GNAT SAS Overview

About This Course

Styles

- *This* is a definition
- `this/is/a.path`
- `code is highlighted`
- `commands are emphasised --like-this`

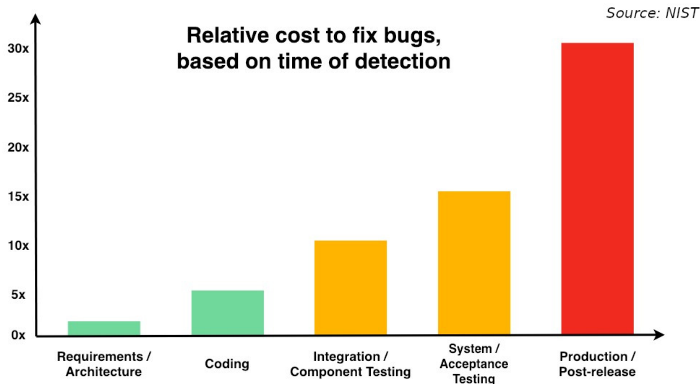
GNAT Static Analysis Suite (GNAT SAS)

What Is Static Analysis?

- **Symbolic** interpretation of **source code**
  - Find what could go wrong
  - No execution
- **Formally** verifying **high level** or **abstract** properties
  - Strong guarantees
- *May* be exhaustive
  - All possible errors are reported
  - No false negatives; there may be false positives
    - If the analyzer does not report a problem, there is no problem

How Does Static Analysis Save Money?

- Costs shift
  - From later, **expensive** phases
  - To earlier, **cheaper** phases



What Is GNAT SAS?

- Set of analysis engines with complementary capabilities
- Able to detect range of issues spanning from breaking coding style standards to deep logic errors
- Designed to support large systems and to detect wide range of programming errors such as
  - Misuse of pointers
  - Indexing out of arrays
  - Buffer overflows
  - Numeric overflows
  - Numeric wraparounds
  - Improper use of Application Programming Interfaces (APIs)
  - and more

What Does GNAT SAS Do?

- Pinpoints root cause of each error to the source line of code
- Analyzes partial or full systems to produce reports
- Maintains history to compare current results to a baseline

GNAT Metrics Tool

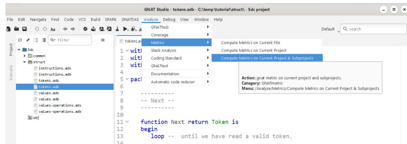
Introduction

Overview of GNAT Metrics Tool GNATMETRIC

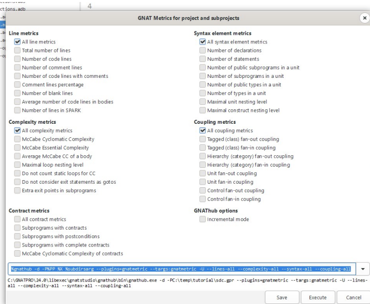
- Utility for computing various program metrics
- Select desired metrics from:
  - Lines
  - Complexity
  - Contract
  - Syntax elements (e.g. nesting levels, number of parameters)
  - Coupling
- Configurable scope of analysis
  - Current file
  - Current project
  - Current project and subprojects
- GNAT STUDIO provides a GUI interface
  - Selecting the metrics to compute
  - Selecting the scope of analysis
  - Displaying the results

Invoking From GNAT Studio

- Start analysis



- Select options and perform analysis



Invoking From the Command Line

- Command line help (partial output)

```
gnatmetric --help
```

```
usage: gnatmetric [options] {filename}
 options:
 --version - Display version and exit
 --help    - Display usage and exit

 -Pproject        - Use project file project. Only one such switch can be used
 -U               - process all sources of the argument project
 -U main          - process the closure of units rooted at unit main
 --no-subprojects - Process sources of root project only
 -Xname=value     - specify an external reference for argument project file
 --subdirs=dir    - specify subdirectory to place the result files into
 -eL              - follow all symbolic links when processing project files

 --verbose    - verbose mode
 --quiet      - quiet mode
```

- Command line invocation

```
gnatmetric -P sdc.gpr -U --lines-all --complexity-all --syntax-all --coupling-all
```

Useful Command Line Options

`--help`	Display usage and exit
`-Pproject`	Use project file project. Only one such switch can be used
`-U main`	Process the closure of units rooted at unit `main`
`--contract-all`	All contract metrics
`--complexity-all`	All complexity metrics
`--lines-all`	All line metrics
`--syntax-all`	All syntax element metrics
`--coupling-all`	All coupling metrics

Output Control

Generated Outputs

GNATMETRIC has three types of outputs

- Execution log
  - Text output from command
- Command results
  - Text files for each unit processed
  - `<ada-filename>.metrix`
- Complete results
  - XML file containing results for all units processed
  - `metrix.xml`

Controlling Output Generation

- From GNAT STUDIO
  - *Execution log* generated and stored in `gnathub/logs/gnatmetric.log` in object file folder
  - *Command results* generated and stored in object file folder
  - *Complete results* generated and stored in object file folder
- From the command line
  - When setting switch `--no-text-output`
    - *Execution log* not generated
    - *Command results* not generated
    - *Complete results* generated and stored in object file folder
  - Without switch `--no-text-output`
    - *Execution log* displayed on console
    - *Command results* generated and stored in object file folder
    - *Complete results* only generated if switches `--generate-xml-output` or `--generate-xml-schema` specified

Execution Log

```
gnatmetric -P default.gpr -U --lines-all
```

```
Line metrics summed over 11 units
  all lines            : 141
  code lines           : 118
  comment lines        : 1
  end-of-line comments : 0
  comment percentage   : 0.84
  blank lines          : 22

Average lines in body: 7.33
```

Command Results

```
gnatmetric -P default.gpr -U --lines-all
```

```
Metrics computed for src\line_metrics_example.adb
containing package body Line_Metrics_Example

=== Code line metrics ===
  all lines           : 19
  code lines          : 15
  comment lines       : 1
  end-of-line comments: 0
  comment percentage  : 6.25
  blank lines         : 3

Average lines in body: 6.00

Line_Metrics_Example (package body - library item at lines  2: 19)

=== Code line metrics ===
   all lines           : 18
   code lines          : 14
   comment lines       : 1
   end-of-line comments: 0
   comment percentage  : 6.66
   blank lines         : 3

   Internal (procedure body at lines  4: 7)

   === Code line metrics ===
      all lines           : 4
      code lines          : 4
      comment lines       : 0
      end-of-line comments: 0
      comment percentage  : 0.00
      blank lines         : 0

   Example (procedure body at lines  10: 17)

   === Code line metrics ===
      all lines           : 8
      code lines          : 8
      comment lines       : 0
      end-of-line comments: 0
      comment percentage  : 0.00
      blank lines         : 0
```

Complete Results

```
gnatmetric -P default.gpr -U --lines-all --generate-xml-output
```

(partial file)

```
<file name="C:\temp\gnatmetric\src\line_metrics_example.adb">
   <metric name="all_lines">19</metric>
   <metric name="code_lines">15</metric>
   <metric name="comment_lines">1</metric>
   <metric name="eol_comments">0</metric>
   <metric name="comment_percentage">6.25</metric>
   <metric name="blank_lines">3</metric>
   <metric name="average_lines_in_bodies">6.00</metric>
   <unit name="Line_Metrics_Example" kind="package body" line="2" col="1">
      <metric name="all_lines">18</metric>
      <metric name="code_lines">14</metric>
      <metric name="comment_lines">1</metric>
      <metric name="eol_comments">0</metric>
      <metric name="comment_percentage">6.66</metric>
      <metric name="blank_lines">3</metric>
      <unit name="Internal" kind="procedure body" line="4" col="4">
         <metric name="all_lines">4</metric>
         <metric name="code_lines">4</metric>
         <metric name="comment_lines">0</metric>
         <metric name="eol_comments">0</metric>
         <metric name="comment_percentage">0.00</metric>
         <metric name="blank_lines">0</metric>
      </unit>
      <unit name="Example" kind="procedure body" line="10" col="4">
         <metric name="all_lines">8</metric>
         <metric name="code_lines">8</metric>
         <metric name="comment_lines">0</metric>
         <metric name="eol_comments">0</metric>
         <metric name="comment_percentage">0.00</metric>
         <metric name="blank_lines">0</metric>
      </unit>
   </unit>
</file>
```

A Little More on Controlling Output Generation

Some more switches to control output generation

output-dir=dirname	Store `<ada-filename>.metrix` into `dirname`
generate-xml-output	Generate XML output
generate-xml-schema	Generate XML output and corresponding schema file
no-text-output	No `<ada-filename>.metrix` or log files
output-suffix=file-suffix	Add `file-suffix` to end of filename for file results *Add "." if you want it to be a file extension*
global-file-name=filename	Full path to the executable log file
xml-file-name=filename	Full path to the XML file
short-file-names	Use short source file names in output

Exploring the Results

Line Metrics Explained

Average Lines In Body	Average number of code lines in subprogram bodies, task bodies, entry bodies and package body executable code
All Lines	Total number of lines in file(s)
Blank Lines	Total number of blank in file(s)
Code Lines	Total lines of code in file(s)
Comment Lines	Total lines of comments in file(s)
Comment Percentage	Comment lines divided by total of code lines and comment lines
End-Of-Line Comments	Count of code lines that also contain comments

Code line is a non-blank line that is not a comment

Line Metrics Code Example

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  package body Line_Metrics_Example is
3
4     procedure Internal (C : Character) is
5     begin
6        Put (C);
7     end Internal;
8
9     -- Print the prompt
10    procedure Example (S1, S2 : String) is
11       S : constant String := S1 & S2;
12    begin
13       for C of S loop
14          Internal (C);
15       end loop;
16       New_Line;
17    end Example;
18
19 end Line_Metrics_Example;
```

Line Metrics Output

```
gnatmetric -Pdefault.gpr --lines-all line_metrics_example.adb
```

line_metrics_example.metrix

```
=== Code line metrics ===
  all lines           : 19
  code lines          : 15
  comment lines       : 1
  end-of-line comments: 0
  comment percentage  : 6.25
  blank lines         : 3

Average lines in body: 6.00

Line_Metrics_Example (package body - library item at lines  2: 19)

=== Code line metrics ===
   all lines           : 18
   code lines          : 14
   comment lines       : 1
   end-of-line comments: 0
   comment percentage  : 6.66
   blank lines         : 3

   Internal (procedure body at lines  4: 7)

   === Code line metrics ===
      all lines           : 4
      code lines          : 4
      comment lines       : 1
      end-of-line comments: 0
      comment percentage  : 0.00
      blank lines         : 0

   Example (procedure body at lines  10: 17)

   === Code line metrics ===
      all lines           : 8
      code lines          : 8
      comment lines       : 0
      end-of-line comments: 0
      comment percentage  : 0.00
      blank lines         : 0
```

Syntax Element Metrics Explained

All Declarations	Total number of objects declared
All Statements	Total number of statements in file(s)
All Subprogram Bodies	Total number of subprograms in file(s)
All Type Definitions	Total number of types in file(s)
Logical SLOC	Total of declarations plus statements
Public Subprograms	Count of subprograms declared in visible part of package
Public Types	Count of types (not subtypes) declared in the visible part of a package plus in the visible part of a generic nested package
Maximal Construct Nesting	Maximal nesting level of composite syntactic constructs
Maximum Unit Nesting	Maximal static nesting level of inner program units

Syntax Element Metrics Code Example

```
1  package body Syntax_Metrics_Example is
2
3    function "&"
4      (L, R : String_T)
5       return String_T is
6    (From_String (To_String (L) & To_String (R)));
7
8    function To_String
9      (S : String_T)
10      return String is
11   (S.Text (1 .. S.Length));
12
13   function From_String
14     (S : String)
15      return String_T is
16      L      : constant Natural
17             := Integer'Min (S'Length, Maximum_Length);
18      Retval : String_T;
19   begin
20      Retval.Length        := L;
21      Retval.Text (1 .. L) := S (S'First .. S'First + L - 1);
22      return Retval;
23   end From_String;
24
25 end Syntax_Metrics_Example;
```

Syntax Element Metrics Output

```
gnatmetric -Pdefault.gpr --syntax-all syntax_metrics_example.adb
```

syntax_metrics_example.metrix

```
Syntax_Metrics_Example (package body - library item at lines  1: 25)

=== Element metrics ===
   all subprogram bodies    : 1
   all statements           : 3
   all declarations         : 9
   logical SLOC             : 12
   maximal unit nesting     : 1
   maximal construct nesting: 2

   "&" (expression function at lines  3: 6)

   === Element metrics ===
      all statements           : 0
      all declarations         : 2
      logical SLOC             : 2
      maximal construct nesting: 1
      all parameters           : 2
      IN parameters            : 2
      OUT parameters           : 0
      IN OUT parameters        : 0

   To_String (expression function at lines  8: 11)

   === Element metrics ===
      all statements           : 0
      all declarations         : 2
      logical SLOC             : 2
      maximal construct nesting: 1
      all parameters           : 1
      IN parameters            : 1
      OUT parameters           : 0
      IN OUT parameters        : 0

   From_String (function body at lines  13: 23)

   === Element metrics ===
      all statements           : 3
      all declarations         : 4
      logical SLOC             : 7
      maximal construct nesting: 1
```

Complexity Metrics Explained

Average Complexity	Total Cyclomatic Complexity divided by total number of subprograms
Cyclomatic Complexity	McCabe cyclomatic complexity (number of independent paths in the control flow graph)
Essential Complexity	McCabe essential complexity (cyclomatic complexity after removing blocks with single entry/exit points)
Expression Complexity	Complexity introduced by short-circuit control forms only
Maximum Loop Nesting	Maximum depth of nested loops
Statement Complexity	Complexity introduced by control statements only, without taking into account short-circuit forms

Understanding McCabe Complexity

http://www.mccabe.com/pdf/mccabe-nist235r.pdf

- Given a control flow graph of a program
  - E - number of edges
  - N - number of nodes
  - P - number of connected components (exit nodes)
- The complexity *v(G)* is computed by:

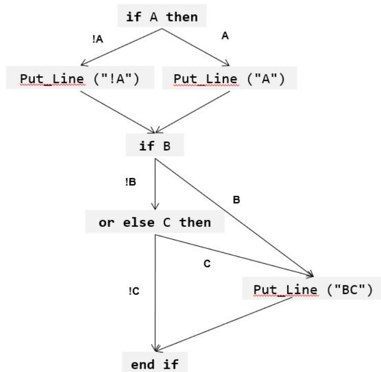
$$v(G) = E - N + 2 * P$$

- Aimed a measuring the complexity of execution paths
- Needs to be adapted for each language

McCabe Example

```
if A then
   Put_Line ("A");
else
   Put_Line ("!A");
end if;

if B or else C then
   Put_Line ("BC");
end if;
```



9 edges - 7 nodes + 2 * 1 exit = complexity 4

Complexity Metrics Code Example

```ada
package body Complexity_Metrics_Example is

   procedure Example (S : in out String) is
      Retval : String (S'First .. S'Last);
      Next   : Integer := S'First;
      procedure Set (C : Character) is
      begin
         Retval (Next) := C;
         Next          := Next + 1;
      end Set;
   begin
      if S'Length > 0 then
         for C of reverse S loop
            Set (C);
         end loop;
      end if;
   end Example;

end Complexity_Metrics_Example;
```

Complexity Metrics Output

```
gnatmetric -Pdefault.gpr --complexity-all complexity_metrics_example.adb
```

complexity_metrics_example.metrix

```
Complexity_Metrics_Example (package body - library item at lines  1: 19)

   Example (procedure body at lines  3: 17)

   === Complexity metrics ===
      statement complexity     : 3
      expression complexity    : 0
      cyclomatic complexity    : 3
      essential complexity     : 1
      maximum loop nesting     : 1
      extra exit points        : 0

      Set (procedure body at lines  6: 10)

      === Complexity metrics ===
         statement complexity     : 1
         expression complexity    : 0
         cyclomatic complexity    : 1
         essential complexity     : 1
         maximum loop nesting     : 0
         extra exit points        : 0

=== Average complexity metrics ===
      statement_complexity     :  2.00
      expression_complexity    :  0.00
      cyclomatic_complexity    :  2.00
      essential_complexity     :  1.00
      max_loop_nesting         :  1.00
```

Coupling Metrics Explained

- Measures dependencies between given entity and other entities in the program
  - High coupling may signal potential issues with maintainability
- Metrics computed:

Object-oriented coupling	Classes in traditional object-oriented sense
Unit coupling	All units making up a program
Control coupling	Dependencies between unit and other units that contain subprograms

Coupling Metrics

- Uses Ada's approach to definition of *class*, but only for polymorphic classes:
  - Tagged types declared within packages
  - Interface types declared within packages
- Two kinds of coupling computed:

Fan-out coupling	Number of classes given class depends on
Fan-in coupling	Number of classes that depend on given class

- Package bodies and specs for *classes* are both considered when computing dependencies

Coupling Metrics Code Example

