OBJ-DIR>/gnatfuzz/harness
- **-X <NAME=VALUE>**
  Sets the external reference `NAME` in the system-under-test's GPR project to the value `VALUE`
- **--disable-styled-output**
  By default GNATfuzz may style some of its output using ANSI escape sequences. This switch can be used to disable this behavior.

2.3 GNATfuzz *"generate-corpus"* mode command line interface

In this mode GNATfuzz has the following command line interface:

Listing 3: GNATfuzz generate-corpus-mode usage

```
$ gnatfuzz generate-corpus [--help|-h] [-P <PROJECT>] [-o <output-directory>]
                           [-X <NAME=VALUE>] [--disable-styled-output]
```

where

- **--help**
  Shows the usage information for the generate-corpus mode

- **-P <PROJECT>**
  Specifies the fuzz-test.gpr project file created during a GNATfuzz **generate** mode associated with the fuzzing campaign we want to generate a corpus for Mandatory
- **-o**
  Path where the generated files are created Mandatory
- **-X <NAME=VALUE>**
  Sets the external reference `NAME` in the system-under-test's GPR project to `VALUE`
- **--disable-styled-output**
  By default GNATfuzz may style some of its output using ANSI escape sequences. This switch can be used to disable this behavior.

2.4 GNATfuzz *"minimize-corpus"* mode command line interface

In this mode GNATfuzz has the following command line interface:

Listing 4: GNATfuzz minimize-corpus-mode usage

```
$ gnatfuzz minimize-corpus [--help|-h] [-P <PROJECT>] [-o <output-directory>]
                        [-i <input-directory>]
```

where

- **--help**
  Shows the usage information for the minimize-corpus mode
- **-P <PROJECT>**
  Specifies the fuzz-test.gpr project file created during a GNATfuzz **generate** mode associated with the fuzzing campaign we want to minimize a corpus for Mandatory
- **-i, --input**
  Path to the starting corpus directory to be minimized. If not specified, GNATfuzz will use the corpus generated by the **generate-corpus** mode
- **-o**
  Path to the directory where the minimized corpus will be stored Default Location: <PROJECT-OBJ-DIR>/gnatfuzz/minimized_corpus

2.5 GNATfuzz *"fuzz"* mode command line interface

In this mode GNATfuzz has the following command line interface:

Listing 5: GNATfuzz fuzz-mode usage

```
$ gnatfuzz fuzz [--help|-h] [-P <PROJECT>] [-X <NAME=VALUE>]
                [--engines ENGINES] [--cores CORES]
                [--corpus-path CORPUS-PATH] [--no-cmin] [--seed SEED]
                [--afl-mode AFL-MODE] [--no-deterministic-phase]
                [--no-GNATcov] [--stop-criteria STOP-CRITERIA]
                [--ignore-stop-criteria] [--disable-styled-output]
```

where

- **--help**
  Shows the usage information for the fuzz mode
- **-P <PROJECT>**
  Specifies the fuzz-test.gpr project file created during a GNATfuzz **generate** mode associated with the fuzzing campaign we want to launch Mandatory
- **-X <NAME=VALUE>**
  Sets the external reference `NAME` in the system-under-test's GPR project to `VALUE`
- **--engines <ENGINES>**
  Comma-separated set of fuzzing engines to enable. Choose from: afl, cmplog, symcc, libfuzzer. Default: afl,cmplog
- **--cores**
  The number of cores the fuzzing campaign can utilize. If 0 (also the default if no value is passed), the campaign will utilize the maximum number of available cores Permitted Values: 0<=Value<=255 Default Value: 0
- **--corpus-path**
  Path to the starting corpus to be used
- **--no-cmin**
  Do not run the corpus minimizer tool "afl-cmin"; afl-cmin reduces the starting corpus down to a set of test cases that all find unique paths through the control flow of the system-under-test
- **--seed**
  Use a fixed seed for AFL random number generators. This is mainly used for deterministic benchmarking Permitted Values: 0 to the maximum file size of the starting corpus test case files (in bytes)
- **--afl-mode**
  Specify the AFL_Mode to run Permitted Values: [afl_defer_and_persist | afl_persist | afl_defer | afl_plain] Default Value: afl_persist
- **--no-deterministic-phase**
  Do not run the deterministic (sequential bit flip) mutation phase
- **--no-GNATcov**
  Do not run the (near) real-time dynamic coverage analysis using GNATcoverage (statement)
- **--stop-criteria**
  Override the default stop criteria rules with the provided
- **--ignore-stop-criteria**
  Ignore stop criteria rules (fuzzer must be manually stopped)
- **--disable-styled-output**
  By default GNATfuzz may style some of its output using ANSI escape sequences. This switch can be used to disable this behavior.

CHAPTER

THREE

GNATFUZZ BY EXAMPLE

This section provides several examples to demonstrate the use of the GNATfuzz tool. These examples incrementally increase in complexity and represent the main capabilities of GNATfuzz. You can run these cases on your machine. All examples are located under:

```
<install_prefix>/share/examples/gnatfuzz/
```

3.1 Prerequisites and possible configurations

However, as discussed below, you need to be aware of some AFL++ and GNATfuzz requirements and configurations prior to using the technology.

3.1.1 AFL++ requirements

The AFL++ engine is only supported on Linux.

To allow the AFL++ tool, and therefore GNATfuzz, to run successfully, you need to execute the following commands as root (either by logging in as `root` or running *su* or *sudo*, as appropriate for your system) before running any fuzzing sessions:

```
$ echo core >/proc/sys/kernel/core_pattern
$ cd /sys/devices/system/cpu
$ echo performance | tee cpu*/cpufreq/scaling_governor
```

The first command avoids having crashes misinterpreted as timeouts and the second is required to improve the performance of a fuzz-testing session. For more information, see the relevant AFL++ implementation details.

3.1.2 GNATfuzz requirements

The current requirements for GNATfuzz are:

1. GNATfuzz is currently supported on x86_64 Linux and Windows. All fuzzing engines are supported on Linux, but on Windows, only the libfuzzer engine is available. The Windows support is currently a beta feature.
2. GNATfuzz requires GNAT Pro Ada x86_64. If the libfuzzer or SymCC engines are enabled, GNAT Pro LLVM Ada x86_64 is also required. If your project uses C++, GNAT Pro C++ x86_64 is additionally required.
3. GNATfuzz requires that Ada projects be built using GPR project files and the GPRbuild compilation tool.

Warning: GNATfuzz has no specific hardware requirements but is a highly resource-intensive application. Executing a fuzz-testing session could run thousands of simultaneous tests, which can occupy the majority of the resources of a computing system. Ensure you allocate the number of cores you feel comfortable with to a fuzz-testing session (see section *GNATfuzz "fuzz" mode command line interface*).

3.1.3 GNATfuzz configurations

`GNATFUZZ_XTERM` : by default, the display of the AFL++ terminals (see Screenshot 3) is disabled during a GNATfuzz testing session. To enable their display set this environment variable.

3.1.4 Accommodating linker flags required by the user project

If your project requires specific linker flags or needs to link against a particular library, these must be added to the test harness project file generated by GNATfuzz. This file is located at `path_to_generated_test_harness/fuzz_testing/fuzz_test.gpr`. For instance, if your project includes C++ code and requires linking to the C++ runtime, you'll need to add the linker switch -lstdc++ to the generated test harness project file.

Listing 1: Adding user's project required linker switches to the generated test harness project

```
1 package linker is
2   for Driver use common_driver;
3   --  add any linker switches required by the user project:
4   for Switches ("Ada") use ("-lstdc++");
5 end linker;
```

3.1.5 GNATfuzz and the AFL++ execution modes

AFL++ provides four fuzz-testing execution modes: `plain`, `persistent`, `defer`, and `defer-and-persistent`. You can choose one of these via the command line switch --afl-mode when running GNATfuzz in `fuzz` mode . If you don't specify this switch, GNATfuzz uses the `persistent` mode.

Different AFL++ executions modes cause the GNATfuzz fuzz-testing session to behave in different ways. These behaviors are explained in the following sections.

GNATfuzz and the AFL++ `plain` mode

When GNATfuzz runs in AFL++ `plain` mode, a new process is spawned for the execution of each test. This guarantees that all application memory is reset between tests, however this has a large overhead because, on top of the time required to spawn a new process, all initialization code is rerun for every test case.

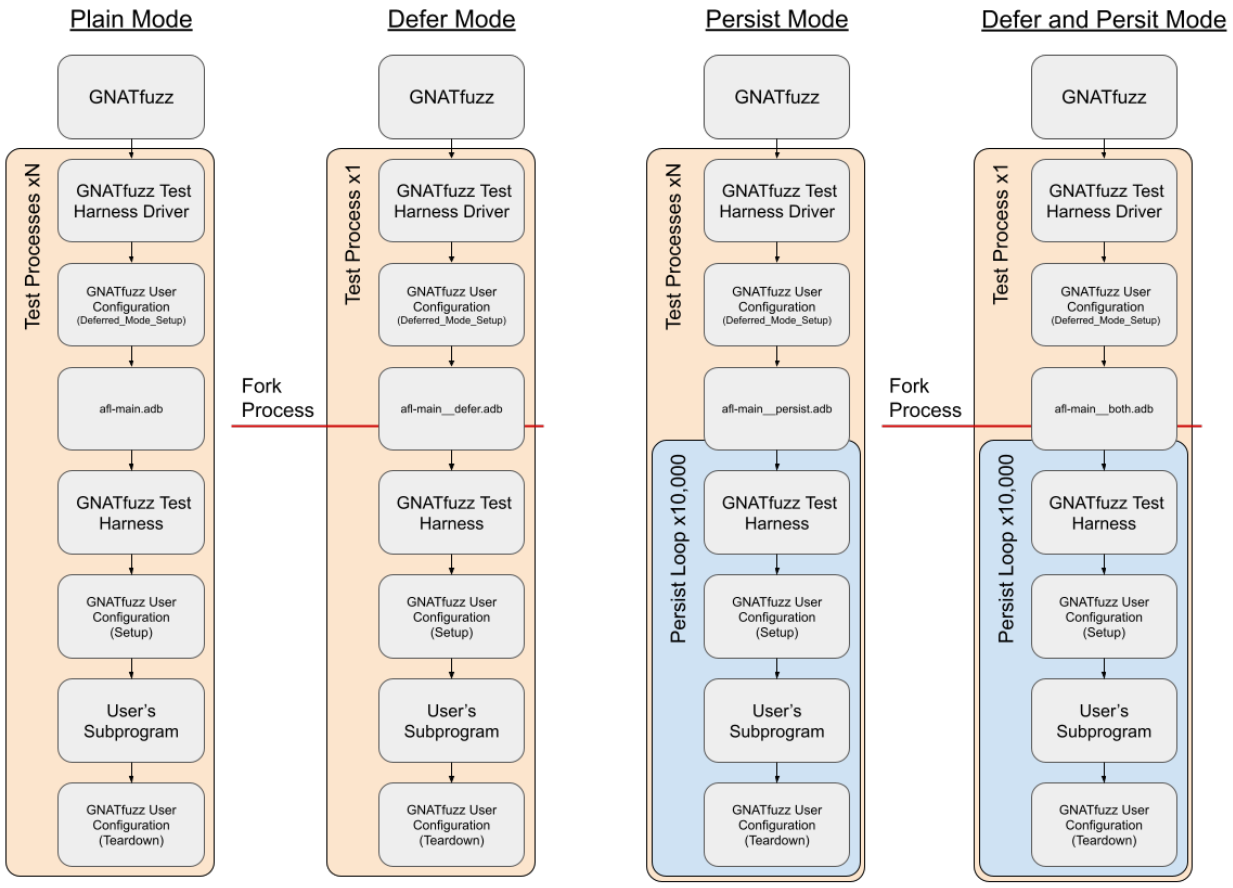


Fig. 1: GNATFuzz AFL++ supported modes overview

GNATfuzz and the AFL++ `persistent` mode

When GNATfuzz runs in AFL++ `persistent` mode, groups of up to 10,000 tests run within the same process before it spawns a new one to run the the next set of tests. This is more efficient, as you only have to pay the time cost of spawning a new process once every 10,000 test cases.

However, as many test cases are run in the same process, any dynamically allocated memory that is modified during execution of a test must be reset, as these changes will otherwise persist while subsequent test cases are executed. Test case failures may not be reproducible in cases where dynamically allocated memory is not fully reset, as the failure could be a result of changes made during previous test runs, instead of depending solely on the data from the current test.

For GNATfuzz, this applies both to any library resources that require you to free memory, as well as to reset any dynamically allocated state, for example, any memory allocated from a pool of available memory (often referred to as a heap) via the Ada keyword `new`.

You can find an example of how to reset dynamically allocated memory in the program under test before each test execution in section *Resetting dynamically allocated memory when fuzz-testing with the AFL++ persistent mode*.

Note: One way to tell if changes are not being fully reset between test runs is to observe the stability value. 100% stability indicates that when the same input data is used, it produces the same result, whereas lower values indicate that this is not always the case. Stability can be monitored through the GNATfuzz stopping criteria, as well as through the AFL++ terminal. If it is not feasible to completely reset state between test runs, consider using the `plain`, `defer`, or `defer_and_persist` modes.

GNATfuzz and the AFL++ `defer` mode

As illustrated in Fig. 1, when running in `defer` mode, only one process is spawned to execute test cases (when running with only one core), and then after some initial setup, this process is forked to run each test case. The benefit of this is that any dynamically allocated memory that is modified during the execution of a test case, is effectively automatically reset before the next test case is run. You do not have to manually clean up after the previous test case.

This is also useful as it allows you to perform large initializations once before the process is forked, and you do not have to worry about re-doing this work between test cases. This can be done using the `Deferred_Mode_Setup` function in the `User_Configuration` package (see section *The "User_Configuration" package*).

`Deferred_Mode_Setup` differs from the `Setup` function in the user configuration package in two ways:

1. It can not depend on test case data
2. When running in `defer` mode, `Deferred_Mode_Setup` will only be called once, whereas `Setup` will be called once per test

That said, in most cases `defer` mode is unlikely to be faster than running in `persist` mode with a optimized set of `Setup` and `Teardown` functions.

Note that if you are running in `plain` mode, `Deferred_Mode_Setup` will be called once per test cases. When running in `persist` mode it will be called once per group of tests (at least every 10,000 test cases).

GNATfuzz and the AFL++ `defer-and-persistent` mode

AFL++ `defer-and-persistent` mode combines the AFL++ `defer` and `persist` modes, as illustrated in Fig. 1. This can be useful in cases where test stability is not 100%, but is close, and so you still want to make use of the performance improvements afforded by the `persist` mode. When running using `defer-and-persistent` mode, a new process is forked to run groups of up to 10,000 test cases, but unlike when just running in `persist` mode, the `Deferred_Mode_Setup` function is only ever called once.

3.2 GNATfuzz automation capabilities and limitations

Section *GNATfuzz overview* provides an overview of the stages you need to use to get a fuzzing session up and running in an automated fashion, each corresponding to running one of the four operating modes of GNATfuzz; namely, **analyze**, **generate**, **generate-corpus**, and **fuzz**.

The degree to which GNATfuzz can automate a testing session is determined mainly by two factors: its ability to auto-generate test harnesses and its ability to auto-generate a starting corpus. We divide subprograms into two categories based on those two factors:

1. **Fuzzable subprograms:** ones for which GNATfuzz can auto-generate both a test harness and a starting corpus.
2. **Non-fuzzable subprograms:** subprograms for which GNATfuzz can neither auto-generate a test harness nor a starting corpus.

Which subprograms belong to each of the two categories will evolve over future releases of GNATfuzz to increase automation.

In the current version, a test harness can be auto-generated for a subprogram unless one of the following are true:

1. Any of the subprogram's "in" or "in out" mode parameters are of an Access type or contain a sub-component of an Access type.
2. Any of the subprogram's "in" or "in out" mode parameters are Subprogram Access Types.
3. Any of the subprogram's "in" or "in out" mode parameters are Limited types.
4. Any of the subprogram's "out" mode parameters are discriminated or variant records with a discriminant without a default value.
5. Any of the subprogram's "in" or "in out" mode parameters types are null records, derive from null records or contain a null record sub-components.
6. Any of the subprogram's "in" or "in out" mode parameters types are tagged types.
7. Any of the subprogram's "in" or "in out" mode parameters types are array types with more than 1000 components.
8. Any of the subprogram's "in" or "in out" mode parameters types is a type composed by another type that that meets any of the rules above.
9. Any of the subprogram's "in" or "in out" mode parameters types is a type with a Static_Predicate aspect.

GNATfuzz can automatically generate a starting corpus for a subprogram unless any of the following are true:

1. A test harness can't be generated for the subprogram (see above).
2. Any of the subprogram's "in" or "in out" mode parameters is an anonymous type or composed by one.
3. The subprogram is a generic procedure or function instantiation.

Section *Fuzzing a simple Ada subprogram* illustrates a case of a fuzzable subprogram, and section *Fuzzing with user-provided seeds* illustrates how you can fuzz a fuzzable subprogram by providing your own seeds via a GNATfuzz-generated interface. Finally, section *Fuzzing instantiations of generic subprograms* illustrates how to fuzz instantiations of generic subprograms.

3.3 Fuzzing a simple Ada subprogram

Let us consider this `simple` project:

Listing 2: simple.adb

```
package body Simple is

   function Is_Divisible (X : Integer; Y : Integer) return Boolean
   is
   begin
      if X mod Y = 0 then
         return true;
      else
         return false;
      end if;
   end Is_Divisible;
end Simple;
```

and:

Listing 3: simple.ads

```
package Simple is

   function Is_Divisible (X : Integer; Y : Integer) return Boolean;
   -- checks if X is divisible by Y

end Simple;
```

This project consists of a single function that takes two integer arguments and returns `true` if the first (`X`) is divisible by the second (`Y`). The code contains a deliberate bug: division by 0 is possible. In a few steps, let's demonstrate how the GNATfuzz toolchain can automatically detect this bug.

3.3.1 Using *analyze* mode

First, we run the GNATfuzz analyze mode on the project:

```
$ gnatfuzz analyze -P simple.gpr
```

Empowered by the *Libadalang* technology, this mode scans the project to detect fuzzable subprograms and generate an analysis file under `<simple-project-obj-dir>/gnatfuzz/analyze.json`, which contains a table listing all the fuzzable subprograms detected. In the case of this project, that file looks like:

Listing 4: analyze.json

```
{
"user_project": "<install_prefix>/share/examples/gnatfuzz/simple/simple.gpr",
"scenario_variables": [],
"fuzzable_subprograms": [
   {
      "source_filename": "<install_prefix>/share/examples/gnatfuzz/simple/src/simple.ads
↪",
```

(continues on next page)

(continued from previous page)

```
        "language": "ada",
        "start_line": 3,
        "start_column": 4,
        "id": 1,
        "label": "Is_Divisible",
    }
],
"non_fuzzable_subprograms": []
}
```

Here we see that GNATfuzz determined that the subprogram declared in the `simple.ads` specification file at line 3 is fuzzable. This corresponds to the `Is_Divisible` function. You can now use the GNATfuzz *generate* mode to autogenerate a test harness suitable for fuzzing.

3.3.2 Using *generate* mode

We can do this in one of two ways.

We can specify the location of the subprogram as given in analyze.json:

```
$ gnatfuzz generate -P simple.gpr -S ./src/simple.ads -L 3 -o generated_test_harness
```

Or we can use the unique identification number (id) for the subprogram as given in the analyze.json file:

```
$ gnatfuzz generate -P simple.gpr --analysis ./obj/gnatfuzz/analyze.json --subprogram-id␣
↪1 -o generated_test_harness
```

Either of these commands generates a new fuzz test-harness project and all the artifacts needed to support a fuzz-test session for the selected subprogram. These are placed under the specified output directory, `generated_test_harness` in our case, with the following directory structure:

Listing 5: Test-harness artifacts

```
├── generate_test_harness
│   ├── fuzz_testing            --  subprogram-specific test-harness artifacts
│   │   ├── build               --  corpus tools
│   │   ├── generated_src       --  test-harness source code
│   │   └── user_configuration  --  user configurable files
│   └── gnatfuzz_shared         --  gnatfuzz lib artifacts
│       ├── ...
│       └── ...
```

In short:

1. `fuzz_testing/fuzz_test.gpr` - the new project file of the test-harness that wraps the subprogram-under-test to enable injecting of test cases, capturing crashes, and coverage analysis.
2. `fuzz_testing/generated_src` - the test-harness source code that you should not modify under normal circumstances.
3. `fuzz_testing/user_configuration` - the test-harness source code that we may modify when needed. For example you may provide its starting corpus (see section *Fuzzing with user-provided seeds*).
4. `fuzz_testing/fuzz_config.json` - contains all the information/configurations needed by the GNATfuzz *fuzz* mode to enable the fuzz-testing of the generated test-harness.

5. `fuzz_testing/build` - includes all the corpus-related tools needed for GNATfuzz to automatically generate the starting corpus as well as the encoding and decoding of test cases.

3.3.3 Using generate-corpus mode

The subprogram under test is suitable for starting corpus auto-generation (see the `"corpus-gen-support": true` field at line 9 of the `<simple-project-obj-dir>/gnatfuzz/analyze.json`). Therefore, we can use the following GNATfuzz command to automatically generate the starting corpus for our fuzzing session:

```
$ gnatfuzz generate-corpus -P ./generated_test_harness/fuzz_testing/fuzz_test.gpr -o␣
↪starting_corpus
```

This generates multiple test_cases under the `starting_corpus` directory. Each is suitable for testing the subprogram chosen using the fuzzing mode. The test cases are in a binary format.

Note: Currently, starting corpus auto-generation support includes all the integral types and arrays of integral types (including strings). There are ongoing efforts to increase the supported types to the full scope of Ada types. Despite this, *GNATfuzz generate* can produce test-harnesses for a more extensive set of subprograms, beyond the ones for which their starting corpus can be auto-generated. GNATfuzz also automatically generates a framework to assist you in manually providing the starting corpus for these cases. We demonstrate this in section *Fuzzing with user-provided seeds*.

Note: The **generate-corpus** mode is currently in its preliminary stages. It includes only one default strategy of auto-generating a starting corpus (using the First, Middle, and Last values for scalar types).

3.3.4 Using *minimize-corpus* mode

The corpus minimization step is optional but recommended for better fuzzing campaigns. Corpus minimization reduces the size of the starting corpus by identifying a minimal subset of test cases that exercise the same code paths, thereby improving the efficiency of the fuzzing campaign.

The following GNATfuzz command minimizes the starting corpus:

```
$ gnatfuzz minimize-corpus -P ./generated_test_harness/fuzz_testing/fuzz_test.gpr -i␣
↪starting_corpus -o minimized_corpus
```

This command:

1. Takes the starting corpus generated by the **generate-corpus** mode (via the `-i` flag)
2. Analyzes the test cases to identify which ones exercise unique execution paths
3. Creates a smaller, optimized corpus in the `minimized_corpus` directory

The minimized corpus can then be used as the starting corpus for the fuzzing session. If the `-i` flag is not provided, GNATfuzz will automatically use the corpus generated by the **generate-corpus** mode. Similarly, if the `-o` flag is not provided, the minimized corpus will be stored in the default location.

Note: Corpus minimization is performed using the **afl-cmin** tool for AFL/CMPLOG engines or the **libFuzzer merge** mode for the libFuzzer engine. The minimization process identifies test cases that trigger unique execution paths in the system-under-test, thereby creating a more efficient corpus for subsequent fuzzing operations.

3.3.5 Decoding an auto-generated test case

To inspect the contents of a generated binary test case, GNATfuzz's decoding utility can be used. Access to this utility can be enabled by setting `GNATFUZZ_BUILD_DECODER` environment variable before the execution of GNATfuzz's `generate` mode. Then, the utility can be found at:

`<simple_example_generated_test_harness_path>/fuzz_testing/build/decode_test`

For example:

```
$ ./decode_test ./starting-corpus/GNATfuzz_1
Parameter: Param 1 [Integer] = -2147483648
Parameter: Param 2 [Integer] = -2147483648
```

Here, the *GNATfuzz_1* auto-generated test case is decoded and its values are printed. *Param 1* and *Param 2* correspond to the *X* and *Y* arguments of the *Is_Divisible* function, respectively.

Note that when decoding a CMPLOG generated test case the switch -CMPLOG can be used with decode_test:

```
$ ./decode_test ./cmplog_used_starting_corpus/GNATfuzz_1 -CMPLOG
```

3.3.6 Using *fuzz* mode

Now everything is in place for fuzz-testing the selected subprogram under test. We can use the following command to launch a fuzzing session:

```
$ gnatfuzz fuzz -P ./generated_test_harness/fuzz_testing/fuzz_test.gpr --corpus-path ./
↪starting_corpus/ --cores=2
```

By default, the `fuzz` command will use the `afl` and `cmplog` engines on Linux, and `libfuzzer` on Windows. The `--engines` switch can be used to defined which fuzzing engines to run.

This command does the following:

1. Builds the test harness.
2. Builds the system-under-test with the fuzzer's instrumentation pass.
3. Builds the system-under-test with GNATcoverage.
4. Minimizes the starting corpus to reduce the starting corpus to the test cases that execute unique paths of the system under test.
5. Launches the fuzz-testing session.
6. Collects dynamic coverage and fuzzing statistics.
7. Monitors the stop criteria and stops the fuzzing session if they're met.
8. Displays status information.

Important: If the optional –corpus-path fuzz mode switch is not used, an internal GNATfuzz temporary directory will be used to place the generated corpus. The generated corpus or its minimized version (if corpus minimization is not disabled) will then be transferred to the `used_starting_corpus` directories shown in Fig. 6. These directories hold the final corpus to be used for starting the fuzz-testing campaign. Furthermore, the default behavior is to execute GNATfuzz's corpus generation. If the user wants only to use their provided starting seeds, either through the –corpus- switch or through the user-configuration package, then GNATfuzz's corpus generation can be disabled through the user's configuration package `User_Provided_Seeds` procedure (see Source Code 12 lines 4 and 5).

You can see the status information displayed in this example during a fuzzing session for this system under test in Fig. 2. GNATfuzz displays various general information points, including the elapsed time since the fuzzing session was started, the number of anomalies detected, information on code coverage based on *GNATcoverage* statement coverage and the progress on meeting the predefined stopping criteria for the session. Both the coverage information and stopping criteria sections can be turned off using the fuzz-mode's --no-GNATcov and --ignore-stop-criteria switches, respectively.

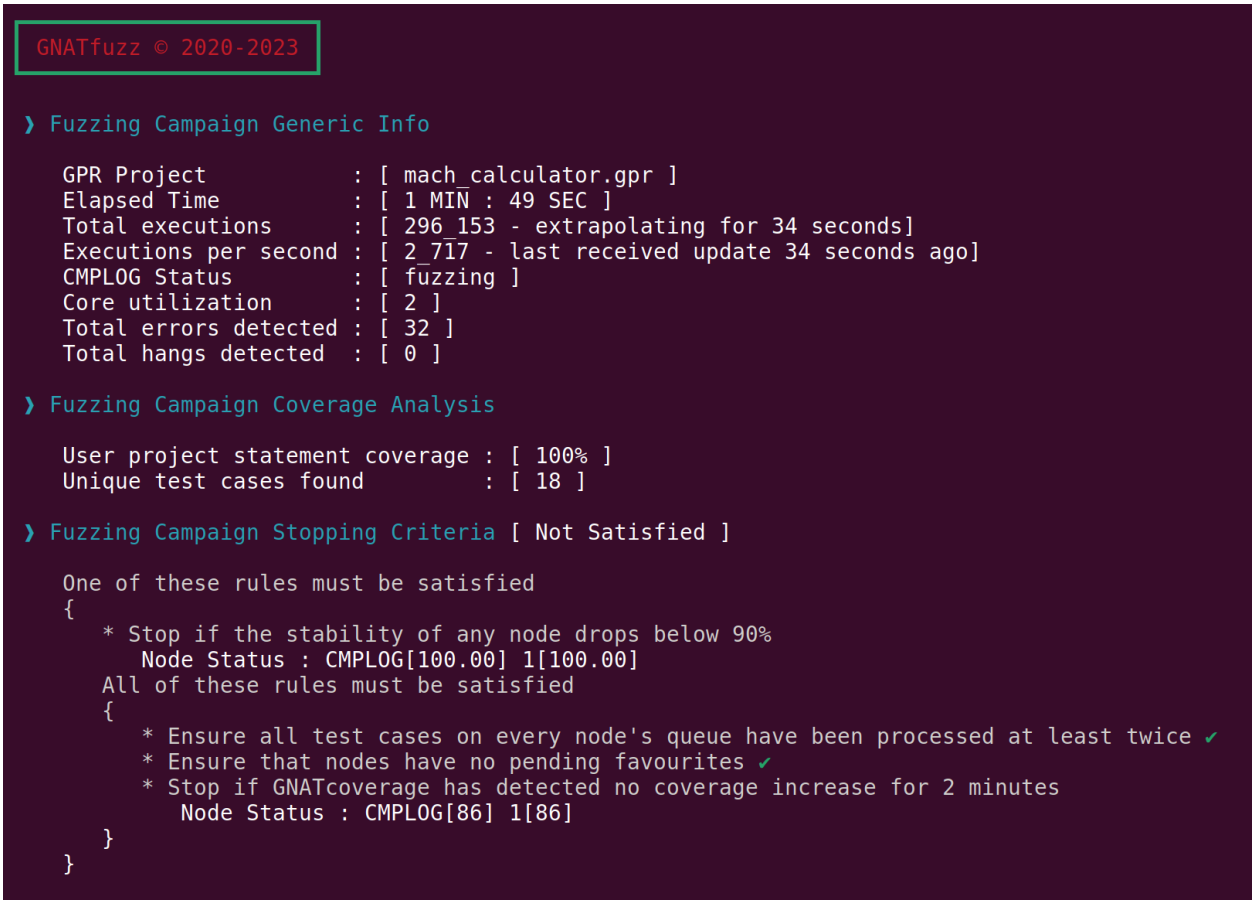


Fig. 2: GNATfuzz fuzzing session status overview

Important: The stopping criteria are defined in *XML* format. GNATfuzz uses a set of default stopping criteria if you don't provide them via the --stop-criteria *fuzz*-mode switch. The *generate* mode also creates a local copy of this default stopping criteria *XML* file within the auto-generated test harness. You can inspect and modify these default criteria and can point to them by specifying the switch --stop-criteria=<path-to-generated-test-harness>/fuzz_testing/user_configuration/stop-criteria.xml. If you want to avoid using the stopping criteria and manually stop a fuzzing session using `ctrl+c` on the terminal, you must use the --ignore-stop-criteria *fuzz*-mode switch. More information on GNATfuzz's stopping criteria can be found in section *GNATfuzz stopping criteria*.

The fuzzing command generates a session directory under `<path-to-generated-test-harness>/fuzz_testing/session:` with the following structure:

Listing 6: Fuzz-session generated artifacts

```
session/
├ afl-fuzz.<N>.log                                    --  The STDOUT from the AFL++ afl-
```

(continues on next page)

(continued from previous page)

```
↪fuzz node executing on core N
 ├ afl-fuzz.<N+1>.log                                  --  The STDOUT from the AFL++ afl-
↪fuzz node executing on core N+1
 ├ fuzz.log                                            --  The diagnostics log file␣
↪generated by GNATfuzz during fuzz mode
 ├ gnatfuzz_stats.log                                  --  The STDOUT from GNATfuzz during␣
↪fuzz mode
 ├── build                                             --  The object-files of various␣
↪compilations needed
 │   ├── obj-<AFL mode>                                --  The build directory of the AFL++␣
↪instrumented test harness
 │   │   ├ gnatfuzz_test_harness.verbose               --  The verbose test harness that␣
↪will print the test results - uses TGen encoding
 │   ├── obj-<AFL mode>_CMPLOG                         --  The build directory of the AFL++␣
↪CMPLOG instrumented test harness
 │   │   └ gnatfuzz-verbose_test_harness               --  The verbose test harness that␣
↪will print the test results - uses CMPLOG encoding
 │   ├── obj-FUZZING_LIBFFUZER                         --  The build directory of the␣
↪libfuzzer instrumented test harness
 │   │   ├ gnatfuzz_test_harness.libfuzzer             --  The libfuzzer test harness driver
 │   └── obj-FUZZING_LIBFFUZER_verbose                 --  The build directory of the␣
↪libfuzzer instrumented verbose test harness
 │       └ gnatfuzz-test_harness.libfuzzer.verbose     --  The verbose libfuzzer test␣
↪harness driver
 ├── coverage_output                                   --  Dynamic coverage reporting
 ├── fuzzer_output                                     --  Fuzzing engines generated␣
↪artifacts
 │   ├── gnatfuzz_1_master                             --  The "master" node's fuzzing-
↪process generated artifacts
 │   │   ├ fuzzer_setup                                --  CL and associated env vars used␣
↪to launch afl-fuzz
 │   │   ├ fuzzer_stats                                --  Latest information about this␣
↪AFL fuzzing node
 │   │   ├── crashes                                   --  Test-cases found to crash the␣
↪system-under-test
 │   │   ├── hangs                                     --  Test-cases found to hang the␣
↪system-under-test
 │   │   └── queue                                     --  Test-cases found to exercise a␣
↪unique execution path
 │   └── gnatfuzz_<2..N>_slave                         --  A "slave" node's fuzzing-process␣
↪generated artifacts
 │       ├ fuzzer_setup                                --  CL and associated env vars used␣
↪to launch afl-fuzz
 │       ├ fuzzer_stats                                --  Latest information about this␣
↪AFL fuzzing node
 │       ├── crashes                                   --  Test-cases found to crash the␣
↪system-under-test
 │       ├── hangs                                     --  Test-cases found to hang the␣
↪system-under-test
 │       └── queue                                     --  Test-cases found to exercise a␣
↪unique execution path
 ├── used_starting_corpus                              --  The starting-corpus used by the␣
```

(continues on next page)

(continued from previous page)

```
↪fuzzing session
 ├── CMPLOG_fuzzer_output                          --  AFL++ CMPLOG generated artifacts␣
↪(same structure as fuzzer_output)
 ├── CMPLOG_used_starting_corpus                   --  The starting-corpus used by the␣
↪CMPLOG fuzzing session
 ├── minimized_starting_corpus                     --  The result of minimizing the␣
↪starting corpus
 ├── symcc_output                                  --  Holds all the SymCC-generated␣
↪test cases
 ├── libfuzzer.log                                 --  The STDOUT of the libfuzzer test␣
↪harness driver
 └── results                                       --  The synchronized results of the␣
↪GNATfuzz fuzz mode
     ├── crashes                                   --  All test-cases found to crash␣
↪the system-under-test
     ├── hangs                                     --  All test-cases found to hang the␣
↪system-under-test
     └── ending_corpus                             --  All test-cases found to exercise␣
↪a unique execution path
```

You should be aware of the following when evaluating or interpreting a fuzzing session:

1. **Multiple fuzzing processes:** The *fuzz* mode switch --cores specifies the total number of fuzzing processes to run across all active engines. Each process is created on an individual core and generates its artifacts directory under the `fuzzer_output` directory.

   - The total number of available cores is allocated among the active engines. **AFL++** and **CMPLOG** (if active) require one full core.
   - The **libfuzzer** engine (if active) requires a minimum of **two cores** for execution. If libfuzzer is the *only* engine active, it will use the number of cores specified by –cores (or default to two, if not specified).
   - For **AFL++**, the first process is always the *master* (e.g., `gnatfuzz_1_master`) and the rest are slave processes.

   The more cores used, the more computing power is allocated to the fuzzing session, and this may speed up coverage and bug finding. However, be cautious not to exhaust your system resources, as this can have a negative impact on fuzzing speed. To monitor the results of a fuzzing session, it is sufficient to inspect only the primary process's artifacts (`gnatfuzz_1_master`), as all secondary processes communicate their findings to the primary process.

2. **Coverage:** The dynamic coverage collected during a fuzzing session is provided in the `coverage_output` directory. This is statement-based coverage using the source-code instrumentation coverage provided by GNATcoverage (see source-traces-based coverage by GNATcoverage). An `index.html` is generated under the coverage directory, which is accessible via a browser. This allows you to navigate the coverage-information annotated code of the system under test. The coverage percentages are currently reported for the entire compilation module and not a subprogram selected from that module for fuzzing. So even though the fuzzing can execute all the statements in the selected subprogram, if the compilation module contains more subprograms than the one chosen for fuzzing, the coverage percentage may never reach 100%. The exception is if the subprogram under test performs calls to all other subprograms. You should keep this in mind when using coverage as a stopping criterion.

3. **Used Starting Corpus:** The *used_starting_corpus* directory of the *session* directory is the starting corpus used by the fuzzing session. This might be the same as the autogenerated or user-provided starting corpus or might be a reduced version if you enable corpus minimization. Note that corpus minimization is currently enabled by default by the fuzz mode of GNATfuzz (use the --no-cmin to disable it.)

4. **AFL++ raw fuzzing stats:** You can inspect the raw AFL++ generated statistics during the fuzzing session by examining the *fuzzer_stats* file under the master's process artifacts directory (*gnatfuzz_1_master*). An example of an AFL++ status file is shown in Fig. 2.

5. **libfuzzer raw fuzzing stats:** You can inspect the raw libfuzzer statistics during during the fuzzing session by examining its log file located in `session/libfuzzer.log`.

When the fuzz-testing session is completed (either because the stopping criteria are satisfied or we stopped it manually), we can check the `results/crashes` directory to examine if the fuzzer was able to capture the expected divide-by-zero bug within the subprogram under test. To inspect the values and the behavior of the crashing test case when executed with the subprogram under test, you can run it with the `gnatfuzz_test_harness.verbose` executable as follows:

Listing 7: Executing found crashes

```
$ cd <path-to-generated-test-harness>/fuzz_testing/session/results
$ ls crashes/
id:000000,sig:06,src:000000,time:64,execs:258,op:flip32,pos:4  README.txt

$  ../build/obj-AFL_PERSIST/gnatfuzz_test_harness.verbose crashes/id\:000000\,sig\:06\,
↪src\:000000\,time\:64\,execs\:258\,op\:flip32\,pos\:4
@@@GNATFUZZ_OUTPUT_START@@@
{
   "Decoded_In_Parameters": [
      {
         "Parameter_Name": "X",
         "Parameter_Type": "Standard.Integer",
         "Parameter_Value": "-1"
      },
      {
         "Parameter_Name": "Y",
         "Parameter_Type": "Standard.Integer",
         "Parameter_Value": " 0"
      }
   ],
   "Subprogram_Under_Test": "Simple.Is_Divisible",
   "Testcase_Exception": {
      "Exception_Information": "raised CONSTRAINT_ERROR : simple.adb:6 divide by zero\n
↪",
      "Exception_Message": "simple.adb:6 divide by zero",
      "Exception_Name": "CONSTRAINT_ERROR"
   }
}
@@@GNATFUZZ_OUTPUT_END@@@
```

As you can see, GNATfuzz produced a single crashing test case (note that the naming of the crashing case can vary). The verbose execution of this crashing test case reveals the cause of the crash: the test case has a value of zero for the divisor (argument `Y` of `Is_Divisible` in Code Sample 3) causing a `CONSTRAINT_ERROR`.