```
package Coupling_Metrics_Dependency is
   type Record_T is tagged private;
   function Set (A, B : Integer) return Record_T;
   function Get (A : Record_T) return Integer;
   function Add (A, B : Record_T) return Record_T;
private
   type Record_T is tagged record
      Field1, Field2 : Integer;
   end record;
end Coupling_Metrics_Dependency;

with Coupling_Metrics_Dependency;
use Coupling_Metrics_Dependency;
package Coupling_Metrics_Example is
   procedure Example (L, R : Record_T);
end Coupling_Metrics_Example;

with Coupling_Metrics_Dependency;
use Coupling_Metrics_Dependency;
with Coupling_Metrics_Example;
procedure Main is
   A : constant Record_T := Set (1, 2);
   B : constant Record_T := Set (30, 40);
begin
   Coupling_Metrics_Example.Example (A, B);
end Main;
```

Coupling Metrics Output

```
gnatmetric -Pdefault.gpr -U --coupling-all
```

```
Coupling metrics:
=================
   Unit Coupling_Metrics_Dependency (coupling_metrics_dependency.ads)
      tagged fan-out coupling   : 0
      hierarchy fan-out coupling: 0
      tagged fan-in coupling    : 0
      hierarchy fan-in coupling : 0
      control fan-out coupling  : 0
      control fan-in coupling   : 2
      unit fan-out coupling     : 0
      unit fan-in coupling      : 2

   Unit Coupling_Metrics_Example (coupling_metrics_example.ads)
      control fan-out coupling  : 1
      control fan-in coupling   : 1
      unit fan-out coupling     : 1
      unit fan-in coupling      : 1

   Unit Main (main.adb)
      control fan-out coupling  : 2
      control fan-in coupling   : 0
      unit fan-out coupling     : 2
      unit fan-in coupling      : 0
```

Lab

GNATmetric Lab Setup

- Copy the `tutorial` folder from the course materials location
- Contents of the tutorial folder:
    - `sdc.gpr` - project file
    - `common` - source directory
    - `struct` - source directory
    - `obj` - object file (and metrics results) directory
- From a command prompt, type `gnatmetric --help` to verify your path is set correctly
    - If not, add the appropriate `bin` directory to your path
    - Typically (for Windows), this is located in `C:\GNATSAS\<version>\bin`

GNATmetric Lab - GUI Part 1

- Use GNAT STUDIO to open the project `sdc.gpr`
- Select `instructions.adb` in the `struct` folder
- Perform metrics analysis to get all line metrics on this file
  - Analyze -> Metrics -> Compute Metrics on Current File
  - Select **All line metrics** and press Execute

Question 1

- How many lines in the file?
  In subprogram `Process`?

GNATmetric Lab - GUI Part 1

- Use GNAT STUDIO to open the project `sdc.gpr`
- Select `instructions.adb` in the `struct` folder
- Perform metrics analysis to get all line metrics on this file
  - Analyze -> Metrics -> Compute Metrics on Current File
  - Select **All line metrics** and press Execute

Question 1

- How many lines in the file? — 59 lines in the file
  In subprogram `Process`? — 20 lines in Process

GNATmetric Lab - GUI Part 1

- Use GNAT Studio to open the project `sdc.gpr`
- Select `instructions.adb` in the `struct` folder
- Perform metrics analysis to get all line metrics on this file
  - Analyze -> Metrics -> Compute Metrics on Current File
  - Select **All line metrics** and press Execute

Question 1

- How many lines in the file?
  In subprogram `Process`?

59 lines in the file
20 lines in Process

Question 2

- Is there any information for the package spec?

GNATmetric Lab - GUI Part 1

- Use GNAT Studio to open the project `sdc.gpr`
- Select `instructions.adb` in the `struct` folder
- Perform metrics analysis to get all line metrics on this file
  - Analyze -> Metrics -> Compute Metrics on Current File
  - Select **All line metrics** and press Execute

Question 1

- How many lines in the file? In subprogram `Process`?

59 lines in the file
20 lines in Process

Question 2

- Is there any information for the package spec?

No - Current File means actual file, not package

GNATmetric Lab - GUI Part 2

Perform metrics analysis to get all complexity metrics in the project

GNATmetric Lab - GUI Part 2

Perform metrics analysis to get all complexity metrics in the project

- Analyze -> Metrics -> Compute Metrics on Current Project
- Select **All complexity metrics** and press Execute

GNATmetric Lab - GUI Part 2

Perform metrics analysis to get all complexity metrics in the project

- Analyze -> Metrics -> Compute Metrics on Current Project
- Select **All complexity metrics** and press Execute

Question 1

- What is the average complexity for the project? `stack.adb`?

GNATmetric Lab - GUI Part 2

Perform metrics analysis to get all complexity metrics in the project

- Analyze -> Metrics -> Compute Metrics on Current Project
- Select **All complexity metrics** and press Execute

Question 1

- What is the average complexity for the project? `stack.adb`?

2.3
1.7

GNATmetric Lab - GUI Part 2

Perform metrics analysis to get all complexity metrics in the project

- Analyze -> Metrics -> Compute Metrics on Current Project
- Select **All complexity metrics** and press Execute

Question 1

- What is the average complexity for the project? `stack.adb`?

2.3
1.7

Question 2

- Which file has an essential complexity of 1?

GNATmetric Lab - GUI Part 2

Perform metrics analysis to get all complexity metrics in the project

- Analyze -> Metrics -> Compute Metrics on Current Project
- Select **All complexity metrics** and press Execute

Question 1

- What is the average complexity for the project? `stack.adb`?

2.3
1.7

Question 2

- Which file has an essential complexity of 1?

`sdc.adb`

GNATmetric Lab - CLI Part 1

- Use the command line to generate syntax elements metrics for the project

```
gnatmetric -Psdc.gpr -U --syntax-all
```

Question 1

- How many total statements and declarations in the project?

GNATmetric Lab - CLI Part 1

- Use the command line to generate syntax elements metrics for the project

```
gnatmetric -Psdc.gpr -U --syntax-all
```

Question 1

- How many total statements and declarations in the project?

Statements - 160
Declarations - 195

GNATmetric Lab - CLI Part 1

- Use the command line to generate syntax elements metrics for the project

```
gnatmetric -Psdc.gpr -U --syntax-all
```

Question 1

- How many total statements and declarations in the project?

Statements - 160
Declarations - 195

Question 2

- What are the number of statements and declarations for procedure `Push` in package `Stack`?

GNATmetric Lab - CLI Part 1

- Use the command line to generate syntax elements metrics for the project

```
gnatmetric -Psdc.gpr -U --syntax-all
```

Question 1

- How many total statements and declarations in the project?

Statements - 160
Declarations - 195

Question 2

- What are the number of statements and declarations for procedure `Push` in package `Stack`?

Statements - 5
Declarations - 2
You need to open the file `obj\stack.adb.metrix` to get the data

GNATmetric Lab - CLI Part 2

Generate a local version of the combined XML metrics file for coupling metrics without generating any of the text files

GNATmetric Lab - CLI Part 2

Generate a local version of the combined XML metrics file for coupling metrics without generating any of the text files

```
gnatmetric -Psdc.gpr -U --coupling-all --no-text-output
--xml-file-name=.\local.xml
```

GNATmetric Lab - CLI Part 2

Generate a local version of the combined XML metrics file for coupling metrics without generating any of the text files

```
gnatmetric -Psdc.gpr -U --coupling-all --no-text-output
--xml-file-name=.\local.xml
```

Question

- How many total lines in the generated XML file?

GNATmetric Lab - CLI Part 2

Generate a local version of the combined XML metrics file for coupling metrics without generating any of the text files

```
gnatmetric -Psdc.gpr -U --coupling-all --no-text-output
--xml-file-name=.\local.xml
```

Question

- How many total lines in the generated XML file? 118

Summary

Closing Remarks

- See the GNAT User's Guide for further details of all the switches
- GNATMETRIC switches can be specified in a GPR file via the "Metrics" package
- GNATMETRIC is based on the LKQL library
  - Allows tool to parse files that may not actually compile

GNATcheck

Introduction

GNATcheck Is...

- An **automated** coding standards checker
- Capable of expressing a variety of rules
  - GNAT compiler **warnings and style** checks
  - Language-defined and GNAT-defined **restrictions**
  - Complexity **metrics**
  - **Specific** GNATcheck rules
- Qualified to DO-178 in several programs
- Integrated in GNAT Studio

Required by DO-178

Table A-5

Verification Of Outputs of Software Coding & Integration Process

	Objective		Applicability by SW Level			
	Description	Ref.	A	B	C	D
	...					
4	Source Code conforms to standards	6.3.4.d	○	○	○	
	...					

● Should be satisfied with independence

○ Should be satisfied

Conformance to Standards Requirement - DO-178

6.3.4 Reviews and Analyses of the Source Code

d. Conformance to standards

The objective is to **ensure that the Software Code Standards were followed** during the development of the code, especially **complexity restrictions and code constraints** that would be consistent with the system safety objectives.

Complexity includes the degree of coupling between software components, the nesting levels for control structures, and the complexity of logical or numeric expressions.

This analysis also ensures that **deviations to the standards are justified.**

GNATCHECK Input Requirements

- Can analyze sources that are not legal
  - But may result in false negatives due to missing/incorrect semantic information
  - Switch `check-semantic` can check if sources are legal
- Can analyze standalone files
  - But will not parse dependencies
  - Use a GNAT Project File as input for better analysis

Getting Started

Basic Usage

Command Line Invocation

```
gnatcheck [options] filename -files=filename [-cargs gcc_switches] -rules rule_switches
```

Argument	Description
{filename}	File to analyze (wildcards allowed)
{files=filename}	`filename` specifies text file containing list of files to analyze
-rules rule_switches	Rules to apply for analysis

Where `rule_switches` can be any combination of the following:

Switch	Explanation
-from=filename	read rule options from `filename`
+R<rule_id>[:param]	turn ON a given rule [with given parameter]
-R<rule_id>	turn OFF a given rule
-R<rule_id>:param	turn OFF some of the checks for a given rule, depending on the specified parameter

Command Line Example Run

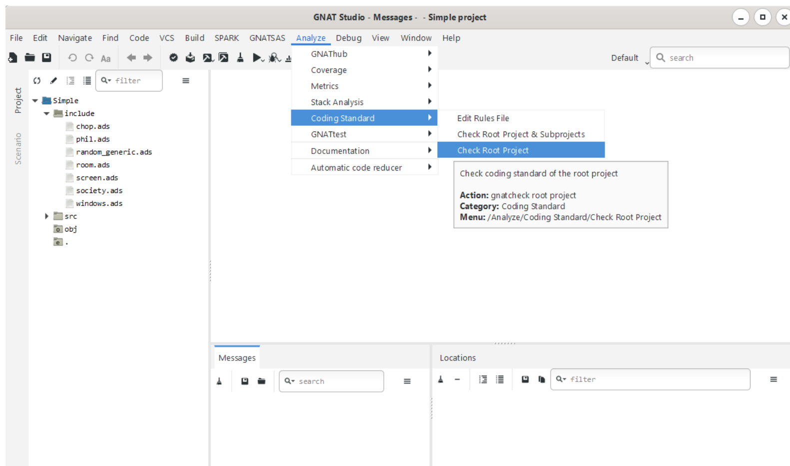
```
gnatcheck -P simple.gpr -rules -from=coding_standard.rules
```

```
chop.adb:14:11: PIck_Up does not have casing specified (mixed)
chop.ads:11:18: Stick does not start with subtype prefix T_
phil.adb:21:11: Think_Times does not start with subtype prefix T_
phil.adb:33:05: "Who_Am_I" is not modified, could be declared constan
phil.ads:12:03: violation of restriction "No_Tasking"
phil.ads:12:13: Philosopher does not start with subtype prefix T_
phil.ads:12:26: My_ID does not have casing specified (mixed)
phil.ads:19:08: States does not end with type suffix _Type
phil.ads:19:08: States does not start with subtype prefix T_
random_generic.ads:5:08: Result_Subtype does not end with type suffix
random_generic.ads:5:08: Result_Subtype does not start with subtype p
room.adb:19:03: violation of restriction "No_Tasking"
room.adb:19:23: anonymous subtype
...
```

These messages are coming from rules specified in `coding_standard.rules`

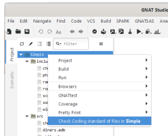
GNATCHECK From GNAT STUDIO Main Menu

Analyze → **Coding Standard** → **Check Root Project**

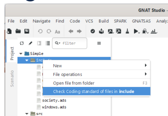


GNATCHECK From GNAT STUDIO Right-Click

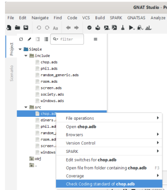
- Right-click on project in Project pane



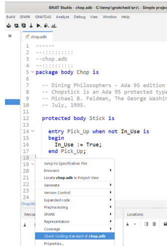
- Right-click on folder in Project pane



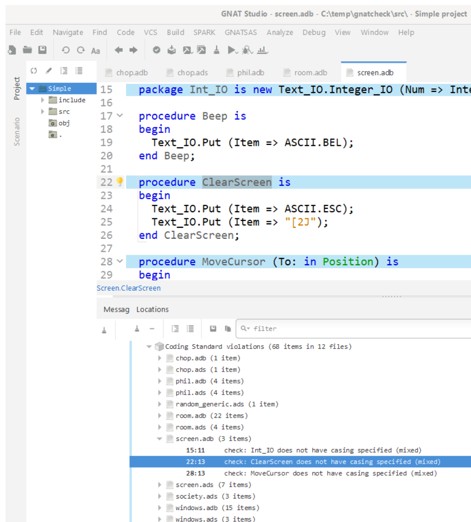
Right-click on file in Project pane



Right-click in source file editor window



GUI Example Run

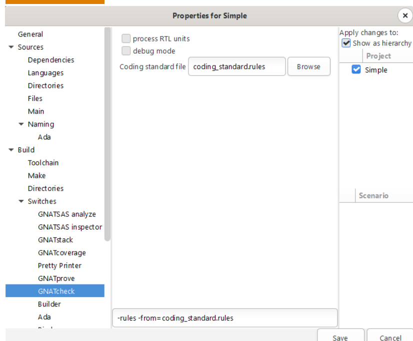


Specifying Rules File

- Rules file can be specified on command line
  - `gnatcheck -rules -from=coding_standard.rules ...`
  - But more commonly defined in project file