Note: The `gnatfuzz_test_harness.verbose` execution of a crashing test case can be used within a debugger to understand further and correct the erroneous behavior of the subprogram under test.

3.4 Fuzzing with non-automatically generated starting corpus

3.4.1 Fuzzing with user-provided seeds

Let's now try to fuzz the `records` example, which contains the following source:

Listing 8: show_date.ads

```
package Show_Date is

   type Months is
     (January, February, March, April,
      May, June, July, August, September,
      October, November, December);

   type Date is record
      Day   : Integer range 1 .. 31;
      Month : Months;
      Year  : Integer range 1 .. 3000 := 2032;
   end record;

   procedure Display_Date (D : Date);
   --  Displays given date

end Show_Date;
```

Listing 9: show-date.adb

```
with Ada.Text_IO; use Ada.Text_IO;

package body Show_Date is

   procedure Display_Date (D : Date) is
   begin
      Put_Line ("Day:" & Integer'Image (D.Day)
                & ", Month: "
                & Months'Image (D.Month)
                & ", Year:"
                & Integer'Image (D.Year));
   end Display_Date;

end Show_Date;
```

First, we run the GNATfuzz analyze mode on the project:

```
$ gnatfuzz analyze -P records_test.gpr
```

This mode scans the project to detect fuzzable subprograms and generates an analysis file `<records_test-project-obj-dir>/gnatfuzz/analyze.json`, which contains a table listing all the fuzzable subprograms detected. In our case, the analysis information file looks like this:

Listing 10: analyze.json

```
{
  "user_project": "records_test.gpr",
  "scenario_variables": [],
   "fuzzable_subprograms": [
      {
         "source_filename": "<install_prefix>/share/examples/gnatfuzz/records/src/show_
↪date.ads",
         "language": "ada",
         "start_line": 14,
         "start_column": 4,
         "id": 1,
         "label": "Display_Date",
      }
   ],
  "non_fuzzable_subprograms": []
}
```

By inspecting the generated analysis file, we can see that GNATfuzz can auto-generate a test harness and a starting corpus for one fuzzable subprogram, namely, the `Display_Date`. However, we want to use a user-provided seed as a starting corpus.

We then proceed to generate a test harness:

```
$ gnatfuzz generate -P records_test.gpr -S ./src/show_date.ads -L 14 -o generated_test_
↪harness
```

Now, instead of using the corpus generation command, you can use GNATfuzz's generated interface to provide seeds that the tool then uses to generate the starting corpus required for fuzzing:

Listing 11: User's seed interface

```
generated_test_harness/fuzz_testing/user_configuration/
├── show_date-gnatfuzz_user_configuration-add_tgen_json_vectors_to_corpus.adb
├── show_date-gnatfuzz_user_configuration.adb
├── show_date-gnatfuzz_user_configuration.ads
└── stop_criteria.xml
```

The actual package namespace is either a child of `GNATfuzz` or, when available, a child of the package holding the subprogram under test. In this case, the package will be generated as a child package of `Show_Date`.

Now we can see the `User_Provided_Seeds` procedure in `show_date-gnatfuzz_user_configuration.adb`:

Listing 12: User_Provided_Seeds procedure

```
procedure User_Provided_Seeds
  (Add_Corpus_Seed : GNATfuzz.Data_Factory.Add_Corpus_Seed_Type) is
begin
   --  If TGen JSON test vectors have been imported, calling
   --  Add_TGen_JSON_Vectors_To_Corpus will import them into the corpus
   Add_TGen_JSON_Vectors_To_Corpus (Add_Corpus_Seed);

   --  Complete here with calls to Add_Corpus_Seed with each set of seed
   --  values.
```

(continues on next page)

(continued from previous page)

```
   --
   --  For instance, add two calls
   --      Add_Corpus_Seed (A, B);
   --      Add_Corpus_Seed (C, D);
   --  to generate two seeds, one with input values (A, B) and one with
   --  (C, D).

end User_Provided_Seeds;
```

The above procedure allows turning off the default starting corpus auto-generation by uncommenting line number 5. We can now edit this procedure to add seeds for the subprogram under test, as follows:

Listing 13: User_Provided_Seeds procedure

```
procedure User_Provided_Seeds
  (Add_Corpus_Seed : GNATfuzz.Data_Factory.Add_Corpus_Seed_Type) is
begin
   --  If TGen JSON test vectors have been imported, calling
   --  Add_TGen_JSON_Vectors_To_Corpus will import them into the corpus
   Add_TGen_JSON_Vectors_To_Corpus (Add_Corpus_Seed);

   Add_Corpus_Seed (Fuzz_Input_Param_1 => ((24, Show_Date.February, 1978)));
   Add_Corpus_Seed (Fuzz_Input_Param_1 => ((29, Show_Date.March, 1978)));
   Add_Corpus_Seed (Fuzz_Input_Param_1 => ((29, Show_Date.March, 1984)));
   Add_Corpus_Seed (Fuzz_Input_Param_1 => ((11, Show_Date.August, 1994)));
end User_Provided_Seeds;
```

where `Fuzz_Input_Param_1` is the parameter of the subprogram under test, `Display_Date (D : Date)`. Let's now generate the starting corpus from the user-provided seeds:

```
$ gnatfuzz build -P ./generated_test_harness/fuzz_testing/fuzz_test.gpr
$ gnatfuzz generate-corpus -P ./generated_test_harness/fuzz_testing/fuzz_test.gpr --
↪strategy=user -o starting_corpus
```

Note: Note that the `build` mode should be rerun whenever the user-configuration source file is updated. Furthermore, the corpus generation strategy should be set to the `user` option using the –strategy command-line switch.

The generated starting corpus is now located in the designated directory, `starting_corpus`, and ready to be used:

```
$ gnatfuzz fuzz -P ./generated_test_harness/fuzz_testing/fuzz_test.gpr --engines=afl --
↪corpus-path=./starting_corpus/ --cores=1 --stop-criteria=./stop_criteria.xml
```

3.4.2 Fuzzing with a dummy starting corpus

While providing a fuzz-testing campaign with a valid set of starting test cases is considered best practice, GNATfuzz also allows seeding a campaign using "dummy" test cases, a test case file containing random data that may or may not be valid. In such cases, GNATfuzz will use coverage-driven fuzzer mutations, coupled with a highly fuzz-efficient marshaller called "Test Generator" (TGen), to convert invalid test cases into valid ones. This behavior holds even if all test cases in the starting corpus are invalid. Using a dummy corpus is an alternative to user-provided seeds and GNATfuzz's corpus generation and minimization operations.

TGen is AdaCore's dynamic marshalling technology that removes much of the memory padding in test cases (typically added to ensure byte alignment of data components). TGen's compact representation of values significantly increases the probability of successfully mutating a dummy corpus to produce a new valid test case.

For this to work, we must place the dummy corpus file in a standalone directory and pass the path to the directory to gnatfuzz when executing the `fuzz` mode via the –corpus-path switch.

When GNATfuzz is given a random-data test case file as the starting corpus, various GNATfuzz test case generation methods (such as random AFL++ mutations and SymCC concolic execution) use this "dummy" test case as a starting point for generating new ones. These new test cases can execute code that was not reached by any previously provided test cases.

Practical experience has shown that a smaller dummy test case helps the fuzzer to create valid test cases faster. However, any dummy test case file should still work and eventually add a valid test case to the campaign queue. Please note that this approach's success will depend on the complexity of the subprogram's under-test argument types.

The fuzz-everything workflow (see section *Introducing the fuzz-everything workflow*) automates the use of a dummy corpus. Let's consider the following supgrogram:

Listing 14: using_dummy_test_cases.ads

```
1  package Using_Dummy_Test_Cases is
2
3     type My_Private_Type is private;
4
5     procedure A_Subprogram_Not_Supported_For_Corpus_Gen
6       (Param_A : My_Private_Type);
7
8  private
9
10    type Component_A_Type is new Integer;
11
12    type My_Private_Type is record
13       Component_A : Component_A_Type;
14    end record;
15
16 end Using_Dummy_Test_Cases;
```

Executing the following command will run a fuzzing session for the subprogram in the above example, achieving 100% coverage shortly.

```
$ gnatfuzz fuzz-everything test -P using_dummy_test_cases.gpr --corpus-strategy=dummy
```

3.5 Fuzzing instantiations of generic subprograms

For GNATfuzz to test a generic subprogram, we must provide an instantiation. The GNATfuzz `analyze` mode can report the instantiation in its generated analysis file if it consists of a fuzzable subprogram (see section *GNATfuzz automation capabilities and limitations* for what is considered a fuzzable subprogram). When multiple instantiations exist, the analyze mode provides a list of all those that are fuzzable to allow you to select the one to be fuzz-tested. When a chain of instantiations exists, the analyze mode also reports that.

Let's now try to fuzz the `generics` example, which has the following source code:

Listing 15: Sorting_Algorithms.ads

```
generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "<" (L, R : Element_Type) return Boolean is <>;
package Sorting_Algorithms is

   type Array_To_Sort is array (Index_Type) of Element_Type;

   procedure Selection_Sort (X : in out Array_To_Sort);

   procedure Bubble_Sort (X : in out Array_To_Sort);

end Sorting_Algorithms;
```

Listing 16: Sorting_Multiple_Arrays.ads

```
with Sorting_Algorithms;

generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "<" (L, R : Element_Type) return Boolean is <>;
package Sorting_Multiple_Arrays is

   package Sort_Array is new Sorting_Algorithms (Index_Type, Element_Type);

   type Sortable_Matrix is array (Index_Type) of Sort_Array.Array_To_Sort;

   procedure Selection_Sort (Matrix : in out Sortable_Matrix);

   procedure Bubble_Sort (Matrix : in out Sortable_Matrix);

end Sorting_Multiple_Arrays;
```

Listing 17: Sorting_Multiple_Arrays.adb

```
package body Sorting_Multiple_Arrays is

   procedure Selection_Sort (Matrix : in out Sortable_Matrix) is
   begin
      for K in Matrix'Range loop
         Sort_Array.Selection_Sort (Matrix (K));
      end loop;
   end Selection_Sort;

   procedure Bubble_Sort (Matrix : in out Sortable_Matrix) is
   begin
      for K in Matrix'Range loop
         Sort_Array.Bubble_Sort (Matrix (K));
      end loop;
   end Bubble_Sort;

end Sorting_Multiple_Arrays;
```

Listing 18: Instantiations.adb

```
with Sorting_Algorithms;
with Sorting_Multiple_Arrays;

package Instantiations is

   type Range_Array_5 is range 1 .. 5;

   package Sort_Integer_Array_Of_Range_5 is new Sorting_Algorithms
     (Range_Array_5, Integer);

   package Sort_Integer_Arrays_Of_Range_5 is
     new Sorting_Multiple_Arrays (Range_Array_5, Integer);

end Instantiations;
```

We didn't display the `Sorting_Algorithms.adb` file, which implements the two sorting algorithms, Selection_Sort, and Bubble_Sort. You can find this file in the example's source directory.

First, we run the GNATfuzz analyze mode on the project:

```
$ gnatfuzz analyze -P generics.gpr
```

This generates an analysis file generated in the `<generics_test-project-obj-dir>/gnatfuzz/analyze.json` file, which contains a table with all the fuzzable subprograms detected:

Listing 19: analyze.json

```
{
"user_project": "<install_prefix>/share/examples/gnatfuzz/generic/generics.gpr",
"scenario_variables": [],
"fuzzable_subprograms": [
```

(continues on next page)

(continued from previous page)


```
    {
       "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/sorting_
↪algorithms.ads",
       "language": "ada",
       "start_line": 9,
       "start_column": 4,
       "instantiations": [
       {
          "id": 1,
          "label": "Selection_Sort",
          "instantiation_chain": [
             {
             "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/
↪instantiations.ads",
             "language": "ada",
             "start_line": 8,
             "start_column": 4,
             "label": "Sort_Integer_Array_Of_Range_5"
             }
          ],
       },
       {
          "id": 2,
          "label": "Selection_Sort",
          "instantiation_chain": [
             {
             "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/
↪instantiations.ads",
             "language": "ada",
             "start_line": 11,
             "start_column": 4,
             "label": "Sort_Integer_Arrays_Of_Range_5"
             },
             {
             "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/
↪sorting_multiple_arrays.ads",
             "language": "ada",
             "start_line": 9,
             "start_column": 4,
             "label": "Sort_Array"
             }
          ],
       }
       ]
    },
    {
       "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/sorting_
↪algorithms.ads",
       "language": "ada",
       "start_line": 11,
       "start_column": 4,
       "instantiations": [
```


(continues on next page)

(continued from previous page)


```
      {
         "id": 3,
         "label": "Bubble_Sort",
         "instantiation_chain": [
            {
            "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/
↪instantiations.ads",
            "language": "ada",
            "start_line": 8,
            "start_column": 4,
            "label": "Sort_Integer_Array_Of_Range_5"
            }
         ],
      },
      {
         "id": 4,
         "label": "Bubble_Sort",
         "instantiation_chain": [
            {
            "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/
↪instantiations.ads",
            "language": "ada",
            "start_line": 11,
            "start_column": 4,
            "label": "Sort_Integer_Arrays_Of_Range_5"
            },
            {
            "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/
↪sorting_multiple_arrays.ads",
            "language": "ada",
            "start_line": 9,
            "start_column": 4,
            "label": "Sort_Array"
            }
         ],
      }
      ]
   },
   {
      "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/sorting_
↪multiple_arrays.ads",
      "language": "ada",
      "start_line": 13,
      "start_column": 4,
      "instantiations": [
      {
         "id": 5,
         "label": "Selection_Sort",
         "instantiation_chain": [
            {
            "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/
↪instantiations.ads",
```


(continues on next page)

(continued from previous page)

```
            "language": "ada",
            "start_line": 11,
            "start_column": 4,
            "label": "Sort_Integer_Arrays_Of_Range_5"
            }
         ],
      }
      ]
   },
   {
      "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/sorting_
↪multiple_arrays.ads",
      "language": "ada",
      "start_line": 15,
      "start_column": 4,
      "instantiations": [
      {
         "id": 6,
         "label": "Bubble_Sort",
         "instantiation_chain": [
            {
            "source_filename": "<install_prefix>/share/examples/gnatfuzz/generic/src/
↪instantiations.ads",
            "language": "ada",
            "start_line": 11,
            "start_column": 4,
            "label": "Sort_Integer_Arrays_Of_Range_5"
            }
         ],
      }
      ]
   }
],
"non_fuzzable_subprograms": []
}
```

Let's now focus on fuzzing the `Selection_Sort` generic subprogram from the `Sorting_Algorithms.ads` package. In analysis file 19 at line 9, we see a list with two possible instantiations of this generic subprogram, with IDs 1 and 2, respectively. By inspecting the source code, we can see the following two instantiations:

```
1. instantiations.ads:Sort_Integer_Array_Of_Range_5  -> sorting_algorithms.ads:Selection_
↪Sort
2. instantiations.ads:Sort_Integer_Arrays_Of_Range_5 -> sorting_multiple_arrays.ads:Sort_
↪Array
                                                     -> sorting_algorithms.ads:Selection_
↪Sort
```

The first is a direct instantiation of the generic `Selection_Sort` subprogram and the second is an indirect instantiation of the same generic via a chain of instantiations. By using the IDs of the instantiations, we can generate a test-harness for the selected instantiation of the `Selection_Sort` and perform a fuzz-testing session. For example:

```
$ gnatfuzz generate -P generics.gpr --analysis obj/gnatfuzz/analyze.json --subprogram-id␣
```

(continues on next page)

(continued from previous page)

```
↪2 -o generated_test_harness
$ gnatfuzz fuzz -P generated_test_harness/fuzz_testing/fuzz_test.gpr
```

3.6 Fuzzing global and private state

The current version of GNATfuzz doesn't support the automatic generation of test harnesses that can fuzz global and private states, which can also affect a subprogram's execution. Fortunately, there are easy ways to enable the fuzzing of such a state (to understand if it's considered fuzzable, see section *GNATfuzz automation capabilities and limitations*). In this example, we demonstrate a simple way of enabling the fuzzing of global and private variables that affect the execution of a subprogram. Let's consider the `global_and_private_state` example with the following source:

Listing 20: Global_And_Private_State_Fuzzing.ads

```
package Global_And_Private_State_Fuzzing is

   Operation_Increase_Enabled : Boolean := True;

   function Increase_Or_Decrease_Value
     (Value : Integer; Amount : Integer) return Integer;
   --  Based on the "Operation_Increase_Enabled" this function will
   --  either increase or decrease the given "Value" by the given "Amount",
   --  and base on the "Absolute_Value_Enabled", it will return either the
   --  absolute value or not of the selected operation's result

private
   Absolute_Value_Enabled : Boolean := False;

end Global_And_Private_State_Fuzzing;
```

Listing 21: Global_And_Private_State_Fuzzing.adb

```
package body Global_And_Private_State_Fuzzing is

   function Increase_Or_Decrease_Value (Value : Integer; Amount : Integer)
                                        return Integer
   is
      Temp_Value : Integer := 0;
   begin

      if Operation_Increase_Enabled then
         Temp_Value := Value + Amount;
      else
         Temp_Value := Value - Amount;
      end if;

      if Absolute_Value_Enabled then
         return abs Temp_Value;
      end if;

      return Temp_value;
```

(continues on next page)

(continued from previous page)

```
20    end Increase_Or_Decrease_Value;
21
22 end Global_And_Private_State_Fuzzing;
```

The operation performed by the `Increase_Or_Decrease_Value` subprogram is controlled by the global parameter `Operation_Increase_Enabled`. The result of the selected operation is conditionally returned as an absolute value depending on the `Absolute_Value_Enabled` private variable. Let's try to fuzz this subprogram as is:

```
$ gnatfuzz analyze -P global_and_private_state.gpr
$ gnatfuzz generate -P global_and_private_state.gpr -S src/global_and_private_state_
↪fuzzing.ads -L 5 -o generated_test_harness/
$ gnatfuzz fuzz -P generated_test_harness/fuzz_testing/fuzz_test.gpr
```

When we allow the fuzzing session to run for a while, we see that the coverage reaches a stable value and doesn't increase any further:

```
GNATfuzz © 2020-2023

❱ Fuzzing Campaign Generic Info

   GPR Project           : [ global_and_private_state.gpr ]
   Elapsed Time          : [ 1 MIN : 23 SEC ]
   Total executions      : [ 1_758_604 - extrapolating for 0 seconds]
   Executions per second : [ 21_188 - last received update 0 seconds ago]
   CMPLOG Status         : [ fuzzing ]
   Core utilization      : [ 6 ]
   Total errors detected : [ 12 ]
   Total hangs detected  : [ 0 ]

❱ Fuzzing Campaign Coverage Analysis

   User project statement coverage : [ 58% ]
   Unique test cases found         : [ 2 ]

❱ Fuzzing Campaign Stopping Criteria [ Not Satisfied ]

   One of these rules must be satisfied
   {
      * Stop if the stability of any node drops below 90%
         Node Status : CMPLOG[100.00] 1[100.00] 2[100.00] 3[100.00] 4[100.00] 5[100.00]
      All of these rules must be satisfied
      {
         * Ensure all test cases on every node's queue have been processed at least twice ✔
         * Ensure that nodes have no pending favourites ✔
         * Stop if GNATcoverage has detected no coverage increase for 10 minutes
            Node Status : CMPLOG[61] 1[61] 2[61] 3[61] 4[61] 5[61]
      }
   }
```

Fig. 3: Coverage can't reach 100% due to not fuzzing the global and private state that affect the subprogram's execution.

We can also examine the coverage report for the source code of the subprogram under test by opening its HTML format in a browser (firefox in this example):

```
$ firefox generated_test_harness/fuzz_testing/session/coverage_output/global_and_private_
↪state_fuzzing.adb.html
```

Because the GNATfuzz autogenerated test-harness doesn't change the `Operation_Increase_Enabled` global or the `Absolute_Value_Enabled` private variable, we can see from the coverage report in Fig. 4 that the code at line 11 and line 15 is never reached.

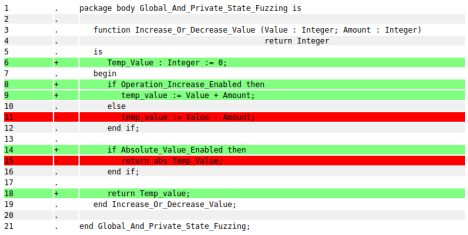


Fig. 4: Coverage report for the `Increase_Or_Decrease_Value` subprogram

We can add the following child package to the `global_and_private_state` project to enable the fuzzing of the `Operation_Increase_Enabled` global and the `Absolute_Value_Enabled` private variable:

Listing 22: Global_And_Private_State_Fuzzing-child.ads

```
package Global_And_Private_State_Fuzzing.Child is

   function Fuzz_Global_And_Private_State
     (Value : Integer; Amount : Integer; Oper_Increase_Enabled : Boolean;
      Absolute_Val_Enabled : Boolean) return Integer;
   --  Wrapper function for the "Increase_Or_Decrease_Value" function
   --  that takes two extra arguments which will allow the fuzzing of
   --  the global and private variables that affect the execution of
   --  the wrapped function.

end Global_And_Private_State_Fuzzing.Child;
```

Listing 23: Global_And_Private_State_Fuzzing-child.adb

```
package body Global_And_Private_State_Fuzzing.Child is

   function Fuzz_Global_And_Private_State
     (Value : Integer; Amount : Integer; Oper_Increase_Enabled : Boolean;
      Absolute_Val_Enabled : Boolean) return Integer
   is
   begin

      Operation_Increase_Enabled := Oper_Increase_Enabled;
      Absolute_Value_Enabled     := Absolute_Val_Enabled;

      return Increase_Or_Decrease_Value (Value, Amount);

   end Fuzz_Global_And_Private_State;

end Global_And_Private_State_Fuzzing.Child;
```

This child package applies a wrapper function, `Fuzz_Global_And_Private_State`, to the function we want to fuzz test. Creating a child of the package containing the subprogram we want to fuzz has two benefits: the child can access the global and private state of the parent and we don't need to modify our original code. The wrapper has the same parameters as the wrapped function, plus two extra: one represents the global and the other the private variable that affect the

wrapped function's execution state. As we can see in Source Code 23 on lines 9 and 10, these extra parameters are used to assign values to the `Operation_Increase_Enabled` global variable and the `Absolute_Value_Enabled` private variable, respectively, and thus can drive their values during fuzz-testing of `Fuzz_Global_And_Private_State`. The wrapped function, `Increase_Or_Decrease_Value`, is called at line 12. This setup now allows the fuzz-testing of the `Increase_Or_Decrease_Value` as before, but it now also drives the global and local state that affects its execution.

Let's now try to fuzz the newly created wrapper function:

```
$ gnatfuzz analyze -P global_and_private_state.gpr
$ gnatfuzz generate -P global_and_private_state.gpr -S src/global_and_private_state_
↪fuzzing-child.ads -L 3 -o generated_test_harness_2/
$ gnatfuzz fuzz -P generated_test_harness_2/fuzz_testing/fuzz_test.gpr
```

```
GNATfuzz © 2020-2023

❱ Fuzzing Campaign Generic Info

   GPR Project           : [ global_and_private_state.gpr ]
   Elapsed Time          : [ 1 MIN : 2 SEC ]
   Total executions      : [ 4_092_000 - extrapolating for 1 seconds]
   Executions per second : [ 66_000 - last received update 1 seconds ago]
   CMPLOG Status         : [ fuzzing ]
   Core utilization      : [ 6 ]
   Total errors detected : [ 30 ]
   Total hangs detected  : [ 0 ]

❱ Fuzzing Campaign Coverage Analysis

   User project statement coverage : [ 100% ]
   Unique test cases found         : [ 8 ]

❱ Fuzzing Campaign Stopping Criteria [ Not Satisfied ]

   One of these rules must be satisfied
   {
      * Stop if the stability of any node drops below 90%
        Node Status : CMPLOG[100.00] 1[100.00] 2[100.00] 3[100.00] 4[100.00] 5[100.00]
      All of these rules must be satisfied
      {
         * Ensure all test cases on every node's queue have been processed at least twice
         * Ensure that nodes have no pending favourites
         * Stop if GNATcoverage has detected no coverage increase for 2 minutes
           Node Status : CMPLOG[41] 1[41] 2[41] 3[41] 4[41] 5[41]
      }
   }
```

Fig. 5: Coverage reaches 100% when global and private variables that affect the execution of the subprogram under test are being fuzzed

We see that the coverage indeed reached 100% in a short time. By inspecting the coverage report generated by GNATcoverage on the source code of the `Increase_Or_Decrease_Value subprogram`, we can see that GNATfuzz now executes all source-code lines of the `Increase_Or_Decrease_Value` subprogram:

```
$ firefox generated_test_harness_2/fuzz_testing/session/coverage_output/global_fuzzing.
↪adb.html
```

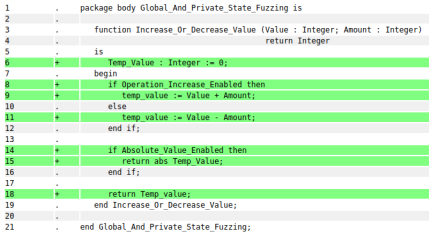


Fig. 6: Coverage report for the `Increase_Or_Decrease_Value` subprogram when fuzzing the private and global variables that affect the subprogram's execution

3.7 Fuzzing Access Types

Currently, GNATfuzz doesn't support the auto-generation of a fuzz-test harness when a subprogram takes as an argument an Access Type of any kind (see section *GNATfuzz automation capabilities and limitations*). Fortunately, we can use the approach used to fuzz-test global and private states (see section *Fuzzing global and private state*) to fuzz test some common forms of Access Types. More specifically, we can do this when the type of the object that the Access Type is granting access to is fuzzable (see section *GNATfuzz automation capabilities and limitations*). We'll use a modified version of the example in section *Fuzzing global and private state*, namely, the `access_types_parameters` example, where we update the parameters of the `Increase_Or_Decrease_Value` function to be *Integer* `access types:

Listing 24: Access_Types_Fuzzing.ads

```
package Access_Types_Fuzzing is

   Operation_Increase_Enabled : Boolean := True;
   type Integer_Access is access Integer;

   function Increase_Or_Decrease_Value
     (Value_Access : Integer_Access; Amount_Access : Integer_Access)
      return Integer;
   --  Based on the "Operation_Increase_Enabled" this function will
   --  either increase or decrease the given "Value" by the given "Amount",
   --  and base on the "Absolute_Value_Enabled", it will return either the
   --  absolute value or not of the selected operation's result

private
   Absolute_Value_Enabled : Boolean := False;

end Access_Types_Fuzzing;
```

Listing 25: Access_Types_Fuzzing.adb

```
package body Access_Types_Fuzzing is

   function Increase_Or_Decrease_Value (Value_Access : Integer_Access;
                                        Amount_Access : Integer_Access)
                                        return Integer
   is
      Temp_Value : Integer := 0;
   begin
      if Operation_Increase_Enabled then
```

(continues on next page)

(continued from previous page)

```
         Temp_Value := Value_Access.all + Amount_Access.all;
      else
         Temp_Value := Value_Access.all - Amount_Access.all;
      end if;

      if Absolute_Value_Enabled then
         return abs Temp_Value;
      end if;

      return Temp_value;
   end Increase_Or_Decrease_Value;

end Access_Types_Fuzzing;
```

By providing a wrapper function to the `Increase_Or_Decrease_Value` function, we enable the auto-generation of a test-harness that allows fuzz-testing of the wrapped function. This is done similarly to the approach used for fuzz-testing global and private variables in section *Fuzzing global and private state*:

Listing 26: Access_Types_Fuzzing-child.ads

```
package Access_Types_Fuzzing.Child is

   function Fuzz_Access_Type_Parameters
     (Value : Integer; Amount : Integer; Oper_Increase_Enabled : Boolean;
      Absolute_Val_Enabled : Boolean) return Integer;
   --  Wrapper function for the "Increase_Or_Decrease_Value" function
   --  that takes two extra arguments which will allow the fuzzing of
   --  the global and private variables that affect the execution of
   --  the wrapped function.

end Access_Types_Fuzzing.Child;
```

Listing 27: Access_Types_Fuzzing-child.adb

```
package body Access_Types_Fuzzing.Child is

   function Fuzz_Access_Type_Parameters
     (Value : Integer; Amount : Integer; Oper_Increase_Enabled : Boolean;
      Absolute_Val_Enabled : Boolean) return Integer
   is
      Value_Access  : constant Integer_Access := new Integer'(Value);
      Amount_Access : constant Integer_Access := new Integer'(Amount);
   begin

      Operation_Increase_Enabled := Oper_Increase_Enabled;
      Absolute_Value_Enabled     := Absolute_Val_Enabled;

      return Increase_Or_Decrease_Value (Value_Access, Amount_Access);
   end Fuzz_Access_Type_Parameters;

end Access_Types_Fuzzing.Child;
```

Now, we can proceed with the standard GNATfuzz modes for fuzz-testing the `Fuzz_Access_Type_Parameters` wrap-

per function:

```
$ gnatfuzz analyze -P access_type_parameter.gpr
$ gnatfuzz generate -P access_type_parameter.gpr -S src/access_types_fuzzing-child.ads -
↪L 3 -o generated_test_harness/
$ gnatfuzz fuzz -P generated_test_harness/fuzz_testing/fuzz_test.gpr
```

Looking at the coverage report generated, we can confirm that the function `Increase_Or_Decrease_Value` was successfully fuzzed:

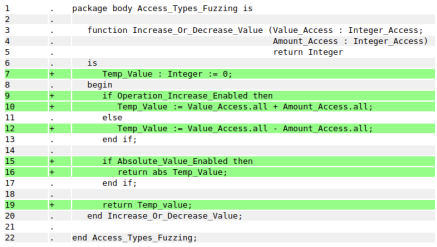


Fig. 7: Coverage report for the `Increase_Or_Decrease_Value` subprogram when fuzzing the private, global variables, and Access Type parameters that affect the subprogram's execution

3.8 Filtering exceptions

A generated GNATfuzz fuzz-test harness captures all exceptions unhandled by the system under test and adds the test cases that caused these exceptions to the `crashes` directory. In some cases, you might want to filter some known and expected exceptions (such as an exception raised by design to notify the calling routine of an error when a library's API is misused) to stop them from being captured as software bugs. The GNATfuzz-generated test-harness provides an easy mechanism to do this.

> **Warning:** You should be careful if you decide to filter out Ada run-time exceptions. Such exceptions, if unhandled, probably indicate software bugs or potential security vulnerabilities.

Let's look at the `filtering_exceptions` example that demonstrates how to tell GNATfuzz not to add a crash to the `crashes` directory when a specific exception is raised. The source code of the example is:

Listing 28: Procedure_Under_Test.ads

```
1  package Procedure_Under_Test is
2
3     Some_Expected_Exception : exception;
4
5     procedure Test (Some_Text : String);
6
7  end Procedure_Under_Test;
```

Listing 29: Procedure_Under_Test.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Procedure_Under_Test is
```

(continues on next page)

(continued from previous page)

```
4
5       procedure Test (Some_Text : String) is
6       begin
7          Put_Line (Some_Text);
8          raise Some_Expected_Exception;
9       end Test;
10
11    end Procedure_Under_Test;
```

As shown in Source Code 29, the procedure `Test` raises an exception at line 8. Let's generate a test harness for this procedure and use it to filter out this exception:

```
$ gnatfuzz analyze -P filtering_exceptions.gpr
$ gnatfuzz generate -P filtering_exceptions.gpr -S src/procedure_under_test.ads -L 5 -o␣
↪generated_test_harness
```

In the `generated_test_harness/fuzz_testing/user_configuration/` directory of the GNATfuzz autogenerated test-harness, there's a file called `procedure_under_test-gnatfuzz_user_configuration.adb`, with the following procedure:

Listing 30: Execute_Subprogram_Under_Test procedure

```
1   procedure Execute_Subprogram_Under_Test
2     (GNATfuzz_Some_Text :  Standard.String)
3   is
4   begin
5
6      --  Call the subprogram under test. The user is not expected to change
7      --  this code.
8      Procedure_Under_Test.Test
9        (Some_Text => GNATfuzz_Some_Text);
10
11     --  Filter out any expected exception types here.
12     --
13     --  Ada run-time exceptions should not be filtered because they probably
14     --  indicate software bugs and potential security vulnerabilities if
15     --  captured. In such a case, GNATfuzz will place the test case
16     --  responsible for raising the exception under the "crashes" directory.
17     --  If an exception is expected, then an exception handler can be added
18     --  here to capture it and allow GNATfuzz to ignore it.
19
20     --  For example,
21     --
22     --  exception
23     --     when Occurrence : Some_Expected_Exception =>
24     --        null;
25
26  end Execute_Subprogram_Under_Test;
```

We can modify this in the following manner:

Listing 31: Execute_Subprogram_Under_Test procedure

```
procedure Execute_Subprogram_Under_Test
  (GNATfuzz_Some_Text :  Standard.String)
is
begin

   --  Call the subprogram under test. The user is not expected to change
   --  this code.
   Procedure_Under_Test.Test
     (Some_Text => GNATfuzz_Some_Text);

exception
   when Occurrence : Procedure_Under_Test.Some_Expected_Exception =>
      null;

end Execute_Subprogram_Under_Test;
```

This modification allows catching the exception before GNATfuzz will record it as a crash. So the test which led to the exception being raised won't be added to the `crashes` directory. By providing the following custom stop criteria, we can confirm the assumption with a fuzz-test session:

Listing 32: stop_criteria.xml

```
<GNATfuzz>
    <afl_stop_criteria>
        <rule_grouper type="and">
            <rule>
                <state>coverage</state>
                <operator>eq</operator>
                <value>100</value>
                <all_sessions>true</all_sessions>
            </rule>
            <rule>
                <state>saved_crashes</state>
                <operator>eq</operator>
                <value>0</value>
                <all_sessions>true</all_sessions>
            </rule>
            <rule>
                <state>run_time</state>
                <operator>gt</operator>
                <value>60</value>
                <all_sessions>true</all_sessions>
            </rule>
        </rule_grouper>
    </afl_stop_criteria>
</GNATfuzz>
```

```
$ gnatfuzz fuzz -P generated_test_harness/fuzz_testing/fuzz_test.gpr --cores=1 --stop-
↪criteria=stop_criteria.xml
```

This fuzz-testing session will stop if 100% coverage of the subprogram under test is achieved and no crashes are found.

This means that line 8 in Source Code 29 will be executed, but the raised exception won't result in the associated test case being added to the `crashes` directory. We also allow the fuzz-testing session to run for one minute to ensure sufficient time was given to place any potential crashing test cases in the `crashes` directory. Our fuzz-testing session exits with this stopping criteria, meaning the raised exception was successfully filtered out.

3.9 Fuzzing C functions through Ada bindings

Currently, GNATfuzz fuzz-testing automation is not supported for C programs. However, it's possible to fuzz-test C functions through Ada bindings.

Let's look at the `testing_C_code_through_Ada_bindings` example to demonstrate how to allow GNATfuzz to fuzz-test a C function using an Ada binding and detect a bug in the C code. The C source code of the example is:

Listing 33: C functions to be fuzz-tested through Ada bindings (calculations.c)

```
1  #include "calculations.h"
2
3  //  simple function performing some basic math operations
4  //  which allows a division by 0
5  int calculations_problematic (int a, int b)
6  {
7    int temp = 0;
8    temp = a * b;
9    temp = temp + a / b;
10   return temp;
11 }
12
13 // same as previous function but prohibits the division
14 // by zero
15 int calculations (int a, int b)
16 {
17   int temp = 0;
18   temp = a * b;
19   if (b != 0)
20      temp = temp + a / b;
21   return temp;
22 }
```

The two functions perform the same simple calculation but the `calculations_problematic` function allows a division by zero. We want to verify that GNATfuzz is able to detect this bug by fuzz-testing the Ada binding for this function.

The Ada bindings for the two C functions are:

Listing 34: Ada bindings for the C functions to be fuzz-tested (procedure_under_test.ads)

```
1  package Procedure_Under_Test is
2
3     function Calculations (A : Integer; B : Integer) return Integer
4       with Import => True,
5       Convention => C,
```

(continues on next page)

(continued from previous page)

```
      External_Name => "calculations";

   function Calculations_Problematic (A : Integer; B : Integer) return Integer
     with Import => True,
     Convention => C,
     External_Name => "calculations_problematic";

   procedure Test
     (Control : Integer;
      In_A    : Integer;
      In_B    : Integer);

end Procedure_Under_Test;
```

A possible driver procedure to test the two C functions could be:

Listing 35: Ada driver subprogram to test the C bindings (procedure_under_test.adb)

```
with Ada.Text_IO; use Ada.Text_IO;

package body Procedure_Under_Test is

   procedure Test
     (Control : Integer;
      In_A    : Integer;
      In_B    : Integer)
   is
      Result : Integer := 0;
   begin

      if Control < 0 then
         Result := Calculations_Problematic (In_A, In_B);
      else
         Result := Calculations (In_A, In_B);
      end if;

      Put_Line (Result'Image);

   end Test;

end Procedure_Under_Test;
```

Note: By default, when using **gprbuild** we only compile Ada source files. To compile C code files as well, we use the Languages attribute in the GPR file and specify c as an option: `for Languages use ("ada", "c");`.

By running the GNATfuzz *analyze* mode:

```
$ gnatfuzz analyze -P testing_C_code.gpr
```

GNATfuzz generates the following analysis file:

Listing 36: analyze.json

```
{
"user_project": "<install_prefix>/share/examples/gnatfuzz/agcf-testing_C_code_through_Ada_bindings/testing_C_code.gpr",
"scenario_variables": [],
"fuzzable_subprograms": [
   {
      "source_filename": "<install_prefix>/share/examples/gnatfuzz/agcf-testing_C_code_through_Ada_bindings/src/procedure_under_test.ads",
      "language": "ada",
      "start_line": 3,
      "start_column": 4,
      "id": 1,
      "label": "Calculations",
   },
   {
      "source_filename": "<install_prefix>/share/examples/gnatfuzz/agcf-testing_C_code_through_Ada_bindings/src/procedure_under_test.ads",
      "language": "ada",
      "start_line": 8,
      "start_column": 4,
      "id": 2,
      "label": "Calculations_Problematic",
   },
   {
      "source_filename": "<install_prefix>/share/examples/gnatfuzz/agcf-testing_C_code_through_Ada_bindings/src/procedure_under_test.ads",
      "language": "ada",
      "start_line": 13,
      "start_column": 4,
      "id": 3,
      "label": "Test",
   }
],
"non_fuzzable_subprograms": []
}
```

Note: As we can see, the Ada bindings for the C functions can also be directly targeted by GNATfuzz for fuzz-testing. For this example, we are using the `Test` procedure as a driver.

We can now use the *generate* and *fuzz* mode to fuzz-test our driver function:

```
$ gnatfuzz generate -P testing_C_code.gpr --analysis obj/gnatfuzz/analyze.json --subprogram-id 3 -o generated_test_harness
$ gnatfuzz fuzz -P generated_test_harness/fuzz_testing/fuzz_test.gpr --cores=1 --stop-criteria=stop_criteria.xml
```

The stop criteria specified only allow the fuzz-testing session to exit when full coverage of the driver function is achieved (implying that both Ada C bindings are called) and at least one test case causing a crash is found (which should be the divide-by-zero allowed by the `calculations_problematic` C function):

Listing 37: stop_criteria.xml

```
<GNATfuzz>
   <afl_stop_criteria>
       <rule_grouper type="and">
           <rule>
               <state>coverage</state>
               <operator>eq</operator>
               <value>100</value>
               <all_sessions>true</all_sessions>
           </rule>
           <rule>
               <state>global_saved_crashes</state>
               <operator>gt</operator>
               <value>0</value>
               <all_sessions>true</all_sessions>
           </rule>
       </rule_grouper>
   </afl_stop_criteria>
</GNATfuzz>
```

By examining the `crashes` directory, we can see that GNATfuzz was able to detect the expected divide-by-zero bug:

Listing 38: Inspecting found crashes

```
$ cd generated_test_harness/fuzz_testing/session/results/
$ ls crashes/
id:000000,sig:06,src:000000,time:10,execs:359,op:flip32,pos:8
id:000001,sig:06,src:000001,time:183,execs:6366,op:int32,pos:4,val:+0

$ ../build/obj-AFL_PERSIST/gnatfuzz-test_harness.verbose /id\:000000\,sig\:06\,src\
↪:000000\,time\:10\,execs\:359\,op\:flip32\,pos\:8
@@@GNATFUZZ_OUTPUT_START@@@
{
"Decoded_In_Parameters": [
   {
      "Parameter_Name": "Control",
      "Parameter_Type": "Standard.Integer",
      "Parameter_Value": "-2147483648"
   },
   {
      "Parameter_Name": "In_A",
      "Parameter_Type": "Standard.Integer",
      "Parameter_Value": "-1"
   },
   {
      "Parameter_Name": "In_B",
      "Parameter_Type": "Standard.Integer",
      "Parameter_Value": " 0"
   }
],
"Subprogram_Under_Test": "Procedure_Under_Test.Test",
   "Testcase_Exception": {
```

(continues on next page)

(continued from previous page)

```
      "Exception_Information": "raised CONSTRAINT_ERROR : s-intman.adb:134 explicit␣
↪raise\n",
      "Exception_Message": "s-intman.adb:134 explicit raise",
      "Exception_Name": "CONSTRAINT_ERROR"
   }
}
@@@GNATFUZZ_OUTPUT_END@@@
```

Note: The test case filenames are not deterministic. The commands shown above are examples that need to be modified with actual filenames generated during the fuzz session.

As we can see, GNATfuzz detected two crashing test cases (note that the names of the cases may vary). By decoding one of the test cases, we can see that the value for the divisor (argument `B` passed to `Calculations_Problematic`) is zero while the value of the *Control* argument ensures the call to the problematic function. We can now use this test case and the `gnatfuzz-test_harness.verbose` executable to debug and correct the C function. Note that arithmetic overflows are also allowed by both C functions, so more crashes can be detected before we get to the division-by-zero crash.

3.10 Resetting dynamically allocated memory when fuzz-testing with the AFL++ `persistent` mode

In section *GNATfuzz and the AFL++ persistent mode*, we explained that `persistent` mode is only suitable for programs whose state can be completely reset between multiple test executions.

Let's now look at the `afl_persist_reset_mem` example that demonstrates how to reset the dynamically allocated memory of the program under test when using the `persistent` execution mode:

Listing 39: Subprogram that will retain state between test executions when fuzz-testing in AFL++ `persistent` mode (test_afl_persist.ads)

```
1  package Test_AFL_Persist is
2
3     Procedure Initialize (Width : Integer; Length : Integer);
4
5  private
6
7     type Floor_Plan_Type is record
8        Width   : Integer;
9        Length  : Integer;
10       Ratio   : Integer;
11    end record;
12
13    type Floor_Plan_Access is access all Floor_Plan_Type;
14
15    Floor_Plan : Floor_Plan_Access;
16
17 end Test_AFL_Persist;
```

Listing 40: Subprogram that will retain state between test executions when fuzz-testing in AFL++ `persistent` mode (test_afl_persist.adb)

```
1  package body Test_AFL_Persist is
2
3     Procedure Initialize (Width : Integer; Length : Integer)
4     is
5        Initialization_Error : exception;
6     begin
7        if Floor_Plan = null then
8           Floor_Plan        := new Floor_Plan_Type;
9           Floor_Plan.Width  := Width;
10          Floor_Plan.Length := Length;
11          Floor_Plan.Ratio  := Width / Length;
12       else
13          raise Initialization_Error with
14            "Error! Object is already initialized";
15       end if;
16    end Initialize;
17
18 end Test_AFL_Persist;
```

The system under test initializes a record containing both the dimensions of a floor and their ratio. The initialization is only supposed to be done once and an exception is raised otherwise. Line 11 of the `Initialize` subprogram is vulnerable to both division-by-zero and overflow. Setting the following stopping criteria stops the fuzz-testing session when two crashes are detected:

Listing 41: Stopping criteria for the `Initialize` subprogram

```
1  <GNATfuzz>
2      <afl_stop_criteria>
3          <rule_grouper type="single">
4              <rule>
5                  <state>saved_crashes</state>
6                  <operator>gt</operator>
7                  <value>1</value>
8                  <all_sessions>true</all_sessions>
9              </rule>
10         </rule_grouper>
11     </afl_stop_criteria>
12 </GNATfuzz>
```

Let's run a fuzz-testing session using the AFL++ `persist` mode by executing the following GNATfuzz commands:

```
$ gnatfuzz analyze -P afl_persist_reset_mem.gpr
$ gnatfuzz generate -P afl_persist_reset_mem.gpr -S ./src/test_afl_persist.ads -L 3 -o ./
↪generated_test_harness
$ gnatfuzz fuzz -P generated_test_harness/fuzz_testing/fuzz_test.gpr --afl-mode=afl_
↪persist --cores=2 --stop-criteria=stop_criteria.xml --no-cmin
```

This results in a failed attempt to launch a fuzz-testing session:

As we can see in Fig. 8, AFL++'s dry-run states that all initial test cases provided to GNATfuzz resulted in crashes. A `dry run` is a phase where AFL++ executes the initial set of test cases multiple times before starting a fuzz-testing

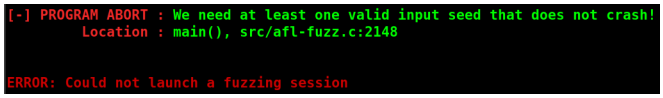


Fig. 8: The fuzz-testing session's dry-run crashes when using AFL++ `persistent` mode without resetting the state of the `Initialize` subprogram

session. During the `dry run` session, stability is calculated by checking if each run of the same test results in the same execution path. If the path differs, the stability drops and a warning is issued in the AFL++ terminal window. For a `dry run` to be successful, at least one of the initial test cases must *always* run to completion without crashing or timing out. If the `dry run` fails, the fuzzing session won't start.

The `Initialize` subprogram under test dynamically allocates memory for the `Floor_Plan` record only when not already allocated by a previous call to `Initialize`. In this example, the first execution of the dry run evaluates the condition on line 7 as True. Line 8 is then executed and the `Floor_Plan` object is created. The second execution evaluates the condition on line 7 as False (i.e. the `Floor_Plan` object is no longer null). This causes the system under test to raise the exception at line 13, which results in a failed `dry run`.

As previously discussed, the `persistent` execution mode retains any program state while executing tests within the same process. To solve this problem, the generated fuzz test-harness requires manual intervention to ensure the memory associated with the `Floor_Plan` record is freed between each test execution. GNATfuzz provides a mechanism you can use to reset dynamically allocated memory before executing each test. To do so, you modify the `TearDown` function within the `User_Configuration` package (see section *The "User_Configuration" package*).

In the `generated_test_harness/fuzz_testing/user_configuration/` directory of the GNATfuzz autogenerated test-harness, there's a file called `test_afl_persist-gnatfuzz_user_configuration.adb`, with the following procedure:

Listing 42: `TearDown` procedure

```
1  procedure Teardown
2    (Width : Standard.Integer;
3     Length : Standard.Integer) is
4  begin
5     --  Complete here with any specialist test deconstruction.
6     --
7     --  For instance, freeing memory.
8     --
9     --  In addition, this subprogram can be used to validate global state
10    --  data or read stored Stub values and check they are as expected.
11    --
12    --  This subprogram can optionally raise exceptions to indicate test
13    --  failure.
14
15    null;
16 end Teardown;
```

We can modify this as follows:

Listing 43: `TearDown` procedure used to reset dynamically allocated memory by the system under test

```
1  procedure Teardown
2    (Width : Standard.Integer;
3     Length : Standard.Integer) is
```

(continues on next page)

(continued from previous page)

```
   procedure Free is new Ada.Unchecked_Deallocation
     (Floor_Plan_Type, Floor_Plan_Access);

begin

   Free (Floor_Plan);

end Teardown;
```

Note that this modification will require adding the statement `with Ada.Unchecked_Deallocation;` at the top of the file. This modification allows you to free memory that was dynamically allocated for the `Floor_Plan` record before GNATfuzz executes the next test. By using the custom stop criteria given in Fig. 41, we can launch a new fuzz-testing campaign using the AFL++ `persistent` mode:

```
$ gnatfuzz fuzz -P generated_test_harness/fuzz_testing/fuzz_test.gpr --cores=1 --afl-
↪mode=afl_persist --stop-criteria=stop_criteria.xml --no-cmin
```

This time, the fuzz-testing session launches and completes successfully:

```
 GNATfuzz © 2020-2023

❱ Fuzzing Campaign Generic Info

   GPR Project           : [ afl_persist_reset_mem.gpr ]
   Elapsed Time          : [ 1 MIN : 13 SEC ]
   Total executions      : [ 1_428_099 - extrapolating for 0 seconds]
   Executions per second : [ 19_563 - last received update 0 seconds ago]
   CMPLOG Status         : [ fuzzing ]
   Core utilization      : [ 2 ]
   Total errors detected : [ 8 ]
   Total hangs detected  : [ 0 ]

❱ Fuzzing Campaign Coverage Analysis

   User project statement coverage : [ 87% ]
   Unique test cases found         : [ 12 ]

❱ Fuzzing Campaign Stopping Criteria [ Satisfied ]

   One of these rules must be satisfied
   {
      * All nodes saved_crashes > 1 ✔
        Node Status : CMPLOG[3] 1[3]
   }
```

Fig. 9: The fuzz-testing session runs and completes successfully when using AFL++ `persistent` mode and resetting the state of the `Initialize` subprogram

3.11 CMPLOG (aka RedQueen) support

CMPLOG is an AFL++ GCC plugin that supports the fuzzer in achieving greater code coverage by exploiting the **input-to-state correspondence** relationship that many programs exhibit. The technique leverages the fact that during the execution of a given program, parts of the input often end up directly (i.e., almost unmodified) in the program state (see related publication for more info). CMPLOG extracts such data that can affect the program's state by dynamically analyzing the code and feeds them to the fuzzer in the form of new test cases that are added to the fuzzer's queue. This can be particularly useful in the instances of magic bytes and (nested) checksum tests. The following function shows a simplified case where CMPLOG can help enter into tight branches whose conditions are based on magic numbers.

Listing 44: Typical scenario that CMPLOG can be useful

```
function CMPLOG_Test
  (Control : in Float) return Boolean
is
begin
   if Control = 1_234_324.45 then
      return True;
   else
      return False;
   end if;
end CMPLOG_Test;
```

In the above example, the fuzzer will need to mutate an existing test case and generate a new test case for the parameter of the `CMPLOG_Test` function that will include the exact value of "1_234_324.45" to get into the branch. The probability of such a mutation is extremely low. CMPLOG, on the other hand, will extract this number by a dynamic analysis stage and try to inject it into a new test case with a CMPLOG-custom mutation. Such a test case will allow the fuzzer to enter the branch as an input-to-state correspondence exists.

Note that CMPLOG feature is enabled by default by the GNATfuzz fuzz mode. A CMPLOG fuzzing campaign runs on a dedicate core. Therefore, in order to run GNATfuzz with CMPLOG at least two cores must be available (can be set by the `--cores` GNATfuzz fuzz mode option).

Also, note that CMPLOG is incompatible with AFL modes `afl_defer` and `afl_defer_and_persist`. Therefore, to use CMPLOG, please ensure either AFL mode `afl_persist` (the default) or `afl_plain` are selected via the –afl-mode switch.

CHAPTER

FOUR

THE "USER_CONFIGURATION" PACKAGE

As we've seen in a previous examples, the GNATfuzz auto-generated test-harness provides a `User_Configuration` package (for example, see Source Code 11). The actual package namespace is either a child of GNATfuzz or, when available, a child of the package holding the subprogram under test. In the latter case, the `User_Configuration` package has access to the global and local state of the package that contains the subprogram under test.

The `User_Configuration` package offers several useful subprograms that you can modify to provide custom functionality for a fuzz-testing session:

1. `Setup` function: This function executes before each test is executed. You can use it to set up any specialized test initialization, such as setting *Stub* return values or assigning values to the global state. Also, you can use the return value of this function to indicate whether the setup of a test was successful or that the test should be skipped. In addition, the "in" parameters of the subprogram under test are passed into this function as "in out" parameters, so you can alter their values before running the test. You should be careful to only use this this feature when necessary, but auto-generated test inputs sometimes require modifications in order to make the test valid. For example, if you have a record containing a checksum, any mutation of the data will invalidate the checksum, so you should add code to this function to recalculate it.
2. `Teardown` procedure: This procedure is called after each test instance is executed. You can use it to add any specialized finalizational functionality, such as freeing memory and also use it to validate global state data or read stored *Stub* values and check that they are as expected. The `Teardown` subprogram can also optionally raise exceptions to indicate test failure.
3. `User_Provided_Seeds` procedure: You can use this procedure to provide custom seed values, which GNATfuzz uses to create a starting corpus and to control if a default starting corpus should also be autogenerated (see section *Fuzzing with user-provided seeds*).
4. `Add_TGen_JSON_Vectors_To_Corpus` separate: This separate procedure allows importing TGen JSON test vectors into the starting corpus for incremental fuzzing workflows. The **import-tgen-vectors** command automatically generates this separate body (see section *Incremental fuzzing with TGen test vectors*).
5. `Execute_Subprogram_Under_Test` procedure: This is the procedure that calls the subprogram under test. You can use it to filter out any expected exceptions (see section *Filtering exceptions*).

CHAPTER

FIVE

GNATFUZZ STOPPING CRITERIA

Through the provided fuzz-test examples, we demonstrated the use of the stopping criteria to provide a predefined set of rules on when a fuzz-testing campaign should be stopped.

The stopping criteria are defined in *XML* format. If you don't provide one via the --stop-criteria *fuzz*-mode switch, GNATfuzz uses a set of default stopping criteria . The *generate* mode also creates a local copy of this default stopping criteria as an *XML* file within the autogenerated test harness. You can inspect and modify these default criteria and point to them by specifying the switch:

```
--stop-criteria=<path-to-generated-test-harness>/fuzz_testing/
user_configuration/stop-criteria.xml
```

Listing 1: GNATfuzz default stopping criteria

```
<GNATfuzz>
    <afl_stop_criteria>
        <rule_grouper type="and">
            <rule>
                <state>cycles_done</state>
                <operator>gt</operator>
                <value>2</value>
                <all_sessions>true</all_sessions>
            </rule>
            <rule>
                <state>pending_favs</state>
                <operator>eq</operator>
                <value>0</value>
                <all_sessions>true</all_sessions>
            </rule>
            <rule>
                <arithmetic_state>
                    <state1>current_time</state1>
                    <state2>last_find</state2>
                    <arithmetic_operator>minus</arithmetic_operator>
                </arithmetic_state>
                <operator>gt</operator>
                <value>360</value>
                <all_sessions>true</all_sessions>
            </rule>
        </rule_grouper>
    </afl_stop_criteria>
</GNATfuzz>
```

Fig. 1 shows the default stopping criteria generated by GNATfuzz. These default stopping criteria specify that a fuzzing session will only stop if:

1. All entries within the test queue are processed at least three times, and
2. There are no "Pending favorites" test cases left (promising test cases for finding new execution paths), and
3. GNATfuzz doesn't find a new path for 6 minutes.

The goal of this set of default stop-criteria rules is to allow sufficient time for obtaining a good snapshot of what GNATfuzz can achieve when testing a subprogram. Small subprograms tend to get the maximum possible code coverage in very little time. Note that the provided stopping criteria offers no guarantees that GNATfuzz will reach its full potential when testing a program. They are only intended as a good starting point for you to modify as needed, e.g., by increasing the time when the last new path is found to allow the fuzzer to exercise more potential corner cases. To see all the possible stop-criteria options, you can view the *XML* relevant schema file that can be found under `<install_prefix>/share/gnatfuzz/auxiliary/config.xsd`.

If you want to avoid using any stopping criteria and manually stop a fuzzing session using `ctrl+c` from the terminal, you must pass the --ignore-stop-criteria *fuzz*-mode switch.

CHAPTER

SIX

GNATFUZZ INTEGRATION IN GNAT STUDIO

Recent versions of GNAT Studio provide an integration with GNATfuzz.

The integration is activated automatically when GNAT Studio finds the command line executable `gnatfuzz` on the PATH.

6.1 The GNATfuzz menu

You can access some GNATfuzz operations through the top level **GNATfuzz** menu. On a regular project, the first two entries can be used to launch an analysis on the code:

- **Analyze Project**: This launches GNATfuzz in *analyze* mode on the entire project.
- **Analyze File**: This launches GNATfuzz in *analyze* mode on the current file only.

When the project loaded in GNAT Studio is a GNATfuzz test-harness project (determined by the presence of the `fuzz_config.json` file next to the project file), it provides additional menu entries:

- **Start/Stop Fuzzing session**: Use this to launch a fuzzing session or stop the current fuzzing session if one is already running.
- **Switch to User Project**: Use this to unload the test-harness project from GNAT Studio and load the original project.

6.2 Viewing analysis results and generating a fuzz-testing harness

The results of GNATfuzz in *analyze* mode, i.e., the subprograms that are identified as automatically fuzzable, are listed in the *Locations* view. Each entry is decorated with an icon you can click to launch GNATfuzz in *generate* mode on the corresponding subprogram and generate a fuzz-test harness project.

You can fine-tune the switches passed to the command line in the intermediate dialog. By default, GNATfuzz will generate the harness project in a `fuzz_testing` directory inside the object directory of the root project.

If generation is successful, GNAT Studio presents a dialog allowing you to load the generated test-harness project.

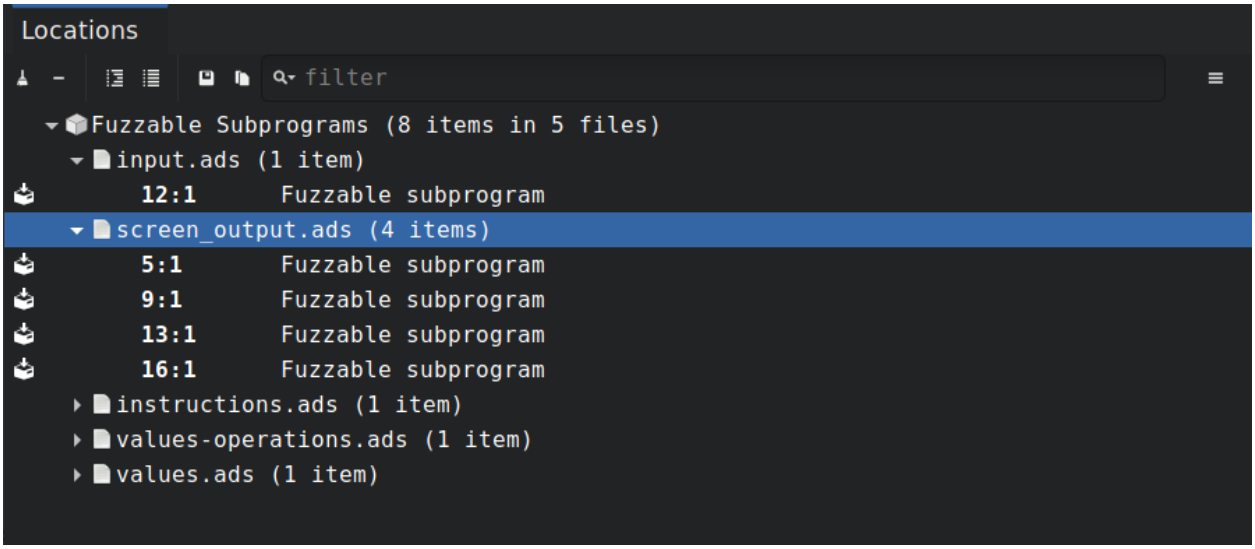


Fig. 1: The *Locations* view showing the results of gnatfuzz analyze

6.3 Running a fuzz session

The first thing you might want to do before running a fuzz session is to add your seeds to pass to the corpus generator phase. This is done by editing the subprogram `User_Provided_Seeds` in `user_configuration/<>-gnatfuzz_user_configuration.adb`. You might want to edit other elements in this package to adjust the test-harness to your code.

When the fuzzing session is running, you'll see the progress in the *Messages* view - the same output you'd see when running `gnatfuzz` interactively in the command line.

GNAT Studio reads the coverage information obtained by `gnatfuzz` during the session, displays a summary in the Coverage report, and annotates source files in the system under test with coverage information, showing which lines have been successfully explored by the fuzz-testing session. This information is cumulative and shows the coverage collectively gained by multiple individual runs of the executable under GNATfuzz.

6.4 Viewing and debugging fuzz crashes

The *Fuzz Crashes* view shows the crashes and hangs found during the fuzzing session.

Each row in this view corresponds to one individual test case that causes the subprogram under test to crash or hang. You can expand a row to view the parameters to the subprogram being tested that correspond to this crash or hang.

You can also launch a debugging session to place yourself at the crash point: you do this by double-clicking on a row. GNAT Studio launches the fuzzed executable in the debugger and runs it with the seed that caused the hang or crash. The debugger stops on exceptions, so you might have to manually execute past any expected exception that might occur prior to the one causing the crash.

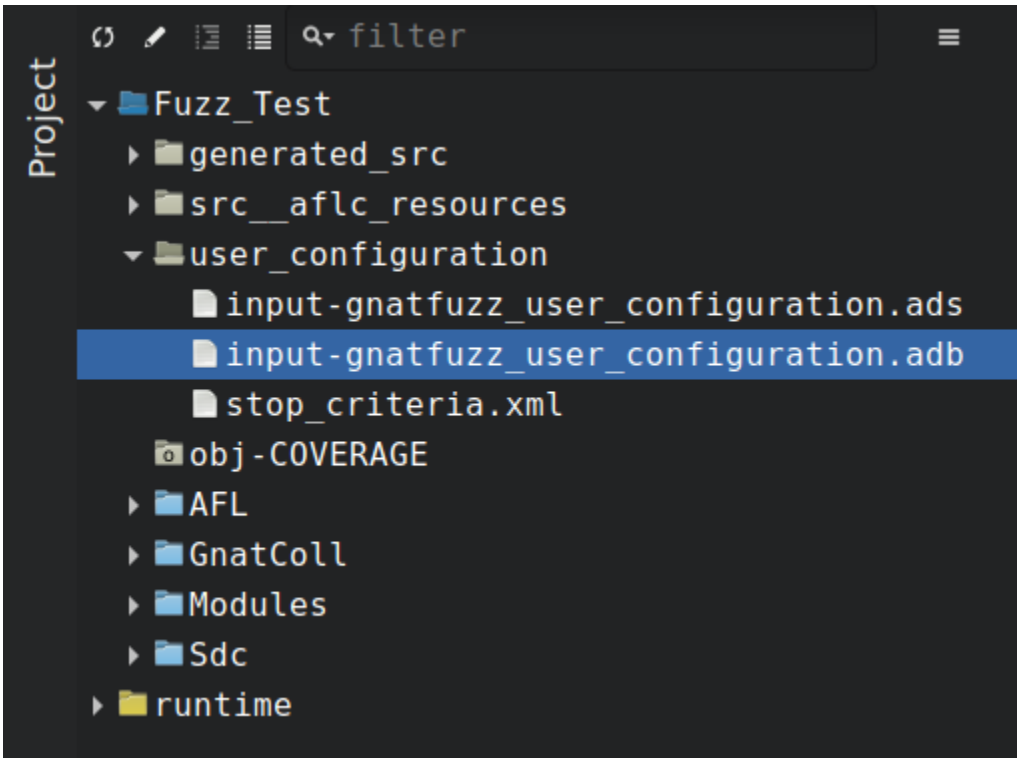


Fig. 2: The *Project* view showing a fuzz-test harness project

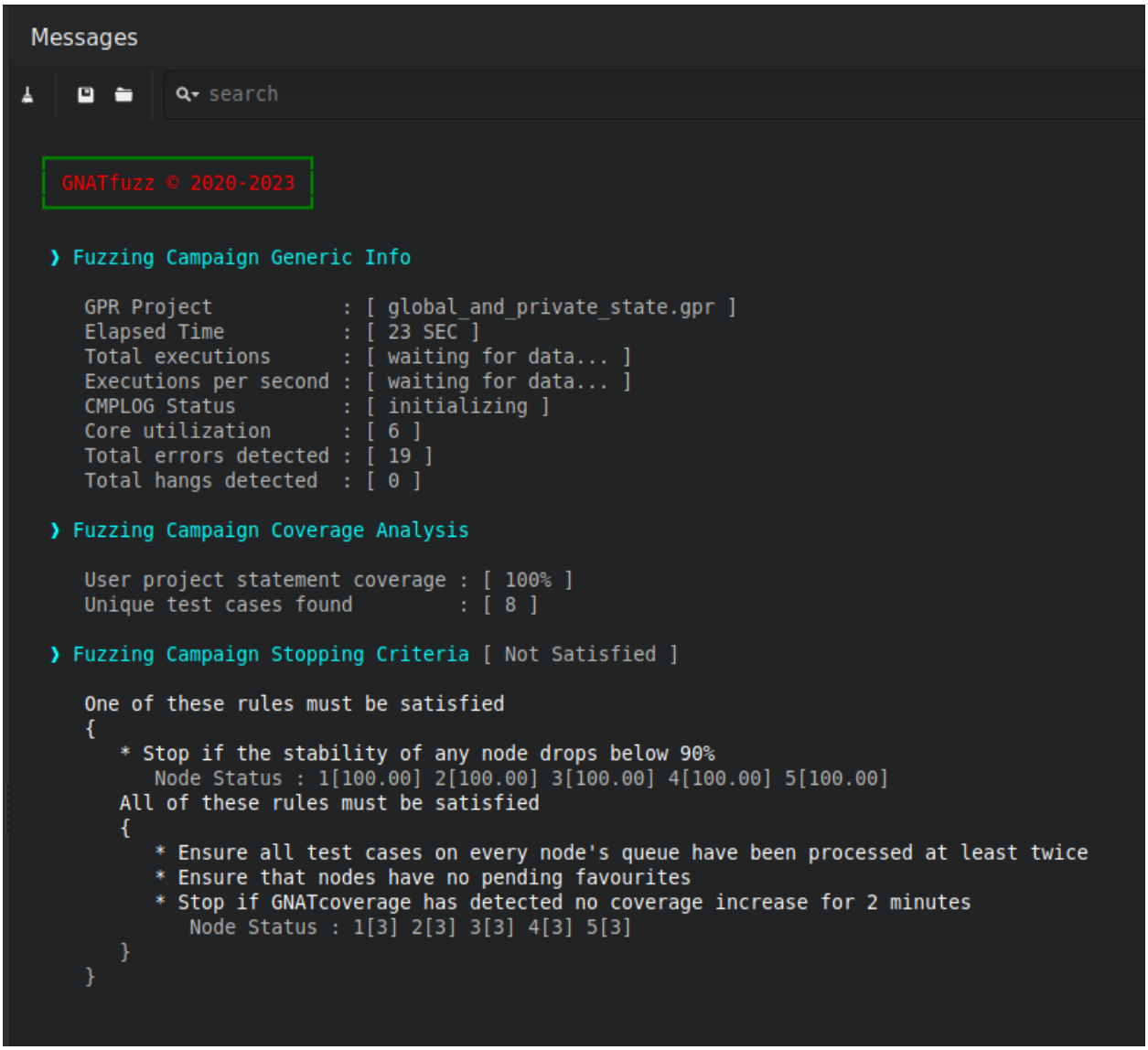


Fig. 3: The Messages view showing the gnatfuzz command line report

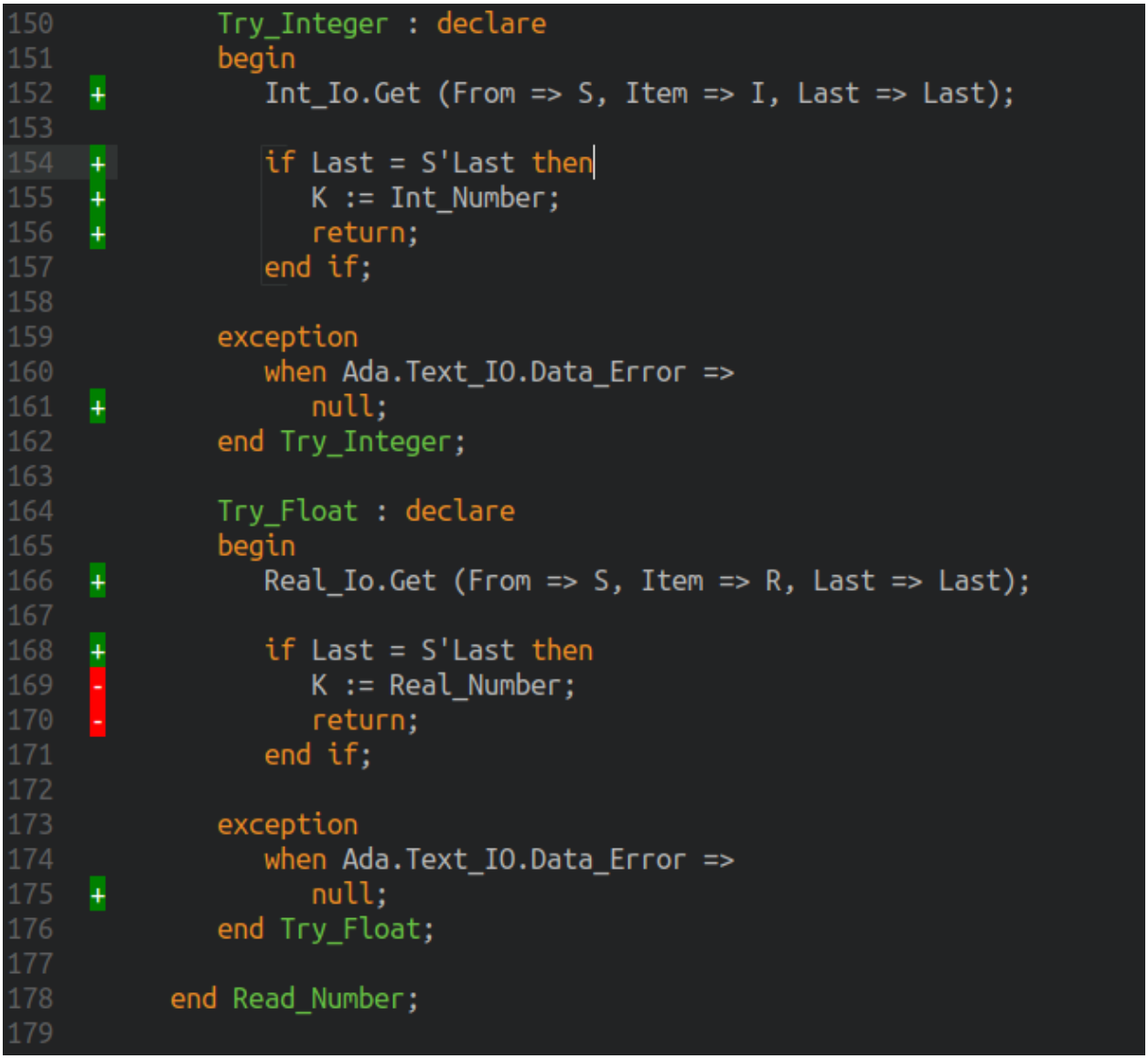


Fig. 4: The source editor annotated with coverage information

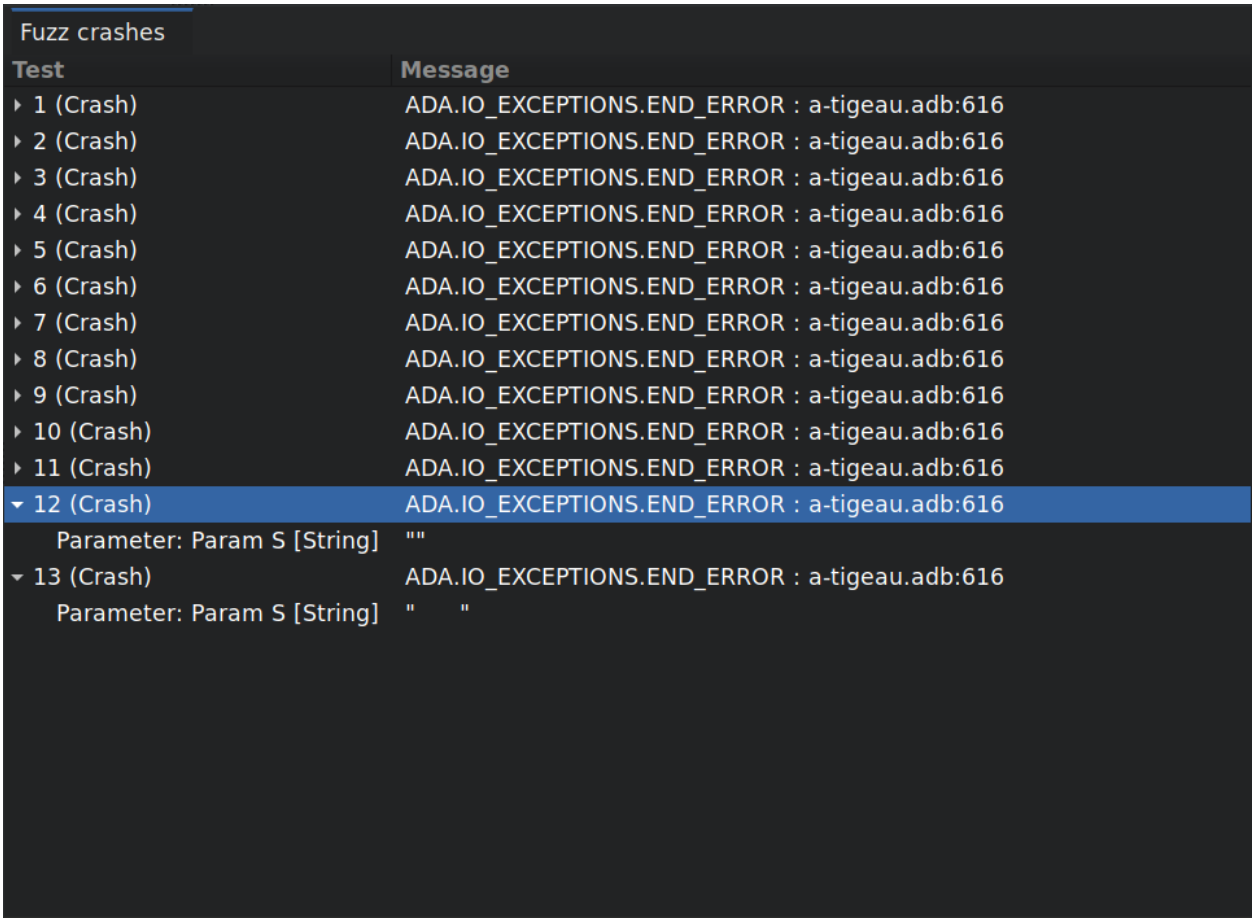


Fig. 5: The *Fuzz Crashes* view showing the crashes found by gnatfuzz

CHAPTER

SEVEN

FAQ FOR GNATFUZZ

Question 1: If my application under test uses Ada tasks, how can I ensure any exceptions raised within task bodies are propagated up to the main test harness?

Answer: Exception propagation in Ada tasking doesn't happen automatically. However, the fuzzing campaign may fail to identify software bugs within Ada tasks unless you ensure that the package `GNATfuzz.Fallback_Handler_Start_Up` is included within the main fuzz-test-harness. To help facilitate this behavior, GNATfuzz will include the required code as an Ada comment within the generated file `fuzz_testing/generated_src/gnatfuzz-fuzz_test_harness.adb`:

Listing 1: Enabling the GNATfuzz.Fallback_Handler_Start_Up

```
1    --  Uncomment the code below if the test application is multi-threaded
2    --  with GNATfuzz.Fallback_Handler_Start_Up;
```

To enable the task-exception-propagation behavior, uncomment the second line above before running the fuzzing campaign.

Question 2: If my Ada application under tests invokes C/C++ via Ada bindings, will the C/C++ libraries also be included in the fuzzing campaign?

Answer: Yes. However, for the C/C++ libraries to be included in a GNATfuzz campaign, you must dynamically build them via the standard **gprbuild** mechanism. GNATfuzz can only add the necessary fuzz-test instrumentation to the C/C++ libraries if it has visibility of the source code.

Question 3: What happens if I want to fuzz test aspects of my application that are statically built?

Answer: For your statically built libraries to be included in the control-flow path awareness feature of GNATfuzz, you have to build them using the additional pass implemented by the **AFL-GCC-fast** compiler plugin. To learn more about how to enable this compiler pass on a statically built library, please contact AdaCore support. Alternatively, you can change the build mechanism for the campaign so that the libraries are built dynamically.

CHAPTER

EIGHT

GNATFUZZ DIAGNOSTICS AND ISSUE REPORTING

GNATfuzz reports any problems or warnings captured during the use of the GNATfuzz modes, whether specified on the command line or by the GNAT Studio IDE. You can find more information on the actions a mode is performing and any potential issues in the auto-generated log files for each mode. The location of these files are:

1. For the **analyze** mode: <obj-dir>/gnatfuzz/analyze.log, where <obj-dir> is the location of the system-under-test project's object directory.
2. For the **generate** mode: <obj-dir>/gnatfuzz/generate.log, where <obj-dir> is the location of the system-under-test project's object directory.
3. For the **generate-corpus** mode: <generated_test_harness>/fuzz_testing/build/gnatfuzz/corpus_gen.log, where <generated-test-harness> is the location specified to the **generate** mode to create the test harness.
4. For the **fuzz** mode: <generated-test-harness>/fuzz_testing/session/fuzz.log, where <generated-test-harness> is the location specified to the **generate** mode to create the test harness. If the AFL++ engine is active, a separate log file *afl-fuzz.<N>.log* will also be generated in this directory for each instance of afl-fuzz being launched. If libfuzzer is active, a separate log file *libfuzzer.log* will also be generated in this directory.

Note that if the GNATfuzz modes are run multiple times targeting the same subprogram, multiple log files will be generated, with increasing numbering as part of their naming convention indicating their order of generation, for example, analyze.log, analyze.log_1, and analyze.log_2

8.1 GNATfuzz Error Codes and Exit Statuses

Below you can find a list of all error types that we expect GNATfuzz could raise, along with the corresponding status code that will have been returned when GNATfuzz exited. Further down you will find a short description of the likely causes of each error, and suggestions for steps you can take to address them.

Error Name	Exit Status Code
Invalid_Command_Line_Options	2
Invalid_GNATfuzz_Configuration_Options	3
Invalid_Fuzzable_Subprogram	4
Error_Initializing_Fuzzing_Campaigns	5
Error_Launching_Fuzzing_Campaigns	6
Error_During_Fuzzing_Campaigns	7
Error_Terminating_Campaigns	8
Error_Initializing_Supporting_Tool	9
Error_Executing_Supporting_Tool	10
Error_Generating_Starting_Corpus	11
Error_Unknown	12

8.1.1 Error_Unknown

This is a catch all exit code for when we don't know what caused GNATfuzz to crash. Please check the log file for whichever mode you were currently running. For more information about where to find the appropriate log file, please see section *GNATfuzz diagnostics and issue reporting*.

8.1.2 Invalid_Command_Line_Options

This exit code is returned when there was an issue with the command line arguments used. See *GNATfuzz "fuzz" mode command line interface* for more information on command line arguments.

8.1.3 Invalid_GNATfuzz_Configuration_Options

This exit code is returned if there is an issue with your one of either: your stopping criteria file, your environment variables, or your Analyze json results file. Check the log file for whichever mode you were currently running. For more information about where to find the appropriate log file, please see section *GNATfuzz diagnostics and issue reporting*.

8.1.4 Invalid_Fuzzable_Subprogram

This exit code is returned if GNATfuzz fails to compile the user's code. Check the log file for whichever mode you were currently running. For more information about where to find the appropriate log file, please see section *GNATfuzz diagnostics and issue reporting*.

8.1.5 Error_Initializing_Fuzzing_Campaigns

This exit code is returned when GNATfuzz fails to build some component that is required for the fuzzing campaign. Check the log file for whichever mode you were currently running. For more information about where to find the appropriate log file, please see section *GNATfuzz diagnostics and issue reporting*.

8.1.6 Error_Launching_Fuzzing_Campaigns

This exit code is returned if GNATfuzz encounters an error will attempting to start your fuzzing campaign. For more information about your issue see the log which can be found at `<generated-test-harness>/fuzz_testing/session/fuzz.log`.

8.1.7 Error_During_Fuzzing_Campaigns

This exit code is returned if GNATfuzz encounters an unexpected issue while running a fuzzing campaign. For more information about your issue see the log which can be found at `<generated-test-harness>/fuzz_testing/session/fuzz.log`.

8.1.8 Error_Terminating_Campaigns

This exit code is returned if GNATfuzz finishes a campaign (either because the stopping criteria were met or because it was interrupted by the user), but then GNATfuzz encountered and issue while trying to stopping one of the running tasks. For more information about your issue see the log which can be found at `<generated-test-harness>/fuzz_testing/session/fuzz.log`.

8.1.9 Error_Initializing_Supporting_Tool

This exit code is returned when GNATfuzz fails to build some component that is required for a supporting tool, for example GNATcov, SymCC, CMPLOG etc. Check the log file for whichever mode you were currently running. For more information about where to find the appropriate log file, please see section If you find that the error was caused by a supporting tool like CMPLOG or SymCC, you can trying disabling the tool as a workaround. See the *GNATfuzz command line interface* section for more information how how to disable specific supporting tools.

8.1.10 Error_Executing_Supporting_Tool

This exit code is returned if GNATfuzz encounters an unexpected issue while running a supporting tool, for example GNATcov, SymCC, CMPLOG etc. For more information about your issue see the log which can be found at `<generated-test-harness>/fuzz_testing/session/fuzz.log`. If you would like to disable the tool that is causing the issue, see the *GNATfuzz command line interface* section.

8.1.11 Error_Generating_Starting_Corpus

This exit code is returned if GNATfuzz encounters an unexpected issue while generating the starting corpus. Check if the User_Provided_Seeds subprogram needs completing within the User Configuration package . For more information about your issue see the log which can be found at `<generated-test-harness>/fuzz_testing/session/fuzz.log`. If you would like to disable the tool that is causing the issue, see the *GNATfuzz command line interface* section.

CHAPTER

NINE

GNATFUZZ BETA FEATURE

This section provides early access to some bleeding-edge GNATfuzz features.

9.1 Incremental fuzzing with TGen test vectors

GNATfuzz provides an incremental fuzzing workflow that allows fuzzing campaigns to build upon previous results. This is achieved through the **import-tgen-vectors** command, which converts TGen JSON test vectors into binary corpus seeds for fuzzing.

TGen JSON test vectors can be generated by GNATtest or by fuzzing campaigns that use the beta experimental flags documented below.

The key benefit of this approach is that each fuzzing campaign can enrich a collection of TGen JSON test vectors. When you run **import-tgen-vectors** before starting a new campaign, you convert accumulated TGen JSON test vectors (including those from previous campaigns) into binary corpus seeds. This means new campaigns can start with all the coverage-increasing test cases discovered by previous campaigns, allowing them to continue exploration from where earlier campaigns ended rather than starting from scratch.

9.1.1 GNATfuzz *"import-tgen-vectors"* mode command line interface

In this mode GNATfuzz has the following command line interface:

Listing 1: GNATfuzz import-tgen-vectors-mode usage

```
$ gnatfuzz import-tgen-vectors [--help|-h] -P <PROJECT> [--json-file <JSON-FILE>]
                               [-f|--force] [--disable-styled-output]
```

where

- **--help**
  Shows the usage information for the import-tgen-vectors mode
- **-P <PROJECT>**
  Specifies the fuzz-test.gpr project file created during a GNATfuzz **generate** mode associated with the fuzzing campaign Mandatory
- **--json-file <JSON-FILE>**
  Override the auto-detected TGen JSON file location. By default, GNATfuzz automatically locates the TGen JSON file based on the project configuration.
- **-f, --force**
  Overwrite the existing separate file without creating a backup. By default, GNATfuzz creates a timestamped backup of the separate file before overwriting it (unless the file contains only the `null;` statement).

- **--disable-styled-output**
  By default GNATfuzz may style some of its output using ANSI escape sequences. This switch can be used to disable this behavior.

Warning: The **import-tgen-vectors** command modifies the `Add_TGen_JSON_Vectors_To_Corpus` separate file in the `user_configuration` directory. This is a user-editable file that can be customized to control how TGen JSON test vectors are imported into the starting corpus. Running the **generate** mode again will recreate this separate file, replacing any customizations. If you need to preserve custom modifications, either back up the file manually or use the incremental workflow where you only run **generate** once and then use **import-tgen-vectors** for subsequent campaigns.

When the **import-tgen-vectors** workflow overwrites the separate file, a timestamped backup is automatically created (unless the -f switch is used or the file contains only the default `null;` statement). One benefit of this separate file architecture is that users can customize how test vectors are imported if needed.

9.1.2 Incremental fuzzing workflow

The **import-tgen-vectors** command enables an incremental fuzzing workflow. When used in conjunction with beta experimental flags (documented below), fuzzing campaigns can automatically generate TGen JSON test vectors during execution.

The recommended workflow for incremental fuzzing is:

1. Run **analyze** and **generate** modes to create a fuzz test-harness
2. (Optional) Use **import-tgen-vectors** if you have existing TGen JSON test vectors to convert into starting corpus seeds
3. Run **fuzz** mode (optionally with the --tgen-gnattest-test-generator flag) to launch the fuzzing campaign. With the flag enabled, newly discovered test cases are automatically written to the TGen JSON directory
4. When the campaign completes, re-run **import-tgen-vectors** to convert the enriched TGen JSON collection (now including test cases from the completed campaign) into corpus seeds for the next campaign
5. Launch a new **fuzz** campaign with the enriched starting corpus, which will continue exploration from where the previous campaign ended
6. Repeat steps 4-5 as needed to iteratively build a comprehensive test suite

This workflow allows you to build increasingly comprehensive test suites over multiple fuzzing iterations, with each campaign benefiting from the discoveries of all previous campaigns.

9.1.3 Beta experimental TGen flags for *"fuzz"* mode

Warning: The following flags are experimental beta features and are not part of the standard GNATfuzz fuzz mode interface. They may change or be removed in future releases.

The following additional flags are available in **fuzz** mode to support incremental fuzzing:

- **--tgen-gnattest-test-generator**
  Enable TGen test case generation during fuzzing. Observes the AFL queue and uses TGen to convert interesting test cases into JSON format that can be used by GNATtest. The generated test vectors are written to the directory specified by --tgen-serialized-test-dir.

- **--tgen-serialized-test-dir**
  Directory where TGen JSON test vectors are written. This option requires --tgen-gnattest-test-generator to be enabled, otherwise it is ignored. Default Location: <FUZZ-TEST-PROJECT-DIR>/tgen_serialized_test_cases

9.2 GNATfuzz and Symbolic Execution

GNATfuzz can leverage SymCC, a symbolic execution tool, to generate test cases that increase coverage in hard-to-reach areas of the code under test.

Symbolic execution is a technique that observes how values are computed in a program, expressing each as a formula for the inputs. This often enables it to compute back from a desired value at an arbitrary point in the program to the input required to produce said value. For example, if an interesting part of a subprogram under test is only entered when the sum of two of its input parameters has a specific value, symbolic execution can find values for the input parameters so that their sum matches the expectation. Both the fuzzer's mutations and CMPLOG are inherently limited in generating test cases for such scenarios. This is because the probability of the fuzzer's random mutations generating a specific value required to access a tight branch (e.g., x=1923333) is pretty low, and CMPLOG would not work in cases where intermediate computation steps break the input-to-state correspondence (see Section *CMPLOG (aka RedQueen) support*). This makes symbolic execution an excellent complementary technology to AFL and CMPLOG, as it helps increase coverage in cases where the other two technologies cannot help. The following function shows a simplified case where SymCC can help the fuzzer enter a tight branch whose conditions are derived from arithmetic algorithms and magic numbers such that there is no input-to-state correspondence between the subprogram parameters and the `c` branch-control variable.

Listing 2: Typical scenario where SymCC adds value

```
package body SymCC_Test is

   function Tight_Branch_Test
   (X : Positive; Y : Positive; Z : Positive) return Boolean
   is
      a : constant Positive := X + Y;
      b : constant Positive := a - Z;
      c : constant Positive := a*b;
   begin
      if  c = 14574000 then
         return true;
      else
         return false;
      end if;
   end Tight_Branch_Test;

end SymCC_Test;
```

SymCC is an open-source symbolic execution implementation. It uses a compiler plugin to instrument the code under test and provides a run-time library to generate new test cases when the program is executed. For more information on SymCC, see the SymCC seminal publication.

9.2.1 Enabling SymCC

To enable SymCC with GNATfuzz, add `symcc` to the –engines switch. Any GPR project dependencies of the code under test must be specified in the environment variable `LLVM_SYMCC_GPR_PROJECT_PATH`. The SymCC feature of GNATfuzz requires configuring the `LLVM_SYMCC_GPR_PROJECT_PATH` environment variable in the same way the `GPR_PROJECT_PATH` environment variable is being configured for the `GNAT Pro gprbuild`. When GNATfuzz is executed with SymCC integration enabled, it runs SymCC on a separate thread to produce interesting test cases. It regularly synchronizes the sets of test cases used by the fuzzer(s) and by SymCC so that SymCC benefits from test cases discovered by the fuzzer(s) and vice versa. Fig. 6 shows SymCC's output directory location where all the SymCC-generated test cases are placed.

9.2.2 Current Limitations

SymCC currently can't reason about floating-point values. Consequently, it won't be able to find new test cases for conditions that depend on floating-point inputs or intermediate values. Note that this doesn't affect its ability to reason about non-floating-point data in the code under test, even if that code also works with floating-point data.

Furthermore, the adaptation of SymCC for Ada can't change the length of unbounded array type parameters. Note that this is particularly relevant to subprograms handling string data (which is represented in Ada as an unbounded array of Characters): if reaching an interesting portion of the subprogram under test requires an input string to have a different length, then SymCC can't find such a test case (but the fuzzer(s) may).

Finally, SymCC requires all code in the program under test to be compilable with GNAT-LLVM, the LLVM-based Ada compiler provided by AdaCore, see the GNAT with the LLVM Back End manual. It works with most Ada constructs but currently still lacks support for a few language features.

9.3 GNATfuzz Adaptive - "On-target" fuzzing (experimental)

Warning: On-target fuzzing is an experimental and evolving feature of GNATfuzz. Please note that aspects of the feature set are subject to change. In addition, the GNATfuzz Adaptive engine has only been tested on CheriBSD on Morello. More extensive testing on Morello and other targets is expected for future releases.

9.3.1 What is it?

The term "On target fuzzing" refers to fuzzing on embedded platforms that may or may not have an operating system. This mode is similar to regular GNATfuzz, except that it involves an executable that should be executed directly on the target machine (or using an emulator).

The workflow is the same, except that GNATfuzz cannot launch the executable built for your target (as normally happens during the GNATfuzz fuzz phase for host-based fuzzing), as this process is specific to the targeted platform. It is also up to the user to push the executable to the targeted hardware or emulator. An example workflow is provided below to help understand the process.

9.3.2 How to use?

The analyze step is the same as with regular GNATfuzz. After invoking `gnatfuzz analyze`, you should get an `analyze.json` file reporting all supported subprograms for a given project.

The generate phase works like in other modes, except that some environment variables must be set to enable the GNATfuzz Adaptive on-target fuzzing mode.

```
export GNATFUZZ_ADAPTIVE=1
export GNATFUZZ_RT_Externally_Built=false
```

The first variable tells GNATfuzz to generate code for Adaptive fuzzing, the second variable allows rebuilding the GNATfuzz runtime for a specified target.

Note: Some platforms may not support converting test cases in JSON format. In such situations it is possible to set the *TGEN_NO_JSON_MARSHALLING* variable to remove JSON test case support from the generated harness. Be aware that if *TGEN_NO_JSON_MARSHALLING* is set, converting binary test case to their JSON representation won't be possible anymore.

On such platforms, it is required to run the regular test decoder **on the the target** to avoid issues because of different memory representation of variables between the host and the target (endianness, floating-point, twos-complement…). The decode values can be retrieved on the test decoder's standard output.

9.3.3 Setup the development environment

The GNAT Dynamic Analysis Suite and a working GNAT Pro cross-compilation toolchain from x86 64-bit Linux to the desired hardware target and operating system specification must be installed. See the GNAT cross platforms user guide for more information.

Note: For this early release of GNATfuzz Adaptive on-target fuzzing, the GNAT Pro Full Ada runtime is required as the runtime requires access to a file system to generate results files. Future releases will allow fuzzing on bareboard targets via the GNAT Pro Ada Embedded runtime.

In addition, the gnatcov runtime must be built and installed for the target. See the gnatcov user guide for more information.

9.3.4 Configure the fuzzer

The following environment variables can be used to configure the engine:

Variable Name	Possible values	Description	Default values
GNATFUZZ_ADAPTIVE	Set to enable	Switch the GNATfuzz generator in adaptive fuzzing mode.	Not set
GNATFUZZ_ADAPTIVE_EN(	Integer value	Manually set a custom value to seed the random number generator within the GNATfuzz Adaptive mutation engine. If set, the mutations are deterministic.	Current time
GNATFUZZ_ADAPTIVE_EN(	Integer value	Maximum execution time allowed for the engine to detect a hang.	60s
GNATFUZZ_ADAPTIVE_EN(	Set to enable	Enable coverage-guided fuzzing.	Not set
GNATFUZZ_ADAPTIVE_EN(	Positive integer value	Number or crashes to stop a fuzzing campaign.	50 crashes

9.3.5 Understanding the results

Each crash and hang is saved in its respective directory. Each saved test case can be decoded using the decode tests executable *Decode_Test_Build_Command*.

9.3.6 Example usage

The following example will explore fuzzing a simple Ada program on the Arm Morello development board using CheriBSD. Arm Morello offers a comprehensive set of hardware based bug finding capabilities via fine grain compartmentalization and makes for an interesting fuzzing target! For more information about Arm Morello and CHERI in general see: CHERI-Morello

Let's pretend that we want to fuzz the following subprogram on a morello-based board using CheriBSD.

```
package Pkg is
   function Compute_Stuff (X : Integer) return Integer;
end Pkg;
```

First, we need to switch GNATfuzz to Adaptive fuzzing mode and analyze the project:

```
$ export GNATFUZZ_ADAPTIVE=1
$ export GNATFUZZ_ADAPTIVE_ENGINE_COVERAGE=1
$ gnatfuzz analyze -Pbuild.gpr
```

The snippet above sets gnatfuzz to use the Adaptive engine (`GNATFUZZ_ADAPTIVE`), with the coverage feature (`GNATFUZZ_ADAPTIVE_ENGINE_COVERAGE`). The last tells the test generator not to try and write new test-cases to the GNATtest TGen JSON files, as this feature isn't supported on some platforms (CheriBSD being one of them).

Now we can use the `generate` subcommand to create a fuzz harness. Since our project contains only one fuzzable subprogram, its `id` will be 1.

```
$ gnatfuzz generate \
    -P build.gpr \
    --analysis obj/gnatfuzz/analyze.json \
    --subprogram-id 1
```

All necessary code is now generated under `obj/gnatfuzz/harness/`. At this point, the generated harness needs to be compiled for CheriBSD on morello.

Before building the harness, the generated code must be instrumented using GNATcoverage to make coverage-based fuzzing work.

```
$ gnatcov instrument \
    -P ./obj/gnatfuzz/harness/fuzz_testing/adaptive_engine.gpr \
    --level=stmt \
    --target=morello-freebsd,15
```

Note: It is possible to instrument the code for different coverage level and even combine the different coverage methods. It may improve fuzzing results on some codebases. We recommend testing different combinations of levels and picking the one that works best for a use case.

Once instrumented, the harness can be built using the following command. Note that `--implicit-with=gnatcov_rts.gpr` and `--src-subdirs=gnatcov-instr` options should be omitted if the coverage feature isn't used.

```
$ gprbuild \
    --target=morello-freebsd,15 \
    -P ./obj/gnatfuzz/harness/fuzz_testing/adaptive_engine.gpr \
    -j$(nproc --all)
```

The resulting artifacts will be located under `./obj/gnatfuzz/harness/fuzz_testing/build/gnatfuzz-test_harness.adaptive`. This executable can be sent to either a physical Arm Morello running CheriBSD or to an emulated version via GNAT Emulator.

While running, the fuzzer will save crashes and hangs in their respective directories. These test cases can be decoded using the `gnatfuzz-decode_test` executable.

Directory	Description
`CRASHES`	Contains test cases that caused an unhandled exception
`HANGS`	Contains test cases that caused a hang
`QUEUE`	Contains test cases that increased code coverage

GNATfuzz test cases are binary files that match the target hardware's in-memory representation. As such, they are susceptible to endianness and are not cross-target compatible. Therefore, binary test cases should be decoded on-target using the generated test case decoder.

Future releases of GNATfuzz Adaptive fuzzing will generate test cases that are target hardware agnostic, such that found errors and hangs can be reproduced on a host-based system. However, for this early release, the test cases can only be debugged on-target or manually reproduced on the host by inspecting the ASCII representation of the test case generated via the decoder.

The test case decoder can be built with the following command:

```
$ gprbuild \
    --target=morello-freebsd,15 \
    -P ./obj/gnatfuzz/harness/fuzz_testing/adaptive_engine.gpr \
    -XGNATFUZZ_INTERNAL_MODE=DECODE_TEST \
    -j$(nproc --all)
```

The resulting executable should be present under `./obj/gnatfuzz/harness/fuzz_testing/build/gnatfuzz-decode_test`.

It is possible to decode a test case in JSON format using the JSON test decoder. The outputted JSON is UTF8 so it is safe to share its content between the host and the target system. It is preferable to use this cross platform format when sharing test cases between architectures that are using incompatible data representation forms (for example, endianness or two's complement). This variant of the test decoder can be built with the following command:

```
$ gprbuild \
    --target=morello-freebsd,15 \
    -P ./obj/gnatfuzz/harness/fuzz_testing/adaptive_engine.gpr \
    -XGNATFUZZ_INTERNAL_MODE=DECODE_TEST_JSON \
    -j$(nproc --all)
```

The resulting executable should be present under `./obj/gnatfuzz/harness/fuzz_testing/build/gnatfuzz-decode_test_json`.

9.3.7 Known limitations

This is a list of known limitations. Some limitations may be addressed in upcoming releases.

- The fuzzer can not recover from a hang. When a hang is detected, the engine saves the corresponding test case and exits with code 2.
- Crashes found by the fuzzer may be duplicated.
- Due to differences in some target hardware in-memory representations (i.e. endianness) the binary test cases in the results folders may not be compatible with a GNATfuzz host-based test harness. The workaround is to use the generated decoder to convert the test case to an ASCII representation.
- The full Ada runtime is required, future releases will allow bareboard fuzzing using the Embedded Ada runtime.
- Corpus generation **must be** supported for the given subprogram. The *analyze.json* contains this information (via the *corpus-gen-supported* attribute).

9.4 Introducing the hot-spot workflow

Note: The hot-spot workflow is a beta feature. Its interface and output format may change in future releases.

The **hot-spot** workflow bridges static analysis and dynamic testing by taking a SARIF file produced by GNATSAS and attempting to corroborate each reported warning (considered a hot-spot for the system-under-test) through fuzzing. For each flagged location, GNATfuzz generates test inputs and tries to produce a concrete reproducer. When fuzzing reaches the location, but cannot trigger the exception, that provides evidence — though not proof — that the warning may not represent a reachable defect.

In short, Static Application Security Testing (SAST) finds the suspicious code, and Dynamic Application Security Testing (DAST) via GNATfuzz tries to verify whether the warning can be triggered at runtime.

9.4.1 When to use the hot-spot workflow

The hot-spot workflow is most useful when:

- You have run GNATSAS and want to triage a SARIF report without manually reviewing every result.
- You are integrating static and dynamic analysis in a CI/CD pipeline and need machine-readable evidence that a warning is exploitable.
- You want to prioritise remediation work by distinguishing warnings confirmed as real bugs from those that fuzzing could not trigger.
- You want to fuzz-test a large codebase but focus on the most suspicious locations identified by static analysis.

9.4.2 Prerequisites

The hot-spot workflow builds on the same GNATfuzz infrastructure as the standard workflow, so the same *AFL++ requirements* and *GNATfuzz requirements* apply.

In addition, you need:

- A SARIF file from a static analysis run on your project. The file must include the path to the GPR project file either as a SARIF property (`properties.gpr_full_path` on the run object) or provided on the command line with -P.

Tip: The hot-spot workflow is most effective when the SARIF is limited to **runtime check failures** — the class of findings that can actually be triggered by a concrete input at runtime (division by zero, out-of-range access, overflow, etc.). Use the `--show=kind+check` flag when invoking `gnatsas report sarif` to filter to this category:

```
gnatsas report sarif -P my_project.gpr --show=kind+check -o checks.sarif
```

Without this filter, the SARIF may contain rules that GNATfuzz cannot corroborate dynamically, such as dead-code warnings, style violations, race-condition heuristics, or memory-leak reports. Including them adds fuzz-testing campaigns that can never produce a verified result, while increasing the overall session's response time.

9.4.3 Hot-spot command line interface

The hot-spot workflow is invoked with the **`gnatfuzz hot-spot`** command:

```
gnatfuzz hot-spot [--sarif <file>] [-P <project>] [-o <dir>]
                  [--sarif-output <file>] [--cores <N>]
                  [--campaign-duration <seconds>] [--engines <list>]
                  [-X <name>=<value>] [--verbose | -v] [--keep-artifacts]
                  [--no-tgen-import] [--root <dir>] [--help]
```

where

- **--sarif <file>**
  Path to the input SARIF file containing static analysis results. Mandatory
- **--sarif-output <file>**
  Path to the SARIF output file: an enhanced version of the input SARIF file with the hot-spot's verification results. Defaults to `<input>_hotspot.sarif`, where `<input>` is the name of the SARIF file provided with the --sarif switch.

- **-P <file>**
  Path to the GPR project file. Extracted from the SARIF `properties.gpr_full_path` field when not provided.
- **-o <dir>, --output-dir <dir>**
  Root directory for all fuzzing artifacts. A timestamped `hot_spot_results/<timestamp>/` subdirectory is created here. Defaults to the current working directory.
- **--cores <N>**
  Number of parallel fuzzing cores per campaign. `0` (the default) auto-detects the available core count.
- **--campaign-duration <seconds>**
  Maximum wall-clock time for each fuzzing campaign in seconds. A campaign also terminates early when all static analysis results for the current hot-spot have been verified. Default: `120`.
- **--engines <list>**
  Comma-separated list of fuzzing engines to use. Currently `afl` and `cmplog` are supported. Default: `afl`.
- **-X <NAME=VALUE>**
  Sets the external reference `NAME` in the system-under-test's GPR project to `VALUE`
- **-v, --verbose**
  Show individual warning listings for each hot-spot, a per-hot-spot bug summary after each campaign, and caller-search diagnostics when a fuzzable entry point could not be found directly.
- **--keep-artifacts**
  Keep all build objects and intermediate fuzzer state on disk after each campaign. By default, artifacts are removed to save disk space.
- **--no-tgen-import**
  Skip importing existing TGen test vectors as corpus seeds. By default, any TGen-generated JSON test vectors for the target subprogram are automatically imported before each fuzz-testing campaign, giving the fuzzer a valid starting corpus.
- **--root <dir>**
  Root directory used to resolve relative source-file Uniform Resource Identifiers (URI)s in the SARIF file. Pass the same directory that was supplied to `gnatsas report sarif --root DIR` so that the relative paths written by GNATSAS resolve to the correct absolute locations in the project tree. When omitted, relative URIs are resolved against the GPR project directory.
- **--help**
  Shows the usage information for the hot-spot workflow.

9.4.4 End-to-end example

The following walkthrough demonstrates a complete hot-spot verification session for hot-spots (warnings for potential run-time check failures) identified by GNATSAS on the `avionics_simulation` example. The example is located under:

```
<install_prefix>/share/examples/gnatfuzz/avionics_simulation/
```

The same example is used throughout all the hot-spot documentation.

Step 1 — Run GNATSAS to produce a SARIF report

GNATSAS supports a range of analysis modes and check levels. The commands below run a comprehensive analysis and export all results to a SARIF file. GNATSAS supports many options; see the GNATSAS User's Guide for the full list of analysis modes, engines, and check configurations.

The following commands demonstrate the optimal settings for running GNATSAS to verify its results with GNATfuzz's hot-spot workflow:

```
# Run comprehensive analysis with all engines and checks
$ gnatsas analyze -P avionics_simulation.gpr --mode deep -j0 --force

# Export only runtime check failures - the findings GNATfuzz can verify
$ gnatsas report sarif -P avionics_simulation.gpr --show=kind=check -o checks.sarif
```

Note: The `--mode deep` flag enables the most thorough inter-procedural analysis. Lighter modes (e.g. `--mode default`) run faster but may produce fewer results. The `-j0` flag uses all available CPU cores. Adjust these options to suit your project and available time.

`--show=kind=check` limits the report to the *check* rule group: `division_check`, `index_check`, `range_check`, `overflow_check`, `access_check`, `discriminant_check`, `tag_check`, and `assertion_check` (including pre/post-condition violations). These are the findings that correspond to Ada runtime exceptions — exactly the class of defects that fuzzing can trigger. Other rule groups (dead code, style, race conditions, memory leaks) are omitted because they cannot produce a concrete crashing input.

Step 2 — Run the hot-spot workflow to verify the findings

```
$ gnatfuzz hot-spot --sarif checks.sarif --cores 4 --campaign-duration 120
```

The above command will perform all the hot-spot workflow operations, including reading the input `checks.sarif` file, matching each GNATSAS-reported runtime-check warning to a fuzzable entry point, and running a campaign for up to 120 seconds per hot-spot. For a full explanation of all the steps performed by the workflow, please refer to *Workflow steps* section.

Furthermore, through the duration of the workflow's execution, a variety of execution statistics are being reported on the terminal's output, depending on the level of verbosity selected through the workflow's -v/--verbose CLI switch. For a complete description of all the information reported to the terminal, please refer to section *Terminal output*.

A workflow's session verification results are being written to the `checks_hotspot.sarif` file. For a description of the structure and semantics of this file, please look at *Augmented SARIF output* section.

The following command demonstrates an example of how to pass any required external references to the hot-spot workflow (e.g., for platform-specific builds):

```
$ gnatfuzz hot-spot --sarif checks.sarif -X PLATFORM=linux -X BUILD_MODE=debug \
    --cores 4 --campaign-duration 300
```

Step 3 — Inspect the machine-readable report

A summary JSON file is always written at the end of the hot-spot workflow run:

```
$ cat hot_spot_results/<timestamp>/hot_spot_report.json
```

ls For a complete description of the structure and semantics of this file, please look at *Machine-readable summary report* section.

Step 4 — Reproduce a crash

The augmented SARIF `properties["Reproduction Command"]` for each verified result contains the exact command to replay the crash. The commands are relative to the root artifacts directory specified by the -o <dir>, --output-dir <dir> switch. Therefore, first navigate to `hot_spot_results/<timestamp>/` and then you can copy the command and run it directly:

```
# Navigate to the root artifacts directory
cd hot_spot_results/<timestamp>/

# Use the reproduction command when no --keep-artifacts switch was used:
$ hotspot<number>_<subprogram_name>/runner/gnatfuzz-test_harness.verbose \
    hot_spot_results/<timestamp>/hotspot<number_<subprogram_name>/crashes/<crash_file_
↪name>

# When using the --keep-artifacts switch, the reproduction command will be:
$ /hotspot<number>_<subprogram_name>/attempt<number>/ \
    fuzz_testing/build/obj-<engine>/gnatfuzz-test_harness.verbose \
    hot_spot_results/<timestamp>/.../fuzz_testing/session/results/crashes/<crash_file_
↪name>
```

The verbose harness prints the full Ada exception information and symbolic traceback, identifying the exact source line and call chain that triggered the crash.

You can also check the pre-decoded output directly — for a fuzz-tested hot-spot, this will be under the `decoded_crashes/` directory alongside `crashes/`, and for a unit-testing hot-spot results, this will be under the `pre-decoded_crashes/` directory alongside `pre-crashes/`. These directories contain the human-readable runner output for every crash, including decoded input values and the full stack trace. Section *Workflow steps* explains when a hot-spot is being unit-tested or fuzz-tested.

Step 5 — Review coverage

The hot-spot workflow produces a project-wide coverage report, which can be viewed via a web browser:

```
# Path also available in hot_spot_report.json as "combined_coverage_dir"
$ xdg-open hot_spot_results/<timestamp>/coverage/index.html
```

The HTML report shows which lines across the system-under-test were reached. For any warnings with `coverage_status: "no_data"` in the JSON report (see the *Per-hot-spot result objects* section), the HTML coverage report will show those lines as unexercised. Consider re-running with a larger value for the --campaign-duration switch to increase the chances of improved path exploration. For a complete description of the coverage reports created by the hot-spot workflow, please refer to section *Coverage reports*.

Step 6 — Re-run to benefit from accumulated knowledge

Because the workflow writes crash and coverage-interesting inputs back to the project's TGen JSON files during each run (see *TGen test vector integration* section), re-running the same command automatically picks up where the previous run ended:

```
# Second run - corpus is already enriched with vectors from the first run
gnatfuzz hot-spot --sarif checks.sarif --cores 4 --campaign-duration 300
```

The `tgen_vectors_added` counter in the new `hot_spot_report.json` shows how many additional vectors this run discovered on top of the previous one. Over repeated runs, the corpus may grow, and warnings could be verified progressively faster.

9.4.5 Workflow steps

When `gnatfuzz hot-spot` is invoked, it performs the following steps for each static analysis result in the SARIF file:

1. **Parse the SARIF** — extract the flagged locations and their associated subprogram names from the `logicalLocations` array.
2. **Correlate with analysis** — match each SARIF location to a fuzzable subprogram in `analyze.json`. When the flagged subprogram is not itself directly fuzzable (for example, because it has no parameters), GNATfuzz searches the call graph for the nearest fuzzable caller. If that caller was already fuzzed earlier in the same hot-spot run, GNATfuzz picks the next-ranked unfuzzed caller instead; when all callers were already fuzzed, the hot-spot is skipped.
3. **Generate harness** — call `gnatfuzz generate` for the selected entry point.
4. **Build harness** — call `gnatfuzz build`.
5. **Generate corpus** — call `gnatfuzz generate-corpus`.
6. **Minimize corpus** — call `gnatfuzz minimize-corpus`.
7. **Screen initial corpus** — before launching the fuzzer, each generated corpus file is executed once through the verbose test harness to detect inputs that immediately crash or hang the subprogram under test.
   - **Crashing inputs** are moved to a `pre_crash/` subdirectory, processed for SAR correlation (identical logic to a crash found during fuzzing), and removed from the corpus so that the fuzzer receives only non-crashing seeds.
   - **Hanging inputs** (execution exceeds one second) are moved to `pre_hang/` and counted toward the hang total.
   - If the screening step finds that all static analysis results for the current hot-spot are already verified, the fuzzer is skipped entirely and TGen test vectors are generated directly from the pre-screened crash inputs.
   - If all corpus files crash or hang, the workflow then tries the next fuzzable caller (if any) rather than launching the fuzzer with an all-crashing corpus.
8. **Fuzz** — run the configured fuzzing engine(s) for up to `--campaign-duration` seconds. The campaign terminates early if all static analysis results for the current hot-spot are verified before the timer expires.
9. **Correlate crashes** — for every crash file discovered by the fuzzer, the verbose test harness is executed and the exception traceback is analysed. Each crash is cross-referenced against *all* static analysis results across *all* hot-spots, not just the current campaign's target (see *Crash correlation and confidence scoring*).
10. **Write augmented SARIF** — the augmented SARIF is written progressively: once immediately when a new unique bug is first found, and again after clean-up (the next step) so that file paths reflect the post-clean-up flat layout. Partial results are therefore always available on disk even if the workflow is interrupted.
11. **Clean up** — unless --keep-artifacts is specified, build objects and intermediate fuzzer state are removed to save disk space. Before deletion, each crash file (fuzzer-detected crashes and corpus-screening pre-crashes) is executed through the verbose runner and the output is saved to a `decoded_crashes/` or `decoded_pre_crashes/` sibling directory. The decoded files contain the human-readable Ada input values, exception message, and symbolic traceback; they are embedded in the SARIF as the `Result file`, `Input Parameters`, `Fuzzed Subprogram`, and `Stack Trace` properties.

9.4.6 Terminal output

The workflow has two levels of terminal output controlled by the -v/--verbose CLI switch.

Default output (no flags) — one line per hot-spot plus a live fuzzing progress counter:

```
GNATfuzz Hot-Spot Workflow

Analyzing SARIF and deriving hot-spots... [ok]

Hotspot 1: function Get_Mach_Value (6 warnings)
    [>]  Preparing fuzzing campaign for Hotspot 1...
    [~] Unit testing - 0 warnings verified, 1 other bugs found - Subprogram covered: 43.0
↪% - Target lines: 1/6 reached

    [>]  Preparing alternative testing campaign for Hotspot 1...
    [~] Unit testing - 0 warnings verified, 1 other bugs found - Subprogram covered: 18.0
↪% - Target lines: 1/6 reached

Hotspot 2: function Compute_Fuel_State (5 warnings)
    [>]  Preparing fuzzing campaign for Hotspot 2...
    [~] Unit testing - 2 warnings verified, 1 other bugs found - Subprogram covered: 100.
↪0% - Target lines: 5/5 reached

    [>]  Preparing alternative testing campaign for Hotspot 2...
    [~] Unit testing - 0 warnings verified, 1 other bugs found - Subprogram covered: 100.
↪0% - Target lines: 5/5 reached

Hotspot 3: function Haversine_Range (3 warnings)
    [>]  Preparing fuzzing campaign for Hotspot 3...
    [~] Fuzzing [16s] - 3 warnings verified, 1 other bugs found - Subprogram covered:␣
↪100.0% - Target lines: 3/3 reached

Hotspot 4: function Compute_Bearing (3 warnings)
    [>]  Preparing fuzzing campaign for Hotspot 4...
    [~] Fuzzing [122s] - 1 warnings verified, 1 other bugs found - Subprogram covered:␣
↪100.0% - Target lines: 3/3 reached

Hotspot 5: function Compute_Navigation (1 warning)
    [>]  Preparing fuzzing campaign for Hotspot 5...
    [OK]  All warnings verified from initial corpus - skipping fuzzing
    [~] Unit testing - 1 warnings verified, 1 other bugs found, 1 corroboration(s) -␣
↪Subprogram covered: 93.0% - Target lines: 1/1 reached

**Workflow complete: 7/18 warnings verified, 3 campaigns run, 9 bugs found in 268s**

  Report:       hot_spot_results/20260101_120000/hot_spot_report.json
  SARIF:        hot_spot_results/20260101_120000/checks_hotspot.sarif
  Coverage:     hot_spot_results/20260101_120000/coverage/index.html
```

The `Unit testing` label (Hotspot 2 above) means all targeted static analysis results were already verified from the initial seed corpus — fuzzing was skipped entirely for that hot-spot.

Verbose output (-v/--verbose) — adds individual warning listings before each campaign, a per-hot-spot bug summary after, and caller-search diagnostics when a direct fuzzable entry point was not found:

```
GNATfuzz Hot-Spot Workflow

Analyzing project... [ok]
Correlating warnings with fuzzable subprograms... [ok]

Hotspot 1: function Get_Mach_Value (6 warnings)
   Warning #1: mach_calculator.adb:57 - range check (note)
   Warning #2: mach_calculator.adb:69 - precondition <range check> (warning)
   Warning #3: mach_calculator.adb:72 - precondition <range check> (warning)
   Warning #4: mach_calculator.adb:78 - precondition <range check> (warning)
   Warning #5: mach_calculator.adb:86 - divide by zero fp (warning)
   Warning #6: mach_calculator.adb:86 - range check (note)

   [>]  Preparing fuzzing campaign for Hotspot 1...
   [~] Unit testing - 0 warnings verified, 1 other bugs found - Subprogram covered: 43.0
↪% - Target lines: 1/6 reached
      Target subprogram cannot start fuzzing because all test cases in the starting␣
↪corpus are crashing the subprogram

   Searching for new entry...
      Explored 2 node(s) across 2 unique location(s) | Maximum depth reached: 1
      Callchain explored:
      └─> Get_Mach_Value (mach_calculator.ads:51)
         └─> Compute_Air_Data (air_data_computer.adb:24)
   [ok] Found fuzzable caller: Compute_Air_Data
      Path: Get_Mach_Value -> Compute_Air_Data
      Location: /home/kg8280/work/ancr/src/gnatfuzz/share/examples/gnatfuzz/avionics_
↪simulation/src/air_data_computer.adb:24

   [>] Preparing alternative testing campaign for Hotspot 1...
   [~] Unit testing - 0 warnings verified, 1 other bugs found - Subprogram covered: 18.0
↪% - Target lines: 1/6 reached
      Target subprogram cannot start fuzzing because all test cases in the starting␣
↪corpus are crashing the subprogram
      No more candidate entry points to try for this hotspot.
   Summary:
      [!] a-ngelfu.adb:92 ARGUMENT_ERROR : a-ngelfu.adb:92 instantiated at mach_
↪calculator.adb:50 (10 instances)

Hotspot 2: function Compute_Fuel_State (5 warnings)
   Warning #1: fuel_computer.adb:59 - divide by zero fp (warning)
   Warning #2: fuel_computer.adb:59 - range check (note)
   Warning #3: fuel_computer.adb:61 - precondition <divide by zero> (warning)
   Warning #4: fuel_computer.adb:61 - precondition <divide by zero, range check>␣
↪(warning)
   Warning #5: fuel_computer.adb:61 - precondition <range check> (warning)

   [>] Preparing fuzzing campaign for Hotspot 2...
   [~] Unit testing - 2 warnings verified, 1 other bugs found - Subprogram covered: 100.0
↪% - Target lines: 5/5 reached
      Target subprogram cannot start fuzzing because all test cases in the starting␣
↪corpus are crashing the subprogram
```

(continues on next page)

(continued from previous page)

```
  Searching for new entry...
     Explored 2 node(s) across 2 unique location(s) | Maximum depth reached: 1
     Callchain explored:
     └─> Compute_Fuel_State (fuel_computer.ads:87)
        └─> Compute_Flight_State (avionics_system.adb:24)
  [ok] Found fuzzable caller: Compute_Flight_State
     Path: Compute_Fuel_State -> Compute_Flight_State
     Location: /home/kg8280/work/ancr/src/gnatfuzz/share/examples/gnatfuzz/avionics_
↪simulation/src/avionics_system.adb:24

  [ok] Preparing alternative testing campaign for Hotspot 2...
  [~] Unit testing - 0 warnings verified, 1 other bugs found - Subprogram covered: 100.0
↪% - Target lines: 5/5 reached
     Target subprogram cannot start fuzzing because all test cases in the starting␣
↪corpus are crashing the subprogram
     No more candidate entry points to try for this hotspot.
  Summary:
     [!] a-ngelfu.adb:92 ARGUMENT_ERROR : a-ngelfu.adb:92 instantiated at mach_
↪calculator.adb:50 (15 instances)

  ...