```
project Simple is
   for Source_Dirs use ("./include", "./src");
   for Main use ("diners");
   for Object_Dir use "./obj";
   package Check is
      for Default_Switches ("ada") use
         ("-rules", "-from=coding_standard.rules");
   end Check;
end Simple;
```

Edit → Project Properties → Switches → GNATcheck



Lab

GNATcheck Getting Started Lab

- Copy the `getting_started` folder from the course materials location
- Contents of the folder:
  - `simple.gpr` - project file
  - `include` - source directory
  - `src` - source directory
  - `coding_standard.rules` - GNATCHECK rules to apply during analysis

Preparing the Command Line

1. Open a command prompt window and navigate to the `getting_started` folder
2. Type `gnatcheck` and press Enter to verify tool is on your path
   - If you see `gnatcheck:  No existing file to process`, you can go to the next step
   - If you see something like `'gnatcheck' is not recognized as an internal or external command`

     Add the appropriate folder to your path (On Windows, typically `C:\GNATSAS\<version>\bin` where version is the GNAT SAS version number)

     ```
     set PATH=C:\GNATSAS\24.0\bin;%PATH%
     ```
3. Type `gnatcheck -h` and press Enter to show list of predefined rules
   - Examine the output to see what kinds of rules are available
   - The keyword at the end (*Easy*, *Medium*, *Major*) indicates the difficulty in remediating the issue

Running From the Command Line

1. Perform gnatcheck on a single file in the `src` folder

   ```
   gnatcheck src\room.adb -rules -from=coding_standard.rules
   ```

   Examine the output to see what parts of the code failed analysis

2. Add the switch to indicate which rule caused the message

   ```
   gnatcheck src\room.adb --show-rule -rules -from=coding_standard.rules
   ```

   Note the actual rule now appears at the end of the message

Preparing the GUI

1. Use GNAT Studio to open the project `simple.gpr`
2. Set the coding standards for the project to `coding_standard.rules`

   Edit -> Project Properties -> Switches -> GNATcheck

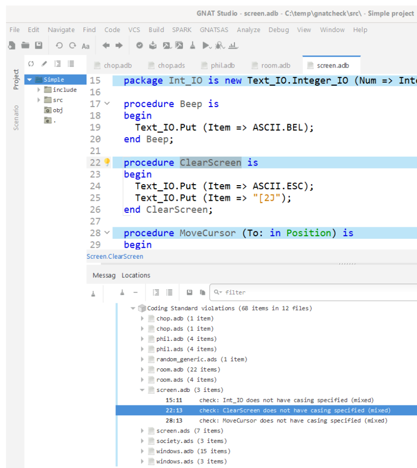
Running From the GUI

1. Perform Coding Analysis on the project

   Analyze ->
   Coding Standard ->
   Check Root Project

2. Double-click on any source line in the **Locations** window to go to the problematic code

   Try fixing the problem and re-running the analysis



Summary

Closing Remarks

- GNATCHECK is a coding standards checker
  - *Rules* are how the tool decides what is the "standard"
- Rules are broken down into two categories:
  - Predefined rules - over 200 rules built into the tool
  - User-defined rules - ability for user to write their own rules

Predefined Rules

Introduction

Accessing Predefined Rules

- Over 200 predefined rules within GNATCHECK
  - Can be found via command `gnatcheck -h`
- Rules have been developed over many years for many uses
  - May have very specialized use cases
  - Some rules may contradict other rules
- Rules can be specified on the command line or via a file
  - Rule on the command line:

    ```
    gnatcheck screen.adb -rules +Runnamed_exits
    ```

    Apply **unnamed_exits** rule (*unnamed exit statement*) in analysis of file `screen.adb`
  - Rules file:

    ```
    gnatcheck screen.adb -rules -from=coding_standard.rules -ROTHERS_In_Aggregates
    ```

    Apply rules from `coding_standard.rules` except for rule **OTHERS_In_Aggregates**

Rules with Parameters

- Some rules have parameters

  `Too_Many_Primitives`

  Flag any tagged type declaration that has more than N user-defined primitive operations

- To specify a parameter, the value comes immediately after the rule separated only by a colon (:)
  - Incorrect

    ```
    gnatcheck screen.adb -rules +RToo_Many_Primitives
    ```

    ```
    gnatcheck: (too_many_primitives) parameter is
    required for +R
    ```

  - Correct

    ```
    gnatcheck screen.adb -rules +RToo_Many_Primitives:3
    ```

Note: Some parameters are optional

Predefined Rules Categories

Style-Related Rules

Tasking Example

`Volatile_Objects_Without_Address_Clauses`
: *Flag each volatile object without an address specification*

Object Orientation Example

`Visible_Components`
: *Flag type declarations located in visible part of a library package or a library generic package that can declare visible component*

Portability Example

`Forbidden_Pragmas`
: *Flag each use of the specified pragmas*

Program Structure Example

`Local_Packages`
: *Flag local packages declared in package and generic package spec*

Programming Practice Example

`Anonymous_Array`
: *Flag all anonymous array type definitions*

Readability Example

`Style_Checks`
: *Flags violations of the source code presentation and formatting rules according to the rule parameter(s) specified*

Feature Usage Rules

Examples

`Abort_Statements`
: *Flag abort statements*

`Numeric_Literals`
: *Flag each use of a numeric literal except for those matching certain requirements*

Metrics-Related Rules

Examples

`Metrics_Cyclomatic_Complexity`
: *Flag program units whose executable body exceeds the specified limit*

`Metrics_LSLOC`
: *Flag program units that exceed the specified limit*

SPARK Rules

Examples

`Overloaded_Operators`
Flag each function declaration that overloads an operator symbol

`Slices`
Flag all uses of array slicing

Writing Your Own Rules

Introduction

Libadalang

- Libadalang (LAL) - library for parsing and semantic analysis of Ada code
    - Meant as building block for integration into other tools (IDE, static analyzers, etc.)
- Provides mainly
    - Complete syntactic analysis with error recovery
        - Precise syntax tree when source is correct, OR
        - Best effort tree when the source is incorrect
    - Semantic queries on top of the syntactic tree such as
        - Resolution of references (what a reference corresponds to)
        - Resolution of types (what is the type of an expression)
        - General cross references queries (find all references to this entity)

LangKit Query Language

- LKQL (LangKit Query Language) - query language enabling users to run queries on top of source code
  - Based on langkit technology
  - Currently hardwired for Ada (and LAL)
- Purely functional, high level, dynamically typed language with general purpose and tree query subsets
- Designed to be simple and concise
- Has a reference manual

Having GNATCHECK rules expressed with a high level interpreted language such as LKQL allows users to write their own rules and test them quickly

LKQL

LKQL Features (General Purpose Subset)

- You can easily define a function in LKQL
  - All functions are first class citizens

```
fun add(x, y) = x * y
fun sub(x, y) = x - y
fun apply(f, x, y) = f(x, y)

print(apply(add, 40, 2))
print(apply((x, y) => x * y), 40, 2)
```

  - You can also define anonymous functions
- LKQL supports list comprehensions with the same syntax as Python

```
val odds = [num for num in [1, 2, 3, 4, 5] if is_odd(num)]
val ids = [node for node in nodes if node is Identifier]
```

- You can use LKQL block expressions to declare local values and add some sequentiality

```
val complex = {
   val part = 40;
   val other_part = 2;
   print("LOGGING");
   part * other_part
}
```

LKQL Features (Query Subset)

- LKQL allows you to write queries to fetch all nodes which satisfy a given pattern
  - LKQL also provides selector operations (e.g. **any children**)

```
val ids = from nodes select Identifier
val if_id_child = select IfStmt(any children is Identifier)
```

- You can define a selector to express a tree traversal logic and use it later as a function or in a pattern
  - This will yield every child but will not recurse for the **if statement** children:

    ```
    selector children_until_if
    | IfStmt  => this
    | AdaNode => rec *this.children
    | *       => ()
    ```

- LKQL patterns use the LAL API to express any filtering logic in a simple and expressive way
  - For more information see the LKQL reference manual

```
val test = select b@BinOp(f_op is OpEq)
           when b.f_left.text == "0" and
                b.f_right is Identifier
```

LKQL API References

- LKQL API can be found at https://docs.adacore.com/live/wave/lkql/html/gnatcheck_rm/gnatcheck_rm/lkql_language_reference.html#lkql-api
- Contains sections on
  - Libadalang API (found at https://docs.adacore.com/live/wave/libadalang/html/libadalang_ug/python_api_ref.html)
    - Includes definitions of all functions like `IfStmt` that we saw above
  - Standard library
    - Includes definitions of typical functions like `print` and `children`

Testing LKQL with Its REPL

- LKQL has an interactive REPL *(Read-Eval-Print-Loop)*
  - Test your ideas and explore available properties and node kinds with auto-completion
- Start the LKQL REPL on a project named **example.gpr** by running the Python script `lkql_repl.py`

```
lkql_repl.py -P example.gpr
```

- Then you can run any LKQL expression or declaration and immediately see the result

```
> select AdaNode  # Get the list of all Ada nodes in your project
[...]
> val ids = select Identifiers  # Assign "ids" value
()
> fun test(nodes) = [n for n in nodes if n.text = "Hello"]  # Define a function
()
> test(ids)  # Call previously defined function with the previously assigned value
[...]
```

Mapping Python API to LKQL API

- Can also refer to Libadalang Python API Reference
- For example, we can find in the Python API documentation:

  class **libadalang.Expr**:
    subclass of AdaNode
    Base class for expressions

  ...

    property **p_expression_type**:
      Return the declaration corresponding to the type
      of this expression after name resolution.

- Thus we know that LKQL has a **Expr** node kind and we can call the **p_expression_type** on this kind of node
  - So we can do

```
val expr_types = [node.p_expression_type() for node in select Expr]
```

In the future LKQL will have its own LAL API documentation.

Integrating LKQL in GNATcheck

- GNATCHECK embeds an LKQL engine to execute rules semantics
- All GNATCHECK rules are expressed using LKQL
- You can make a custom rule written in `my_rules/custom_rule.lkql` available to GNATCHECK with a command line option

```
--rules-dir=my_rules
```

  - Option will trigger the loading of all `.lkql` files in the provided directory
  - Makes their associated rules available

- Example of a GNATCHECK call to load rules inside the `my_rules` folder and apply the **custom_rule** rule

```
gnatcheck -P prj.gpr --rules-dir=my_rules/ -rules +Rcustom_rule
```

Rules

Boolean Rules

- Defined by function which takes a *node* as first parameter
- Returns a *boolean* indicating if given node should be flagged by GNATCHECK
- Called on every node of LAL AST
- To define custom *boolean* rule
  - Create an LKQL function annotated with **@check**
  - Function name should be same as LKQL file name
  - Custom boolean rule which flags every **BodyNode** in Ada sources
    - Function should be in `bodies.lkql`

```
@check
fun bodies(node) = node is BodyNode
```

Example of Boolean Rules

- Flag every `goto` and `if` statemnt

```
@check
fun goto_and_if(node) =
   match node
   | GotoStmt => true
   | IfStmt   => true
   | *        => false
```

- Flag every **Identifier** called dummy (case-insensitive)

```
@check
fun dummy_id(node) =
   node is id@Identifier
   when id.p_name_is("dummy")
```

- Flags every *Binary Operator* with any child a *Numeric Literal*

```
@check
fun op_with_num(node) =
   node is BinOp(any children
                 is NumLiteral)
```

```
1  procedure Test is
2     My_Int : Integer := 10 * 5;
3     Dummy  : String  := "Hello World!";
4  begin
5     if My_Int = 15 then
6        Put_Line (Dummy);
7     else
8        Goto label;
9     end if;
10    <<label>>
11 end Test;
```

Running GNATcheck with these rules on this Ada source will produce:

```
test.adb:02:24: op_with_num
test.adb:03:04: dummy_id
test.adb:05:04: goto_and_if
test.adb:05:07: op_with_num
test.adb:06:17: dummy_id
test.adb:08:07: goto_and_if
```

Unit Rules

- Defined by function which takes an *analysis unit* as its first parameter
- Return list of LKQL objects containing message and location
- Called on every LAL analysis unit
- Meant to be more flexible than boolean rules
  - Fulfill needs that the latter cannot express
  - Example: emitting multiple messages for the same node
- To create custom *unit* rule
  - Create an LKQL function annotated with **@unit_check**
  - Function name should be the same as the LKQL file name (same as **@check**)

Example of Unit Rules

Flag every **goto** *statement and give target label line in associated message*

```
@unit_check
fun goto_line(unit) = [
   {message: "go to line " &
             img(node.f_label_name
                     .p_referenced_decl()
                     .token_start()
                     .start_line),
    loc: node}
   for node in (from unit.root select GotoStmt)
]
```

```
1  procedure Test is
2  begin
3     <<start>>
4     goto label;
5     <<label>>
6
7     goto start;
8  end Test;
```

Running GNATCHECK *with this rule will produce:*

```
test.adb:04:04: go to line 5 [goto_line]
test.adb:07:04: go to line 3 [goto_line]
```

Rule Arguments

You configure an LKQL rule behavior with annotation arguments

message	Message of the rule *(boolean rules only)*
help	Help message for the rule usage
follow_generic_instantiations	Whether to follow generic instantiations in Ada sources *(boolean rules only)*
category / subcategory	Category and subcategory of a rule
remediation	Mediation complexity for technical debt computation

Example of an LKQL rule with rule arguments

```
@check(
   message: "There is a body node",
   help: "This rule flags all body nodes",
   follow_generic_instantiations: false,
   remediation: "EASY"
)
fun bodies(node) = node is BodyNode
```

Rule Function Parameters

- LKQL rule (*boolean* or *unit*) is defined by a function
- Rule function can have more than one parameter
  - Allows GNATCHECK rule arguments being forwarded
- Rule function parameter must have a default value
  - In case none is provided
- You can configure the rule below with the **threshold** argument when running it with GNATCHECK:
  - Flag all *Identifier* nodes with too many characters according to given threshold

```
@check
fun too_long_id(node, threshold=15) =
   node is Identifier
   when node.text.length >= threshold
```

- Flag all *Identifier* nodes with more than 42 characters using **too_long_id** rule

```
gnatcheck -P prj.gpr --rules-dir=.  -rules +Rtoo_long_id:42
```

Configuring a GNATcheck Run with LKQL

- You can configure GNATCHECK run with an LKQL file
  - Chooses rules you want to run (with arguments)
  - Possible alias
  - Whether to run them on Ada code, SPARK code or both
- Example LKQL configuration file

```
val rules = @{
   identifier_suffixes: [
      {access_suffix: "_PTR",
       type_suffix: "_T",
       constant_suffix: "_C",
       interrupt_suffix: "_Hdl"},
      {access_suffix: "_A",
       alias_name: "other_convention"}
   ]
}
val ada_rules = @{ goto_statements }
val spark_rules = @{ recursive_subprograms }
```

- Example GNATCHECK call configured via `config.lkql`

```
gnatcheck -P prj.gpr -rules -from-lkql=config.lkql
```

Lab

GNATcheck LKQL Lab

This lab is a hands-on walkthrough of creating your own LKQL rule for use with GNATCHECK. You can use any text editor to create this rule file.

We want to create a rule that will flag all `integer` types that could be replaced by an enumeration type. To flag those type declarations we must define a criteria list:

- No use of any arithmetic or bitwise operator on the type
- No type conversion from or to the type
- No subtype definition
- No type derivation
- No reference to the type in generic instantiations

We're going to see how to express those criteria using LKQL.

Source Code Specification

```
with Ada.Text_IO;
package Test_Pkg is

   type Good_Candidate is range 0 .. 100;
   function Supplier1 (X : Good_Candidate) return Good_Candidate;

   type Operator_T is range 0 .. 100;
   function Supplier2 (X : Operator_T) return Operator_T;

   type Conversion_Tgt_T is range 0 .. 100;
   function Supplier3 (X : Integer) return Conversion_Tgt_T;

   type Conversion_Source_T is range 0 .. 100;
   function Supplier4 (X : Conversion_Source_T) return Integer;

   type Subtype_Parent_T is range 0 .. 100;
   subtype Subtype_T is Subtype_Parent_T range 1 .. Subtype_Parent_T'Last;
   function Supplier5 (X : Subtype_T) return Subtype_T;

   type Derived_Parent_T is range 0 .. 100;
   type Derived_T is new Derived_Parent_T range 1 .. Derived_Parent_T'Last;
   function Supplier6 (X : Derived_T) return Derived_T;

   type Generic_Instantiaion_T is range 0 .. 100;
   package IO is new Ada.Text_IO.Integer_IO (Generic_Instantiaion_T);

end Test_Pkg;
```

Source Code Body

```
package body Test_Pkg is

   function Supplier1 (X : Good_Candidate) return Good_Candidate is
     (X);

   function Supplier2 (X : Operator_T) return Operator_T is
     (X + 1);

   function Supplier3 (X : Integer) return Conversion_Tgt_T is
     (Conversion_Tgt_T (X));

   function Supplier4 (X : Conversion_Source_T) return Integer is
     (Integer (X));

   function Supplier5 (X : Subtype_T) return Subtype_T is
     (X);

   function Supplier6 (X : Derived_T) return Derived_T is
     (X);

end Test_Pkg;
```

Step 1 - Flag All Integers

1. Create rule **enum_for_integer**
   a. In file `enum_for_integer.lkql`
   b. Use **@check** annotation
2. Flag all integers
   a. Look for **p_is_int_type** LAL property using a *node kind* pattern

```
@check
fun enum_for_integer(node) = node is TypeDecl(p_is_int_type() is true)
```

3. Test it out - see what happens when you run the rule:

```
gnatcheck -P prj.gpr --rules-dir=. -rules +Renum_for_integer
```

- This gives us the output:

```
test_pkg.ads:4:09: enum_for_integer
test_pkg.ads:9:09: enum_for_integer
test_pkg.ads:14:09: enum_for_integer
test_pkg.ads:19:09: enum_for_integer
test_pkg.ads:24:09: enum_for_integer
test_pkg.ads:30:09: enum_for_integer
test_pkg.ads:36:09: enum_for_integer
```

- All integer types are reported - we need to add filters

Step 2 - Improve Message

- Default message for boolean rules is just the name of the rule:

```
test_pkg.ads:4:09: enum_for_integer
```

- To improve message, add **message** attribute to **@check** token

```
@check(message="Integer type could be replaced by an enumeration")
fun enum_for_integer(node) =
   node is TypeDecl(p_is_int_type() is true)
```

- Gives much more information:

```
test_pkg.ads:4:09: Integer type could be replaced by an enumeration
test_pkg.ads:9:09: Integer type could be replaced by an enumeration
test_pkg.ads:14:09: Integer type could be replaced by an enumeration
test_pkg.ads:19:09: Integer type could be replaced by an enumeration
test_pkg.ads:24:09: Integer type could be replaced by an enumeration
test_pkg.ads:30:09: Integer type could be replaced by an enumeration
test_pkg.ads:31:09: Integer type could be replaced by an enumeration
test_pkg.ads:36:09: Integer type could be replaced by an enumeration
```

Step 3 - Implement First Criteria

1. Implement the first criteria: **No use of any arithmetic or bitwise operator on the type**.
   a. Need to fetch all operators - use global **select** with **BinOp** and **UnOp** node kind patterns. (Field **f_op** contains the kind of the operator.)

   ```
   select BinOp(f_op is OpDiv or OpMinus or OpMod or OpMult or OpPlus or
                        OpPow or OpRem or OpXor or OpAnd or OpOr) or
          UnOp(f_op is OpAbs or OpMinus or OpPlus or OpNot)
   ```

   b. **select** returns list of **BinOp** and **UnOp**
      - Both inherit from the **Expr** node - so we use **p_expression_type** property to retrieve **TypeDecl** node associated with expression's actual type.
2. Implement function named **arithmetic_ops** to return the list of **TypeDecl** used in arithmetic and logical operations

```
fun arithmetic_ops() =
   [op.p_expression_type()
    for op in select
       BinOp(f_op is OpDiv or OpMinus or OpMod or OpMult or OpPlus or
                     OpPow or OpRem or OpXor or OpAnd or OpOr) or
       UnOp(f_op is OpAbs or OpMinus or OpPlus or OpNot)].to_list
```

Step 4 - Use First Criteria in Rule

1. Update **enum_for_integer** function to filter integer type declarations by excluding all **TypeDecl** used in operators

```
@check
fun enum_for_integer(node) =
   node is TypeDecl(p_is_int_type() is true)
   when not [t for t in arithmetic_ops() if t == node]
```

2. Test it out - see what happens when you run the rule:

```
gnatcheck -P prj.gpr --rules-dir=. -rules +Renum_for_integer
```

   - This gives us the output:

```
test_pkg.ads:4:09: Integer type could be replaced by an enumeration
test_pkg.ads:14:09: Integer type could be replaced by an enumeration
test_pkg.ads:19:09: Integer type could be replaced by an enumeration
test_pkg.ads:24:09: Integer type could be replaced by an enumeration
test_pkg.ads:30:09: Integer type could be replaced by an enumeration
test_pkg.ads:31:09: Integer type could be replaced by an enumeration
test_pkg.ads:36:09: Integer type could be replaced by an enumeration
```

Note we are no longer reporting on the type at line 9

Step 5 - Implement Second Criteria

- Criteria: **No type conversion from or to the type**
  - In the LAL tree type conversions appear as **CallExpr** whose referenced declaration is a **TypeDecl**

1. Implement new function **types** to return list of **TypeDecl** used as target type in a conversion

```
fun types() =
    [c.p_referenced_decl()
     for c in select CallExpr(p_referenced_decl() is TypeDecl)].to_list
```

   - **to_list** member is necessary if we want to combine lists later

2. Add our new filtering function in the rule body.

```
@check
fun enum_for_integer(node) =
   node is TypeDecl(p_is_int_type() is true)
   when not [t for t in arithmetic_ops() if t == node] and
        not [t for t in types() if t == node]
```

- This version of **types** only returns **TypeDecl** used as target in conversions - we also want to filter out source of conversions

Step 6 - Improve Types Filter

1. Update the **types** function to also return types used as source type in conversions
   - LAL field **f_suffix**
     - Returns **ParamAssocList** with a single element - source expression
     - Use on type conversion nodes to get source of conversions

```
fun types() =
   concat ([[c.p_referenced_decl(), c.f_suffix[1].f_r_expr.p_expression_type()]
            for c in select CallExpr(p_referenced_decl() is TypeDecl)].to_list)
```

   - **concat** function takes a list of lists and returns the one-dimensional result of concatenation of all lists.

2. Test it out - see what happens when you run the rule:

```
gnatcheck -P prj.gpr --rules-dir=. -rules +Renum_for_integer
```

   - This gives us the output:

```
test_pkg.ads:4:09: Integer type could be replaced by an enumeration
test_pkg.ads:24:09: Integer type could be replaced by an enumeration
test_pkg.ads:30:09: Integer type could be replaced by an enumeration
test_pkg.ads:31:09: Integer type could be replaced by an enumeration
test_pkg.ads:36:09: Integer type could be replaced by an enumeration
```

   - List of integers that meet our criteria is shrinking!

Step 7 - Implement Third Criteria

- Criteria: **No subtype definition**

1. We can use global **select** with list comprehension filtering

```
[s.f_subtype.f_name.p_referenced_decl() for s in select SubtypeDecl]
```

   - Expression gives list of subtype **TypeDecl**. We can now add it to the result of the **types** function.

```
fun types() =
   concat ([[c.p_referenced_decl(), c.f_suffix[1].f_r_expr.p_expression_type()]
           for c in select CallExpr(p_referenced_decl() is TypeDecl)].to_list) &
   [s.f_subtype.f_name.p_referenced_decl() for s in select SubtypeDecl].to_list
```

2. And once again test it out

gnatcheck -P prj.gpr --rules-dir=. -rules +Renum_for_integer

   - This gives us the output:

```
test_pkg.ads:4:09: Integer type could be replaced by an enumeration
test_pkg.ads:30:09: Integer type could be replaced by an enumeration
test_pkg.ads:31:09: Integer type could be replaced by an enumeration
test_pkg.ads:36:09: Integer type could be replaced by an enumeration
```

   - Even fewer integers meet our criteria

Step 8 - Implement Fourth Criteria

- Criteria: **No type derivation**

1. We can implement this similar to the subtype check using

```
[c.f_type_def.f_subtype_indication.f_name.p_referenced_decl()
 for c in select TypeDecl(f_type_def is DerivedTypeDef)].to_list
```

2. Add this expression to the **types** function

```
fun types() =
   concat([[c.p_referenced_decl(), c.f_suffix[1].f_r_expr.p_expression_type()]
          for c in select CallExpr(p_referenced_decl() is TypeDecl)].to_list) &
   [s.f_subtype.f_name.p_referenced_decl() for s in select SubtypeDecl] &
   [c.f_type_def.f_subtype_indication.f_name.p_referenced_decl()
    for c in select TypeDecl(f_type_def is DerivedTypeDef)].to_list
```

Step 9 - Implement Final Criteria

- Criteria: **No reference to the type in generic instantiations**

1. Look in every each generic instantiation for identifiers referring to the type

```
from (select GenericInstantiation) select Identifier
```

   - Gives list of each **Identifier** used in **GenericInstantiation**
   - Use **p_referenced_decl** property we to get associated declaration (that may be a **TypeDecl**

2. Express our query as a function

```
fun instantiations() =
    [id.p_referenced_decl()
     for id in from select GenericInstantiation select Identifier].to_list
```

3. Add to **enum_for_integer** function to finalize filtering

```
@check
fun enum_for_integer(node) =
   node is TypeDecl(p_is_int_type() is true)
   when not [t for t in arithmetic_ops() if t == node] and
        not [t for t in types() if t == node] and
        not [t for t in instantiations() if t == node]
```

Complete Rules File

Here is the final view of our `enum_for_integer.lkql` file.

```
fun arithmetic_ops() =
   [op.p_expression_type()
    for op in select
       BinOp(f_op is OpDiv or OpMinus or OpMod or OpMult or OpPlus or
                      OpPow or OpRem or OpXor or OpAnd or OpOr) or
       UnOp(f_op is OpAbs or OpMinus or OpPlus or OpNot)].to_list

fun instantiations() =
    [id.p_referenced_decl()
            for id in from select GenericInstantiation select Identifier].to_list

fun types() =
    concat ([[c.p_referenced_decl(), c.f_suffix[1].f_r_expr.p_expression_type()]
            for c in select CallExpr(p_referenced_decl() is TypeDecl)].to_list) &
            [s.f_subtype.f_name.p_referenced_decl() for s in select SubtypeDecl].to_list &
            [c.f_type_def.f_subtype_indication.f_name.p_referenced_decl()
             for c in select TypeDecl(f_type_def is DerivedTypeDef)].to_list

@check(message="Integer type could be replaced by an enumeration")
fun enum_for_integer(node) =
   node is TypeDecl(p_is_int_type() is true)
   when not [t for t in arithmetic_ops() if t == node]
    and not [t for t in types() if t == node]
    and not [t for t in instantiations() if t == node]
```

Final Result

- One more run to get the "correct" result

  gnatcheck -P prj.gpr --rules-dir=. -rules +Renum_for_integer

- This gives us the output

```
test_pkg.ads:4:09: Integer type could be replaced by an enumeration
test_pkg.ads:31:09: Integer type could be replaced by an enumeration
```

Improving the Behavior Part 1

- Speed of the rule as written is slow
  - Repeated calls to global **select** query in **arithmentic_ops**, **types**, **instantiations**
- Query functions can be instructed to cached their results
  - Use **@memoized** attribute

```
@memoized
fun arithmetic_ops() =
   [op.p_expression_type()
    for op in select
       BinOp(f_op is OpDiv or OpMinus or OpMod or OpMult or OpPlus or
                     OpPow or OpRem or OpXor or OpAnd or OpOr) or
       UnOp(f_op is OpAbs or OpMinus or OpPlus or OpNot)].to_list
```

Improving the Behavior Part 2

- Take advantage of conditional short circuiting
    - Typically more arithmentic/logical operations than conversions, subtypes, instantiations
    - Swap filtering order to check for those last

```
@check(message="integer type may be replaced by an enumeration")
fun enum_for_integer(node) =
   node is TypeDecl(p_is_int_type() is true)
   when not [t for t in types() if t == node] and
        not [t for t in instantiations() if t == node] and
        not [t for t in arithmetic_ops() if t == node]
```

Summary

Future Evolutions of LKQL

- Adding a custom LAL API documentation for LKQL (for now user can rely on the LAL Python API documentation)
- Support of the LAL rewriting API to express code transformation
- Adding a static type system to improve performance and debugging processes
- Making LKQL available for all Langkit defined languages

GNAT Static Analysis Suite (GNAT SAS)

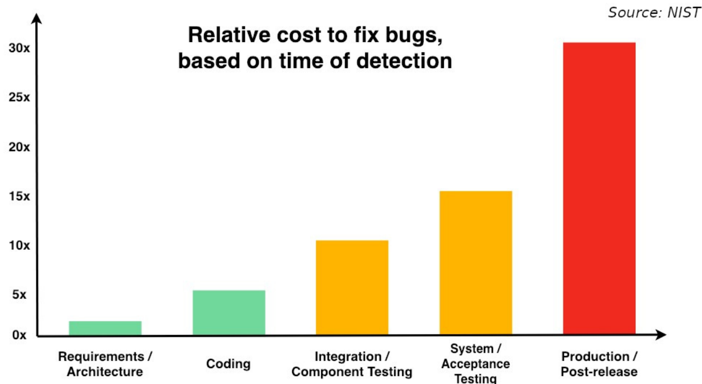
Advanced Static Analysis

What Is Static Analysis?

- **Symbolic** interpretation of **source code**
  - Find what could go wrong
  - No execution
- **Formally** verifying **high level** or **abstract** properties
  - Strong guarantees
- *May* be exhaustive
  - All possible errors are reported
  - No false negatives; there may be false positives
  - If the analyzer does not report a problem, there is no problem

Why Static Analysis Saves Money

- Costs shift
  - From later, **expensive** phases
  - To earlier, **cheaper** phases



Why Use GNAT SAS?

- Efficient automated code reviewer
    - Identifies run-time errors with a **level of certainty**
        - E.g. buffer overflows, division by zero
    - Flags legal but **suspect** code
        - Typically logic errors
- Detailed subprograms analysis
- Can analyze existing code bases
    - Detect and remove **latent bugs**
    - Legacy code
    - Code from external sources

Detailed Subprogram Analysis

- **Explicit** specification
  - Written in the code
  - Types
  - Contracts
  - Assertions
  - etc...
- **Implicit** specification
  - Assumptions by GNAT SAS
  - *Deduced preconditions*

GNAT SAS Overview

What Is GNAT SAS?

GNAT SAS in a Nutshell (1/2)

- GNAT SAS is a static analysis tool
  - Provides feedback **before** execution and test
  - Provides *as-built documentation* for code reviews
- Helps identify and eliminate **vulnerabilities and bugs** early
- Modular
  - Analyze entire project or a single file
  - Configure for speed or depth
- Review of analysis report
  - Filtering messages by category, severity, package...
  - Comparative analysis between runs
  - Maintain historical comments

GNAT SAS in a Nutshell (2/2)

- Large Ada support
  - Usable with Ada83 through Ada2022
  - Compiler agnostic
    - Supports GNAT, Apex, GHS, ObjectAda, VADS
- Bundled with a Coding Standards Checker and a Metrics Calculation Tool
  - GNATCHECK and GNATMETRIC
- Detects runtime and logic errors
  - Initialization errors, run-time errors and assertion failures
  - Race condition errors: unprotected access to globals
- Warns on dead or suspicious code

GNAT SAS Integration

- Output: textual, XML, CSV, HTML, SARIF, CodeClimate
- Integrated with GPRBUILD
  - Tool configuration can be source controlled
- Scriptable command-line tool for easy deployment in CI/CD technologies (e.g. GitLab, Jenkins)
- Interactive use in GNAT STUDIO
- Integration with SONARQUBE (continuous inspection of code quality)

Integrated Analysis Engines

- Inspector
  - Excels in detecting possibly failing run-time checks as well as wide range of logical errors
  - Determines preconditions on the inputs necessary to preclude run-time failures
  - Makes presumptions about return values of external subprograms
  - Identifies postconditions that characterize the range of outputs
- Infer
  - https://fbinfer.com/
  - Specialized to Ada by AdaCore
  - Fast analysis with low false positive rate
  - Especially good in detecting problems occurring for certain execution paths, such as null-pointer dereferences or memory leaks
- GNAT Warnings
  - Provides warning issued by GNAT compiler frontend
  - Detects things like suspicious constructs and warnings when the compiler is sure an exception will be raised at run-time
- GNATcheck
  - Tool used to check for suspicious code constructs and compliance with specified coding standard rules
  - Fully integrated with GNAT SAS

Typical Users and Use Cases

- Developers, during code-writing
  - **Fix** (local) problems before integration
- Reviewers
  - **Annotate** code with analysis of potential problems
  - **Analyse** specific CWE issues
- Project managers and quality engineers
  - **Track** reported vulnerabilities regularly
  - **Identify** new issues quickly
- Software auditors
  - **Identify** overall vulnerabilities or hot spots
  - **Verify** compliance to quality standards

Analyzing Code

Running GNAT SAS

Running the Analysis

When running the analysis, the tool maintains data files to store messages, including history and user reviews

SAM file — *Static Analysis Messages* file containing messages generated by analysis engines

SAR file — *Static Analysis Review* file containing user-specified message reviews

Running From the Command Line

- This command only performs the analysis (typically used for simple testing or automation)

```
gnatsas analyze -Psdc
```

- To view results of analysis, you need to generate a report

```
gnatsas report -Psdc
```

  - More information in next section

Running From the GUI

- From GNAT Studio

  GNATSAS → Analyze All

- Report automatically displayed based on user-specified filters
  - *Message Ranking* controls level of messages displayed
  - Other filters control types of messages and categories

GNATSAS report

Base run: sdc.deep.baseline.sam
2024-01-08 13:08:21

Current run: sdc.deep.sam
2024-01-08 13:08:21

Messages Race conditions

Entity	High	Med	Low
RTL and removed			
Sdc		6	24
tokens.adb		2	2
values.adb		2	1
stack.adb		2	
sdc.adb			15
input.adb			5
values-operations.adb			1
Total:		6	24

Warning categories: module analyzed, suspicious precondition, unused out parameter

Check categories: access check, array index check, conditional raise, divide by zero, overflow check, precondition

CWE categories: CWE-120, CWE-190, CWE-369, CWE-457, CWE-476, CWE-563

Message history: added, unchanged, removed

Message ranking: informational, low, medium, high

Message review status: Uncategorized, Pending, Not a bug, False positive, Intentional, Bug

Report with **Low, Medium,** *and* **High** *ranking messages*

Analysis

Analysis Modes

- Deals solely with *Inspector* engine
- *Fast mode*
  - Analyze each library unit separately
  - Allows for *incremental* analysis
    - Only units that change will be re-inspected
- *Deep mode*
  - Analyzes groups of unit
    - Partitioning options to determine size of group
  - Analysis always starts from scratch
- Each mode has its own *baseline*

Timelines

- Default: Separate baselines for comparing *deep* or *fast* runs
- Custom timelines available
  - `timeline <name>` switch to create custom baseline
  - First execution becomes baseline for that name
  - Allows creating specialized timelines based on switches
    - Such as `no-subprojects` which might drastically change number of messages

Settings

Analysis Settings

- Filters can remove uninteresting messages
  - e.g. `show` to control messages to be displayed
- Skip problematic source files
  - *Excluded_Source_Files* project attribute
  - `pragma Annotate (GNATSAS, Skip_Analysis);` embedded in code

Performance Settings

- Simplistic methods
  - Disable specific analysis engine(s)
  - `-j0` jobs switch
  - High-performance machines (multiple cores, etc)
- Identifying problematic units
  - For *Inspector*, examine output for units taking a long time

    ```
    analyzed main.scil in 0.05 seconds
    analyzed main__body.scil in 620.31 seconds ←
    analyzed pack1__body.scil in 20.02 seconds
    analyzed pack2__body.scil in 5.13 seconds
    ```
  - For *Infer*, use progress bar to see where the process is slow

    ```
    -Q --progress-bar-style multiline
    ```

Viewing Results

Report Command

Generating a Report

- To view results, you must generate a **report**
  - From the command line

```
gnatsas report -Psdc.gpr
```

Default output format (text), written to standard output

```
gnatsas report csv -Psdc.gpr --out report.csv
```

Generate a comma-separated values file, save in `report.csv`

Available Output Formats

text Compiler-like listing of messages

html HTML output generated by GNATHUB. Output always stored in `index.html` in `gnathub/html-report` subfolder of object directory

csv Comma-separated values, useful for input into third-party tools like spreadsheets

security HTML report focusing on certain vulnerabilities

code-climate JSON output useful with tools such as BitBucket and Gitlab

sarif Output for integration with any SARIF viewer tool

exit-code Number of messages (up to 255) will be returned as the report exit code. Useful for automation processes

For more information, refer to *GNAT SAS User's Guide Section 5.4 - Report Formats in Detail*

Selecting Results to Display

- `gnatsas report` always displays results of last run (regardless of run's switches)
- To generate report for other runs
  - Specify a timeline

    ```
    gnatsas report text -P sdc.gpr --timeline <timeline>
    ```
  - Specify a SAM file

    ```
    gnatsas report text <sam-file>
    ```

Message Kinds

- Message kinds fall into one of the following categories
  - Warning - compilation warnings issued by GNAT front end
  - Check - possible run-time check failures
  - Informational - extra information about a message
  - Race Condition - messages about synchronization objects
  - Annotation - Information about a subprogram determined by analysis
- Each of these categories has multiple messages
  - GNAT SAS reporting can call out message kinds by category or individual kind

See section *10 GNAT SAS Messages Reference* of the documentation for more detailed information

Message Categories

Messages can be grouped by *category*. These categories can be used to determine which messages are displayed in the report.

Age	Compared to the previous run, is this message the same, new, or no longer there
Kind	Kind of message (category (e.g. `check`) or kind (e.g. `range_check`)
Rank	Severity - likelihood that message identifies a defect that could lead to incorrect results
Tool	Which analysis engine generated the message
CWE	*Common Weakness Enumeration*
Review Status	Actual status of message review (see section on *Message Review*)
Review Kind	Category of review status (see section on *Message Review*)
Project	Project containing source file with the message
File	Specific file containing message

Filtering Messages by Category

- Use `show` switch to add or remove messages from report

```
gnatsas report --show [category_constraint]*
```

where *category_constraint* can be specified as

`<category>=<constraint>`	Restrict report to messages that match constraint
`<category>+<constraint>`	Add to report messages that match constraint
`<category>-<constraint>`	Remove from report messages that match constraint

Switches for Filtering Messages by Category

```
gnatsas report -P sdc.gpr --show <filter=value>
```

Filter	Value Choices
`default`	Default categories with constraints
`all`	Only specified categories with constraints
`age`	unchanged, added, removed
`kind`	Message kind (category or individual kind)
`rank`	info, low, medium, high
`tool`	inspector, infer, gnatcheck, gnat
`cwe`	Specific CWE or "none"
`review_status`	Any review statuses or "none"
`review_kind`	not_a_bug, pending, bug, uncategorized, none
`prj`	runtime or project base name, or relative paths
`file`	Source filename basename or relative path

Note: **none** matches those messages that do not have corresponding information attached (e.g., no CWE or no review)

Comparing GNAT SAS Runs

Using History Data

- *Baseline run* is first run performed at appropriate mode
  - *fast* and *deep* have different baselines
- Report indicates if message is *new*, *unchanged*, or *removed* relative to baseline
- Can change baseline with `gnatsas baseline` command:
  - `bump-baseline` switch sets last analysis run as a baseline
  - `set-baseline <sam-file>` switch sets specified SAM file to be the baseline
- To compare different runs without updating baseline, use `gnatsas report --compare-with <sam-file>`
  - Current run will be compared to specified run without impacting baseline

Classifying Message Changes

- In determining if message is *unchanged*, *added*, or *removed* even when surrounding source changes, GNAT SAS checks for:
  - Full name of procedure where message was generated
  - Analysis engine that emitted message
  - Kind of message
  - Selected content within the message (depending on kind)
- If all the above matches multiple messages, GNAT SAS uses order of appearance in code

Note: default behavior is to not mention removed messages and to call out specifically new messages

GUI Reports

Viewing Reports Via GNAT Studio

- To view report from within GNAT Studio
  - Perform analyis (GNATSAS → Analyze All)
    - Report appears when analysis completes
  - GNATSAS → Display Code Review
    - Will open report if analysis has ever been done
  - GNATSAS → Advanced → Regenerate Report
    - Brings up dialog for report generation
    - Allows user to specify options such as `compare-with` or `show`

GNAT Studio Analysis Report

- Baseline / Current run SAM file
  - Hover over these filenames gives switches used in run
- Filters
  - Control which messages appear in report table/locations view
- Locations View
  - Click on any message to go to appropriate source line
  - Click on pencil icon to add review/annotion

Reviewing Results and Improving Code

Reviewing Messages

Documenting Review Comments

- GNAT SAS generates many messages
  - Sometimes the code is OK as-is
  - Sometimes we might want to say we'll worry about it later
- Two methods of documenting human response to the message
  - Interactive review via
    - GNAT STUDIO
    - CSV import
  - Code-based review via `pragma Annotate`
- Benefits of interactive review
  - No source code modification
  - Can be performed by non-Ada reviewers
  - Additional review statuses available
- Benefits of code-based review
  - Review appears with source code
  - Review less likely to be affected by other source changes
  - Editing/Source code control can be used to manage review

Review Actions

- Left-click pencil icon in *Locations* window to get review choices

GNAT Studio - tokens.adb - C:\temp\tutorial\struct\ - Sdc project

```
19
20          begin
21             --  Figure out which kind of token we have from the first
22             --  character and delegate the full token recognition to
23             --  the Read routine in the appropriate Instruction, Values
24             --  or Values.Operations package.
25
26             case Word (Word'First) is
27
28                when '0' .. '9' | '.' =>
29                   return Token'(Kind => Val, Val => Values.Read (Word));
30
31                when '+' | '*' | '/' =>
32                   return
33                     Token'(Kind => Op, Op => Values.Operations.Read (Word));
```

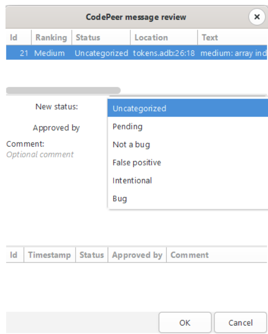
Locations

GNATSAS: messages (30 items in 6 files)
- input.adb (5 items)
- sdc.adb (15 items)
- stack.adb (2 items)
- tokens.adb (4 items)
  - 26:18 medium: array index check [CWE 120] (Inspector): requires (Input.Next_Word'First) <= (Input.Next_Word'Last)

Manual review
Annotate

Manual Review

- **Manual review** brings up dialog to add review comments



- **Annotate** inserts `pragma Annotate` after source code
  - Reviewer updates *<insert review>* text

```
pragma Annotate
  (CodePeer, False_Positive, "array index check", "<insert review>");
```

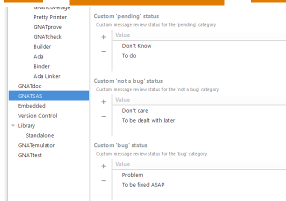
Default Review Statuses

- GNAT SAS groups statuses into three categories
  - **Pending**
  - **Not a bug**
  - **Bug**
  - *By default,* GNAT STUDIO *does not show messages in category* **Not a bug**
- GNAT SAS predefines the following review statuses
  - `Uncategorized`
  - `Pending`
  - `Not a bug`
  - `Bug`
  - `False positive`
  - `Intentional`
  - *Note that* `False positive` *and* `Intentional` *fall into the* **Not a bug** *category*
- For `pragma Annotate`, only `False_Positive` and `Intentional` are allowed

Custom Review Statuses

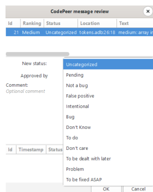
It is possible to create your own statuses for the **Manual review** dialog

Edit → Edit Project Properties → GNATSAS




```
project Sdc is
  package Analyzer is
    for Pending_Status use ("Don't Know",
                            "To do");
    for Not_A_Bug_Status use ("Don't care",
                              "To be dealt with later");
    for Bug_Status use ("Problem",
                        "To be fixed ASAP");
  end Analyzer;
```

Resulting in an updated **Manual review** dialog



Code Annotations Via GNAT Studio

Understanding Code Annotations

- The *Inspector* engine generates documentation for each analyzed subprogram
  - Appears as virtual comments in GNAT STUDIO source editor
  - General reasoning behind analysis that caused message to appear

`Pre`	Requirements subprogram imposes on inputs
`Presumption`	Presumptions about results of external subprogram (when code is unavailable or in separate partition)
`Post`	Behavior of subprogram in terms of outputs
`Unanalyzed`	External subprograms that are unanalyzed (Participate in determination of presumptions)
`Global inputs`	All global objects referenced by subprogram
`Global outputs`	All global objects and components modified by subprogram
`New Objects`	List of heap-allocated objects created but not reclaimed

Annotation Example

GNATSAS report | stack.adb

```
53      ---------
54      -- Pop --
55      ---------
56
        --
        --  Subprogram: stack.pop
        --
        --  Post:
        --    stack.pop'Result = Tab(Last'Old)
        --    stack.pop'Result /= null
        --    Last = Last'Old - 1
        --    Last <= 199
        --
        --  Pre:
        --    V.E'Initialized
        --    Tab(Last) /= null
        --    Last in 1..200
        --
        --  Global_outputs:
        --    Last
        --
        --  Global_inputs:
        --    Last, Tab, Tab(1..200)
        --
        --  Presumption:
        --    'Image'Result@44'Last in 1..1_234
        --    'Image'Result@44'First = 1
        --
57      function Pop return Value is
58         V : Value;
59
60      begin
61         if Empty then
62            raise Underflow;
63         end if;
64
65         V := Tab (Last);
66         Last := Last - 1;
67
68         Screen_Output.Debug_Msg ("Popping <- " & Values.To_String (V));
69
70         return V;
71      end Pop;
72
```

Annotation Syntax Explanations

`--  Post:`	*On completion of the subprogram*
`--    stack.pop'Result = Tab(Last'Old)`	*The return value will be the value in* `Tab` *at the location specified by* `Last` *on entry into the subprogram*
`--    stack.pop'Result /= null`	*The return value will not be* `null`
`--    Last = Last'Old - 1`	`Last` *will be its value on entry minus 1*
`--    Last <= 199`	`Last` *will be less than 200*
`--  Pre:`	*On entry into the subprogram*
`--    V.E'Initialized`	`V.E` *has been initialized*
`--    Tab(Last) /= null`	`Tab(Last)` *is not null*
`--    Last in 1..200`	`Last` *is in range 1 .. 200*
`--  Global_outputs:`	*List of global objects modified*
`--    Last`	
`--  Global_inputs:`	*List of global objects read*
`--    Last, Tab, Tab(1..200)`	
`--  Presumption:`	*Presumptions about* `Image` *call in* `To_String`
`--    'Image'Result@44'Last in 1..1_234`	
`--    'Image'Result@44'First = 1`	

For more information about annotation syntax, refer to Inspector Annotations *chapter in* **GNAT SAS User's Guide**

GNAT SAS Tutorial - Step by Step

Introduction

Getting Started

- This module is a lab-based version of the *GNAT SAS Tutorial* found here
- Copy the `tutorial` folder from the course materials location
- Contents of the tutorial folder:
  - `sdc.gpr` - project file
  - `common` - source directory
  - `struct` - source directory
  - `obj` - object file (and metrics results) directory

Starting GNAT Studio

- From a command prompt, type `gnatsas --help` to verify your path is set correctly
  - If not, add the appropriate `bin` directory to your path
  - Typically (for Windows), this is located in `C:\GNATSAS\<version>\bin`
- Start GNAT Studio and open the `sdc.gpr` project file by one of these methods:
  - From the application library, select GNAT Studio and use File → Open Project to navigate to and open `sdc.gpr`
  - From the command prompt navigate to the `tutorial` directory and enter `gnatstudio sdc.gpr` to open the project
    - You don't actually need `sdc.gpr` - GNAT Studio will automatically open a GPR file if it is the only GPR file in the folder

Running GNAT SAS

First Analysis

Perform a deep static analysis on the project

First Analysis

Perform a deep static analysis on the project

- GNATSAS → Analyze
- Set **Analysis mode** to *deep*
- Press Execute

Filter Messages by Rank

- In the *GNATSAS Report*, note the count of *High*, *Medium*, and *Low* messages
  - In the **Locations** window, note the actual messages displayed

Filter Messages by Rank

- In the *GNATSAS Report*, note the count of *High*, *Medium*, and *Low* messages
    - In the **Locations** window, note the actual messages displayed
- Check/uncheck the *Medium* and *Low* items in **Message ranking**
    - Note the **Locations** window content changes based on which messages are displayed

Check Messages

Finding a Check Message

In the **Locations** window, click on the *medium* message for line 26 of `tokens.adb`

Finding a Check Message

In the **Locations** window, click on the *medium* message for line 26 of `tokens.adb`

- Click the triangle next to `tokens.adb` to show all the messages
- Select the *medium* message for line 26

Note that the file appears and the line is highlighted

Understanding a Check Message

```
17  Read_A_Valid_Token : declare
18     Word : String := Input.Next_Word;
19
20  begin
21     --  Figure out which kind of token we have from the first
22     --  character and delegate the full token recognition to
23     --  the Read routine in the appropriate Instruction, Values
24     --  or Values.Operations package.
25
26     case Word (Word'First) is
```

Message Part	Description
`tokens.adb:26:18`	Source location
`medium`	Message ranking
`array index check [CWE 120] (Inspector)`	Short description of message
`requires (Input.Next_Word'First) <= (Input.Next_Word'Last)`	Explanation / possible remediation

- GNATSAS is warning that line 26 indexes into array* `Word` without ever checking if the array is not empty, possibly raising a `Constraint_Error`
  - So we need to investigate how `Word` is initialized, so we will look at `Input.Next_Word`

Determining Cause of Message

- To investigate the behavior of `Input.Next_Word`, right-click on it and select Go to Body or Full Declaration
  - This brings us to the implementation, including the GNATSAS annotations

```
180  ---------------
181  -- Next_Word --
182  ---------------

     --
     --  Subprogram: input.next_word
     --
     --  Post:
     --    possibly_updated(input.next_word'Result(1..2_147_483_647))
     --    possibly_updated(Line(1..1_024))
     --    input.next_word'Result'Last in 0..1_023
     --    input.next_word'Result'First <= 1_024
     --    Line_Num'Initialized
     --    Last_Char /= 0
     --    First_Char <= 1_024
     --    First_Char - input.next_word'Result'First in 0..1_023
     --

184  function Next_Word return String is
```

Interpreting Annotations

- Our interest here is in the result of the call, so we're looking at the postconditions as determined by GNATSAS

```
--      input.next_word'Result'Last in 0..1_023
--      input.next_word'Result'First <= 1_024
```

- This is indicating that for the result (return value) of `Input.Next_Word`, `'Last` can be 0 to 1023, and `'First` just has to be less than 1024
    - This means the last index can be less than the first index, which, in Ada, is an indication of a 0-length array

Fixing Our Problem

- So we need to add a check in `Tokens.Next` to deal with this issue
  - On line 25, add the following code:

```
if Word = "" then
   declare
      Temp : Token := (Kind => Val,
                       Val  => Values.Read (""));
   begin
      return Temp;
   end;
end if;
```

- Rerun the analysis, and see that the totals changed, and the *check* message is no longer there

Warnings

Potential Logic Errors

- In the **Locations** window, click on the message for line 41 of `stack.adb`

```
stack.adb:41:4: medium warning: suspicious precondition (Inspector): precondition for Last
does not have a contiguous range of values
```

```
--  Subprogram: stack.push
--
--  Post:
--    Tab(1..198 | 200) = One-of{V, Tab(1..198 | 200)'Old}
--    Last in (1..198 | 200)
--    Last = Last'Old - 1
--
--  Pre:
--    V.E'Initialized
--    V /= null
--    Last in (2..199 | 201)
--
--  Global_outputs:
--    Last, Tab(1..198 | 200)
```

- The non-contiguous values on line 4, 5, 11, and 14 indicate a possible issue

Determining Cause of Message

- Precondition of `-- Last in (2..199 | 201)` indicates that 199 and 201 are legal, but 200 is not
  - 200 is an interesting number - it happens to be the length of `Tab`
  - What happens in the code when Last is 199, 200, or 201?