Hotspot 1: function avionics_simulation.navigation.compute_bearing (2 warnings)
    Warning #1: navigation.adb:43 - range check (warning)
    Warning #2: navigation.adb:57 - overflow check (warning)

    [>] Preparing testing campaign for Hotspot 1...
    [~] Fuzzing [87s] - 2 warnings verified, 1 other bugs found - Subprogram covered:␣
↪94.1% - Target lines: 2/2 reached

Summary:
    [ok] Warning #1: navigation.adb:43 CONSTRAINT_ERROR [high] (7 instances)
    [ok] Warning #2: navigation.adb:57 CONSTRAINT_ERROR [medium] (3 instances)
    [!] navigation.adb:91 PROGRAM_ERROR (1 instance)

Hotspot 2: function avionics_simulation.fuel.calculate_reserve (3 warnings)
    Warning #1: fuel.adb:28 - range check (warning)
    Warning #2: fuel.adb:35 - division check (error)
    Warning #3: fuel.adb:41 - overflow check (warning)

    [>]  Preparing testing campaign for Hotspot 2...
    [~] Unit testing [0s] - 1 warnings verified, 0 other bugs found, 2 corroboration(s)␣
↪- Subprogram covered: 78.3% - Target lines: 1/3 reached

**Workflow complete: 7/18 warnings verified, 3 campaigns run, 9 bugs found in 268s**

  Report:       hot_spot_results/20260101_120000/hot_spot_report.json
  SARIF:        hot_spot_results/20260101_120000/checks_hotspot.sarif
  Coverage:     hot_spot_results/20260101_120000/coverage/index.html
```

To enable low-level debug logging (GPR build command lines, coverage data, correlation decisions, etc.), set the `GNATFUZZ_LOG=DEBUG` environment variable. This output is voluminous and intended for diagnosing analysis or build failures rather than routine use.

The following symbols appear in the terminal output:

Indicator	Meaning
`[>]`	A testing campaign is being prepared (harness generation, build, and corpus generation are in progress).
`[~] Fuzzing [VALUEs]`	An fuzzing campaign is running. The elapsed time counter updates in place while fuzzing.
`[~] Unit testing`	All targeted warnings were verified from the seed corpus alone; the fuzzer was skipped for this hot-spot.
`[-] Skipped`	The hot-spot could not be fuzzed (e.g. no fuzzable entry point found, or harness generation/build failed).
`Searching for new entry...`	Verbose caller-search diagnostics are being printed to explain how GNATfuzz is locating a fuzzable caller when the target is not directly fuzzable.
`[ok]`	A static analysis result was verified: the fuzzer triggered an exception that matches (or is traceable to) the flagged source location.
`[corr] Corroborates`	A crash found during this campaign matches a static analysis result that was already verified by a previous hot-spot's campaign, or matches a SAR belonging to a different hot-spot. This provides additional evidence that the flagged location is reachable, but verification credit is attributed to the campaign that first triggered it. Corroborations are counted separately from new verifications in the progress line (`N corroboration(s)`).
`[!]`	A crash was found during fuzzing but does not correspond to any flagged static analysis location. It is reported as a newly discovered bug and written to the augmented SARIF as a `dynamic-bug-discovery` result.
`[x]`	No crashes were found for this hot-spot within the campaign duration.

Instances — the number in parentheses after each bug entry (e.g. `(7 instances)`) is the count of distinct crash files that all produced the same exception at the same source location. A high instance count suggests the crash is easy to reach with many different inputs, which is a useful indicator of real-world severity.

9.4.7 Crash correlation and confidence scoring

When a crash is found, GNATfuzz analyses the exception traceback to determine whether any flagged SARIF location is responsible. Two correlation modes are supported:

Direct match (high confidence)
: The crash site in the traceback (innermost user-code frame) is the same source location as the SARIF result. This is the strongest form of evidence: the fuzzer triggered an exception at exactly the line that static analysis warned about.

Stack-based match (medium / low confidence)
: The crash occurs inside a callee, but the SARIF location appears higher in the call stack — i.e. the SARIF location is on the path that led to the crash. Confidence is assigned by call depth:

  - *Medium* — the SARIF location is one frame above the crash site (the direct caller of the crashing function).
  - *Low* — the SARIF location is two or more frames above the crash site.

Stack-based correlation is important when SAST flags a call site that passes an unsafe value to a callee: the crash happens inside the callee, but the root cause is the argument prepared at the call site.

GNATfuzz infrastructure frames (harness wrapper and test harness internals) are automatically filtered out of the traceback before correlation, so only user-project source files are considered and included in the results SARIF file.

Cross-hot-spot verification — crashes found during the campaign for one hot-spot are checked against *all* SARIF results across all hot-spots. A single fuzzing run can therefore verify multiple static analysis results simultaneously.

When a crash matches a SAR that was already verified by a *previous* hot-spot's campaign, or matches a SAR from a *different* hot-spot than the current one, the match is displayed as ✓ `Corroborates Hotspot N Warning #M` (cyan) rather than as a new verification (green ✓). This distinguishes fresh evidence from independent confirmation that a location is reachable. The progress line shows the two counts separately: `N warnings verified, M corroboration(s)`.

9.4.8 Augmented SARIF output

The hot-spot workflow produces an augmented copy of the input SARIF file. Every result that GNATfuzz processed receives a `GNATfuzz Result` property that records the fuzzing outcome for that warning.

GNATfuzz Result values

The `GNATfuzz Result` property records the fuzzing outcome for each result. All results processed by GNATfuzz have their `message.text` prefixed with a `[GNATfuzz - ...]` label so the outcome is visible at a glance in any SARIF viewer. Newly discovered bugs that had no matching SAST warning are added as separate results with a `[GNATfuzz - New]` prefix.

`GNATfuzz Result`	Meaning and recommended action
`Verified`	GNATfuzz triggered a runtime exception that can be traced to this location. The `Correlation Confidence` property (`"High"` or `"Medium"`) indicates how directly the crash maps to the flagged line. The result message is updated with a `[GNATfuzz - High/Medium]` prefix. **Action: investigate and fix the bug; use the Reproduction Command to replay it.**
`Not Triggered`	GNATfuzz executed this source line during fuzzing (confirmed by GNATcoverage) but did not trigger a crash. The warning condition was not reached by the inputs explored — fuzzing has limits and may not have found the right input. **Action: consider this lower priority for triage, but do not dismiss the SAST warning. A longer campaign or additional corpus seeds may find a crash.**
`Not Reached`	Coverage data shows this source line was never executed during fuzzing. The SAST finding is inconclusive — the fuzzing campaign did not explore the code path leading to this warning. **Action: consider running with a longer `--campaign-duration` or more cores; or a customer corpus the warning is unverified, not confirmed safe.**
(absent)	The result was not targeted by any fuzzing campaign in this session.

Verified static analysis results

Each SARIF result with `GNATfuzz Result: "Verified"` has:

- **message.text** updated with a `[GNATfuzz - <Confidence>]` prefix followed by the exception name and the original SAST message. For example:

```
[GNATfuzz - High] CONSTRAINT_ERROR - requires Speed_Of_Sound > 0.0
```

  For stack-based matches (where the crash occurred in a callee):

```
[GNATfuzz - Medium] CONSTRAINT_ERROR (raised in callee) - precondition might fail␣
↪on call to risky_function
```

Only the Ada exception *name* appears in the headline — the full Ada runtime message (which includes a redundant file/line reference) is available in the `Exception Type` property.

- **properties** augmented with:

Property	Description
`GNATfuzz Result`	`"Verified"` — the warning was confirmed by GNATfuzz.
`Correlation Confidence`	`"High"` — crash directly at the SAR location. `"Medium"` — crash in a direct callee with the SAR location one frame above the crash site. `"Low"` — crash two or more frames below the SAR location. See *Crash correlation and confidence scoring*.
`Exception Type`	Full Ada exception string, e.g. `"CONSTRAINT_ERROR : example.adb:42 range check failed"`.
`Crash Location`	`"file.adb:N"` — the source line where the exception was actually raised. Present only for stack-based (Medium/Low) matches where the crash site differs from the SAR location.
`Test Case Path`	Path to the binary crash input file that reproduces the exception.
`Reproduction Command`	Shell command to reproduce the crash, e.g. `/path/to/runner/gnatfuzz-test_harness.verbose /path/to/crash_file`.
`Verification Timestamp`	ISO-8601 timestamp when the crash was first found.
`Instance Count`	Number of distinct crash inputs that triggered the same exception at the same location. A high count indicates the crash is easy to reach.
`Result file`	URI of the decoded output file (in `decoded_crashes/` or `decoded_pre_crashes/`) containing the human-readable verbose-runner output for this crash. Present when decoding succeeded.
`Input Parameters`	The decoded Ada input values passed to the subprogram, extracted from the verbose harness JSON output.
`Fuzzed Subprogram`	Fully-qualified Ada name of the subprogram that was fuzzed (which may be a caller of the SAST-flagged subprogram).
`Stack Trace`	Human-readable symbolic traceback extracted from the verbose harness output, showing the full call chain from the entry point to the crash site. GNATfuzz-internal frames are stripped; only user-project frames are shown. See *Stack trace format* for the format and machine-readable equivalent.

Stack trace format

The `Stack Trace` property is a plain-text symbolic traceback with one frame per line, numbered from the innermost (crash) frame outward:

```
#0  Avionics_Simulation.Navigation.Compute_Bearing at navigation.adb:43
#1  Avionics_Simulation.Flight_Plan.Update_Leg at flight_plan.adb:112
#2  Avionics_Simulation.Mission.Run_Step at mission.adb:78
```

Frame `#0` is the crash site — the source line where the exception was raised. Frame `#1` is the direct caller, and so on. This ordering matches the `Correlation Confidence` levels: a SAR at frame `#0` is `"High"`, at frame `#1` is `"Medium"`, and at frame `#2` or deeper is `"Low"`.

The same information is also available as a machine-readable SARIF `result["stacks"]` array. Each entry is a SARIF `stack` object with a `frames` list, where every frame carries both a `logicalLocation` (fully-qualified Ada subprogram name) and a `physicalLocation` (source file and line number). SARIF viewers that understand the `stacks` schema (e.g. Microsoft SARIF Viewer) will render an interactive call-stack panel; viewers that do not will fall back to the `location.message` text, which mirrors the plain-text format above.

Non-verified results (Not Triggered / Not Reached)

Results with `GNATfuzz Result: "Not Triggered"` or `"Not Reached"` have their `message.text` prefixed with the corresponding `[GNATfuzz - ...]` label so they are visually consistent with verified results in SARIF viewers. The original SAST message follows the prefix and is otherwise unchanged. The `engine: "GNATfuzz"` property is also set to identify that GNATfuzz processed the result.

Newly discovered bugs (dynamic-bug-discovery)

Crashes found during fuzzing that do not correspond to any location in the original SARIF are appended as new results with `ruleId` set to `dynamic-bug-discovery`. A rule definition named `GNATfuzz - dynamic-bug-discovery` is injected into `tool.driver.rules` so that SARIF viewers display a human-readable rule name.

These results carry a `[GNATfuzz - New]` prefix in `message.text` to indicate that they are genuinely new findings — bugs the static analyser did not flag but that GNATfuzz discovered through dynamic testing. This makes them easy to distinguish from verified SAST warnings (`[GNATfuzz - High/Medium]`) in any SARIF viewer. For example:

```
[GNATfuzz - New] CONSTRAINT_ERROR raised at mach_calculator.adb:109 during fuzzing (1␣
↪instance)
```

Action: treat these as new bug reports. Use the `Reproduction Command` property to replay the crash and investigate the root cause.

Each dynamically discovered result includes:

- `message.text` — `[GNATfuzz - New] <ExceptionName> raised at <file>:<line> during fuzzing (<N> instance(s))`
- `locations` — the source location of the crash site.
- `properties` — `Test Case Path`, `Reproduction Command`, `Exception Type`, and `Result file` (when decoded output is available).

9.4.9 Machine-readable summary report

At the end of every hot-spot workflow invocation, GNATfuzz writes `hot_spot_report.json` inside the timestamped artifacts directory. This file is intended for CI pipelines, dashboards, and post-run tooling that need to consume results programmatically without screen-scraping terminal output. All path fields in the report are relative to the working directory so the file remains interpretable after the run directory is moved or archived.

Top-level fields

Field	Description
`gnatfuzz_version`	GNATfuzz version string (e.g. `"26.0w"`).
`timestamp`	ISO-8601 UTC timestamp of when the workflow started.
`elapsed_seconds`	Total wall-clock time from workflow start to completion.
`command`	Full command line used to invoke the workflow (`sys.argv`).
`status`	`"complete"` for a full run; `"interrupted"` if the workflow was stopped early with Ctrl-C.
`warnings_verified`	Number of static-analysis warnings confirmed by dynamic testing.
`total_warnings`	Total static-analysis warnings processed in this run.
`bugs_found`	Unique dynamically discovered bugs (crashes with no matching SAST warning) across all campaigns.
`campaigns_run`	Fuzzing campaigns that completed successfully.
`hangs_found`	Total hang inputs found across all hot-spots.
`run_directory`	Relative path to the timestamped run directory.
`log_file`	Relative path to the `hot-spot.log` debug log.
`sarif_input`	Relative path to the input SARIF supplied to the workflow.
`sarif_output`	Relative path to the augmented SARIF output, or `null` when no crashes or verified warnings were found.
`tgen_vectors_files`	List of relative paths to the project TGen JSON files updated during this run (see *TGen test vector integration*).
`tgen_vectors_added`	Number of new test vectors written across all updated TGen files.
`tgen_vectors_dirs`	List of relative paths to per-hot-spot `tgen_vectors/` directories.
`coverage_html_dirs`	List of relative paths to per-hot-spot `coverage/` HTML directories.
`combined_coverage_dir`	Relative path to the project-wide combined coverage report directory, or `null` when no coverage data was produced.
`hotspots`	Array of per-hot-spot result objects (see below).

Per-hot-spot result objects

Each object in the `hotspots` array describes one subprogram that carries static-analysis warnings:

Field	Description
`hotspot_number`	One-based number as displayed in the workflow terminal output.
`subprogram_name`	Fully-qualified Ada subprogram name.
`subprogram_kind`	`"function"`, `"procedure"`, or `null`.
`source_uri`	Source file URI from the SARIF input, or `null`.
`sar_warnings`	Array of per-warning result objects (see below).
`crashes_found`	Array of deduplicated crash result objects (see below).
`hangs_found`	Number of hang inputs found for this hot-spot.
`tgen_vectors_dir`	Relative path to the `tgen_vectors/` directory for this hot-spot, or `null`.
`coverage_html_dir`	Relative path to the `coverage/` HTML report directory, or `null`.
`campaigns_run`	Number of successful campaigns run for this hot-spot.
`skipped`	`true` if no campaign completed (already verified, non-fuzzable, or all attempts failed).
`skip_reason`	Machine-readable reason string when `skipped` is `true`, or `null`.

Each `sar_warnings` entry:

Field	Description
`index`	One-based warning number within the hot-spot.
`location`	`"filename.adb:line"` from the SARIF result.
`rule_id`	SARIF rule identifier (e.g. `"RANGE_CHECK"`).
`level`	SARIF severity (e.g. `"warning"`, `"error"`).
`message`	Human-readable warning text from the SARIF result.
`verified`	`true` when a fuzzing crash confirmed this warning.
`verified_at`	`"file:line"` where the verifying crash occurred, or `null`.
`coverage_status`	`"hit"` — line executed during fuzzing; `"not_reached"` — line never executed; `"no_data"` — coverage run produced no data for this line; `null` — hot-spot was skipped or no coverage run completed.
`reproduction_command`	Shell command to reproduce the crash, or `null` if not verified.

Each `crashes_found` entry:

Field	Description
`location`	`"file.adb:line"` of the crash site.
`exception_type`	Ada exception name and message.
`instance_count`	Distinct crash inputs that triggered this location.
`correlation_confidence`	Best confidence level: `"high"`, `"medium"`, or `"low"`.
`matching_sar_indices`	One-based indices of SARs verified by this crash (empty for new bugs).
`test_case_path`	Relative path to the crash input file.
`decoded_output_path`	Relative path to the decoded verbose-runner output, or `null`.
`reproduction_command`	Shell command to reproduce the crash.

Example report snippet


```
{
  "gnatfuzz_version": "24.0w",
  "timestamp": "2026-01-01T12:00:00+00:00",
  "elapsed_seconds": 187,
  "status": "complete",
  "warnings_verified": 3,
  "total_warnings": 6,
  "bugs_found": 1,
  "campaigns_run": 1,
  "sarif_output": "hot_spot_results/20260101T120000/sast_dast_combined.sarif",
  "combined_coverage_dir": "hot_spot_results/20260101T120000/coverage_report",
  "tgen_vectors_added": 4,
  "hotspots": [
    {
      "hotspot_number": 1,
      "subprogram_name": "mach_calculator.get_mach_value",
      "campaigns_run": 1,
      "skipped": false,
      "sar_warnings": [
        {
          "index": 1,
          "location": "mach_calculator.adb:57",
          "rule_id": "RANGE_CHECK",
          "verified": true,
          "verified_at": "mach_calculator.adb:57",
          "coverage_status": "hit",
          "reproduction_command": "hot_spot_results/.../runner/gnatfuzz-test_harness.
↪verbose hot_spot_results/.../crashes/id:000000,..."
        }
      ],
      "crashes_found": [
        {
          "location": "mach_calculator.adb:57",
          "exception_type": "CONSTRAINT_ERROR : range check failed",
          "instance_count": 7,
          "correlation_confidence": "high",
          "matching_sar_indices": [1],
          "test_case_path": "hot_spot_results/.../crashes/id:000000,...",
          "decoded_output_path": "hot_spot_results/.../decoded_crashes/id:000000,...",
          "reproduction_command": "hot_spot_results/.../runner/gnatfuzz-test_harness.
↪verbose hot_spot_results/.../crashes/id:000000,..."
        }
      ]
    }
  ]
}
```

Using the report in CI

The `status`, `warnings_verified`, and `bugs_found` fields make it easy to integrate the hot-spot workflow into a CI pipeline:

```
gnatfuzz hot-spot --sarif analysis.sarif --cores 4 --campaign-duration 300

# Parse the machine-readable report
python3 - <<'EOF'
import json, sys
report = json.load(open("$(ls -d hot_spot_results/*/hot_spot_report.json | tail -1)"))
verified  = report["warnings_verified"]
total     = report["total_warnings"]
bugs      = report["bugs_found"]
print(f"Verified {verified}/{total} warnings, {bugs} new bugs found")
if verified > 0 or bugs > 0:
    sys.exit(1)   # Fail the pipeline: real crashes were found
EOF
```

9.4.10 Coverage reports

The hot-spot workflow produces two levels of GNATcoverage HTML report.

Per-hot-spot coverage (`coverage/`)

After each fuzzing campaign, GNATcoverage analyses the source traces collected during fuzzing and produces a per-hot-spot HTML report in the `coverage/` directory under the hot-spot's flat layout folder. This report shows which lines of the subprogram and its callees were exercised by the fuzzing session.

Each SAR location's line coverage is also recorded in the machine-readable report as `coverage_status`:

- `"hit"` — the line was reached and executed during fuzzing. A warning with this status that was *not* verified by a crash is a genuine `"Not Triggered"` result: the condition was reachable but the fuzzer did not find an input that triggered the exception.
- `"not_reached"` — the line was never executed. This is a `"Not Reached"` result: the fuzzing campaign did not explore the code path leading to the warning. Consider using a longer --campaign-duration or more --cores.
- `"no_data"` — the coverage run completed but produced no data for this specific line (usually a compiler optimisation or an inline).

Combined project-wide coverage (`coverage_report/`)

After all hot-spot campaigns finish, GNATfuzz merges the per-hot-spot GNATcoverage checkpoints (`coverage_checkpoint.cov`) into a single project-wide DHTML (dynamic HTML) report written to `coverage_report/` in the run directory. This report shows aggregate source coverage across every subprogram exercised during the entire hot-spot run, making it straightforward to assess total dynamic test coverage alongside the SAST verification results.

Open `coverage_report/index.html` in a browser to explore the combined coverage data. The path to this directory is also available in `hot_spot_report.json` as `combined_coverage_dir`.

9.4.11 TGen test vector integration

GNATfuzz integrates with the GNATtest TGen test-vector format to build an accumulating knowledge base of interesting inputs. Each time the workflow runs, it learns from the crashes and coverage-increasing inputs it discovers and stores them so future runs start from a richer corpus.

How it works

The workflow performs two TGen operations for each hot-spot campaign:

1. **Import before building** — GNATfuzz silently locates the project's TGen JSON test-vector file (`<obj>/gnattest/tests/JSON_Tests/<Package>.json`) and, if found, imports the stored vectors into the harness's `user_configuration/` directory. The harness is then compiled with those vectors baked in as additional corpus seeds alongside the TGen type-model generated corpus. This gives each campaign the full benefit of all previously discovered test cases.
2. **Write-back during fuzzing** — with the TGen import active, the fuzzing campaign is started with write-back enabled (`--tgen-gnattest-test-generator`). Whenever the fuzzer discovers a new coverage-increasing or crash-triggering input, the harness converts it to TGen JSON format and appends it to the project TGen file.

The result is a **self-improving corpus**: every time you run the hot-spot workflow on the same project, the fuzzer starts from the enriched corpus produced by all previous runs. Over repeated CI runs the corpus grows to cover more code paths, making it increasingly likely that new warnings are verified quickly.

The `tgen_vectors_added` field in `hot_spot_report.json` records how many new vectors were written to TGen files in the current run. The `tgen_vectors/` directory inside each hot-spot folder stores the per-campaign TGen JSON subdirectories alongside `dast_index.json`, which records which vectors triggered crashes and which SARIF warnings they matched.

The `tgen_vectors/` directory

After the clean-up stage, each hot-spot directory contains a `tgen_vectors/` subdirectory with:

File	Description
`all_crashes.tgen`	Aggregate TGen JSON file containing every crash-triggering input vector found during this campaign, in TGen format. Can be fed directly to GNATtest to reproduce all crashes without the fuzzer.
`id_XXXXXX/<package>.json`	Individual TGen JSON vector file for each crash input. Each crash lives in its own subdirectory named after the AFL crash ID prefix with colons replaced by underscores (e.g. `id_000000` from `id:000000,sig:06,...`). The filename matches `TGen.LAL_Utils.JSON_Test_Filename` for the subprogram (`<parent_package>.lower() + ".json"`).
`id_XXXXXX/<crash_name>.results`	Copy of the raw binary crash input, co-located in the crash subdirectory alongside the TGen vector.
`dast_index.json`	Correlation index recording which vectors triggered crashes, the exception type and source location of each crash, and which SARIF warnings they matched.

`dast_index.json` schema

```
{
  "subprogram": {
    "fully_qualified_name": "Mach_Calculator.Get_Mach_Value",
    "checksum": "a1b2c3d4e5f60718",
    "fuzz_harness_gpr": "../generated_harness/fuzz_test.gpr"
  },
  "aggregate_tgen_file": "all_crashes.tgen",
  "crash_vectors": [
    {
      "tgen_file": "id_000000/mach_calculator.json",
      "results_file": "id_000000/id:000000,sig:06,...results",
      "exception_type": "CONSTRAINT_ERROR : range check failed",
      "location": "mach_calculator.adb:57",
      "sar_matches": [
        {
          "hotspot_idx": 0,
          "sar_idx": 1,
          "confidence": "high"
        }
      ]
    }
  ]
}
```

Reproducing crashes with GNATtest

After each campaign the hot-spot workflow writes the crash vectors back to the project's standard GNATtest TGen JSON location (`<obj>/gnattest/tests/JSON_Tests/<Package>.json`). GNATtest picks them up from that location automatically when you generate a test harness against the user project:

```
gnattest -P myproject.gpr --gen-test-vectors --harness-dir=my_repro_harness/

gprbuild -P my_repro_harness/test_driver.gpr
my_repro_harness/test_runner
```

Each test case reproduces the same exception observed during fuzzing. After fixing the defect, rebuild and re-run to verify all tests pass.

In addition, individual crash vector files (`id_XXXXXX/<package>.json`) can be fed directly to GNATtest to reproduce specific crashes:

```
# Copy the desired crash vector to a separate folder
cp hot_spot_results/.../hotspot1_calculations.calculate/tgen_vectors/id_000000/mach_
↪calculator.json \
    my_crash_folder/

# Generate a test harness with the crash vector directory as input
gnattest -P myproject.gpr --gen-test-vectors --harness-dir=my_repro_harness/ \
    --serialized-test-dir=my_crash_folder
```

For full instructions on setting up the GNATtest runtime and running TGen-based test harnesses, refer to the GNATtest

documentation.

Disabling TGen integration

Pass --no-tgen-import to skip both the import and write-back steps. This is useful when:

- The project's TGen JSON files are managed externally and should not be modified by the hot-spot workflow.
- You want a clean, isolated campaign that does not inherit any prior corpus.

9.4.12 Output directory layout

All fuzzing artifacts are written under `<output-dir>/hot_spot_results/<timestamp>/`. Within that timestamped run directory, each hot-spot has its own subdirectory. After each campaign completes, the deep `attempt*/fuzz_testing/` tree is restructured into a flat layout and build objects are removed to save disk space:

```
hot_spot_results/
└── 20260101T120000/                        <-- timestamped run
    ├── sast_dast_combined.sarif            <--augmented SARIF (updated progressively)
    ├── hot_spot_report.json                <--machine-readable summary for CI/tooling
    ├── hot-spot.log                        <--full debug log for the run
    ├── coverage_report/                    <--combined project-wide GNATcoverage report
    ├── hotspot1_calculations.calculate/    <--per-hotspot flat layout (after clean-up)
    │   ├── crashes/                        <--Fuzzer-detected crash inputs
    │   ├── pre_crashes/                    <--inputs that crashed during corpus screening
    │   ├── decoded_crashes/                <--decoded runner output for each fuzzer-
↪detected crash
    │   ├── decoded_pre_crashes/            <--decoded runner output for each pre-crash
    │   ├── runner/
    │   │   └── gnatfuzz-test_harness.verbose
    │   ├── coverage/                       <--per-hot-spot GNATcoverage HTML report
    │   ├── coverage_checkpoint.cov         <--checkpoint for combined report
    │   ├── tgen_vectors/                   <--per-crash JSON subdirs and dast_index.json
    │   ├── corpus/                         <--seed corpus
    │   ├── logs/
    │   │   ├── gnatfuzz_stats.log
    │   │   └── afl-fuzz.1.log
    │   └── config/                         <--user_configuration/ (if present)
    ├── hotspot2_processor.validate/
    └── hotspot3_engine.run/
```

When a hot-spot's target subprogram is not directly fuzzable, GNATfuzz searches for a fuzzable caller and labels the campaign `attempt2_<caller_name>` (and so on for deeper searches).

If --keep-artifacts is given, the `attempt*/fuzz_testing/` subtree is retained instead of the flat layout above. Crash files are still decoded in-place in this mode so that decoded outputs and stack traces appear in the SARIF.

The `decoded_crashes/` and `decoded_pre_crashes/` directories are populated automatically during clean-up. Each file has the same name as the corresponding crash input and contains the full verbose runner output — decoded Ada input values, the exception message, and the symbolic traceback — making crashes immediately human-readable without re-running the runner manually.

9.5 Introducing the fuzz-everything workflow

Up to this point, GNATfuzz can be utilized for testing an individual subprogram by manually invoking all the necessary GNATfuzz commands, either through the command line interface (CLI) or via IDE integration. This workflow is well-suited for using GNATfuzz as a companion tool during software development to identify and eliminate code vulnerabilities at the earliest stage possible of the software development lifecycle, just before the code leaves the programmer's local development environment.

The GNATFuzz Fuzz-everything workflow aims to automate and scale further GNATfuzz's usage, making it suitable for project-level testing and integration with modern DevOps testing solutions, such as CI/CD pipelines and nightly testing approaches.

The automation aspect is enabled mainly by:

- Automating the execution of all GNATfuzz commands needed to prepare and launch a fuzz-testing session for a targeted subprogram.
- Automating post-processing steps, such as the creation of project-wide coverage reports and decoding the unique test cases and crashes discovered by the fuzzer. These steps make it easier and faster for the user to debug any issues and gain confidence in their software's robustness.

The scaling aspect is enabled mainly by:

- Being able to fuzz-test multiple (or all) subprograms of a project.
- Enabling the parallel execution of GNATfuzz commands for multiple subprograms when the GNATfuzz command permits it (currently, GNATfuzz supports parallelization for its **generate**, **generate-corpus**, and **minimize-corpus** modes). This significantly speeds up a fuzz-everything multiple-subprogram testing session. Future versions of GNATfuzz will explore the parallelization of other GNATfuzz modes.

Fuzz-everything automation and scaling enable valuable features that significantly enhance GNATfuzz's added value:

- It amalgamates the collected coverage from all fuzz-tested subprograms to provide a project-wide coverage report, enabling the user to understand which parts of their project's source code were tested.
- It generates valuable, human-readable, and machine-parsable reports for all results, helping to automate any pre-processing required by the user to gain valuable insights from the testing campaign.
- It supports various corpus strategies that can significantly increase code coverage.

9.5.1 Accessing the fuzz-everything workflow

Fuzz-everything can be invoked via the GNATfuzz Command Line Interface (CLI):

Listing 3: GNATfuzz's CLI workflows

```
# Command executed
$ gnatfuzz --help

# Output
Usage: gnatfuzz [OPTIONS] COMMAND [ARGS]...

GNATfuzz command-line interface.

Options:
--version  Show the version and exit.
--help     Show this message and exit.
```

(continues on next page)

(continued from previous page)

```
Commands:
analyze     Analyze a GPR project file and find all fuzzable subprograms.
generate    Generate a fuzz test harness for a subprogram.
generate-corpus  Generate the starting corpus used to seed the fuzzing campaign.
fuzz        Build, run, monitor and stop a fuzzing campaign.

Workflows:
fuzz-everything  Top-level group for fuzz-everything commands.
```

Then, the --help switch can be passed to the fuzz-everything workflow for further guidance:

Listing 4: Fuzz-everything CLI

```
# Command executed
$ gnatfuzz fuzz-everything --help

# Output
Usage: gnatfuzz fuzz-everything [OPTIONS] COMMAND [ARGS]...

Top-level group for fuzz-everything commands.

Options:
--help  Show this message and exit.