```
41 procedure Push (V : Value) is
42 begin
43    if Last = Tab'Last then
44       raise Overflow;
45    end if;
46
47    Screen_Output.Debug_Msg ("Pushing -> " & Values.To_String (V));
48
49    Last := Last - 1;
50    Tab (Last) := V;
51 end Push;
```

Determining Cause of Message

- Precondition of `-- Last in (2..199 | 201)` indicates that 199 and 201 are legal, but 200 is not
  - 200 is an interesting number - it happens to be the length of `Tab`
  - What happens in the code when Last is 199, 200, or 201?

```
41 procedure Push (V : Value) is
42 begin
43    if Last = Tab'Last then
44       raise Overflow;
45    end if;
46
47    Screen_Output.Debug_Msg ("Pushing -> " & Values.To_String (V));
48
49    Last := Last - 1;
50    Tab (Last) := V;
51 end Push;
```

- If `Last` is 199, the **if** statement is False, and we assign `Tab(198)` to `V`
- If `Last` is 201, the **if** statement is False, and we assign `Tab(200)` to `V`
- If `Last` is 200, the **if** statement is True, and we raise an overflow exception

If this is a `Push` routine, why are we *decrementing* `Last`?

Fix the issue, and re-run the analysis.

False Positive

Messages for Something That Is Correct

- Not all messages reported by GNAT SAS are actual errors
  - *False positive* - result of performing static analysis on complex code
- In the **Locations** window, click on the message for line 191 of `input.adb`

```
input.adb:191:13: low: array index check [CWE 120]
(Inspector): requires First_Char <= 1_024
```

Why is this a false positive?

Messages for Something That Is Correct

- Not all messages reported by GNAT SAS are actual errors
  - *False positive* - result of performing static analysis on complex code
- In the **Locations** window, click on the message for line 191 of `input.adb`

```
input.adb:191:13: low: array index check [CWE 120]
(Inspector): requires First_Char <= 1_024
```

Why is this a false positive?

- `Skip_Spaces` uses `Get_Char` to get the next printable character
- `Get_Char` increments `First_Char` to a maximum of `Line'Last + 1`
- `Skip_Spaces` calls `Unread_Char` to decrement `First_Char`
- So `First_Char` will never be greater than `Line'Last`

Review Message

- In the **Locations** window, click on the message for line 191 of `input.adb`
- Click then pencil icon next to the message and select Manual Review
- Set the status to **False positive** and press OK
- Rerun the analysis
  - Note the number of messages decreased
  - To include the message in the report, select **False positive** from the *Message review status* filter

Running GNAT SAS Again

Comparing to Baseline

- Note that each of the previous runs have new timestamps (upper right corner of *GNATSAS Report* tab), but our baseline hasn't changed (upper left corner)
  - Messages removed by fixing code are still in the history
  - Select **removed** in *Message history* filter to see old messages
    - Old messages appear in *Locations* window in italics
  - **added** displays messages added since baseline run
  - **unchanged** displays messages in baseline and also in current run

Resetting Baseline

- To set current state to be baseline
  - GNATSAS → Baseline → Bump Baseline to Current Run
  - History is lost
  - All future runs will be compared to this new baseline

Note: You can also use the `timeline` *switch when comparing runs. See the* **Timelines** *chapter in the* **GNAT SAS User's Guide**

GNAT DAS Overview

About This Course

Styles

- *This* is a definition
- `this/is/a.path`
- `code is highlighted`
- `commands are emphasised --like-this`

GNAT Dynamic Analysis Suite (GNAT DAS)

What Is Dynamic Analysis?

- Process of testing and evaluating an application while it is running
- Dynamic analysis finds properties that hold for one or more executions
  - Can't prove a program satisfies a particular property
  - But can detect violations and provide useful information

What Is GNAT DAS?

- Two tools that can work together to analyze code execution
  - GNATcoverage
    - Indicates which lines/decisions/branches have been reached during execution
  - GNATtest
    - Creates framework to build software tests for your codebase

GNATcoverage

Introduction

GNATcoverage

- Provides range of coverage analysis facilities with support for
  - Variety of measurement methods, coverage criteria and output formats
  - Consolidation features to report across multiple program executions
- Sometimes, we only want coverage on certain units
  - Referred to as *Units of Interest*
  - Typically new/modified units
  - Usually excludes any units used for testing

Coverage Data Gathering

- Coverage computed from two kinds of trace files
  - *Binary traces*
    - Produced by instrumented execution environment with unmodifed version of program.
    - Traces contain low level information about executed blocks of machine instructions
  - *Source traces*
    - Produced by modified version of program
    - Original source *instrumented* to generate coverage data

Note: This course will focus on **Source Traces** *coverage*

Coverage Types

Statement Coverage

- Each executed line gets flagged as *covered*
  - Including object initialization

```
3   procedure Test_Statement
4     (A, B, C :     Integer;
5      Z       : out Integer) is
6 +    Local : Integer := A;
7   begin
8 +    Local := Local * 10;
9 +    Local := Local + B * 100;
10 +   if C > 0 then
11 +      Local := Local * 2;
12 +      Z     := Local + C * 1_000;
13     end if;
14  end Test_Statement;
```

- Call `Test_Statement` with `(1, 2, Integer'Last)`
  - Congratulations: 100% Statement Coverage! But...

Statement Coverage

- Each executed line gets flagged as *covered*
  - Including object initialization

```
3    procedure Test_Statement
4      (A, B, C :     Integer;
5       Z       : out Integer) is
6 +     Local : Integer := A;
7    begin
8 +     Local := Local * 10;
9 +     Local := Local + B * 100;
10 +    if C > 0 then
11 +       Local := Local * 2;
12 +       Z     := Local + C * 1_000;
13      end if;
14   end Test_Statement;
```

- Call `Test_Statement` with `(1, 2, Integer'Last)`
  - Congratulations: 100% Statement Coverage! But...
- We have not tested `C <= 0`
  - Which is a problem because we don't assign `Z` in this case
- We cannot tell if `Z := Local + C * 1_000;` raised an exception
  - Statement coverage shows we *reached* a line, not that it executed successfully

Decision Coverage

- Adds evaluation of boolean expressions to statement coverage
  - Not just branches - boolean objects as well

```
16      procedure Test_Decision
17        (A, B, C :     Integer;
18         Z       : out Integer) is
19 +       Check : constant Boolean := A > 0 and then (B**2 > 0 or else C**2 > 0);
20      begin
21 +       if Check then
22 +          Z := A + B + C;
23         else
24 +          Z := A * B * C;
25         end if;
26      end Test_Decision;
```

- Call `Test_Decision` with `(0, 0, 0)` and `(1, 1, Integer'Last)`
  - Congratulations: 100% Decision Coverage! But...

Decision Coverage

- Adds evaluation of boolean expressions to statement coverage
  - Not just branches - boolean objects as well

```
16    procedure Test_Decision
17      (A, B, C :     Integer;
18       Z       : out Integer) is
19 +     Check : constant Boolean := A > 0 and then (B**2 > 0 or else C**2 > 0);
20    begin
21 +     if Check then
22 +        Z := A + B + C;
23       else
24 +        Z := A * B * C;
25       end if;
26    end Test_Decision;
```

- Call `Test_Decision` with `(0, 0, 0)` and `(1, 1, Integer'Last)`
  - Congratulations: 100% Decision Coverage! But...
- `Check` can be `True` or `False` without ever examining `C**2 > 0`
  - `False` when `A <= 0`
  - `True` when `A > 0` and `B >= 1`

Modified Condition/Decision Coverage

- Decision Coverage plus *Unique-Cause* verification
  - *Independent Influence* For each subcondition, changing just the subcondition can change the expression result
- Simple example: `A and then (B or else C)`

Row	A	B	C	Result
1)	F	F	F	*F*
2)	F	F	T	*F*
3)	F	T	F	*F*
4)	F	T	T	*F*
5)	T	F	F	*F*
6)	T	F	T	*T*
7)	T	T	F	*T*
8)	T	T	T	*T*

- Note that rows 2 and 6 show that, if `B` is False and `C` is True, changing `A` changes the result
  - Similarly for rows 5 and 7 for `B` and rows 5 and 6 for `C`
  - There can be multiple pairs of rows depending on the expression
- So, to prove MCDC for subcondition A, coverage results must show that **both** rows 2 and 6 have been executed
- Note that there are two types of MCDC coverage implementations
  - Unique Cause MCDC, where every subcondition must be shown to affect the outcome of the result
  - Masking MCDC, which allows conditions to be grouped, necessary with coupled conditions

Modified Condition/Decision Coverage Example

```
28  procedure Test_Mcdc
29    (A, B, C :     Integer;
30     Z       : out Integer) is
31  begin
32     if A > 0 and then B > 0 then
33        Z := A * B;
34     end if;
35     if A > 0 or else B > 0 or else C > 0 then
36        Z := Z + A + B + C;
37     end if;
38  end Test_Mcdc;
```

- Call `Test_Mcdc` with `(1, 0, 0)`, `(0, 1, 0)`, and `(1, 1, 0)`
  - Better test results, but we need more tests
  - In general, if there are N subconditions, need N+1 sets of data to get complete MCDC coverage

```
34     end if;
35     if A > 0 or else B > 0 or else C > 0 then
condition "B > 0" at 35:24 has no independent influence pair, MC/DC not achieved
condition "C > 0" at 35:38 has no independent influence pair, MC/DC not achieved
36        Z := Z + A + B + C;
```

Basic Workflow

Workflow Overview

General Process

1. Set up instrumentation runtime
2. Instrument sources for coverage
3. Build the executable from the instrumented sources
4. Run the executable
   - Possibly many times
5. Generate and analyze code coverage reports

A Simple Example

Unit of Interest

We want to get code coverage on this unit

```
package Ops is
   type Op_Kind is (Increment, Decrement, Double, Half);

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

package body Ops is
   procedure Apply
     (Op :        Op_Kind;
      X  : in out Integer) is
   begin
      case Op is
         when Increment => X := X + 1;
         when Decrement => X := X - 1;
         when Double    => X := X * 2;
         when Half      => X := X / 2;
      end case;
   exception
      when others =>
         null;
   end Apply;
end Ops;
```

Supplying an Execution Context

- Sometimes, we have an application for which we want coverage
  - But more often, we want coverage on a package or collection of packages
- In our example, we have a package, so we need to create a main program to run

```
with Ada.Text_IO; use Ada.Text_IO;
with Ops;
procedure Test_Driver is
   procedure Run_One
     (Kind  : Ops.Op_Kind;
      Value : Integer) is
      X : Integer := Value;
   begin
      Ops.Apply (Kind, X);
      Put_Line ("Before:" & Value'Image & " After:" & X'Image);
   end Run_One;
begin
   Run_One (Ops.Increment, 4);
end Test_Driver;
```

Setup

- First need to verify our project builds cleanly

```
gprbuild -P default.gpr
```

- Then we need to install the instrumentation context, giving a directory (*prefix*) where the data will be stored

```
gnatcov setup --prefix=.\gnatcov-rts
```

- We need to update the environment variable `GPR_PROJECT_PATH` to add this context:

```
set GPR_PROJECT_PATH=%GPR_PROJECT_PATH%;\path\to\gnatcov-rts\share\gpr

OR

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

Instrument

We now need to add the instrumentation to the source code that will collect data

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

level is the type of coverage information you will gather

`stmt`	Statement coverage
`stmt+decision`	Statement and decision coverage
`stmt+mcdc`	Statement and Masking MCDC
`stmt+uc_mcdc`	Statement and Unique Cause MCDC

Build

- To build the instrumented executable, we just need some extra switches

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

where

`-f` Force recompilation

`-p` Create missing object (and library/executable) directories

`--src-subdirs` Instruct the builder to search for the instrumented versions of the sources

`--implicit-with` Provide visibility to the builder over the coverage runtime referenced by the instrumented sources

Execute

- We can now execute the test program as we would normally

```
obj\test_driver.exe
```

```
Before:  4 After:  5
```

- This generates a source trace file in the working directory that looks like `test-driver.exe-<stamp>.srctrace`
  - *stamp* will be a unique identifier to prevent clashes from multiple executions

Analyze

- Analysis of coverage is done by processing the source trace file(s) generated

```
gnatcov coverage --level=stmt --annotate=xcov test_driver*.srctrace -Pdefault.gpr
```

where

`--level=stmt` indicates we are looking for statement coverage

`--annotate=xcov` indicates we want an annotated source report in text format

`-Pdefault.gpr` indicates we want the report for all units for the executable in the project

- This generates `*.xcov` files in the `obj` directory for each unit in the project

Viewing the Report

- The report file (for package body ops) looks like:

```
C:\temp\gnatcov\src\ops.adb:
33% of 6 lines covered
33% statement coverage (2 out of 6)

Coverage level: stmt
   1 .: package body Ops is
   2 .:    procedure Apply
   3 .:      (Op :        Op_Kind;
   4 .:       X  : in out Integer) is
   5 .:    begin
   6 +:       case Op is
   7 +:          when Increment => X := X + 1;
   8 -:          when Decrement => X := X - 1;
   9 -:          when Double    => X := X * 2;
  10 -:          when Half      => X := X / 2;
  11 .:       end case;
  12 .:    exception
  13 .:       when others =>
  14 -:          null;
  15 .:    end Apply;
  16 .: end Ops;
```

- Coverage information appears after the line number, where

  . indicates uncoverable line

  + means covered line

  - means uncovered line

Lab

Basic Workflow Lab

- We are going to get 100% Statement Coverage on the example from the module
- Copy the `cover_020_basic_workflow` lab from the course materials location
- Contents of the folder:
  - `default.gpr` - project file
  - `src` - source directory

Note: Many of the following pages use animation to first give you a task and then show you how to do it. Page Down *does not always go to the next page!*

Preparing the Command Line

1. Open a command prompt window and navigate to the directory containing `default.gpr`
2. Type `gprbuild -Pdefault.gpr` and press Enter to verify tool is on your path and you can build an executable
   - You can even run the executable (`obj\test_driver.exe` on Windows or `obj/test_driver` on Linux) to see what happens when you run `Increment`
3. Prepare the coverage libraries
   - Windows

     ```
     gnatcov setup --prefix=.\gnatcov-rts
     ```

   - Linux

     ```
     gnatcov setup --prefix=./gnatcov-rts
     ```

   - This creates the `gnatcov-rts` folder containing the coverage libraries
4. Add the coverage project to the `GPR_PROJECT_PATH` environment variable
   - Windows

     ```
     set GPR_PROJECT_PATH=%GPR_PROJECT_PATH%;\path\to\gnatcov-rts\share\gpr
     ```

   - Linux (bash)

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

Instrument, Build, and Execute

1. Perform statement instrumentation on your source code

Instrument, Build, and Execute

1. Perform statement instrumentation on your source code

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

2. Then build the instrumented executable

Instrument, Build, and Execute

1. Perform statement instrumentation on your source code

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

2. Then build the instrumented executable

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

3. And then run it

Instrument, Build, and Execute

1. Perform statement instrumentation on your source code

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

2. Then build the instrumented executable

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

3. And then run it

- Windows

```
obj\test_driver.exe
```

- Linux

```
obj/test_driver
```

If you did this correctly, there should be a `*.srctrace` *file*

Viewing Coverage

1. Add the coverage information into the project

Viewing Coverage

1. Add the coverage information into the project

```
gnatcov coverage --level=stmt --annotate=xcov test_driver*.srctrace -Pdefault.gpr
```