Commands:
inspect  Provide information to help configure a fuzz-everything testing campaign.
test     Fuzz-test all GNATfuzz-supported subprograms in a GPR project.
```

Fuzz-everything currently suppors two modes, **inspect** and **test**.

9.5.2 Fuzz-everything *"inspect"* mode command line interface

In this mode fuzz-everything has the following command line interface:

Listing 5: Fuzz-everything inspect mode help menu

```
# Command executed
$ gnatfuzz fuzz-everything inspect --help

# Output
Usage: gnatfuzz fuzz-everything inspect [OPTIONS]

Provide information to help configure a fuzz-everything testing campaign.
```

Where the available options are:

- **-P <PROJECT>**
  Path to the user project GPR to inspect Mandatory
- **-X <NAME=VALUE>**
  Sets the external reference `NAME` in the system-under-test's GPR project to `VALUE`
- **--disable-styled-output**
  By default fuzz-everything may style some of its output using ANSI escape sequences. This switch can be

used to disable this behavior

- **--include-subprojects**
  Process the root project and any included subprojects
- **--help**
  Shows the usage information for the **inspect** mode and exits

9.5.3 Fuzz-everything *"test"* mode command line interface

In this mode fuzz-everything has the following command line interface:

Listing 6: Fuzz-everything test mode help menu

```
# Command executed
$ gnatfuzz fuzz-everything test --help

# Output
Usage: gnatfuzz fuzz-everything test [OPTIONS]

Fuzz-test all GNATfuzz-supported subprograms in a GPR project.
```

Where the available options are:

- **-P <PROJECT>**
  Path to the user project GPR to fuzz-test Mandatory
- **-X <NAME=VALUE>**
  Sets the external reference `NAME` in the system-under-test's GPR project to `VALUE`
- **--disable-styled-output**
  By default fuzz-everything may style some of its output using ANSI escape sequences. This switch can be used to disable this behavior
- **--include-subprojects**
  Process the root project and any included subprojects
- **--engines**
  A comma-separated set of fuzzing engines to be enabled. Unix-like OS permitted Values: afl, libfuzzer, cmplog, symcc [Unix-like OS default: afl,cmplog] Windows OS permitted Values: libfuzzer [Windows OS default: libfuzzer]
- **--serial-mode**
  Avoid any parallelism
- **--core-utilization**
  The percentage of cores to utilize (minimum required cores: 2) Permitted Values: 25<=Value<=100 Default Value: 90
- **--corpus-strategy**
  Set the corpus-generation strategy. "auto" - use only valid corpus when automatic generation is supported, "dummy" - always use a dummy value Permitted Values: [auto | dummy] Default Value: auto
- **--disable-project-build-validation**
  Disable project build validation. Note that an invalid project might cause several GNATfuzz stages to fail
- **--stop-criteria**
  Path to the XML stopping criteria file to be used Default Value: printed on terminal

- **--timeout**
  Max duration in minutes for testing each subprogram (0 to disable). When this limit is reached, testing aborts regardless of any stop-criteria configuration. Default Value: 0
- **--subprogram-ids-to-test**
  Only test subprograms with the given IDs. Input must be a range (e.g. 1..7) or a comma-separated list (e.g. 1,2,3)
- **--help**
  Shows the usage information for the **inspect** mode and exits

9.5.4 Fuzz-everything *"inspect"* mode usage example

It is a good practice to always "inspect" a project to understand several factors that can affect its testing via the fuzz-everything workflow.

Let us **inspect** the `fuzz_everything_usage` project located under:

```
<install_prefix>/share/examples/gnatfuzz/fuzz_everything_test/
```

The example package contains the following subprogram definitions:

Listing 7: Fuzz-everything usage example

```
package Test is

   function Is_Divisible (X : Integer; Y : Integer) return Boolean;
   --  checks if X is divisible by Y

   function Division (X : Integer; Y : Integer) return Integer;
   --  devide and return X if X/Y = 1 otherwise return Y

end Test;
```

By running the fuzz-everything **inspect** mode on this project, we get the following output:

Listing 8: Example output from the fuzz-everything **inspect** mode

```
# Command executed
$ gnatfuzz fuzz-everything inspect -P test.gpr --disable-styled-output

# Output
Inspecting project: test.gpr

Executing GNATfuzz's analyze stage: ✓

-------------------------------------------------------
               Inspect-Mode Information
-------------------------------------------------------

Subprogram Information
----------------------
Detected fuzzable subprograms             : 2
Detected non-fuzzable subprograms         : 0
Fuzzable subprograms ID range             : [1-2]
```

(continues on next page)

(continued from previous page)

```
Fuzz-Testing Information
------------------------
Detected cores                              : 20
Usable cores @ 90% utilization (default)  : 18
Recommended corpus-generation strategy    : auto

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

The **inspect** mode will first execute gnatfuzz's **analyze** mode and then produce a screen report. The report informs the user of the number of subprograms for which GNATfuzz can automatically generate a test harness and a starting corpus. It will also report the range of the IDs for these subprograms. This could be useful in selecting which subprograms to fuzz-test when using the fuzz-everything test mode.

The output also indicates the number of cores available on the machine where the **inspect** mode was executed and how many will be utilized based on the default core utilization percentage of the fuzz-everything test mode. This is crucial for the user to understand how to allocate the resources on their testing machine effectively. The more cores you use, the better it is for speeding up the fuzz-testing session and achieving increased code coverage; however, we must be mindful of other processing requirements on the testing machine. Trying to allocate all cores to the fuzz-everything utility while other applications are executed might stall the machine.

The number of available cores and the number of fuzzable subprograms can also guide the user in building stopping criteria for a fuzz-everything campaign that can strike a good balance between the campaign's feedback time and the degree of allowing the fuzzer to exploit more of the project's source code.

Finally, the **inspect** mode recommends the most suitable corpus strategy for fuzz-testing the project with fuzz-everything. We will examine the various fuzz-everything corpus strategies in the following section.

9.5.5 Fuzz-everything *"test"* mode usage example

By running the fuzz-everything **test** mode on the `fuzz_everything_usage` example project, we get the following output:

Listing 9: Example output from the fuzz-everything **test** mode

```
# Command executed
$ gnatfuzz fuzz-everything test -P test.gpr
-------------------------------------------------------------------
                     Runtime Information
-------------------------------------------------------------------

Resources & Strategy
--------------------
Cores detected        : 20
Cores to be used      : 18
Corpus strategy used  : auto

Reporting Format
----------------
Progress format       : subprograms-actioned/fuzzable-subprograms
Estimated wait format : D:H:M:S
```

(continues on next page)

(continued from previous page)

```
User's project build validation: ✓

Executing GNATfuzz's analyze stage: ✓

Executing GNATfuzz's generate stage:

Progress: 2/2, Wait: 0:00:00:00

Executing GNATfuzz's build stage:

Progress: 2/2, Wait: 0:00:00:00

Executing GNATfuzz's corpus generation stage:

Progress: 2/2, Wait: 0:00:00:00

Executing GNATfuzz's minimize corpus stage:

Progress: 2/2, Wait: 0:00:00:00

Executing GNATfuzz's fuzz-testing stage:

Progress: 2/2, Wait: 0:00:00:00

Elapsed Time (D:H:M:S): 0:00:02:39


------------------------------------------------------------------
                          Results
------------------------------------------------------------------

Files & Paths
-------------
Project                              : <path_to_fuzz_everything_example>/user_project.
↪gpr
Results JSON                         : <path_to_fuzz_everything_example>/session_
↪artifacts_2025-06-30T14-55-49_utc_iso/results.json
GNATfuzz issues log                  : <path_to_fuzz_everything_example>/session_
↪artifacts_2025-06-30T14-55-49_utc_iso/issues.log
Session log file                     : <path_to_fuzz_everything_example>/session_
↪artifacts_2025-06-30T14-55-49_utc_iso/session.log
Coverage report                      : <path_to_fuzz_everything_example>/session_
↪artifacts_2025-06-30T14-55-49_utc_iso/coverage_reports/index.html

Generation Information
----------------------
Detected fuzzable subprograms         : 2
Number of subprograms set for testing : 2
Failed generate mode                  : 0
Failed build mode                     : 0
Failed corpus-gen mode                : 0
```

(continues on next page)

(continued from previous page)

```
Failed corpus-minimization mode       : 0
Subprograms using dummy-corpus        : 0

Fuzz-Testing Information
------------------------
Failed fuzz mode                      : 0
Fuzz mode timeouts                    : 0
Subprograms with crashes              : 2
Subprograms with hangs                : 0
Statements covered                    : 7
Statements partially covered          : 0
Statements not covered                : 0

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

The test mode screen output is divided into two sections:

1. **Runtime Information Section**: This section informs the user about critical configurations of the testing session, such as the core utilization number on the test machine and the corpus strategy to be deployed. It also provides the expected formatting of the mode's screen progress reporting. The section also provides live updates on the progress of each stage, as executed by the **test** mode of fuzz-everything, and the total elapsed time for completing all stages.
2. **Results Section**: This section includes paths to all the essential generated artifacts of the completed test session, as well as an overview of the results.

Let's examine some of the most useful **test** mode output information and artifacts in conjunction with their corresponding fuzz-everything functionality:

Core utilization implications

The fuzz-everything test mode output displays the progress of each stage. A ✓ represents the successful completion of stages that are not concerned with multiple subprograms, and they only invoke a single execution. This includes the user's project build validation and GNATfuzz's analysis stages. These stages are not concerned with the specified core utilization. A failure at any of these stages will result in termination of the fuzz-everything test mode. The project build validation can also be disabled via the --disable-styled-output, if the user is confident that the project builds successfully.

Fuzz-everything test mode, by default, invokes parallel executions for GNATfuzz's **generate** and **generate-corpus** modes. It will execute batches of these modes, with each batch working on as many subprograms as the number of cores specified with the --core-utilization switch. The process will continue until all subprograms scheduled for execution by the currently executing GNATfuzz mode are completed. Note that if a GNATfuzz mode fails on a subprogram, then the subprogram is not scheduled for execution by the next GNATfuzz mode.

The fuzz stage always tests one subprogram at a time, but uses all the specified number of cores to launch multiple fuzz-testing nodes.

For the generate, gen-corpus, and fuzz stages, fuzz-everything provides an estimated wait time for the completion of each stage. This time is updated periodically depending on the cores' utilization and the number of subprograms to be executed by each stage. If the user requires all stages to execute in serial mode, they can use the --serial-mode option.

Test-mode generated artifacts directory

Executing a fuzz-everything test mode session on a project will create an artifacts directory at the exact location of the project's GPR file. Listing 10 demonstrates the structure of this directory:

Listing 10: Fuzz-everything **test** mode session directory structure

```
├── session_artifacts_<timestamp>/
│   ├── coverage_reports/
│   ├── generation_for_subprogram_with_id_1/
│   ├── generation_for_subprogram_with_id_2/
│   ├── ...
│   ├── generation_for_subprogram_with_id_n/
│   ├── gnatcov_merged_checkpoint.cov
│   ├── issues.log
│   ├── results.json
│   └── session.log
```

Here is a description of all the artifacts generated by a fuzz-everything testing session:

- `session_artifacts_<timestamp>/` – Top-level directory that holds all the artifacts for a fuzz-everything testing session. The timestamp naming convention allows for the independent storage of artifacts from multiple testing sessions.
- `coverage_reports/` – Holds all GNATcoverage generated reports (e.g., HTML, XCOV, SARIF, Cobertura) for the session.
- `generation_for_subprogram_with_id_n/` – GNATfuzz generation folder, annotated with the subprogram's GNATfuzz analysis mode-allocated ID (which can be found within the mode's generated `analysis.json`). For further information on the contents of these directories, please refer to Listing 5.
- `gnatcov_merged_checkpoint.cov` – The final GNATcoverage checkpoint with the total coverage achieved for the project under test. Suppose the user decides to test their project incrementally, using the fuzz-everything test-mode's --subprogram-ids-to-test configuration, instead of testing all the fuzzable subprograms of a project in a single session. In that case, the checkpoints from the individual sessions can be combined to retrieve the project-wide collective coverage reports. For more information on how to achieve this, please refer to the GNATcoverage User's Guide.
- `issues.log` – **Test** mode will attempt to fuzz-test all avaliable fuzzable subprograms and may uncover some GNATfuzz issues in the process. In that case, fuzz-everything will silently document a bug report to this log file, containing all the necessary information to reproduce the problem, and proceed to the following subprograms. If you encounter such issues, we encourage you to report them to AdaCore.
- `session.log` – The diagnostics log file generated by fuzz-everything during a testing session.
- `results.json` – The information in this file is comprehensive, including overall campaign statistics such as the number of successfully tested subprograms and failures, for each of GNATfuzz's modes. It will also include the paths to all the essential generated artifacts, such as the overall project's code coverage reports and the stopping criteria used. Listing 11 demonstrates the structure of this file for the running example in this section. The information provided by this file should be sufficient for investigating any crashes or hangs captured, as well as identifying test cases that increase a project's code coverage and can potentially be used as part of a unit testing framework for the same project. Note that fuzz-everything will automatically decode and store all test cases, crashes, and hangs in a human-readable format for easy inspection of their values and an initial indication of what caused a specific issue. The paths to the folders of decoded test cases are also provided in the `results.json` for each fuzz-tested subprogram.

Listing 11: Fuzz-everything test-mode's `results.json` file example

```
{
   "datetime_utc_iso_format": "2025-07-03T12:44:10.036487+00:00",
   "utc_timestamp": "1751546650.036487",
   "project_gpr": "<path_to_project_under_test_directory>/test.gpr",
   "stop_criteria": "<path_to_project_under_test_directory>/stop_criteria.xml",
   "global_coverage_reports_path": "<path_to_session_artifacts_directory>/coverage_
↪reports",
   "stats": {
      "fuzzable_subprograms_detected": 2,
      "subprograms_using_dummy_corpus": 0,
      "failed_generate_mode": 0,
      "failed_build_mode": 0,
      "failed_corpus_gen_mode": 0,
      "failed_corpus_minimization_mode": 0,
      "failed_fuzz_mode": 0,
      "fuzz_mode_timeouts": 0,
      "subprograms_with_crashes": 2,
      "subprograms_with_hangs": 0,
      "statements_covered": 7,
      "statements_partially_covered": 0,
      "statements_not_covered": 0
   },
   "results": [
      {
            "id": 1,
            "generic_instantiation": false,
            "source_filename": "<path_to_project_under_test_sources_directory>/simple.
↪ads",
            "start_line": 3,
            "generate_folder": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_1",
            "corpus_type": "automatically_generated",
            "gnatfuzz_command": "<the exact gnatfuzz fuzz command and arguments␣
↪executed>",
            "unique_test_cases_count": 6,
            "crashes_count": 2,
            "hangs_count": 0,
            "ending_corpus_path": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_1/fuzz_testing/session/results/ending_corpus",
            "crashes_path": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_1/fuzz_testing/session/results/crashes",
            "hangs_path": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_1/fuzz_testing/session/results/hangs",
            "decoded_ending_corpus_path": "<path_to_session_artifacts_directory>/
↪generation_for_subprogram_with_id_1/fuzz_testing/session/results/decoded_ending_corpus
↪",
            "decoded_crashes_path": "<path_to_session_artifacts_directory>/generation_
↪for_subprogram_with_id_1/fuzz_testing/session/results/decoded_crashes",
            "fuzz_log_path": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_1/fuzz_testing/session/fuzz.log",
            "coverage_html_report": "<path_to_session_artifacts_directory>/generation_
```

(continues on next page)

(continued from previous page)

```
↪for_subprogram_with_id_1/fuzz_testing/session/coverage_output/index.html",
41              "lines_fully_covered_in_project": 3,
42              "gnatcov_checkpoint": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_1/fuzz_testing/session/.temp/GNATfuzz_Checkpoint",
43              "timeout": false
44          },
45          {
46              "id": 2,
47              "generic_instantiation": false,
48              "source_filename": "<path_to_project_under_test_sources_directory>/simple.
↪ads",
49              "start_line": 6,
50              "generate_folder": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_2",
51              "corpus_type": "automatically_generated",
52              "gnatfuzz_command": "<the exact gnatfuzz fuzz command and arguments␣
↪executed>",
53              "unique_test_cases_count": 4,
54              "crashes_count": 4,
55              "hangs_count": 0,
56              "ending_corpus_path": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_2/fuzz_testing/session/results/ending_corpus",
57              "crashes_path": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_2/fuzz_testing/session/results/crashes",
58              "hangs_path": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_2/fuzz_testing/session/results/hangs",
59              "decoded_ending_corpus_path": "<path_to_session_artifacts_directory>/
↪generation_for_subprogram_with_id_2/fuzz_testing/session/results/decoded_ending_corpus
↪",
60              "decoded_crashes_path": "<path_to_session_artifacts_directory>/generation_
↪for_subprogram_with_id_2/fuzz_testing/session/results/decoded_crashes",
61              "fuzz_log_path": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_2/fuzz_testing/session/fuzz.log",
62              "coverage_html_report": "<path_to_session_artifacts_directory>/generation_
↪for_subprogram_with_id_2/fuzz_testing/session/coverage_output/index.html",
63              "lines_fully_covered_in_project": 4,
64              "gnatcov_checkpoint": "<path_to_session_artifacts_directory>/generation_for_
↪subprogram_with_id_2/fuzz_testing/session/.temp/GNATfuzz_Checkpoint",
65              "timeout": false
66          }
67      ]
68  }
```

Corpus strategy

For our running example, we have used automatic corpus generation. The user can also experiment with using the **dummy** corpus strategy. This strategy will bypass the corpus generation and minimization steps entirely and test all fuzzable subprograms with a dummy corpus. Avoiding the corpus generation stage will speed up execution in **test** mode. It could also yield a different set of interesting test cases compared to the automatic corpus generation strategy. Furthermore, in some cases, it can compensate when the corpus minimization stage fails on an automatically generated starting corpus. Note that, although experimentation demonstrated that a dummy corpus could be used with a high success rate, there are no guarantees that the fuzzer will always be able to convert dummy test cases into valid ones, depending on the complexity of the parameter types for the subprogram under test.

Project-wide coverage reporting

Fuzz-everything will automatically aggregate the code coverage achieved from all the subprograms it successfully fuzz-tested to create project-level coverage reports in multiple formats (e.g., HTML, XCOV, SARIF, Cobertura). These can be found under the generated session-artifacts directory. The coverage reports can provide valuable insights into which parts of your project have yet to be tested. Such information can help you target any subsequent fuzz-everything testing sessions to gain extra code coverage and, thus, increase your confidence in your project's code robustness.

Timeout configuration usage

Fuzz-everything **test** mode supports a --timeout configuration. This sets the maximum time in minutes for which GNATfuzz's fuzz mode should be executed for each fuzzable subprogram to be tested. It is a good practice to use this configuration, especially in cases of large campaign sessions, which are expected to test more than a couple of subprograms. This can safeguard against the unlikely scenario of an internal non-termination issue in GNATfuzz's fuzz mode. When the configuration is set, the user must ensure that their provided timeout does not override their stopping criteria; for example, not setting a timeout shorter than the value specified for the elapsed time stopping criterion.

Default stopping criteria

Listing 12 shows the default stopping used by the fuzz-everything **test** mode. These stopping criteria will be applied for each subprogram targeted within a fuzz-everything test session.

Listing 12: Fuzz-everything test mode default stopping criteria

```
<GNATfuzz>
   <afl_stop_criteria>
      <rule_grouper type="or">
            <rule>
               <display_formuliac>false</display_formuliac>
               <display_node_status>true</display_node_status>
               <state>stability</state>
               <operator>lt</operator>
               <value>90</value>
               <all_sessions>false</all_sessions>
         </rule>
         <rule>
               <state>elapsed_time</state>
               <operator>gt</operator>
               <value>40</value>
```

(continues on next page)

(continued from previous page)

```
            <all_sessions>false</all_sessions>
        </rule>
        <rule>
            <state>coverage</state>
            <operator>eq</operator>
            <value>100</value>
            <all_sessions>false</all_sessions>
        </rule>
    </rule_grouper>
  </afl_stop_criteria>
</GNATfuzz>
```

These default stopping criteria specify that each subprogram tested by the fuzz-everything will only terminate its testing if:

1. the stability of the main AFL++ node drops below 90%, or
2. the elapsed time is greater than 40 seconds, or
3. the achieved code coverage reached 100%.

The provided default stopping criteria are not meant to allow GNATfuzz to reach its full potential when testing a subprogram. Instead, they aim for a quick exploitation session of all the subprograms scheduled to be tested by fuzz-everything. This is particularly useful when the user wants to conduct a brief, first-time explanatory session to see how the fuzz-everything **test** mode performs on a new project. For example, this session can provide an initial project-wide coverage report, enabling the user to understand which areas of the project require the most focus for testing. It can also capture any potential GNATfuzz issues on specific subprograms that the user can report for resolving before engaging in extended testing sessions.

As discussed in the *Fuzz-everything "inspect" mode usage example* section, the fuzz-everything **inspect** mode can help users estimate their total test session execution time based on their defined stopping criteria. The user goal is to define a set of stopping criteria rules that allow sufficient time for GNATfuzz to explore each fuzzable subprogram, while also ensuring the completion of the testing session within a timeframe that the users consider reasonable. For more information on defining appropriate stopping criteria, please refer to the *GNATfuzz stopping criteria* section.

9.5.6 Utilizing fuzz-everything-generated test cases with GNATtest

All test cases generated by a fuzz-everything testing session are stored in JSON format in files under the `<obj_dir>/gnattest/tests/JSON_Tests/` directory. GNATtest can then utilize these test cases for a GNATtest-generated test harness of the same project. A workflow to achieve this is:

1. Execute a fuzz-everything testing session on the selected project.
2. Verify that the `<obj_dir>/gnattest/tests/JSON_Tests/` directory was created and is not empty.
3. Set up the test generation runtime (for instructions, refer to the relevant section in the GNATtest documentation.)
4. Open the user project with GNATstudio.
5. First, create a GNATtest harness project if it doesn't already exist, using the `Analyze -> GNATtest -> Generate Harness Project Entries` in the menu bar (you will be able to select the `no-subprojects` option on the pop-up window if you have not included subprojects during the fuzz-everything testing session).
6. Once a test harness is generated, you will be able to see all the fuzz-everything generated test cases loaded in the Tests View of GNATstudio.
7. You can also create a coverage report based on the imported test-cases by using the `Analyze -> Coverage -> GNATcoverage Source Traces -> Run All Actions` entries in the menu bar.

9.6 Libfuzzer

9.6.1 Libfuzzer basics

This section provides a high-level overview of libfuzzer, an in-process, coverage-guided fuzzer that works by linking directly against the library under test. In contrast to AFL++, libfuzzer does not execute the system under test as a separate process; instead, it repeatedly calls a dedicated fuzzing target function with the mutated input.

To utilize libfuzzer with GNATfuzz, the libfuzzer option must be passed to GNATfuzz's fuzz mode --engines switch. All the other GNATfuzz commands and operations required no further configuration.

Note: This document is not intended to be a substitute for the libfuzzer documentation, but rather to provide some information needed to understand libfuzzer and how it works. In this document, we mostly focus on the added value of the GNATfuzz toolchain. For more information about libfuzzer, you can visit the LLVM Compiler User's Manual on Libfuzzer.

Key Statistics

Libfuzzer provides its own set of execution statistics. This can be viewed and monitored under the `<generated-test-harness>/fuzz_testing/session/libfuzzer.log`, where `<generated-test-harness>` is the location specified to the **generate** mode to create the test harness. The main statistics included and their meaning are:

- **runs:** Total number of executions of the target function.
- **exec/s:** Execution rate (executions per second).
- **cover:** Total unique code coverage edges found.
- **corp:** Current size of the in-memory corpus.
- **reloads:** Number of times the fuzzer has reloaded the shared corpus from disk for synchronization.

9.6.2 Libfuzzer requirements

GNAT Pro LLVM toolchain installation is required to enable the use of the libfuzzer engine with GNATfuzz.

No special operating system configuration commands are required for libfuzzer to run successfully.

9.6.3 GNATfuzz and the libfuzzer execution configuration

Libfuzzer is a highly configurable fuzzing engine. GNATfuzz utilizes a predefined set of configurations to execute libfuzzer, striking a good balance between required functionality, resource utilization, and logging progress. The current configurations are:

- -fork: 2 minimum. If libfuzzer is the only active engine and --cores is specified, then --fork is set by the --cores value - Enables multi-process execution. The main libfuzzer process acts as a manager, creating and synchronizing other fuzzing processes (which try to allocate individual cores). This allows GNATfuzz to handle the multi-core aspects of the fuzzing campaign.
- -ignore_crashes: 1 - Required when using -fork. This disables libfuzzer's default behavior of stopping when it detects the first crash, allowing the fuzzing campaign to continue and find more issues.

- -detect_leaks: `0` - Used when the address sanitizer is enabled. This prevents the fuzzer from stopping upon the first detected memory leak.
- -use_value_profile: `1` - Activates value profile tracking, which improves fuzzer effectiveness by tracking values used in comparisons and using them to generate better test cases.
- -timeout: `1` (second) - Sets the maximum time (in seconds) for a single test case execution before it is declared a hang. This one-second value matches the default behavior of AFL++.
- -reload: `10` (seconds) - Specifies the interval (in seconds) at which libfuzzer reloads test cases from the synchronization folder. This enables libfuzzer to use test cases discovered by other fuzzing engines (like AFL++) running concurrently.
- -print_final_stats: `1` - Instructs libfuzzer to print a final summary of all overall statistics at the end of the fuzzing session.
- -verbosity and -print_funcs: `1` - Controls the output level. Level `1` is set along with -print_funcs=1 to improve reporting by including newly covered functions.

Part IV

Using GNATtest with GNATcoverage

Using GNATcoverage to obtain coverage information on the project being tested is the next logical step in the GNATtest workflow. There are two main ways to accomplish this: *using the facilities generated by GNATtest* (namely a Makefile) during the harness generation, or *deriving a custom solution*, while keeping a few important concepts in mind.

CHAPTER

ONE

USING THE MAKEFILE GENERATED BY GNATTEST

GNATtest will always generate a Makefile to automate as much as possible the production of a coverage report on the project being tested.

This Makefile is generated in the same directory as the harness project, and consists of two files:

- The main `Makefile` which is regenerated for each invocation of `gnattest`
- The `coverage_settings.mk` secondary makefile, which holds most of the options that will be passed to gnatcoverage. The values present when the file is generated are extracted from the root project file that was passed to GNATtest. This file will not be re-generated by subsequent invocation of `gnattest`, so it is a suitable place to store settings for GNATcoverage if they are not all defined in the project file.

The makefile also assumes that the GNATcoverage runtime is already built and installed, and that the path to its project file is in the environment variable `GPR_PROJECT_PATH` so it can be automatically located by GPRbuild. It is also possible to specify the path to the project file in the `GNATCOV_RTS` variable in the `coverage_settings.mk` file. See *Setting up the coverage runtime library* for instructions on building and installing the runtime.

If all the switches that need to be passed to GNATcoverage are correctly defined in `coverage_Settings.mk`, then running:

```
make coverage
```

will build and run all the harness projects and then generate a coverage report for all the units under test.

By default, coverage data for a native project is obtained using *source traces* and coverage for projects targeting a cross environment is assessed using *binary traces*.

For cases where the default behavior isn't appropriate for the project under test, the rules for producing a coverage report from both kind of traces are always generated.

1.1 Special considerations for cross environment targets

Using GNATcoverage to assess coverage for a project with a cross-environment target may require some special adaptation to be able to run the program and collect execution traces. The generated makefile tries to implement an execution and trace collection strategy that is the most likely to work for most projects, based on GNATemulator for execution.

For binary traces, this is done through the `gnatcov run` command (see *gnatcov run command line*), which takes care of setting up the trace collection mechanism in GNATemulator. For this configuration to work properly, the `Board` attribute in the package `Emulator` needs to be specified in the root project file, or alternatively the `GNATEMU_BOARD` variable must be set in the `coverage_settings.mk` secondary makefile.

For source traces, the instrumented program will be run using GNATemulator directly, and assumes that the standard package Ada.GNAT_IO allows data to be output on a serial port. The instrumentation process overrides the two switches `--dump-trigger` and `--dump-channel` to the values `main-end` and `base64-stdout` respectively as they

are the most adapted to the GNATtest harness project and to the GNATemulator execution environment. The program is run under GNATemulator, and the output of the first serial port is captured as it contains the trace in a base-64 encoding. This text trace is then decoded and converted into a regular source trace.

There is currently no out-of-the-box support for running the GNATtest harness projects on target and collect execution traces from a hardware probe; the current makefile can still be used to compile the executables and process the traces once they are available, but the execution rules will need rewriting.

1.2 Usage example

In this example the project has two units, `Pkg1` and `Pkg2`. The code in `Pkg2` calls all the subprograms in `Pkg1`, and we wish to unit test each unit in isolation. To do so, we will use the *Individual Test Drivers* feature of GNATtest so each unit under test gets its own test executable, and we'll be able to create each time a checkpoint containing coverage for the unit being tested. See *Handle incidental coverage effects* on how checkpoints help avoid incidental coverage.

The project is structured as follows:

```
<project root directory/>
  |prj.gpr
  |src/
    |pkg1.ads
    |pkg2.ads
    ...
  |tests/
    |<gnattest generated test skeletons>
  |harness/
    |Makefile
    |coverage_settings.mk
    |<GNATtest generated harness files>
```

and is defined with the following project file:

```
--  prj.gpr
project Prj is
   for Source_Dirs use ("src");
   for Object_Dir use "obj";

   package Coverage is
      Cov_Level := ("--level=stmt");
      for Switches ("instrument") use ("--dump-trigger=atexit") & Cov_Level;
      for Switches ("run") use Cov_Level;
      for Switches ("coverage") use ("--annotate=report") & Cov_Level;
   end Coverage;

   package Gnattest is
      for Tests_Dir use "../tests";
      for Harness_Dir use "../harness";
   end Gnattest;

end Prj;
```

Specifying the coverage preferences in the project file spares us modifying the `coverage_settings.mk` after generation.

The tests will be stored in the `tests` directory, as specified by the project attribute `Tests_Dir`, and the GNATtest generated harness in the `harness` directory.

After invoking GNATtest as follows:

```
gnattest -P prj.gpr --separate-drivers=unit
```

The following two files can be found in the harness directory:

- The `Makefile` with (amongst other things) three rules that execute the full coverage workflow for each test driver project, and then generates a coverage report combining the results:

  ```
  bin-coverage: ...

  inst-coverage: ...

  coverage: inst-coverage
  ```

  The first rule (`bin-coverage`) runs the binary traces workflow for GNATcoverage, whereas the second rule (`inst-coverage`) runs the source-trace (or instrumentation based) workflow. The last rule (`coverage`) is defined to use the workflow the most likely to work given the current target.

- The `coverage_settings.mk` file, which, when generated, copied all the values of the relevant root project attributes into corresponding variables:

  ```
  # Settings in this file were extracted from the source project
  # or are gnattest default values if they weren't specified in the source
  # project. They may need adjustments to fit your particular coverage needs.
  # This file won't be overwritten when regenerating the harness.

  # Switches for the various gnatcov commands
  SWITCHES_INSTRUMENT=--dump-trigger=main-end --level=stmt

  SWITCHES_RUN=--level=stmt

  SWITCHES_COVERAGE=--annotate=report --level=stmt

  # Path to the installed gnatcov rts project file.
  # No need to specify it if the project file path was added to the
  # GPR_PROJECT_PATH environment variable.
  GNATCOV_RTS=
  ```

  There is an empty `GNATCOV_RTS` variable defined, which we can set to the path to the installed gnatcov runtime project file.

Once the tests are all written, generating the coverage report (on the standard output in this example) can be done by simply invoking:

```
make -C harness/ coverage
```

This outputs the sequence of commands issued to perform the coverage computation, then the results if the "report" format is selected. On our example, this would be like:

```
Instrumenting project Pkg1.Test_Data.Tests/test_driver.gpr:
gnatcov instrument -PPkg1.Test_Data.Tests/test_driver.gpr --dump-trigger=main-end --
↪level=stmt   --projects=Prj --units=@Pkg1.Test_Data.Tests/units.list
```

(continues on next page)

(continued from previous page)

```
Building Pkg1.Test_Data.Tests/test_driver.gpr:
gprbuild  -PPkg1.Test_Data.Tests/test_driver.gpr  -o test_driver --src-subdirs=gnatcov-
↪instr --implicit-with=gnatcov_rts
...

Running Pkg1.Test_Data.Tests/test_driver.gpr:
GNATCOV_TRACE_FILE=Pkg1.Test_Data.Tests/test_driver-gnattest_td.srctrace Pkg1.Test_Data.
↪Tests/test_driver
pkg1.ads:2:4: info: corresponding test FAILED:  Test not implemented. (pkg1-test_data-
↪tests.adb:44)
pkg1.ads:3:4: error: corresponding test FAILED: Test not implemented. (pkg1-test_data-
↪tests.adb:65)
2 tests run: 0 passed; 2 failed; 0 crashed.

Creating checkpoint for Pkg1.Test_Data.Tests/test_driver.gpr:
gnatcov coverage --save-checkpoint=Pkg1.Test_Data.Tests/test_driver-gnattest.ckpt -PPkg1.
↪Test_Data.Tests/test_driver.gpr  --annotate=report --level=stmt  --cancel-annotate --
↪projects=Prj Pkg1.Test_Data.Tests/test_driver-gnattest_td.srctrace --units=@Pkg1.Test_
↪Data.Tests/units.list

Instrumenting project Pkg2.Test_Data.Tests/test_driver.gpr:
gnatcov instrument -PPkg2.Test_Data.Tests/test_driver.gpr --dump-trigger=main-end --
↪level=stmt   --projects=Prj --units=@Pkg2.Test_Data.Tests/units.list

Building Pkg2.Test_Data.Tests/test_driver.gpr:
gprbuild  -PPkg2.Test_Data.Tests/test_driver.gpr  -o test_driver --src-subdirs=gnatcov-
↪instr --implicit-with=gnatcov_rts
...

Running Pkg2.Test_Data.Tests/test_driver.gpr:
GNATCOV_TRACE_FILE=Pkg2.Test_Data.Tests/test_driver-gnattest_td.srctrace Pkg2.Test_Data.
↪Tests/test_driver
pkg2.ads:2:4: info: corresponding test PASSED
pkg2.ads:3:4: info: corresponding test PASSED
2 tests run: 2 passed; 0 failed; 0 crashed.

Creating checkpoint for Pkg2.Test_Data.Tests/test_driver.gpr:
gnatcov coverage --save-checkpoint=Pkg2.Test_Data.Tests/test_driver-gnattest.ckpt -PPkg2.
↪Test_Data.Tests/test_driver.gpr  --annotate=report --level=stmt  --cancel-annotate --
↪projects=Prj Pkg2.Test_Data.Tests/test_driver-gnattest_td.srctrace --units=@Pkg2.Test_
↪Data.Tests/units.list

Creating coverage report:
gnatcov coverage -P../prj.gpr -CPkg1.Test_Data.Tests/test_driver-gnattest.ckpt -CPkg2.
↪Test_Data.Tests/test_driver-gnattest.ckpt --annotate=report --level=stmt
** COVERAGE REPORT **

============================
== 1. ASSESSMENT CONTEXT ==
============================
```

(continues on next page)

(continued from previous page)

```
Date and time of execution: 2021-08-23 15:35:58 +02:00
Tool version: XCOV development-tree

Command line:
gnatcov coverage -P../prj.gpr -CPkg1.Test_Data.Tests/test_driver-gnattest.ckpt -CPkg2.
↪Test_Data.Tests/test_driver-gnattest.ckpt --annotate=report --level=stmt

Coverage level: stmt

Trace files:

Pkg1.Test_Data.Tests/test_driver-gnattest_td.srctrace
  kind     : source
  program  : Pkg1.Test_Data.Tests/test_driver
  date     : 2021-08-23 15:35:56 +02:00
  tag      :
  processed: gnatcov coverage --save-checkpoint=Pkg1.Test_Data.Tests/test_driver-
↪gnattest.ckpt -PPkg1.Test_Data.Tests/test_driver.gpr --annotate=report --level=stmt --
↪cancel-annotate --projects=Prj Pkg1.Test_Data.Tests/test_driver-gnattest_td.srctrace --
↪units=@Pkg1.Test_Data.Tests/units.list @ 2021-08-23 15:35:56 +02:00

Pkg2.Test_Data.Tests/test_driver-gnattest_td.srctrace
  kind     : source
  program  : Pkg2.Test_Data.Tests/test_driver
  date     : 2021-08-23 15:35:57 +02:00
  tag      :
  processed: gnatcov coverage --save-checkpoint=Pkg2.Test_Data.Tests/test_driver-
↪gnattest.ckpt -PPkg2.Test_Data.Tests/test_driver.gpr --annotate=report --level=stmt --
↪cancel-annotate --projects=Prj Pkg2.Test_Data.Tests/test_driver-gnattest_td.srctrace --
↪units=@Pkg2.Test_Data.Tests/units.list @ 2021-08-23 15:35:58 +02:00

============================
== 2. COVERAGE VIOLATIONS ==
============================

2.1. STMT COVERAGE
------------------

pkg1.adb:13:7: statement not executed
pkg1.adb:18:7: statement not executed

2 violation.

=========================
== 3. ANALYSIS SUMMARY ==
=========================

2 STMT violation.

** END OF REPORT **
```

The log shows all the steps necessary to obtain coverage results from the multiple test drivers, and end with the report.

From the coverage report, we see that the only lines not covered are in `pkg1.adb`, which is expected as the tests corresponding to that unit are not implemented. By using separate drivers, although the code in `Pkg2` uses the subprograms defined in `Pkg1`, we were able to not have the coverage results from the unit tests on `Pkg1` be polluted by the tests on `Pkg2`.

Regenerating the harness to use a single monolithic driver, and re-generating a coverage report shows that without the separate drivers, `Pkg1` is marked as covered despite not having any test implemented:

```
gnattest -P prj.gpr && make -C harness/ coverage
```

which outputs:

```
Instrumenting project test_driver.gpr:
gnatcov instrument -Ptest_driver.gpr --dump-trigger=main-end --level=stmt   --
↪projects=Prj

Building test_driver.gpr:
gprbuild  -Ptest_driver.gpr  -o test_driver --src-subdirs=gnatcov-instr --implicit-
↪with=gnatcov_rts
...
Running test_driver.gpr:
GNATCOV_TRACE_FILE=test_driver-gnattest_td.srctrace ./test_driver
pkg2.ads:2:4: info: corresponding test PASSED
pkg2.ads:3:4: info: corresponding test PASSED
pkg1.ads:2:4: error: corresponding test FAILED: Test not implemented. (pkg1-test_data-
↪tests.adb:44)
pkg1.ads:3:4: error: corresponding test FAILED: Test not implemented. (pkg1-test_data-
↪tests.adb:65)
4 tests run: 2 passed; 2 failed; 0 crashed.

Creating checkpoint for test_driver.gpr:
gnatcov coverage --save-checkpoint=test_driver-gnattest.ckpt -Ptest_driver.gpr  --
↪annotate=report --level=stmt  --cancel-annotate --projects=Prj test_driver-gnattest_td.
↪srctrace

Creating coverage report:
gnatcov coverage -P../prj.gpr -Ctest_driver-gnattest.ckpt --annotate=report --level=stmt
** COVERAGE REPORT **

============================
== 1. ASSESSMENT CONTEXT ==
============================

Date and time of execution: 2021-08-23 15:52:36 +02:00
Tool version: XCOV development-tree

Command line:
gnatcov coverage -P../prj.gpr -Ctest_driver-gnattest.ckpt --annotate=report --level=stmt

Coverage level: stmt

Trace files:

test_driver-gnattest_td.srctrace
```

(continues on next page)

(continued from previous page)

```
  kind     : source
  program  : ./test_driver
  date     : 2021-08-23 15:52:35 +02:00
  tag      :
  processed: gnatcov coverage --save-checkpoint=test_driver-gnattest.ckpt -Ptest_driver.
↪gpr --annotate=report --level=stmt --cancel-annotate --projects=Prj test_driver-
↪gnattest_td.srctrace @ 2021-08-23 15:52:35 +02:00

============================
== 2. COVERAGE VIOLATIONS ==
============================

2.1. STMT COVERAGE
------------------

No violation.

=========================
== 3. ANALYSIS SUMMARY ==
=========================

No STMT violation.