2. Examine the coverage data for the ops unit by viewing the file `ops.adb.xcov` in the `obj` folder

```
33% of 6 lines covered
33% statement coverage (2 out of 6)

Coverage level: stmt
   1 .: package body Ops is
   2 .:    procedure Apply
   3 .:      (Op :        Op_Kind;
   4 .:       X  : in out Integer) is
   5 .:    begin
   6 +:       case Op is
   7 .:          when Increment =>
   8 +:             X := X + 1;
   9 .:          when Decrement =>
  10 -:             X := X - 1;
  11 .:          when Double =>
  12 -:             X := X * 2;
  13 .:          when Half =>
  14 -:             X := X / 2;
  15 .:       end case;
  16 .:    exception
  17 .:       when others =>
  18 -:          null;
  19 .:    end Apply;
  20 .: end Ops;
```

Improving Coverage

- Two ways of getting more coverage
  1. Modify `test_driver` to test a different value for `Ops.Apply Op` parameter
     - When you run the executable to generate coverage, you will get a **`srctrace`** file with a different timestamp to analyze
  2. Expand `test_driver` to test all values for `Ops.Apply Op` parameter in one execution
     - When you run the executable to generate coverage, you will get a **`srctrace`** containing all the coverage information
- Using whichever method you want, get 100% statement coverage
  - One possible solution on next page

Possible Solution

```
with Ada.Text_IO; use Ada.Text_IO;
with Ops;
procedure Test_Driver is
   procedure Run_One
     (Kind  : Ops.Op_Kind;
      Value : Integer) is
      X : Integer := Value;
   begin
      Ops.Apply (Kind, X);
      Put_Line ("Before:" & Value'Image & " After:" & X'Image);
   end Run_One;
begin
   for Op in Ops.Op_Kind loop
      Run_One (Op, 4);
   end loop;
   Run_One (Ops.Increment, Integer'Last);
end Test_Driver;
```

Hints

Possible Solution

```
with Ada.Text_IO; use Ada.Text_IO;
with Ops;
procedure Test_Driver is
   procedure Run_One
     (Kind  : Ops.Op_Kind;
      Value : Integer) is
      X : Integer := Value;
   begin
      Ops.Apply (Kind, X);
      Put_Line ("Before:" & Value'Image & " After:" & X'Image);
   end Run_One;
begin
   for Op in Ops.Op_Kind loop
      Run_One (Op, 4);
   end loop;
   Run_One (Ops.Increment, Integer'Last);
end Test_Driver;
```

Hints

- Whenever you update your source code, you need to re-instrument your project
- If you modify your source code, previous `srctrace` files will be out-of-date, generating a message like:

```
warning: traces for body of test_driver (from test_driver.exe-65ba6772-4f18-65baa1dd.srctrace)
    are inconsistent with the corresponding Source Instrumentation Data
```

Advanced GNATcoverage Capabilities

Introduction

More Options and Capabilities

- Various options exist to include/exclude instrumentation for
  - Subprojects
  - Specific units
  - Specific files
  - Parts of a file
- In addition, the coverage report mechanism allows
  - Multiple output formats
  - Control over what information is reported
  - Coverage on instances of generics or the generics themselves
  - Plus more

Project-Based Instrumentation Control

Simple Instrumentation

- As we saw before, it is easy to instrument a simple project

```
project Default is
   for Source_Dirs use ("src");
   for Object_Dir use "obj";
   for Main use ("test_driver.adb");
end Default;
```

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

- But what happens in a more complicated build environment with multiple projects?

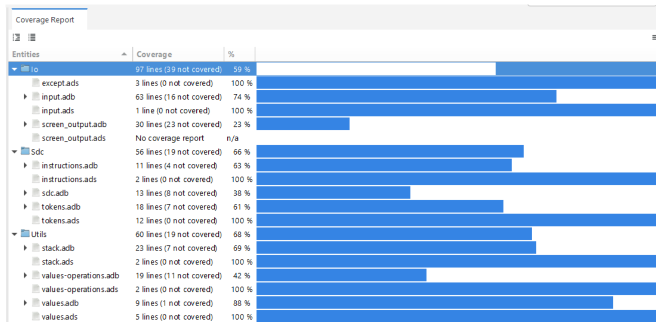
```
with "io/io.gpr";
with "utils/utils.gpr";
project Sdc is
   for Languages use ("ada");
   for Source_Dirs use ("src");
   for Object_Dir use "obj";
   for Main use ("sdc.adb");
end Sdc;
```

Multiple Projects - Simple Case

- As we would expect, the simple case (we want to instrument the entire application) works the same way:

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

- This will instrument all source files within the `src` folder, plus any source files from projects `io` and `utils`



Coverage Report

Entities	Coverage	%
Io	97 lines (39 not covered)	59 %
except.ads	3 lines (0 not covered)	100 %
input.adb	63 lines (16 not covered)	74 %
input.ads	1 line (0 not covered)	100 %
screen_output.adb	30 lines (23 not covered)	23 %
screen_output.ads	No coverage report	n/a
Sdc	56 lines (19 not covered)	66 %
instructions.adb	11 lines (4 not covered)	63 %
instructions.ads	2 lines (0 not covered)	100 %
sdc.adb	13 lines (8 not covered)	38 %
tokens.adb	18 lines (7 not covered)	61 %
tokens.ads	12 lines (0 not covered)	100 %
Utils	60 lines (19 not covered)	68 %
stack.adb	23 lines (7 not covered)	69 %
stack.ads	2 lines (0 not covered)	100 %
values-operations.adb	19 lines (11 not covered)	42 %
values-operations.ads	2 lines (0 not covered)	100 %
values.adb	9 lines (1 not covered)	88 %
values.ads	5 lines (0 not covered)	100 %

Indicating Projects of Interest

- But what if we don't want coverage on a particular subproject?
  - Might be externally built
  - Might be a re-used project that we don't need coverage for
- Specifying *projects of interest* is handled by the instrumentation command
- We build the command using the following options
  - `--no-subprojects` tells the instrumenter to only process the root project
  - `--projects=utils` tells the instrumenter to only process the `utils` project
- You can combine the options for better control
  - Root project only

```
gnatcov instrument -Psdc.gpr --no-subprojects --level=stmt
```

  - Subproject `utils` only (not `sdc`)

```
gnatcov instrument -Psdc.gpr --no-subprojects --projects=utils --level=stmt
```

  - Root project and subproject `io`

```
gnatcov instrument -Psdc.gpr --projects=sdc --projects=io --level=stmt
```

Indicating Units of Interest Within Projects

- By default, all units within project(s) of interest are considered *units of interest*
- We can control units of interest from the project file's `Coverage` package

```
package Coverage is
  for Units use ("instructions", "tokens");
end Coverage;
```

  - Note that **units** refer to the Ada name, not the source file name
    - So `package Naming` would have no affect
- The four keywords to control units of interest

`units`	List of units to instrument
`units_list`	Filename containing list of units to instrument
`excluded_units`	List of units **not** to instrument
`excluded_units_list`	Filename containing list of units **not** to instrument

What About Separates?

- Sometimes you build your application using a **separate** body for a package or subprogram

```
package body Input is
   procedure Read_New_Line is separate;
end Input;
```

  - Useful when your build process wants a different body based on various situations

```
package Naming is
   case Build is
      when "DEBUG" =>
         for Body ("input.read_new_line")
            use "read_new_line_from_console.adb";
      when "PRODUCTION" =>
         for Body ("input.read_new_line")
            use "read_new_line_from_file.adb";
   end case;
end Naming;
```

- As a **separate** is not a unit, how do we prevent instrumentation of this subprogram when DEBUG is set?

```
package Coverage is
  for Ignored_Source_Files use ("input-*.adb");
  for Ingored_Source_Files_List use "files_to_skip.txt";
end Coverage;
```

Units of Interest and Their Dependents

- In a large project, we might want coverage on a unit PLUS every unit it calls
  - Analyzing the entire project is overkill, but we don't want to find all the units that our unit needs
- When instrumenting files, GNATCOVERAGE creates an *obligation file* (file extension `.sid`)
  - This file contains information regarding all the dependents for the unit
- To get coverage on a unit and all its dependents, use the `--sid` option for the unit(s) you want
  - Use the unit name for individual files, or `@<filename>` to specify a file containing a list

```
gnatcov coverage -Pdefault.gpr --level=stmt --sid=tokens.sid --sid=@sidfiles.lst
```

Source-Based Instrumentation Control

Coverage Exemptions

- Sometimes there are blocks of code for which you do not want coverage reporting
  - Typically for defensive coding purposes

```
1  function My_New return Access_T is
2     Retval : Access_T;
3  begin
4     Retval := new Record_T;
5     if Retval = null then
6         raise Program_Error;
7     end if;
8     return Retval;
9  end My_New;
```

- The likelihood of **line 5** being `True` should be small, so we don't want the `False` branch (and the `raise` statement) to reduce our coverage totals

Coverage Exemption Region

- We need to modify the My_New subprogram to indicate where we do not want coverage
  - Use **pragma** Annotate to indicate source that should not be tracked

```
function My_New return Access_T is
   Retval : Access_T;
begin
   Retval := new Record_T;
   pragma Annotate (Xcov, Exempt_On, "justification");
   if Retval = null then
      raise Program_Error;
   end if;
   pragma Annotate (Xcov, Exempt_Off);
   return Retval;
end My_New;
```

  - Note that we are turning on/off the *exemption*, not the *instrumentation*
    - That's why we start the block with Exempt_On on **line 5** and end with Exempt_Off on **line 9**

Coverage Exemption Reporting

- Coverage reports can be generated in multiple ways

```
gnatcov coverage --level=stmt+decision
--annotate=<form>*trace* -P default.gpr
```

where *<form>* is one of the following:

report
: Summary that lists all coverage violations

xcov
: For each source file, the **xcov** file contains a global summary of assessment results followed by annotated source lines

xcov+
: Same as **xcov** except it provides extra details below lines with improperly satisfied obligations

html
: Web-based reporting mechanism to show coverage data for the project

xml
: XML database containing all necessary coverage information

xcov Output File

utils.adb.xcov

Without exemptions :

```
60% of 5 lines covered
80% statement coverage (4 out of 5)
0% decision coverage (0 out of 1)

Coverage level: stmt+decision
   1 .: package body Utils is
   2 .:
   3 .:    function My_New return Access_T is
   4 +:       Retval : Access_T;
   5 .:    begin
   6 +:       Retval := new Record_T;
   7 !:       if Retval = null then
   8 -:          raise Program_Error;
   9 .:       end if;
  10 +:       return Retval;
  11 .:    end My_New;
  12 .:
  13 .: end Utils;
```

With exemptions

```
60% of 5 lines covered
60% statement coverage (3 out of 5)
0% decision coverage (0 out of 1)

Coverage level: stmt+decision
   1 .: package body Utils is
   2 .:
   3 .:    function My_New return Access_T is
   4 +:       Retval : Access_T;
   5 .:    begin
   6 +:       Retval := new Record_T;
   7 *:       pragma Annotate (Xcov,
   8 *:                        Exempt_On,
   9 *:                        "justification");
  10 *:       if Retval = null then
  11 *:          raise Program_Error;
  12 *:       end if;
  13 *:       pragma Annotate (Xcov,
  14 .:                        Exempt_Off);
  15 +:       return Retval;
  16 .:    end My_New;
  17 .:
  18 .: end Utils;
```

Note different coverage indicator for exempted code

report Output File

- Exemptions appear in the coverage summary report

```
gnatcov coverage --level=stmt+decision
--annotate=report *trace* -P default.gpr
```

```
** COVERAGE REPORT **
===========================
== 1. ASSESSMENT CONTEXT ==
===========================
Date and time of execution: 2024-02-21 15:08:45 -05:00
Tool version: XCOV 24.0 (20231011)
Command line:
C:\GNATPRO\24.0\bin\gnatcov.exe coverage --level=stmt+decision --annotate=report main.exe-65d6573e-9358-65d65741.srctrace -P default.gpr
Coverage level: stmt+decision
Trace files:
main.exe-65d6573e-9358-65d65741.srctrace
  kind     : source
  program  : C:\temp\temp\obj\main.exe
  date     : 2024-02-21 15:04:17 -05:00
  tag      :
=========================================
== 2. NON-EXEMPTED COVERAGE VIOLATIONS ==
=========================================
2.1. STMT COVERAGE
------------------
No violation.
2.2. DECISION COVERAGE
----------------------
No violation.
=========================
== 3. EXEMPTED REGIONS ==
=========================
utils.adb:7:7-13:7: 2 exempted violations, justification:
"justification"
Exempted violations:
utils.adb:10:10: decision outcome TRUE never exercised
utils.adb:11:10: 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.
** END OF REPORT **
```

Lab

Advanced Topics Lab

- We are going to demonstrate different ways of controlling coverage information on the supplied base project
  - Which happens to include subprojects
- Copy the `cover_030_advanced_topics` lab from the course materials location
  - This is basically a copy of the *Tutorial* example from the GNAT distribution
- Directories in the folder:
  - `io` - I/O support subproject
  - `utils` - Utilities subproject
  - `sdc` - base project
- Each directory contains a project file and `src` directory

Note: Many of the following pages use animation to first give you a task and then show you how to do it. Page Down *does not always go to the next page!*

Quick Note on the Application

- **SDC** stands for *Simple Desktop Calculator*
- The input is a list of operands an operators in programmatic order, not mathematic
  - The calculator does have a simple memory
- Example: To add **11** to **22** you would enter `11 22 +`
  - To multiply that by **10**, the next line might be `10 *`
- In additon to numbers, you have the commands **Clear**, **Print**, and **Quit**

  Example run `sdc.exe`

Display	Description
Welcome to SDC. Go ahead type your commands ...	
100 20 +	User Input
100 20 +	Command echo
3 +	User Input
3 +	Command echo
print	User Input
print	Command echo
-> 123	Result of **Print**
quit	User Input
quit	Command echo
Thank you for using SDC.	