** END OF REPORT **
```

1.3 Instrumenting test harnesses for a SPARK project

General information about SPARK code instrumentation can be found in section *Instrumentation and coverage of SPARK code*. The compilation of instrumented user code needs to be controlled by a configuration pragma file. When using GNATtest, the main project is the generated test driver project, not the original user code project. As such, the configuration pragma files that need to be passed during compilation cannot be specified in the project under test.

There are two possibilities to specify the configuration pragma file to be used when building the instrumented harness projects. The first one is to modify the `gnattest_common.gpr` project file (which is not overwritten when the harness is regenerated), as in:

```
--- harness/gnattest_common.gpr
+++ harness/gnattest_common.gpr
  type TD_Compilation_Type is ("contract-checks","no-contract-checks", "no-config-file");
  TD_Compilation : TD_Compilation_Type := external ("TEST_DRIVER_BUILD_MODE", "no-config-
↪file");

  package Builder is
     case TD_Compilation is
        when "contract-checks" =>
           for Global_Configuration_Pragmas use "suppress.adc";
        when "no-contract-checks" =>
           for Global_Configuration_Pragmas use "suppress_no_ghost.adc";
        when "no-config-file" =>
-          null;
+          for Global_Configuration_Pragmas use "<path to file>/instrument-spark.adc";;
```

(continues on next page)

(continued from previous page)

```
    end case;
  end Builder;
```

An alternative solution is to specify the configuration pragma file on the command line when invoking the Makefile:

```
make BUILDERFLAGS='-cargs:Ada -gnatec=instrument-spark.adc' coverage
```

CHAPTER

TWO

DEVELOPING A CUSTOM SOLUTION

If the generated makefile is not suitable to use with the execution environment, there are a few things to keep in mind in order not to have unexpected coverage results.

2.1 Single test driver

In the case where a monolithic test driver is generated by GNATtest, obtaining coverage results for your project is relatively simple, and the only aspect which needs attention is the specification of units of interest, particularly in the case of using source traces.

When using source traces, GNATcoverage needs to instrument the main so that execution traces are dumped at the end of the test run. So despite none of the units in the harness project being of interest, is is important that the root project passed to all **gnatcov** commands is `test_driver.gpr`.

The projects generated by GNATtest all specify that none of the units are of interest, so none of the units generated by GNATtest should appear in the reports.

2.2 Separate test drivers

Using separate test drivers is advisable to avoid incidental coverage of one unit from the testing of other units (see *Handle incidental coverage effects*). Note that since the smallest division of a project supported by GNATcoverage is the unit, there is no benefit in specifying `--separate-drivers=test` instead of `--separate-drivers=unit` to GNATtest, as far as incidental coverage is concerned.

When using mutliple drivers, there will be a `test_driver.gpr` generated for each unit. For each generated driver, the project needs to be instrumented (if source traces are used), built, run and a *checkpoint* must be created from the execution trace. Then a call to **gnatcov coverage** merges the coverage data from all the checkpoints and generates the desired report.

The key point in the process is to specify, when creating all the individual checkpoints, which unit is being tested, so that the checkpoint only records the coverage information about that unit, and discard any incidental coverage on other units. During harness generation, a file named `units.list` will be created in the same directory as each `test_driver.gpr` file. This file contains the name of the unit tested by the driver, and can be used to specify to **gnatcov** to only process the unit under test, by adding the switch `--units=@units.list`.

Part V

Using GNATtest with GNATfuzz (experimental)

This section presents how a pre-existing GNATtest test harness can be used as a starting corpus for a GNATfuzz fuzzing campaign, and how inputs of interest found by GNATfuzz can be imported back into a GNATtest harness. These features are still experimental; the workflow and command line interface may change in the future.

CHAPTER

ONE

SETTING UP THE ENVIRONMENT

To ensure the entire workflow functions properly, it's crucial to configure the various tools by setting specific environment variables.

The first step is to setup the value generation runtime library. For detailed instructions on how to do so, see *Setting up the test generation runtime*.

CHAPTER

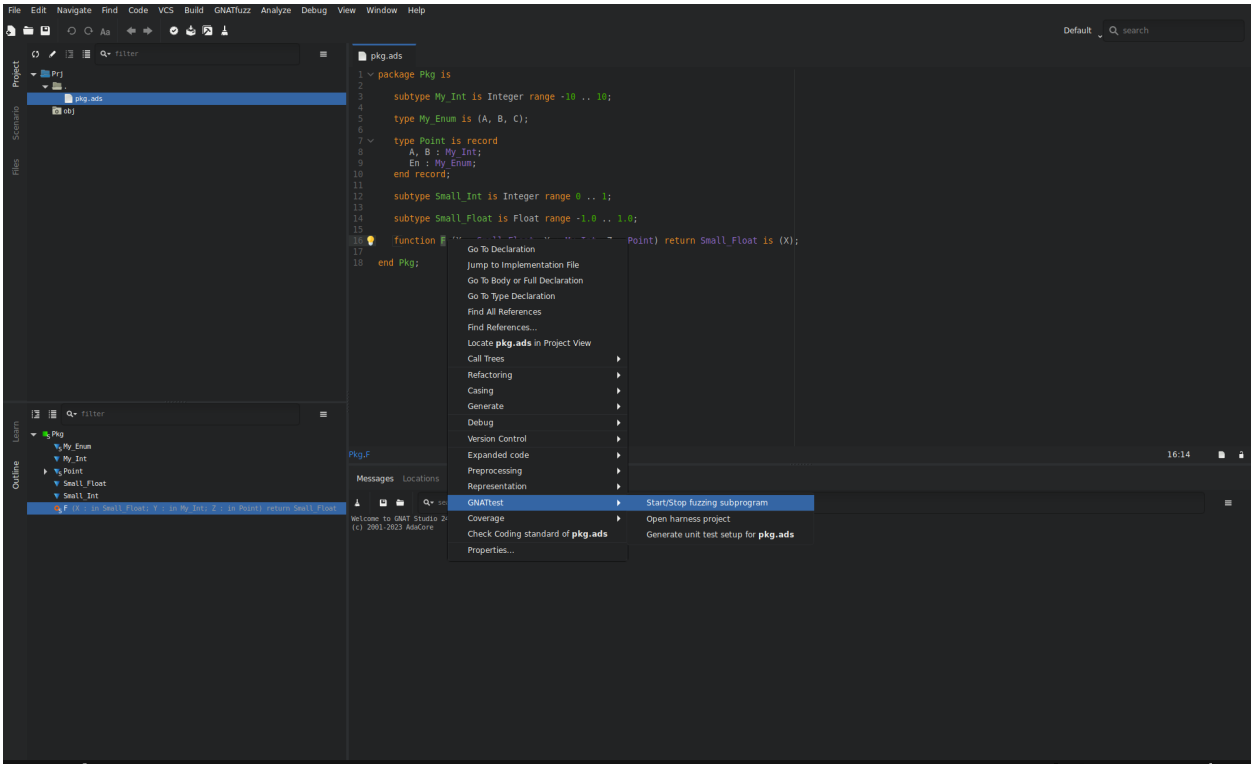
TWO

USING GNATSTUDIO TO PERFORM THE INTEGRATION

The simplest way to sequence the invocations of the two tools is to use the GNATstudio integration plugin.

First, create a GNATtest harness project if it doesn't already exist, using the `Analyze -> GNATtest -> Generate harness project` entries in the menu bar. Note that in the dialog box opening there is an option to generate tests inputs if needed.

Then, exporting tests inputs from GNATtest to GNATfuzz and running a fuzzing campaign on a specific subprogram can be done by right-clicking on the declaration of the target subprogram, then selecting `GNATtest -> Start/Stop fuzzing subprogram`, as illustrated bellow.



This will first instrument sources and re-generate the GNATtest harness in order to be able to intercept the inputs passed to the subprogram, then run the test harness, dumping the inputs in a binary format that can be used by GNATfuzz. GNATstudio will then setup a fuzzing session on the subprogram, for which the parameters can be controlled through the various pop-up windows that will be displayed during the process.

`gnatfuzz` will periodically export newly found inputs in a human readable JSON format under `<obj>/gnattest/test/JSON_Tests`, where `<obj>` designates the object directory of the project.

The fuzzing session will stop once all the stopping criteria have been met. The fuzzing session can also be stopped early by right clicking on the subprogram declaration, then selecting `GNATtest => Start/Stop fuzzing subprogram`.

After the fuzzing sessions has ended, a new GNATtest harness will be regenerated, including the tests exported by the GNATfuzz session. These will appear in `<obj>/gnattest/tests/<unit_name>-test_data-test_<subp_name>_<subp_hash>.adb`, where `<unit_name>` is the name of the unit in which the subprogram is declared, `<subp_name>` is the name of the subprogram, and `<subp_hash>` is a hash based on the profile of the subprogram, in order to differentiate overloads.

Part VI

Appendix A. Using GNATcoverage Binary Traces

CHAPTER

ONE

INTRODUCTION

1.1 Definition and main characteristics

Unlike *source traces* produced by the execution of instrumented programs in their regular environment, *binary traces* are coverage trace files produced by the execution of uninstrumented programs under the control of an instrumented execution environment. *"Binary"* in this designation refers to the nature of the collected data, which consists in low level information about executed blocks of machine instructions.

From binary traces, GNATcoverage is able to compute both *source coverage* metrics as with source traces, and *object coverage* metrics assessing the coverage of machine level instructions. GNATcoverage supports two criteria of the latter kind:

- *Instruction Coverage*, where we evaluate for each machine instruction whether it has been executed at least once or not; and
- *Branch Coverage*, where we also evaluate for each conditional branch instruction whether it was only taken, or went fall-through, or both.

Source coverage metrics are only available for a subset of the target configurations supported with source traces, though, and new tool functionalities are most often implemented for source traces only.

The matrix below summarizes the main characteristics of the two trace production modes:

	With Source Traces	**With Binary Traces**
Tracing mechanism	Program instrumentation	Execution environment
~	~	~
Native applications	Yes (including shared-libraries)	No
Cross configurations with RTOS	Yes	No
Bare metal configurations	Yes	Specific CPUs only, through GNATemulator or hardware probes
~	~	~
Object coverage analysis	No	Yes
New features	Often not	Yes (source level metrics for C++, scoped metrics in html)

1.2 Workflow overview

A high level view comparing the processes involved in the use of binary or source traces is provided on Fig. 1. The use of coverage *traces* in both modes allows lots of similarities between workflows, in particular in the steps processing traces after they have been produced.

We don't support mixing the two kinds of traces together, though, and have depicted a separate analysis/consolidation step to emphasize this:

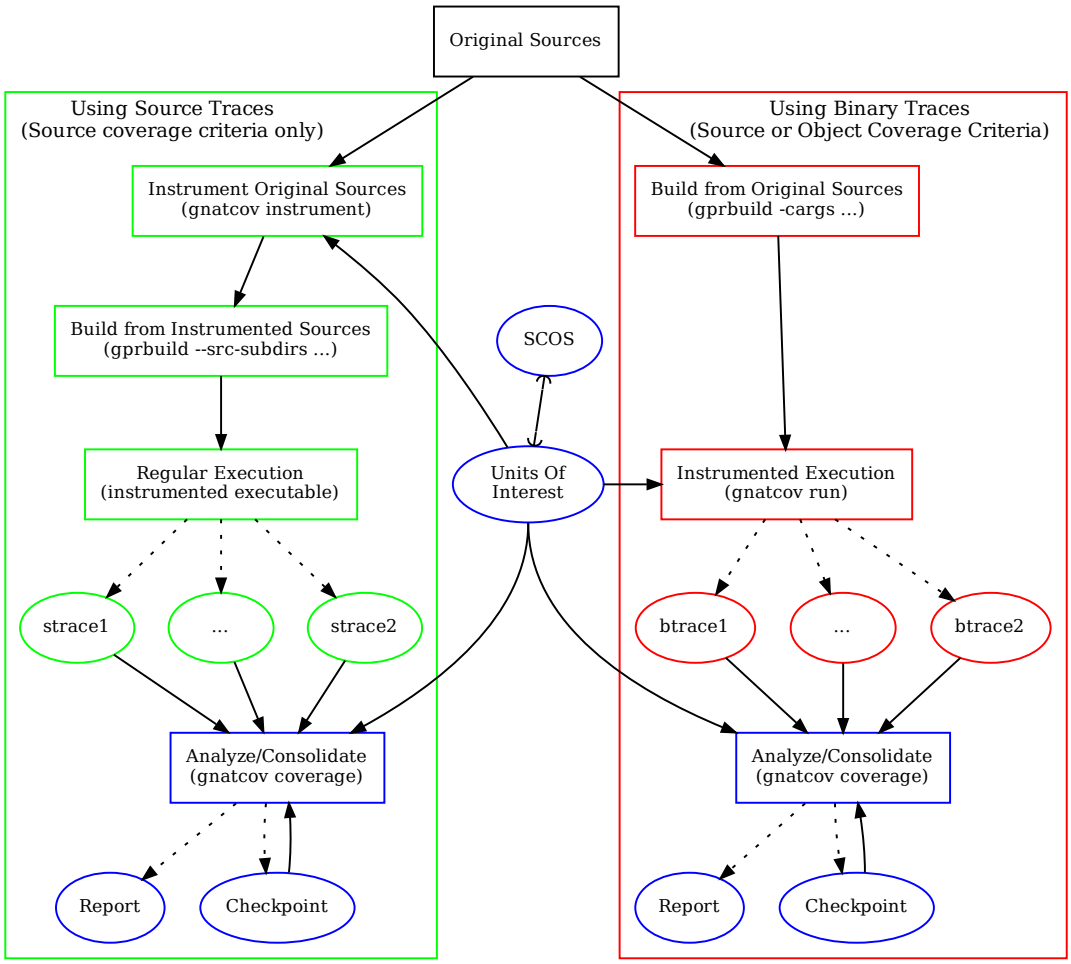


Fig. 1: Workflows using source or binary coverage traces

For binary traces, **gnatcov** relies on an instrumented execution environment to produce the traces instead of having to instrument the program itself with extra code and data structures. GNATemulator provides such an environment for bare metal configurations, as well as hardware probes assisted with means to convert their output to the format **gnatcov** expects.

Programs are built from their original sources, only requiring the use of `-g -fpreserve-control-flow -fdump-scos` compilation options to generate coverage obligation lists and let us associate execution traces to these

obligations afterwards.

1.3 A simple example

For the original "Getting Started" use case, we first use the GNAT Pro toolset for `powerpc-elf` to build, using **`gprbuild`** as follows:

```
gprbuild --target=powerpc-elf --RTS=light-mpc8641 -Ptests.gpr \
  -cargs -g -fpreserve-control-flow -fdump-scos
```

We pass the project file with `-P`, the required compilation flags with `-cargs` and request the use of a `light` runtime library tailored for the `mpc8641` board.

The build command produces a `test_inc` executable in the object subdirectory. To automate the execution of this program within GNATemulator to produce a trace, we provide the **`gnatcov run`** command. For the use case at hand, we would simply do:

```
gnatcov run --target=powerpc-elf obj-tests/test_inc
```

This produces a `test_inc.trace` binary trace file in the current directory. By default, such a trace is amenable to statement and decision coverage at most. If MCDC analysis is needed, `--level=stmt+mcdc` must be passed to **`gnatcov run`** as well and we recommend also providing source coverage obligations in this case.

Producing report is then achieved as for sources traces, using the binary trace files as inputs to the **`gnatcov coverage`** commands.

Units of interest can be conveyed with either the GPR oriented facilities as with source traces, or with `--scos` switches to provide lists of files holding coverage obligations in a similar fashion as `--sid` switches for `.sid` files out of source instrumentation.

The files to list with `--scos` are the *Library Information* files produced by the compiler along with object files, which are `.ali` files for Ada and `.gli` files for C.

The GNAT toolchain provides a helpful device in this process for Ada units: the `-A` command line argument to **`gnatbind`** which produces a list of all the `.ali` files involved in an executable construction. By default, the list goes to standard output. It may be directed to a file on request with `-A=<list-filename>`, and users may of course filter this list as they see fit depending on their analysis purposes.

Below is a complete example sequence of commands to illustrate, using the standard Unix `grep` tool to filter out test harness units:

```
# Build executable and produce the corresponding list of ALI files. Pass
# -A to gnatbind through gprbuild -bargs then filter out the test units:
gprbuild -p --target=powerpc-elf --RTS=light-mpc8641 -Ptests.gpr
  -cargs -fdump-scos -g -fpreserve-control-flow -bargs -A=all.alis

# Run and analyze all units except the test harness, filtering out
# the correspond ALI files from the list:
grep -v 'test_[^/]*.ali' all.alis > nontest.alis
gnatcov run --level=stmt+mcdc --scos=@nontest.alis
gnatcov coverage --level=stmt+mcdc --annotate=xcov --scos=@nontest.alis
```

1.4 Going Further

As the source and binary trace based workflows share commonalities, a lot of information from the main documentation also applies to the use binary traces, such as

- The use of project files to specify command switches or designate units of interest at analysis time,
- Using coverage checkpoints or traces for consolidation,
- Defining exemptions regions with pragmas for Ada. The support for other languages is not available for binary traces.

Then this appendix holds separate chapters dedicated to binary trace specific considerations:

- *Producing binary traces with gnatcov run*
- *Object coverage analysis with gnatcov coverage*
- *Object/Source level metrics considerations*
- *Importing binary traces with gnatcov convert*

CHAPTER

TWO

PRODUCING BINARY TRACES WITH `GNATCOV RUN`

The simplest way to produce binary traces for GNATcoverage is to run the program in an environment that knows how to produce such traces directly. For bare metal environments operating within GNATemulator, the **`gnatcov run`** command is available for this purpose.

2.1 Compilation prerequisites

2.1.1 General rules

If the traces are intended to be used for *source coverage analysis*, regardless of the actual criteria to be assessed afterwards, sources must be compiled with `-g -fpreserve-control-flow -fdump-scos` to control optimizations and allow mapping trace info to source constructs. `-gno-strict-dwarf` is required in addition for VxWorks targets, together with `-mlongcall` for PowerPC configurations.

Optimization is supported up to `-O1`, with inlining allowed. However, if the binary version of the code under test during the coverage campaign is intended to be used directly in operational conditions, care is needed to prevent inlining of the code under test within the testing code at compilation time in unit-testing campaigns. With GCC based toolchains such as GNAT Pro, this can be achieved by adding `-fno-inline` to the compilation options of the testing code.

The `-fprofile-arcs` or `--coverage` compilation switches, intended for analysis with the gcov tool, are incompatible with the needs of GNATcoverage for binary traces and must not be used. Likewise for the `-gnatV` family of GNAT Pro switches requesting data validity checks other than those mandated by the Ada RM (special case of `-gnatVd`).

For object coverage assessments, only `-g` is of possible use, if you intend to produce annotated sources as coverage reports. If source coverage analysis is to be performed *as well*, and you have no particular constraint on the form of the machine code for which object coverage is needed, the whole process might turn simpler if the same compilation options are used for both kinds of analysis. There is no incompatibility with object coverage analysis incurred by compiling the code compiled with options required for source coverage.

2.1.2 Consolidation related rules

For object or source level criteria, **`gnatcov`** computes the coverage achieved for the full set of routines or source units declared to be of interest amongst those exposed by the union of the exercised executables, as designated by the set of consolidated traces;

For the purpose of computing combined coverage achievements, two symbols are considered overlapping when all the following conditions are met:

- Both symbols have identical names at the object level,
- Both symbols have DWARF debug information attached to them,

- According to this debug information, both symbols originate from the same compilation unit, denoted by the full path of the corresponding source file.

By this construction, a symbol missing debug information is never considered overlapping with any other symbol. Whatever coverage is achieved on such a symbol never gets combined with anything else and the only kind of report where the symbol coverage is exposed is the `=asm` assembly output for object level criteria.

Moreover, for object level coverage criteria, **gnatcov coverage** will issue a consolidation error when two symbols are found to overlap but have structurally different machine code, which happens for example when the same unit is compiled with different different optimization levels for different executables.

2.2 gnatcov run command line

gnatcov run offers a unified interface to launch programs for a specific target machine and produce execution traces. The general interface synopsis is available from `gnatcov --help`, as follows:

```
gnatcov run [OPTIONS] [EXE] [-eargs [EXE] EARGS...]
```

EXE :
Name of the program file to execute. The provided path is stored in the output trace header that **gnatcov coverage** fetches for analysis purposes later on. Relative paths there will be considered relative to the location where **gnatcov coverage** is launched. If the program name is not provided on the base command line (before eargs), the first earg or a project file attribute can be used for this purpose instead. See the *Program selection and Arguments passing* section of this chapter for more details on this process.

-eargs :
For cross configurations (with `--target`), pass the items not used as the program name to the machine simulator that eventually executes the program. For native configurations, pass the items not used as the program name as command line arguments to the executed program.

Then the available `[OPTIONS]` are:

--level :
Convey the most precise kind of analysis that is intended from the produced traces later on. This defaults to `stmt+decision` and must be set to `stmt+mcdc` if you intend to perform MCDC analysis. In this case, it is highly recommended, for efficiency purposes, to also state the units on which you will perform such analysis with `-P` or `--scos` options, as described in *Specifying Units Of Interest*.

-t, --target :
State the target toolchain configuration used to build the analyzed programs, as provided to `gprbuild`. For cross or 32bit native configurations, this switch together with its possible `--RTS` companion is required for all commands using project files unless the root project provides the information with `Target`/`Runtime` attributes.

This is used to control the underlying execution engine used to run the program, assumed to be the host environment by default. On the command line, a possible `,<board-name>` extension is allowed as well. See the *Execution environment* section of this chapter for additional details.

--kernel :
Tell the underlying emulator that the executable program actually is a module to be loaded on top of the provided kernel binary. This is typically for VxWorks kinds of targets, where the kernel is a tailored version built to include GNATemulator support.

-v, --verbose :
Request verbose output. In particular, this displays the commands launched to run the program within the underlying instrumented environment.

-o, --output :
Request the selection of an alternate output file name for the execution trace. Otherwise, **gnatcov run** writes the execution trace in the current directory, in a file named like the executable input with a `.trace` suffix. For example **gnatcov run** */path/to/myexecfile* produces a *myexecfile.trace* file in the current directory by default.

-T, –*tag* :
Store the provided string argument 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.

-P :
Designate a root project file for various possible purposes. This can first be used together with `--projects` as an alternative to `--scos` to collect the set of units of interest for later coverage assessments if mcdc computation is needed. See the *Example program and assessments* and *Specifying Units Of Interest* sections of this manual for more details on this. Another possible use is the specification in the root project file of attributes controlling the execution, such as the coverage level intended to be assessed later on or the name of the main subprogram unit, from which the name of the executable to run can be inferred. In such cases, project attributes provide default values, ignored when the corresponding item is specified on the command line.

2.3 Execution environment

For cross configurations, the `--target` command line option allows specifying the target environment for which the program was built and for which **gnatcov run** should pick a suitable execution environment. The option states a base target name possibly followed by a board specialization after a separating '`,`' (`--target=powerpc-eabispe,mpc5566` for example). When a simple target name alone is sufficient, it is possible to let **gnatcov** detect it from the `Target` attribute from the project file (see *Specifying the Target and language Runtime* section). Then:

- When GNATemulator for the base target is available on your PATH, as <*base-target*>*-gnatemu*, **gnatcov** uses this to run the program. GNATemulator acts as a wrapper around the real machine emulator in this case, taking care of low-level interfacing details. If an optional board extension is provided in the `--target` argument, the specified board name is passed as an extra `--board=<board-name>` command line option to GNATemulator.
- Otherwise, **gnatcov** resorts to a builtin low level emulator statically configured for the base target. An `unsupported target` error is issued and **gnatcov** exits if no such configuration is found to match.

2.4 Program selection and Arguments passing

To facilitate the coverage analysis integration within different kinds of environments, **gnatcov run** supports several ways of selecting the program to execute:

- If an *[EXE]* argument is provided on the base command line, before `eargs`, use it as the candidate program name to execute. Otherwise:
- If *eargs* are provided and the first item designates an executable file, use this as the program to execute. Otherwise:
- If a root project file is designated with `-P`, and this project features a single Main attribute, use the corresponding executable name as the program to execute.

The `eargs` switches not used as the program name are then used differently in cross or native configurations, as differentiated by the use of a `--target` option.

In cross configurations, `eargs` switches are passed straight to the low-level emulation engine. They are not interpreted by GNATemulator when it is used. Passing arguments to GNATemulator via **gnatcov run** can be achieved by way of a GNAT project file, with a Switches attribute in the Emulator package. For example:

```
-- covemu.gpr
project Covemu is
  package Emulator is
    for Switches use ("--gnatbus=bridge.bar.com:1800");
  end Emulator;
end Covemu;
```

Then:

```
gnatcov run --target=<target> -Pcovemu.gpr
```

In native configurations, the program executes in the host environment and the `-eargs` switches are passed as command line arguments to the executable program.

This way users can just prefix a regular host command line by "gnatcov run … -eargs" to produce execution traces, as illustrated by the native case in the set of valid **gnatcov** command line examples below:

```
gnatcov run --target=powerpc-elf myprog
# Run "myprog" using powerpc-elf-gnatemu as the execution environment.
# Produce myprog.trace in the current directory.

gnatcov run --target=powerpc-elf myprog -o myrun.trace
# Likewise, producing myrun.trace instead, still in the current directory

gnatcov run --target=powerpc-elf myprog -o myrun.trace -eargs --version
# Likewise, also requesting version output from the low level execution
# engine, *not* from gnatemulator if it happens to be involved.

gnatcov run --target=powerpc-elf -o myrun.trace -eargs myprog --version
# Likewise, providing the executable program to run as the first earg

gnatcov run --target=powerpc-elf -o myrun.trace -Pmyprj.gpr -eargs --version
# Likewise, inferring the executable program name from the Main attribute
# in myprj.gpr.

gnatcov run --target=powerpc-elf myprog -T "trace for doc example"
# Providing a trace tag, that can de retrieved with trace dump facilities
# and which is displayed in some output reports.

gnatcov run --target=powerpc-eabispe myprog
# Run "myprog" using powerpc-eabispe-gnatemu as the execution environment.
# Produce myprog.trace in the current directory.

gnatcov run --target=powerpc-eabispe,mpc5566 myprog
# Likewise, instructing gnatemu to select the "mpc5566" board emulation.

gnatcov run -eargs myprog arg1 arg2
# Where supported, run "myprog" in the native environment through an
# instrumentation layer to produce the execution trace. Pass arg1 and arg2
# as command line arguments to "myprog".
```

2.5 Trace file contents

A trace file essentially consists in

- A header with general information about the trace generation context (name of the binary executable passed to **gnatcov run**, `--tag` argument value, production date & time, ...), followed by
- The machine execution trace entries (roughly, one per execution basic block, with information on the branch decision at the end)

gnatcov offers a `dump-trace` option to display the contents of trace files passed as arguments, displaying tags passed to **gnatcov run** amongst other things. For example:

```
gnatcov dump-trace test_divmod2.trace

Tag  : DATE_TIME (Date)
Len  : 8
Data : dc 07 02 15 08 00 25 00
       2012-02-21 08:00:37

Tag  : EXEC_FILE_NAME
Len  : 16
Data : obj/test_divmod2

Tag  : USER_DATA (User_Tag)
Len  : 10
Data : sample tag

Traces:
fffffffc-fffffffb ?: 20 ---- fault
fffffffc-ffffffff ?: 11 ---t block
fff0067c-fff006b3 ?: 11 ---t block
fff006bc-fff006bf ?: 12 --t- block
[...]
```

Prior to the execution traces per-se (list of executed instruction blocks past the `Traces:` label), we see a few information entries aimed at helping GNATcoverage and users on various accounts. Each entry has a tag identifying it's kind, then some associated data dependent on the kind. Our example above features the following information entries:

DATE_TIME :
The trace production time stamp, always 8 bytes long.

EXEC_FILE_NAME :
Path to the binary program that was executed to produce the trace. This path can be exactly the one that was passed to **gnatcov run** or a derived path, for instance if **gnatcov run** had to add an extension to find the actual file (see *Target specific points of note with Binary traces*). **gnatcov coverage** uses this path to find the program file again at analysis time, to find which machine code corresponds to which address for example.

USER_DATA :
User string tag for this trace, when one was passed with `-T` to **gnatcov run**.

The precise structure is described in the `qemu_traces.ads` unit of the gnatcov sources.

CHAPTER

THREE

OBJECT COVERAGE ANALYSIS WITH `GNATCOV COVERAGE`

Object coverage analysis computes metrics focused on machine-level object code, concerned with machine basic instructions or conditional branches. Conditional general instructions (i.e.non-branch instructions), such as ARM conditional instructions are not supported. If such instructions are present in the subprograms to be analyzed, their reported coverage status may be incorrect.

On request, the metrics can be presented on sources, with an annotation on each line synthesizing the coverage status of all the instructions generated for this line. This mapping relies on debug information, so sources must be compiled with `-g` for this to work. There is no further compilation requirement for object coverage alone.

Once your application is built, the analysis proceeds in two steps: **`gnatcov run`** is used to produce execution traces, then **`gnatcov coverage`** to generate coverage reports. *Object* coverage is queried by passing a specific `--level` argument to **`gnatcov coverage`**; `=insn` or `=branch`, described in detail in the following sections. As for source coverage, there is never a requirement to recompile just because a different criterion needs to be analyzed.

The *Producing binary traces with gnatcov run* chapter of this document provides details on the trace production interface. The remainder of this chapter explains the use of **`gnatcov coverage`** in particular, to analyze traces once they have been produced. The general command line structure is always like:

```
gnatcov coverage --level=<criterion> --annotate=<format>
                 [--routines=<names>] ... <traces>
```

The optional `--routines` argument provides the set of object level subprogram names on which the analysis should focus. This set defaults to the full set of symbols defined by all the executables associated with the provided execution traces.

Later in this chapter, *Focusing on subprograms of interest (--routines)* explains how to construct the relevant list of names for `--routines`. Prior to this comes a section describing the *available report formats*, then more details regarding *Object Instruction Coverage analysis (--level=insn)*, *Object Branch Coverage analysis (--level=branch)*, and specificities regarding *Inlining & Ada Generic units*. Finally, *Full object coverage considerations* describes tools that help analyze low-level object files for issues of interest when aiming at full object coverage.

3.1 `gnatcov coverage` command line

Coverage analysis with GNATcoverage is performed by invoking **`gnatcov coverage`** for a set of criteria queried via the `--level` command line option. The general interface synopsis is available from **`gnatcov`** `--help`:

```
gnatcov coverage OPTIONS TRACE_FILES
```

The available options are as follows:

-c, --level *(mandatory)*:
Tell the set of coverage criteria to be assessed. The possible values for object coverage analysis are `insn`, `branch`, both explained later in this chapter.

-a, --annotate *(mandatory)*:
Request a specific output report format. The two possible criteria support `xcov[+]`, `html`, and `asm`, with interpretations that vary depending on the assessed criteria. See the corresponding documentation later in this chapter for more details. This option accepts comma separated values and/or can be specified multiple times on the command line, in which case there will be one report produced for each of the requested annotation formats.

--timezone :
Select the timezone to use when displaying trace creation dates in reports, which could be either `local` or `utc`. The local timezone is used by default.

--output-dir :
Request that the report files (index and annotated sources for the `xcov` and `html` output formats) be output in the provided directory. If not specified, the default is the root project's object directory if using projects, and the current directory if not. If more than one of the above annotation formats is requested, then each report will be placed in a subdirectory named accordingly.

--report-title :
Request that generated HTML documents (index for the `html` output format) are assigned a customized title. For instance, passing `--report-title="Project ABC"` will yield titles such as: `Project ABC - GNATcoverage Report`. If passed multiple times, passing an empty string last will restore the default behavior. This option is ignored is the selected output format does not support titles.

-T, --trace *(mandatory)*, possibly repeated and accepting *@listfile arguments* :
Provide the set of execution traces for which a report is to be produced. When multiple traces are provided, **gnatcov** produces a consolidated result, as described in detail in the *Coverage Consolidation* chapter of this manual.

--exec:
Override executable from traces. Trace files contain an indication of the executable used to generate them. This option causes the named executable to be loaded for coverage analysis, and to override the indication contained in any trace specified after it on the command line. An empty executable name may be specified to restore the default behavior of using the indication contained in each trace file. Note that `--exec` may appear last on the command line, in which case it applies to no trace file, but still causes the indicated executable to be included in the coverage analysis. This ensures that any code in that executable that is not exercised by some trace file will be reported as not covered.

--warnings-as-errors:
Treat warnings as errors, i.e. exit with a non-zero status code if a warning is emitted.

--routines, possibly repeated and accepting *@listfile arguments*:
Provide the list of object symbol names that correspond to routines for which the coverage assessment is to be performed. Each instance of this option on the command line adds to what is to be assessed eventually. See the *Focusing on subprograms of interest (--routines)* section for extra details and use examples.

--alis, possibly repeated and accepting *@listfile arguments*:
Provide set of *Library Information files* for units where there might be applicable exemption regions to account for, as explained in the *Object coverage exemptions* section of this manual.

-P:
Use the indicated project file to find default options. This can also be used, possibly combined with other project related switches, as an alternative to `--alis` to determine the set of units for which coverage exemptions should be honored. In this case, the target/runtime/scenario contextual options that would be passed to build the project should also be passed for proper interpretation of the project files. This is similar to the use of project facilities for the determination of SCOs for source coverage analysis, and described in the *Specifying Units Of Interest* chapter of this manual.

-t, --target :
State the target toolchain configuration used to build the analyzed programs, as provided to `gprbuild`. For cross or 32bit native configurations, this switch together with its possible `--RTS` companion is required for all commands using project files unless the root project provides the information with `Target`/`Runtime` attributes.

3.2 Output report formats (`--annotate`)

Object coverage reports may be produced in various formats, as requested with the `--annotate` command line argument of **`gnatcov coverage`**.

The `asm` format produces an annotated assembly output, with a coverage indication attached to every single instruction. This is the base information of interest to object coverage analysis, simply presented in different manners through the other possible output formats. The `xcov` and `html` formats produce a set of annotated source files, in the directory where **`gnatcov`** is launched unless overridden with a `--output-dir` option. Even though presented on sources, the annotations remain representative of object coverage metrics, synthesized for all the instructions associated with each source line.

Later in this chapter we name output formats by the text to add to `--annotate` on the command line. For example, we use "the `=asm` outputs" to mean "the coverage reports produced with `--annotate=asm`". We also sometimes use *in-source* reports or outputs to designate the set of outputs in annotated source forms.

We illustrate the various formats with coverage analysis excerpts on the following example Ada support unit:

```
--  raise Program_Error if T is False. Do nothing otherwise.

procedure Assert (T : Boolean) is
begin
   if not T then
      raise Program_Error;
   end if;
end Assert;
```

As the contents suggest, this subprogram is expected never to be called with T False in nominal situations.

3.2.1 Machine level reports (`=asm`)

For object coverage analysis, `--annotate=asm` produces annotated assembly code for all the selected routines on standard output. The annotations are first visible as a special character on each machine code line to convey the coverage status of the corresponding instruction. The following output excerpt, for example, is part of a coverage report for our `Assert` subprogram compiled for the PowerPc architecture:

```
Coverage level: branch
_ada_assert !: 0c4-123
0c4 +:  94 21 ff e0      stwu   r1,-0x0020(r1)
...
0ec +:  2f 80 00 00      cmpiw  cr7,r0,0x0000
0f0 +:  41 9e 00 18      beq-   cr7,0x108 <_ada_assert+00000044>
...
100 -:  38 80 00 04      li     r4,0x0004
104 -:  48 00 00 a1      bl     0x1a4 <__gnat_last_chance_handler>
108 +:  60 00 00 00      ori    r0,r0,0x0000
...
120 +:  4e 80 00 20      blr
```

A - annotation for a line always conveys that the instruction was not executed at all. The instruction is also said to be *uncovered* in this case. Conversely, a + means that the instruction is *fully covered* with respect to the analyzed criterion, with a meaning which depends on both the criterion and the kind of instruction – whether the instruction is a conditional branch and whether we are doing mere instruction or object branch coverage analysis. Annotations conveying *partial coverage* might show up as well, also depending on the criterion and kind of instruction.

More details on the instruction specific annotations are provided in the sections that follow. Then, as the first line of the example suggests, the report also annotates each subprogram symbol as a whole, with the range of addresses that the subprogram spans and a synthetic coverage indication according to the following table:

Symbol Annotation	Meaning
-	All the subprogram instructions are uncovered (none executed)
+	All the subprogram instructions are fully covered
!	Some of the subprogram instructions were fully or partially covered

In our example, the code features both fully covered and uncovered instructions, and the `_assert` symbol as a whole is marked partially covered with a `!` annotation.

3.2.2 Annotated sources, text (`=xcov[+]`)

For object coverage analysis, `--annotate=xcov` produces annotated source files with the `.xcov` extension, one per original compilation unit in the selected output directory.

The annotations are visible at the beginning of every source line, as a single character which synthesizes the coverage status of all the machine instructions generated for this line. The following table provides a uniform description of this synthesis for all the object level criteria:

Source Annotation	Meaning
.	no machine code associated with this line
-	all the instructions associated with the line are - (uncovered)
+	all the instructions associated with the line are + (fully covered)
!	otherwise

The report also includes a short header, which features a global coverage count with respect to the total number of lines with associated code, as well as an indication of the assessed criterion. Below is an example of report obtained for our Assert unit:

```
examples/src/assert.adb:
75% of 4 lines covered

Coverage level: insn
  1 +: procedure Assert (T : Boolean) is
  2 .: begin
  3 +:    if not T then
  4 -:       raise Program_Error;
  5 .:    end if;
  6 +: end Assert;
```

To *lines* with associated object code we apply qualifiers similar to those for individual instructions: when the synthetic coverage indication for a line is `-`, `+` or `!`, we qualify the line as *uncovered*, *fully covered*, or *partially covered*, respectively. Note that even though they are rendered on source lines, the annotations are really meant to convey object code

properties here, hence are of a different nature than what the DO-178B source structural coverage criteria refer to. See our *Object/Source level metrics considerations* section for further details on this aspect.

With `--annotate=xcov+` (extra + at the end), the machine instructions and their individual coverage status are printed next to their associated source line.

3.2.3 Annotated sources, html (`=html`)

For object coverage criteria, **`gnatcov coverage`** `--annotate=html` produces an annotated version of each source file. Each annotated source page contains a summary of the assessment results followed by the original source lines, all numbered and marked with a coverage annotation as in the `--annotate=xcov` case. In addition, lines with obligations are colorized in green, orange or red for `+`, `!` or `-` coverage respectively. An *index.html* page is also produced, which contains a description of the assessment context (assessed criteria, set of trace files involved, ...) and a summary of the coverage results for all the units, with links to their annotated sources.

Similarly to the `xcov` format case, `--annotate=html` attaches to each line details about the coverage status of all the individual instructions generated for the line. These are folded within the line and expanded when a mouse click hits it.

3.3 Object Instruction Coverage analysis (`--level=insn`)

Object *Instruction* Coverage treats basic and conditional branch instructions identically, as either executed or not, hence fully covered or uncovered. The `=asm` instruction annotations are as follows:

Insn Annotation	Meaning
-	the instruction was not executed
+	the instruction was executed

The `=asm` excerpt below provides a representative example of the PowerPC instruction coverage achieved for our `Assert` procedure by nominal executions where the subprogram is never called with T False:

```
_ada_assert !: 0c4-123
0c4 +:  94 21 ff e0      stwu   r1,-0x0020(r1)
...
0ec +:  2f 80 00 00      cmpiw  cr7,r0,0x0000
0f0 +:  41 9e 00 18      beq-   cr7,0x108 <_ada_assert+00000044>
...
100 -:  38 80 00 04      li     r4,0x0004
104 -:  48 00 00 a1      bl     0x1a4 <__gnat_last_chance_handler>
108 +:  60 00 00 00      ori    r0,r0,0x0000
...
120 +:  4e 80 00 20      blr
```

Expectedly, the coverage annotations report all the instructions as executed except the two issuing the call to `__gnat_last_chance_handler`, which correspond to the `raise` statement in the GNAT high integrity profiles without exception propagation support. The two instructions at offsets 0ec and 0f0 are the comparison and branch conditioned on the comparison result that implement the *if* construct. We note here that the conditional branch is reported fully covered, as merely executed, even though always taken.

The corresponding `=xcov` output follows:

```
1 +: procedure Assert (T : Boolean) is
2 .: begin
3 +:    if not T then
4 -:       raise Program_Error;
5 .:    end if;
6 +: end Assert;
```

The annotations on lines 3 and 4 correspond to immediate expectations from comments we made on the `=asm` output. We can also observe annotations on lines 1 and 6, to which the subprogram prologue and epilogue code is attached, and executed as soon as the procedure is called.

3.4 Object Branch Coverage analysis (`--level=branch`)

Object *Branch* Coverage treats basic and conditional branch instructions differently. Basic instructions are considered fully covered as soon as executed, as in the Instruction Coverage case. Conditional branches, however, have to be executed at least twice to be claimed fully covered : once taking the branch and once executing fall-through, which we sometimes abusively refer to as *taken both ways* even if one case actually corresponds to the branch not being taken. The `=asm` instruction annotations are as follows:

Insn Annotation	Meaning
-	the instruction was never executed
+	the instruction was executed and taken both ways for a conditional branch
>	the instruction is a conditional branch, executed and always taken
v	the instruction is a conditional branch, executed and never taken

The `v` and `>` annotations are representative of situations where a conditional branch instruction is executed and *taken one way* only, which constitutes *partial coverage* of the instruction.

An example of partial coverage is observable on our Assert case, where the conditional branch at offset 0f0 is always taken, jumping over the raise statement code as expected for nominal executions:

```
_ada_assert !: 0c4-123
0c4 +:  94 21 ff e0      stwu   r1,-0x0020(r1)
...
0ec +:  2f 80 00 00      cmpiw  cr7,r0,0x0000
0f0 >:  41 9e 00 18      beq-   cr7,0x108 <_ada_assert+00000044>
...
100 -:  38 80 00 04      li     r4,0x0004
104 -:  48 00 00 a1      bl     0x1a4 <__gnat_last_chance_handler>
108 +:  60 00 00 00      ori    r0,r0,0x0000
...
120 +:  4e 80 00 20      blr
```

The corresponding `=xcov` output follows:

```
examples/src/assert.adb:
50% of 4 lines covered
Coverage level: branch
  1 +: procedure Assert (T : Boolean) is
  2 .: begin
```

(continues on next page)

(continued from previous page)

```
  3 !:    if not T then
  4 -:       raise Program_Error;
  5 .:    end if;
  6 +: end Assert;
```

The partial branch coverage logically translates into a partial coverage annotation on the line to which the branch is attached, here the line of the *if* statement that the conditional branch implements. This is confirmed by the `=xcov+` output, where the individual instructions are visible as well together with their own coverage indications:

```
examples/src/assert.adb:
Coverage level: branch
   1 +: procedure Assert (T : Boolean) is
<_ada_assert+00000000>:+
0c4 +:  94 21 ff e0  stwu   r1,-0x0020(r1)
...
0dc +:  98 1f 00 08  stb    r0,0x0008(r31)
   2 .: begin
   3 !:    if not T then
<_ada_assert+0000001c>:!
0e0 +:  88 1f 00 08  lbz    r0,0x0008(r31)
...
0ec +:  2f 80 00 00  cmpiw  cr7,r0,0x0000
0f0 >:  41 9e 00 18  beq-   cr7,0x108 <_ada_assert+00000044>
   4 -:       raise Program_Error;
<_ada_assert+00000030>:-
0f4 -:  3c 00 ff f0  lis    r0,-0x0010
...
104 -:  48 00 00 a1  bl     0x1a4 <__gnat_last_chance_handler>
   5 .:    end if;
   6 +: end Assert;
<_ada_assert+00000044>:+
...
120 +:  4e 80 00 20  blr
```

3.5 Focusing on subprograms of interest (`--routines`)

By default, in absence of a `--routines` argument to **`gnatcov coverage`**, object coverage results are produced for the full set of subprogram symbols defined by the executables designated by the analyzed traces.

`--routines` allows the specification of a set of subprogram symbols of interest so reports refer to this (sub)set only. Each occurrence of `--routines` on the command line expects a single argument which specifies a subset of symbols of interest. Multiple occurrences are allowed and the subsets accumulate. The argument might be either a single symbol name or a *@listfile argument* expected to contain a list of symbol names.

For example, focusing on three symbols `sym1`, `sym2` and `sym3` can be achieved with either one of the following set of `--routines` combinations:

```
--routines=sym1 --routines=sym2 --routines=sym3
or --routines=@symlist123
or --routines=sym3 --routines=@symlist12
```

... provided a `symlist12` text file containing the first two symbol names and a `symlist123` text file containing the three of them.

It is often convenient to compute the lists of symbols for a *@listfile argument*, for example as "the full set of defined subprograms except those with `test_` or `harness_` at the beginning of their name". **gnatcov** provides the **gnatcov disp-routines** sub-command for this purpose.

The general synopsis of **gnatcov disp-routines** is as follows:

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

gnatcov 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 here, as *conforming to a supported object file format, typically 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, until contradicted, symbols defined in object files are added to the result set", and `--exclude` states "from now on, until contradicted, symbols defined in object files are removed from the result set". An implicit `--include` is assumed right at the beginning, and each argument may be either the direct name of an object file or a *@listfile argument* containing a list of such names.

Below are a few examples of commands together with a description of the set they build:

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

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

$ gnatcov 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)
```

Annotated source reports, when requested, are generated for sources associated with the selected symbols' object code via debug information, and coverage annotations are produced only on the corresponding. Inlining can have surprising effects in this context, as the following section describes in greater details.

3.6 Inlining & Ada Generic units

The generated code for an inlined subprogram call or a generic instantiation materializes two distinct source entities: the expanded source (of the inlined subprogram or of the instantiated generic body) and the expansion request (the subprogram call or the generic instantiation). While this is of no consequence for `=asm` outputs, which just report coverage of raw machine instructions within their object level subprograms, regardless of the object code origin, this raises a few points of note for in-source outputs.

For inlined calls, potentially surprising results might show up when a specific set of object routines is queried. Indeed, when the code for a symbol A in unit Ua embeds code inlined from unit Ub, a request for an annotated source report for routine A, intuitively expected to yield a report for Ua only, will typically produce an output file for Ub as well, for lines referenced by the machine code inlined in A.

Consider the following Ada units for example, with a functional unit in `intops.ads` and `intops.adb`, then a test driver in `test_inc0.adb`:

```
package Intops is                          -- intops.ads
   procedure Inc (X : in out Integer);
   pragma Inline (Inc);
end Intops;

package body Intops is                     -- intops.adb
   procedure Inc (X : in out Integer) is
   begin
      X := X + 1;
   end Inc;
end Intops;

procedure Test_Inc0  is                    -- test_inc0.adb
   X : Integer := 0;
begin
   Inc (X);
end Test_Inc0;
```

The following analysis:

```
gnatcov coverage --level=insn --routines=_test_inc0 --annotate=xcov+ test_inc0.trace
```

... requests, with `--routines`, to report about the Test_Inc0 procedure only, so we intuitively expect a single `test_inc0.adb.xcov` annotated source result. If the `Inc(X)` call in Test_Inc0 is inlined, however, the command actually produces an `intops.adb.xcov` report as well because the object code of Test_Inc0 also contains inlined code coming from the other unit.

For generic units, information for all the instances is aggregated on the generic source, so each line annotation is a super synthesis of the coverage achieved for all the instructions attached to this line through all the instances.

Let us consider the generic Ada unit below to illustrate:

```
generic
   type Num_T is range <>;
package Genpos is
   procedure Count (X : Num_T);
   --  Increment N_Positive is X > 0

   N_Positive : Natural := 0;
   --  Number of positive values passed to Count
end Genpos;

package body Genpos is
   procedure Count (X : Num_T) is
   begin
      if X > 0 then
         N_Positive := N_Positive + 1;
      end if;
   end Count;
end Genpos;
```

Then two distinct instances in their own package, producing separate object code for each instance:

```
package POSI is
   type T1 is new Integer;
   package Pos_T1 is new Genpos (Num_T => T1);

   type T2 is new Integer;
   package Pos_T2 is new Genpos (Num_T => T2);
end POSI;
```

And now a simple test driver that executes all the code for `Count` in the first instance (going within the *if* statement), and only part of the code for `Count` in the second instance (not going within the *if* statement):

```
procedure Test_Genpos is
begin
   Pos_T1.Count (X => 1);
   Pos_T2.Count (X => -1);
end Test_Genpos;
```

The precise `insn` coverage difference is first visible in the `=asm` report. The conditioned part of `Count` clearly shows up as uncovered in the `Pos_T2` instance (`-` at offset 204 and on), while it is reported covered as expected in the `Pos_T1` instance (`+` at offset 1b4 and on):

```
posi__pos_t1__count +: 1ac-1e7
1ac +:  2f 80 00 00      cmpiw  cr7,r0,0x0000
1b0 +:  40 9d 00 24      ble-   cr7,0x1d4 <posi__pos_t1__count+0000003c>
1b4 +:  3c 00 00 00      lis    r0,0x0000  | cond branch not taken,
...                                        | fallthrough down to 1d4
...                                        v
1d4 +:  60 00 00 00      ori    r0,r0,0x0000
...
posi__pos_t2__count !: 1fc-237
1fc +:  2f 80 00 00      cmpiw  cr7,r0,0x0000
200 +:  40 9d 00 24      ble-   cr7,0x224 <posi__pos_t2__count+0000003c>
204 -:  3c 00 00 00      lis    r0,0x0000  | cond branch taken,
...                                        | skip everything up to 224
...                                        v
224 +:  60 00 00 00      ori    r0,r0,0x0000
...
```

The presence of uncovered instructions yields a partial coverage annotation for the corresponding source line in the `=xcov` output (`!` on line 10):

```
 6 .: package body Genpos is
 7 +:    procedure Count (X : Num_T) is
 8 .:    begin
 9 +:       if X > 0 then
10 !:         N_Positive := N_Positive + 1;
11 .:       end if;
12 +:    end Count;
13 .: end Genpos;
```

And the `=xcov+` (or `=html`) output gathers everything together, with the blocks of instructions coming from different instances identifiable by the associated object symbol names:

```
  10 !:          N_Positive := N_Positive + 1;
<posi__pos_t1__count+0000001c>:+
1b4 +:  3c 00 00 00  lis    r0,0x0000
...
<posi__pos_t2__count+0000001c>:-
204 -:  3c 00 00 00  lis    r0,0x0000
...
```

3.7 Full object coverage considerations

The previous sections focus on the coverage analysis of code attached to symbols. When full object level coverage is to be reached, extra care is required regarding *orphaned code regions*, which are not attached to any symbol, and *empty symbols*, for which the reported code size is null.

Orphaned regions usually show up out of legitimate code alignment requests issued for performance or target ABI specificities. Empty symbols most often result from low level assembly programmed parts missing the assembly directives aimed at populating the symbol table. Both are typically harmless so information about them is only emitted on explicit request. **gnatcov** provides the `scan-objects` command for this purpose. The command expects the set of object files to examine on the command line, as a sequence of either object file or *@listfile argument*, and reports about the two kinds of situations described above.

CHAPTER

FOUR

OBJECT/SOURCE LEVEL METRICS CONSIDERATIONS

Even though the executable code reflects semantics expressed in the application sources, Object and Source level coverage metrics are of very different nature, concerned with machine instructions vs high level constructs respectively.

Our purpose here is to illustrate this through a few examples, *not* to perform a qualitative comparison between the two kinds of criteria, way beyond the scope of this toolset user guide. The essential point is twofold:

- Stress that annotated source reports for object criteria remain focused on object level metrics, and that source representations are just a means to present the results in this case.
- Illustrate the GNATcoverage ability to compute accurate results for both kinds of criteria.

4.1 Main differences exemplified

To illustrate the main differences between the two kinds of metrics, we exercise the following functional Ada unit:

```
--  Return whether X divides Y, print a message when True

function Divides (X, Y : Integer) return Boolean is
begin
   if Y mod X = 0 then
      Put_Line (Integer'Image (X) & " divides " & Integer'Image (Y));
      return True;
   else
      return False;
   end if;
end Divides;
```

Using the basic test driver below:

```
procedure Test_Ops1  is
begin
   Assert (Divides (2, 4));
   Assert (not Divides (2, 5));
end Test_Ops1;
```

`Divides` features a simple decision controlling an *if* statement exercised both ways so the driver achieves statement and decision coverage. It even achieves mcdc since the decision has a single condition, which is reported by GNATcoverage with a 100% achievement for the *stmt+mcdc* coverage level and + annotations everywhere in the `=xcov` output:

```
gnatcov coverage --level=stmt+mcdc --scos=@alis --annotate=xcov test_ops1.trace
...

100% of 4 lines covered
Coverage level: stmt+mcdc
...
 5 .:    function Divides (X, Y : Integer) return Boolean is
 6 .:    begin
 7 +:       if Y mod X = 0 then
 8 +:          Put_Line (Integer'Image (X) & " divides " & Integer'Image (Y));
 9 +:          return True;
10 .:       else
11 +:          return False;
12 .:       end if;
13 .:    end Divides;
```

If we consider object coverage now, we have to consider that the Ada `mod` operator needs special treatment to handle negative operands, which incurs an internal test (conditional branch) and dedicated sequences of instructions. The operation normally also entails a check to raise the predefined `Constraint_Error` exception if X happens to be null. These sequences are not exercised by our basic driver, and object coverage for the same execution trace correctly reports partial achievement only:

```
gnatcov coverage --level=insn --annotate=xcov test_ops1.trace
...
67% of 6 lines covered
Coverage level: insn
...
 5 +:    function Divides (X, Y : Integer) return Boolean is
 6 .:    begin
 7 !:       if Y mod X = 0 then
 8 !:          Put_Line (Integer'Image (X) & " divides " & Integer'Image (Y));
 9 +:          return True;
10 .:       else
11 +:          return False;
12 .:       end if;
13 +:    end Divides;
```

Another difference we can notice here is the presence of coverage annotations on lines 5 and 13, which had `.` in the source coverage reports. This materializes the fact that there is machine code associated with these lines (prologue and epilogue sequences, in particular), but no entity of source level relevance (what we call *Source Coverage Obligation*) at all there.

4.2 Full branch coverage vs MCDC

The second example we look at is the canonical case which exposed that object branch coverage does not necessarily imply mcdc coverage, contrary to what was believed for long. Consider this source and the associated decision Binary Decision Diagram:

```
function Orand (A, B, C : Boolean) return Boolean is
begin
   return (A or else B) and then C;
end Orand;
```

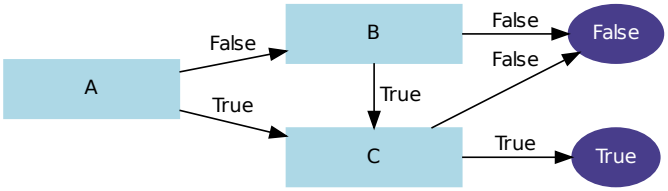


Fig. 1: BDD for `(A or else B) and then C`

The following simple driver exercises all the paths through this BDD:

```
procedure Test_Orand  is
   X : constant Boolean := True;
begin
   Assert (Orand (True, X, True) = True);
   Assert (Orand (False, False, X) = False);
   Assert (Orand (False, True, False) = False);
end Test_Orand;
```

As we will be comparing with the mcdc assessment, we pass `--scos` and `--level` to **`gnatcov run`** prior to anything else, so we will be able to reuse the same execution trace for both our object and source level experiments:

```
gnatcov run --scos=@alis --level=stmt+mcdc test_orand
```

Now we verify that GNATcoverage reports full object coverage as expected:

```
gnatcov coverage --level=branch --annotate=xcov test_orand.trace
...
100% of 3 lines covered
Coverage level: branch

1 +: function Orand (A, B, C : Boolean) return Boolean is
2 .: begin
3 +:    return (A or else B) and then C;
4 +: end Orand;
```

With 3 tests for 3 conditions, mcdc cannot be achieved yet and GNATcoverage reports this correctly as well. Using `=xcov+` to see the reason for partial coverage attached to line 3, we indeed get:

```
gnatcov coverage --level=stmt+mcdc --scos=@alis --annotate=xcov+ test_orand.trace
...
0% of 1 lines covered
Coverage level: stmt+mcdc

1 .: function Orand (A, B, C : Boolean) return Boolean is
2 .: begin
3 !:    return (A or else B) and then C;
CONDITION "B" at 3:22 has no independent influence pair, MC/DC not achieved
4 .: end Orand;
```

We have a clear illustration of the GNATcoverage ability to perform accurate assessments of distinct source and object criteria here, actually based on solid theoretical grounds established as part of the *Couverture* research project from

which GNATcoverage originates. The core particularity allowing full branch coverage without mcdc is the presence of decisions with BDDs which are not trees, as we have in this specific case,

CHAPTER

FIVE

USING GNATCOVERAGE FROM GNAT STUDIO

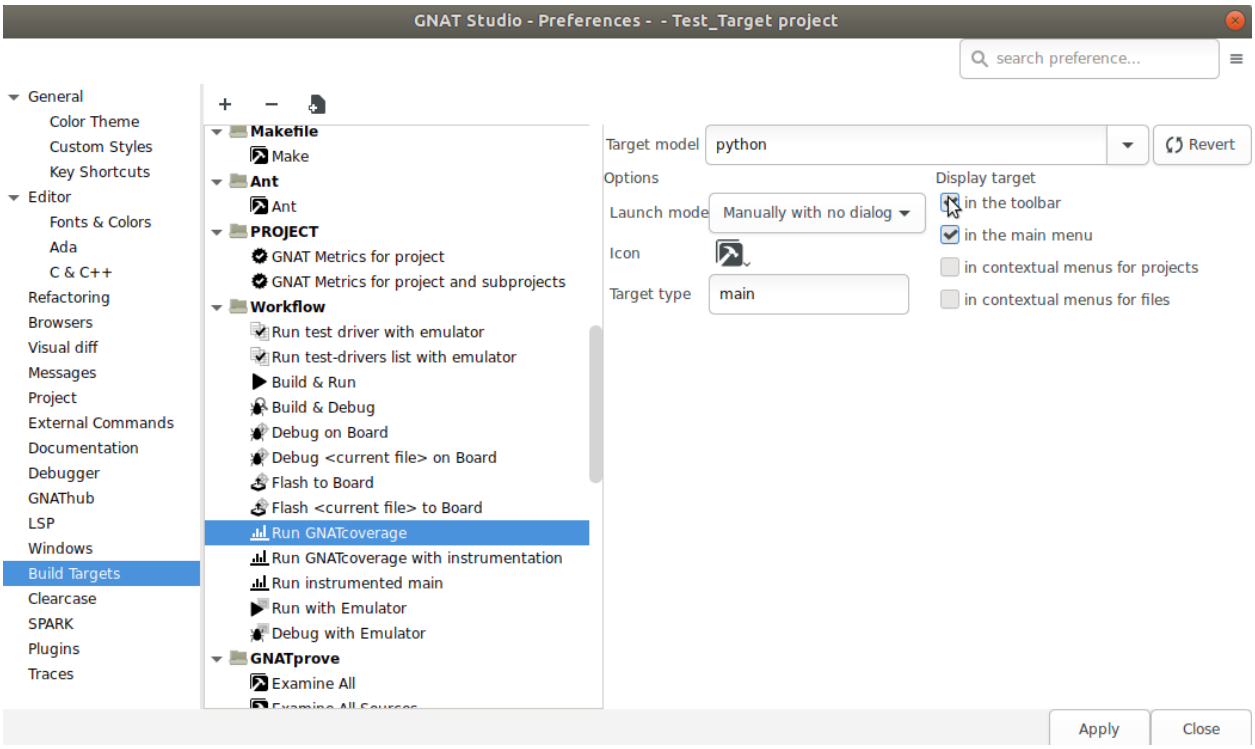
Starting from GNATcoverage version 22.0, binary traces (using gnatcov run) are no longer supported in native profiles. This mode is still supported for cross configurations, and the IDE integrated use in such cases is similar to the use for source traces:

Go to the *Analyze* → *Coverage* → *GNATcoverage Binary Traces* menu to perform each step individually, or use the `Run All Actions` item to do everything at once.

In configurations where binary traces are not supported, the menu entry just wouldn't show up.

You can also replace the toolbar button (dedicated to the instrumentation-based scheme) with an alike button, but for binary traces. Go to GNATstudio Preferences in *Edit* → *Preferences*.

Then, go to the `Build Targets` section, and find the `Run GNATcoverage` workflow: add it to the toolbar as shown below.



Remove the `Run GNATcoverage with instrumentation` entry from the toolbar in the same fashion.

CHAPTER

SIX

IMPORTING BINARY TRACES WITH GNATCOV CONVERT

Traces generated from external sources (other than by using **gnatcov run**) can be imported into GNATcoverage by converting them to the GNATcoverage format using **gnatcov convert**.

6.1 gnatcov convert command line

gnatcov convert converts trace data from foreign sources to the format used by **gnatcov coverage**. The general interface synopsis is available from `gnatcov --help`:

```
gnatcov convert --trace-source=SOURCE_ID --exec=EXECUTABLE
  --input=INPUT_TRACE --hw-trigger-traces=START_ID,START_ADDR,STOP_ID
  [OPTIONS]
```

Details about the available options follow:

--trace-source :
: An identifier of the source of the input trace data is provided using this option. The input source effects which other options are required. Currently, the only values for this option are `iSystem-5634` (see *Generating Coverage Information from Nexus Traces on MPC5634M*) and `Trace32-Branchflow` (see *Generating Coverage Information from Lauterbach Trace32*). For the `iSystem-5634` source, the `--hw-trigger-traces` option is required.

--exec :
: This option provides the filename of a file containing the target executable from which the trace data was generated, and upon which the coverage analysis is done.

--input :
: This option provides the name of the file containing the input trace data to be converted.

--output :
: The name of the file to write the output trace into can be provided by by this option. If `--output` is not used, the output filename is derived from the name of the executable, by taking the basename of the executable file and appending `.trace`.

--hw-trigger-traces :
: For Nexus modules that support the Class 4 HW Watchpoint Trigger feature, the generation of program trace data can be started and stopped in response to watchpoint events. When this mechanism is used, in order to interpret the Nexus traces, information about how the triggers were set during the run of the executable needs to be provided. For the `iSystem-5634` trace-source, **gnatcov convert** allows the use of the Instruction Address Compare registers for the `start/stop` watchpoints. `START_ID` is one of `IAC1`, `IAC2`, `IAC3`, or `IAC4`, and `START_ADDR` is the address set in the IAC identified by `START_ID`. The address can be specified numerically using either the usual C syntax (0xHHHH), Ada syntax (16#HHHH_HHHH#), or by providing an ELF symbol

table symbol name (e.g. `main`). `STOP_ID` can identify another of the IACs, or be `0`, to indicate that a stop trigger was not set up.

`--level=`
This is used in an identical manner as for **`gnatcov run`**.

`--scos=`
This is used in an identical manner as for **`gnatcov run`**.

6.2 Generating Coverage Information from Nexus Traces on MPC5634M

Nexus realtime Program Trace data can be produced while running an executable on a processor supporting the needed combination of Nexus capabilites. Such data can be processed by GNATcoverage in two steps:

- Invoke **`gnatcov convert`** on the Nexus trace data file, producing an intermediate file
- Invoke **`gnatcov coverage`** on this intermediate file

Currently, the freescale MPC5634M SOC is supported as a source of Nexus data, using iSystem tools to interface with the processor and to produce a file containing Nexus data.

The iSystem winIDEA software can be used to control program loading and execution on target 5634 hardware, and, through the Analyzer tool, to configure and execute Program Trace data collection and output. winIDEA is a GUI, and the information in this section describes the user interaction with the GUI. winIDEA APIs are provided for several languages (including Python and C++), and these are useful when not working in interactive mode.

winIDEA runs on a host computer that is connected via USB to iSystem hardware (the "Blue Box"), which in turn is connected to a Mictor connector on the target 5634 processor through a Nexus cable.

While the program and analyzer are running, Program Trace data and other Nexus data are sent over the processor's Nexus interface to the Blue Box, where it is annotated and buffered. (The buffering is shared between memory on the Blue Box and resources on the host computer; this affects how to configure the Analyzer).

The Analyzer can stop as a result of any of several conditions; e.g. when the program halts upon encountering a breakpoint, or when some condition set in the Analyzer configuration is met.

Once the Analyzer has stopped, it should not be used to collect additional data in the same buffer (the trace data will be inaccurate if there are gaps in its collection). The Nexus data collected can be written to a file suitable for **`gnatcov convert`** using the Analyzer's `export` function.

6.2.1 Nexus Program Trace Data: the Hardware Perspective

The 5634 hardware supplies two options for controlling when Nexus Program Trace Data is output while the CPU is running:

- A bit in a control register in the Nexus Module

  When this bit is set, Nexus data will be produced whenever the CPU is running.

- Watchpoint triggers

  If the control register bit is not set, Watchpoint Triggers may be used to generate trace data. Not all of the functionality of Watchpoint Triggers is supported by GNATcoverage, but what is used is built around the four Instruction Address Compare registers on the 5634 (IAC1 - IAC4). These registers can be used to cause watchpoint events when the PC attains specified values. In the basic usage, an address is set in one of the IAC registers and a watchpoint occurs when the PC matches the value set in the IAC. In the Nexus module, an IAC watchpoint event can be set to `start` or to `stop` the production of Nexus Program Trace messages.

6.2.2 Configuring the winIDEA Analyzer

The Watchpoint Triggers functionality of the 5634 is the (only) option supported by GNATcoverage. The Analyzer must be configured accordingly, and options to the **gnatcov convert** command reflect the trigger settings that were set by the Analyzer when the trace data was collected:

- the IAC that was used for the `start` trigger,
- the address that was in the `start` trigger IAC,
- the IAC used for the `stop` trigger.

The address in the `stop` trigger is not needed, and the `stop` trigger can be omitted completely.

Here are two examples:

- A program continually runs a loop where it receives a command and dispatches based on that command. The command input is implemented as a busy loop. To prevent a very large output of useless data from the busy loop, IAC1 is used as the `start` trigger, with the address set to the end of the busy loop, and IAC2 is set as the `stop` trigger, with its contents set to the beginning of the busy loop.
- For coverage from unit testing, small programs are run to completion. In this scenario, `start` can be set at the entry point for `main`. No `stop` trigger is needed, since a breakpoint is set upon exit from `main`.

In the GUI, the hammer with a sheet of paper icon brings up the `Analyzer Configuration List` dialog, and the hammer plus Trace icon brings up the `Analyzer Configuration` dialog. The following steps will create a configuration that will be applicable across a wide range of test execution scenarios:

- Open the `Analyzer Configuration List` dialog, choose New->Trace and choose a name for your configuration. This will bring up a trigger configuration window.
- The first, `Recorder` tab is used to set values for four properites (click on the initial values for the first three to see that they are drop down lists). For `Start`, `Buffer Size` and `Trigger Position` choose `Immediately`, `1 GB` (+/-) and `Begin`. Do not check `Break On Trigger`.
- Select the `MP5634M` tab, and check `Enabled`. IAC1 and IAC2 will be used for the `start` and `stop` triggers; check them and enter addresses. In the `Record` box, only `Program` should be selected, and for `Start` and `Stop`, IAC1 and IAC2 should be chosen. `Type` needs to be `Individual Branch Messages`, and Watchpoints should be `All`. Click `OK`.

  > *Note*: The address set for the Start Trigger should not contain a branch instruction, since such a trigger may cause the Analyzer to lose the context for interpreting some Nexus messages. **gnatcov convert** will generate an error message when it is passed a start trigger address for a branch instruction.

- Back at the `Analyzer Configuration` window, the new configurataion will be shown to be the active one with a red arrow on the left. Check both `Start Analyzer when CPU starts` and `Stop Analyzer when CPU stops`. The first is a minor convenience, making it unnecessary to explicitly start the Analyzer before performing the `start` from winIDEA.

6.2.3 Exporting the Trace Data

To set the relevant arguments for the `export` command (whose icon looks like a diskette with a green arrow):

- In the window that pops up for `export`, specify the file desired and choose `Binary` from the dropdown list for `Format`;
- Select `Options...`, and in the resulting window, choose only `On Chip Data`;
- In the `export` window choose `Entire Session` for the `Time Scope`;
- Clicking `OK` will write the file; this may take a long time for large trace collections.

6.3 Generating Coverage Information from Lauterbach Trace32

Lauterbach Trace32 BranchFlow trace data can be produced while running an executable on a processor supporting the needed execution trace output.

In environments where a Lauterbach probe is available for a target board with a trace port, the Trace32 software can be used to control program loading and execution on target hardware, and configure program trace data collection and output. Trace32 APIs are provided for several languages (including Python and C), and these are useful when not working in interactive mode.

Trace32 runs on a host computer that is connected via USB or Ethernet to a hardware probe, which in turn is connected to a target processor.

Coverage analysis from Trace32 data involves four steps, requiring the specific ability to export BranchFlow traces from Trace32:

- Run the executable on the target using Lauterbach probe and Trace32;
- Export BranchFlow trace with Trace32;
- Invoke **gnatcov convert** on the BranchFlow trace data file, producing an intermediate file;
- Invoke **gnatcov coverage** on this intermediate file.

While the program is running, trace data is sent over the processor's trace interface (depending on the architecture) to the probe. The trace data is then sent to Trace32 on the host computer to be exported to a file suitable for **gnatcov convert**.

6.3.1 Enabling Trace Data

For Trace32 to produce BranchFlow execution traces, the target hardware and the probe have to be configured properly. This configuration depends on the target architecture and the specific device.

We recommend starting from one of the examples provided for your hardware in the Trace32 demo folder, for instance `demo/arm/hardware/stm32/stm32f4/sieve/stm32f4_discovery_offchip_trace.cmm`, and to contact Lauterbach support for advice on the configuration for your target.

6.3.2 Running the executable

The executable can be loaded on the target using the Trace32 command `Data.LOAD.auto <ELF_executable>`.

Before starting the program, a termination point has to be defined using breakpoint. We recommend setting a breakpoint on the `_exit` function using the Trace32 command `Break.Set _exit`.

To run the program, use the `Go.direct` Trace32 command.

6.3.3 Exporting the Trace Data

Once the program reached the termination breakpoint, the trace data can be exported using the `Trace.export.BranchFlow <filename> /NOSYMBOL /CALLER` command. The last two options are required to produce a trace file in a format that **gnatcov convert** can translate.

The BranchFlow trace should then be converted to GNATcoverage format using **gnatcov convert** command (see *gnatcov convert command line*).

6.3.4 Automation

Trace32 provides an external API in C and Python to control the execution of commands. This interface can be used to automate the process described above: enable trace, run executable, export trace. Please refer to the Trace32 documentation for more information, for instance `Controlling TRACE32 via Python 3`.

CHAPTER

SEVEN

TARGET SPECIFIC POINTS OF NOTE WITH BINARY TRACES

The following list summarizes points of note for each target where some aspects of the build/execution/analysis process depart from the general instructions.

For **cross** configurations in general:

- Need to convey the target to **gnatcov run**, either with a `--target` switch or a `Target` project file attribute;
- Need to convey the target and Ada runtime when using project files to state units of interest, for **gnatcov coverage**, **gnatcov instrument** or **gnatcov run**. See *Specifying Units Of Interest*, *Other switches or attributes of importance*.

For **powerpc-vxworks6**:

- Need to compile with -gno-strict-dwarf -mlongcall for source coverage analysis;
- Need to add a –kernel argument on **gnatcov run**;
- The provided kernel has to be augmented with a specific module for gnatcov purposes. Please refer to the GNATemulator documentation for this part of the process;
- Support for the `kernel` Ada RTS and Downloadable Kernel Modules only.

CHAPTER

EIGHT

KNOWN LIMITATIONS WITH BINARY TRACES

8.1 Incomplete statement coverage on lines with multiple statements

On lines with multiple statements, the tool may not be able to infer accurate statement coverage results for each individual statement. The tool emits explicit diagnostics in this case.

In `=report` outputs (with `--annotate=report`), this translates as a statement coverage violation like:

```
<sloc>: multiple statements on line, unable to establish full statement coverage
```

where <sloc> is a source-filename:line:column source location of the problematic line.

In annotated sources kinds of outputs, this materializes as '!' note indicating partial coverage on the affected lines, with the `=report` violation text quoted above available as part of the line extra-details expanded on demand (`=html` and `xcov+` output formats).

8.2 MCDC inaccuracies with interrupts and multi-threaded applications

There is one limitation in GNATcoverage with respect to MCDC assessments from binary traces: potential inaccuracies in results reported for particular decisions when these decisions are evaluated concurrently by different threads or mixed with interrupt processing in bare metal configurations.

Technically, the decisions of concern are those for which the associated binary decision diagram is not a tree, that is, those with at least one condition node joining several possible evaluation paths.

The code sample below illustrates the simplest possible problematic decision and the following figure depicts the corresponding Binary Decision Diagram (commonly abbreviated as *BDD*), which states how sequence of operand evaluations, starting from the left, eventually lead to the expression outcome, here on the right:

```
function Mp (A, B, C : Boolean) return Boolean is
begin
  return (A or else B) and then C;
end;
```

The expression BDD is indeed not a tree, as the node representing the evaluation of C is reachable either directly from A, when A is True, or via B when A is False.

According to measures performed on a few large real code bases, occurrences of such decisions are statistically rare. **gnatcov** can report about them on demand, thanks to the **scan-decisions** command together with the the set of coverage obligations to examine. Below is an excerpt of a an execution for a project which encompasses this function, where we see that **gnatcov** provides the source location of conditions reachable through multiple paths:

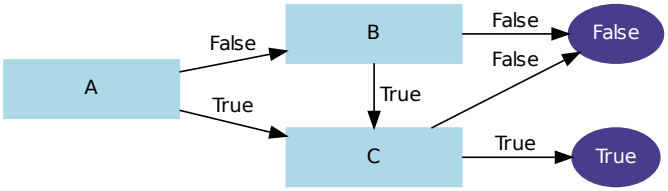


Fig. 1: BDD for `(A or else B) and then C`, not a tree

```
gnatcov scan-decisions -Pmytest.gpr
...
*** mp.adb:4:33: warning: condition is reachable through multiple paths
```

INDEX

Symbols

@listfile argument, **128**
-U (*gnattest*), 137
-X (*gnattest*), 137
--RTS (*gnattest*), 137
--additional-tests (*gnattest*), 137
--copy-environment (*gnattest*), 141
--exclude-from-stubbing (*gnattest*), 138
--exit-status (*gnattest*), 138
--harness-dir (*gnattest*), 138
--harness-only (*gnattest*), 137
--ignore (*gnattest*), 137
--inheritance-check (*gnattest*), 138
--no-command-line (*gnattest*), 139
--no-inheritance-check (*gnattest*), 138
--no-subprojects (*gnattest*), 137
--omit-sloc (*gnattest*), 139
--passed-tests (*gnattest*), 138, 141
--queues (*gnattest*), 141
--recursive-stub (*gnattest*), 138
--reporter (*gnattest*), 139
--separate-drivers (*gnattest*), 137
--skeleton-default (*gnattest*), 138
--strict (*gnattest*), 137
--stub (*gnattest*), 137
--stubs-dir (*gnattest*), 138
--subdirs (*gnattest*), 138, 141
--test-duration (*gnattest*), 139
--tests-dir (*gnattest*), 138
--tests-root (*gnattest*), 138
--validate-type-extensions (*gnattest*), 138
-files (*gnattest*), 137
-j (*gnattest*), 141
-q (*gnattest*), 137
-r (*gnattest*), 137
-v (*gnattest*), 137

B

Binary Trace, **127**

C

Complex Boolean Expression, **127**
Condition, **127**
Coverage Checkpoint, **127**
Coverage Checkpoints, 95
Coverage Consolidation, 104
Coverage Runtime, **127**
Coverage Violation, **127**

D

Decision, **127**

E

Evaluation Vector, **127**
Exemption Region, **127**

G

gnatcov convert command line, 339
gnatcov coverage command line for object coverage, 321
gnatcov coverage command line for source coverage, 51
gnatcov run command line, 316
gnattest, **133**

I

Incremental Coverage Analysis, **127**
Independence Pair, **128**

L

Library Information file, **128**

S

Simple Boolean Expression, **128**
Source Coverage Obligation, **128**
Source Instrumentation Data file, **128**
Source Trace, **128**

T

Test Driver, **128**

U

Units of Interest, **128**