Initialization

- Make sure your project builds

Initialization

- Make sure your project builds

```
cd /path/to/sdc.gpr
gprbuild -P sdc.gpr
```

- Prepare the coverage libraries

Initialization

- Make sure your project builds

```
cd /path/to/sdc.gpr
gprbuild -P sdc.gpr
```

- Prepare the coverage libraries

```
gnatcov setup --prefix=.\gnatcov-rts

OR

gnatcov setup --prefix=./gnatcov-rts
```

Don't forget to set the environment variable `GPR_PROJECT_PATH` to point to the folder containing the `gnatcov_rts.gpr` file

Workflow One - Coverage on Base Project

- Typically we only care about coverage on the project we are working with
  - We know we won't get 100% coverage on things like utility packages, so we don't want to instrument them
- Instrument the `sdc` project for statement and decision coverage *without* instrumenting the `utils` or `io` projects

Workflow One - Coverage on Base Project

- Typically we only care about coverage on the project we are working with
  - We know we won't get 100% coverage on things like utility packages, so we don't want to instrument them
- Instrument the `sdc` project for statement and decision coverage *without* instrumenting the `utils` or `io` projects
- Two ways to do this
  - Method 1 - focus only on base project

    ```
    gnatcov instrument -Psdc.gpr --no-subprojects
    --level=stmt+decision
    ```

  - Method 2 - specify particular project

    ```
    gnatcov instrument -Psdc.gpr --projects=sdc
    --level=stmt+decision
    ```

Workflow One Continued

- Build your application

Workflow One Continued

- Build your application

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

- Run your application and add the coverage to the project

Workflow One Continued

- Build your application

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

- Run your application and add the coverage to the project

```
obj/sdc
```

- Add the coverage to the project

Workflow One Continued

- Build your application

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

- Run your application and add the coverage to the project

```
obj/sdc
```

- Add the coverage to the project

```
gnatcov coverage --level=stmt+decision --annotate=xcov
*.srctrace -Psdc.gpr
```

- Inspect the coverage

Workflow One Continued

- Build your application

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

- Run your application and add the coverage to the project

```
obj/sdc
```

- Add the coverage to the project

```
gnatcov coverage --level=stmt+decision --annotate=xcov
*.srctrace -Psdc.gpr
```

- Inspect the coverage

It should be in the `obj/*.xcov` files

Workflow Two - Excluding Source Files

- We only want coverage on package bodies
  - Modify the base project file to ignore all `*.ads` files

- Instrument the base project, run the executable, and analyze the coverage data

Workflow Two - Excluding Source Files

- We only want coverage on package bodies
  - Modify the base project file to ignore all `*.ads` files
- We need to add the following to `sdc.gpr`

```
package Coverage is
   for Ignored_Source_Files use ("*.ads");
end Coverage;
```

- You could also use `for Ignored_Source_Files_List use ("list.txt");` where `list.txt` lists all the spec files

- Instrument the base project, run the executable, and analyze the coverage data

Workflow Two - Excluding Source Files

- We only want coverage on package bodies
  - Modify the base project file to ignore all `*.ads` files
- We need to add the following to `sdc.gpr`

```
package Coverage is
   for Ignored_Source_Files use ("*.ads");
end Coverage;
```

- You could also use `for Ignored_Source_Files_List use ("list.txt");` where `list.txt` lists all the spec files

- Instrument the base project, run the executable, and analyze the coverage data

```
gnatcov instrument -Psdc.gpr --level=stmt
gprbuild -f -p -Psdc.gpr --src-subdirs=gnatcov-instr --implicit-with=gnatcov_rts.gpr
obj/sdc
gnatcov coverage --level=stmt --annotate=xcov *.srctrace -Psdc.gpr
```

- When looking for `*.xcov` files, note they exist only for `*.adb` files
- Note the warning that no information was found for unit `Except`
  - Because it's only a package spec

Workflow Three - Excluding Source Code

- We do not want coverage on any exception processing in unit `Sdc`
    - Use `pragma Annotate` to turn off coverage in the exception blocks

Workflow Three - Excluding Source Code

- We do not want coverage on any exception processing in unit Sdc
  - Use pragma Annotate to turn off coverage in the exception blocks

```
34 pragma Annotate (Xcov, Exempt_On, "Exception Handler");
35 exception
36    when Stack.Underflow =>
37       Error_Msg ("Not enough values in the Stack.");
38
39    when Stack.Overflow =>
40      null;
41 pragma Annotate (Xcov, Exempt_Off);
```

- Now run your code and generate a summary report

Workflow Three - Exemption Reporting

- When you look look at `sdc.adb.xcov` you'll notice the exempted lines are marked with *

```
29 .:      begin
30 .:
31 +:         Process (Next);
32 .:         --  Read the next Token from the input and process it.
33 .:
34 *:      pragma Annotate (Xcov, Exempt_On, "Exception Handler");
35 *:      exception
36 *:         when Stack.Underflow =>
37 *:            Error_Msg ("Not enough values in the Stack.");
38 *:
39 *:         when Stack.Overflow =>
40 *:           null;
41 *:      pragma Annotate (Xcov, Exempt_Off);
42 .:      end;
```

- While the summary report contains a description of the exemption region

```
=========================
== 3. EXEMPTED REGIONS ==
=========================

sdc.adb:34:7-41:7: 2 exempted violations, justification:
"Exception Handler"

Exempted violations:
sdc.adb:37:13: statement not executed
sdc.adb:40:13: statement not executed

1 exempted region, 2 exempted violations.
```

GNATcoverage From GNAT Studio

Introduction

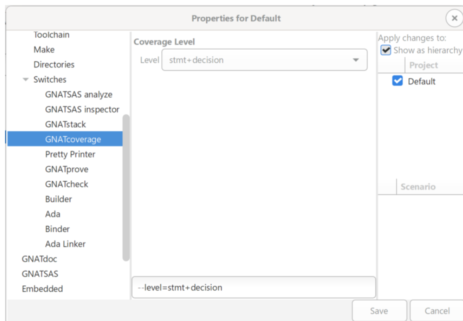
Coverage Integrated with IDE

- Benefits of using GNAT Studio for coverage processing
  - One click to instrument, build, run, analyze
  - Color-coded coverage view
  - Edit source code from coverage view

Generating Coverage From GNAT Studio

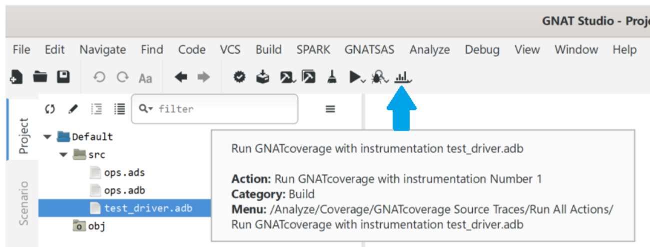
Setting Coverage Type

- On the command line, we used `--level=stmt` for each step to specify coverage type
  - This allows us to generate a **Statement** report even if we've instrumented for **Statement** and **Decision**
- For GNAT Studio, we simplify by using the same coverage type for all steps
  - Edit → Project Properties

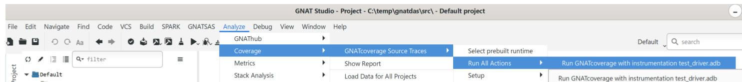


One-Click Coverage

- Once the coverage type is set, you can use the menu or shortcut icon to instrument, build, and execute

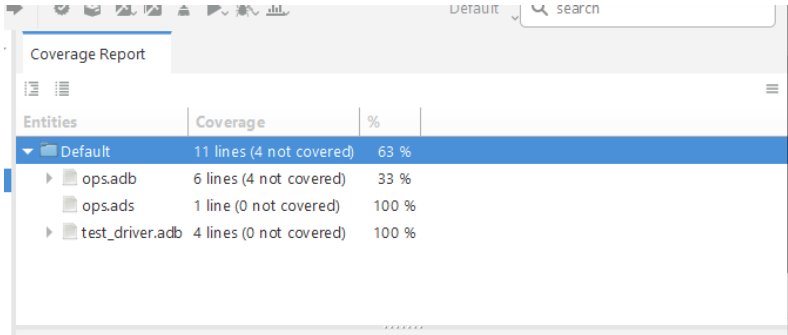


- Analyze → Coverage → GNATcoverage Source Traces → Run All Actions



Viewing Coverage

- Coverage report is displayed after execution



- Benefits from running coverage from GNAT STUDIO
  - Coverage report is graphical and interactive
  - Annotated source code view

Coverage Report

Coverage report default view

Coverage report with units expanded (shows subprograms)

- Shows all source within project(s)
  - Numeric percentage of coverage
  - Graphical representation of coverage
- Click on column title to sort by
  - Unit name (**Entities**)
  - Absolute coverage numbers (**Coverage**)
  - Relative coverage numbers (**%**)

Annotated Source Code

- Double-click on row in **Entities** column to show annotated code
  - Unit or subprogram

```
Coverage Report   ops.adb
1  package body Ops is
2     procedure Apply
3       (Op :        Op_Kind;
4        X  : in out Integer) is
5     begin
6 +      case Op is
7           when Increment =>
8 +            X := X + 1;
9           when Decrement =>
10 -            X := X - 1;
11          when Double =>
12 -            X := X * 2;
13          when Half =>
14 -            X := X / 2;
15       end case;
16    exception
17       when others =>
18 -         null;  -- don't do anything
19    end Apply;
20 end Ops;
```

- Selecting row in report vs unit in project view
  - Double-clicking row in report always brings up annotated view
  - Selecting unit in project view will
    - Open normal source view if unit not already displayed
    - Switch to displayed unit if tab already open (normal or annotated)
  - These are different *views* of the same file
    - So edits in one view automatically appear in other view
    - (That's why you don't have two tabs open!)

Updating Code with Coverage

Typical Development Process

- During development (typically called *Code and Test*) you
  - Write your code
  - Run simple tests to make sure things don't blow up
  - Make sure all paths through your code are tested
- How does this process integrate with coverage instrumentation?

Updating Coverage When Code Changes

- Coverage executable is different from normal executable
    - Changes to code need to be added to coverage information
    - Only one executable is maintained
        - So if you build your normal executable, you need to rebuild the instrumented one to get coverage
- Simplest method of re-running to get updated coverage data
    - One-click method
        - `Run GNATcoverage...` icon
        - Analyze → Coverage → GNATcoverage Source Traces → Run All Actions
- Or manually do each step in the menu
    - Setup
    - Instrumentation
    - Build
    - Run
    - Generate Report

Lab

GNATcoverage From GNAT Studio

- We are going to get 100% Statement Coverage on the example from the module
  - But now we're doing it from the IDE!
- Copy the `cover_040_gnatstudio` lab from the course materials location
- Contents of the folder:
  - `default.gpr` - project file
  - `src` - source directory

Note: Many of the following pages use animation to first give you a task and then show you how to do it. Page Down *does not always go to the next page!*

Start GNAT Studio

- Start GNAT STUDIO and open project `default.gpr`
  - From the **Start Menu** or **Application Launcher**
    - Then File → Open Project and navigate to the file
  - From a command prompt
    - `gnatstudio -P /path/to/default.gpr` **OR**
    - `gnatstudio` if `default.gpr` is the only project in the current directory

Instrument Your Project

- Always best to make sure your code compiles first
  - Build → Project → Build All **OR**
  - *Build target Build All* icon
- Set the coverage type to **Stmt**

Instrument Your Project

- Always best to make sure your code compiles first
  - Build → Project → Build All **OR**
  - *Build target Build All* icon
- Set the coverage type to **Stmt**

Edit → Project Properties → Coverage

- Instrument the project

Instrument Your Project

- Always best to make sure your code compiles first
  - Build → Project → Build All **OR**
  - *Build target Build All* icon
- Set the coverage type to **Stmt**

Edit → Project Properties → Coverage

- Instrument the project

Analyze → Coverage → GNATcoverage Source Traces → Run All Actions **OR**

Run GNATCoverage with instrumentation test_driver.adb* icon

- Your coverage report should be displayed

Navigating the Coverage Report

- Experiment with the coverage report
    - Click on column titles to change order
    - Click expansion triangle to see coverage per subprogram
    - Double-click on an entity to see annotated coverage
- Next, edit the test driver to test more subprograms and run the new driver

Navigating the Coverage Report

- Experiment with the coverage report
  - Click on column titles to change order
  - Click expansion triangle to see coverage per subprogram
  - Double-click on an entity to see annotated coverage
- Next, edit the test driver to test more subprograms and run the new driver
- If you clicked your normal *Build & Run test_driver.adb* icon, coverage didn't update!
  - You need to rebuild the coverage to get the updated code

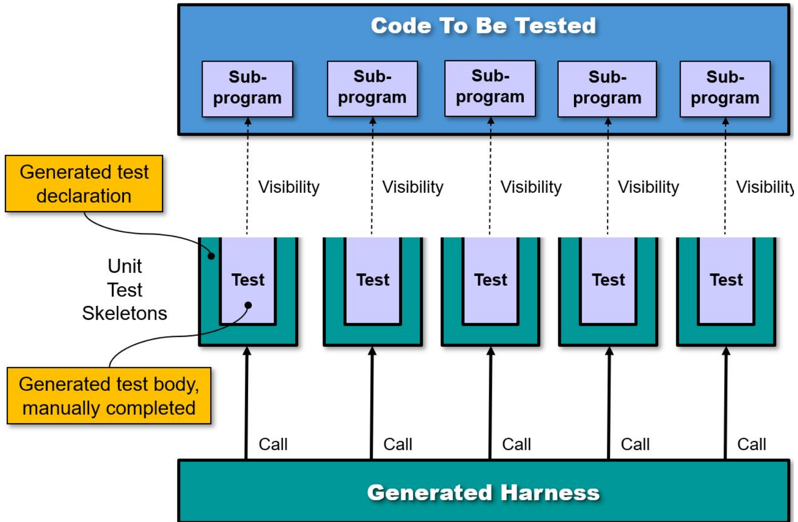
GNATtest

Introduction

Why Automate the Process?

- Developing tests is labor-intensive
- Much of the effort is not specific to the tests
  - Developing the harness and driver
    - How to test generic units, etc.
  - Verifying output is as expected
  - Maintenance and update when new units to be tested
- Ideally developers should concentrate on the high-value part: the test cases themselves
- GNATTEST makes that ideal possible

What Can Be Automated?



GNATtest

- Tool to create unit test framework
    - Creates skeleton for each visible subprogram in packages under consideration
- Automatic unit test infrastructure generation including
    - Test harness
    - Stub generation
    - Aggregates results from multiple test drivers

Legal Ada Code

- Sources must be compilable
  - Warnings issued otherwise
  - If not, GNATTEST will skip it and continue to any others
- All source dependencies must be available
  - Those units named in **`with`** clauses, transitively
  - Whether or not they are to be analyzed themselves

Based on AUnit

- Unit test framework based on CppUnit for C++
- Generates the boilerplate code for test harnesses, suites, and cases needed to use the framework
- For more information on **AUnit** view the series of tutorials created by Daniel Bigelow
  - http://www.youtube.com/user/DanielRBigelow

Usage

Overview

Test Generation Methods

- Framework Generation Mode
  - Used to generate framework for writing individual unit tests
    - Test drivers, stubs, etc
  - Creates one executable to run all tests
- Test Execution Mode
  - Used to generate a driver to call individual test executables

Simple Test Generation

Building a Test Harness

- Build test harness for a simple project

```
gnattest --harness-dir=driver -P default.gpr
```

  - Where `--harness-dir=driver` creates the test harness in a folder called **driver** inside the **obj** directory

- To run the driver, build and run the executable in the **obj/driver** folder

```
cd obj/driver
gprbuild -P test_driver
test_runner
```

Gives the result:

```
simple.ads:3:4: error: corresponding test FAILED: Test not implemented. (simple-test_data-tests.adb:44)
simple.ads:7:4: error: corresponding test FAILED: Test not implemented. (simple-test_data-tests.adb:65)
2 tests run: 0 passed; 2 failed; 0 crashed.
```

- Note that the tests fail!
  - We have only built a harness - it's up to the tester to implement the test

Test Data Structure

- GNATTEST builds a child package for each unit (e.g. `Simple`) to test called `Simple.Test_Data` which contains
  - Type `Test` to contain test information
    - Extensible by tester if necessary
  - `Set_Up`/`Tear_Down` procedures to call before/after test execution
    - Useful for initialize and verify global data
- GNATTEST also builds child package `Simple.Test_Data.Tests` containing test driver for each visible subprogram
  - `Test_XXXX_YYYY` where **XXXX** is the subprogram name and **YYYY** is a unique identifier (prevents overloading/scoping issues)
  - Implementation seeded with failure case ("Test not implemented") - should be replaced with test implementation
- When editing generated files, make sure **not** to edit between *begin read only* and *end read only* comments
  - Anywhere else will remain when test harness is regenerated

Test Case Format

- Test example

```
33 --  begin read only
34    procedure Test_Inc (Gnattest_T : in out Test);
35    procedure Test_Inc_4f8b9f (Gnattest_T : in out Test) renames Test_Inc;
36 --  id:2.2/4f8b9f38b0ce8c74/Inc/1/0/
37    procedure Test_Inc (Gnattest_T : in out Test) is
38    --  simple.ads:3:4:Inc
39 --  end read only
40
41       pragma Unreferenced (Gnattest_T);
42
43    begin
44
45       AUnit.Assertions.Assert
46         (Gnattest_Generated.Default_Assert_Value,
47          "Test not implemented.");
48
49 --  begin read only
50    end Test_Inc;
51 --  end read only
```

  - Line 33-39 - test declaration *(do not modify)*
  - Line 41 - suppress unused parameter warning (if necessary)
  - Line 45-47 - Test assertion (if first parameter is `False`, test fails - print second parameter)
  - Line 49-51 - end of test *(do not modify)*
- By default, `Default_Assert_Value` is `False`, so that unimplemented tests fail
  - It is possible to change the value to `True` so that unimplemented tests do not clutter report

Test Implementation

- For `Test_Inc`, we modify the test to verify that `Increment` succeeded

```
45  AUnit.Assertions.Assert
46    (Inc(1) = 2,
47     "Incrementation failed");
```

- Then we rerun the test

```
gprbuild -P test_driver
test_runner
```

Giving the result:

```
simple.ads:3:4: info: corresponding test PASSED
simple.ads:7:4: error: corresponding test FAILED: Test not implemented. (simple-test_data-tests.adb:66)
2 tests run: 1 passed; 1 failed; 0 crashed.
```

Lab

Usage Lab

- Test a simplistic stack

```
package Simple_Stack is

   procedure Push (Item : Integer);
   function Pop return Integer;
   function Empty return Boolean;
   function Full return Boolean;
   function Top return Integer;
   function Count return Natural;

   procedure Reset;

end Simple_Stack;
```

  - There is a bug in the code - your testing should find it!

- Copy the `test_020_usage` lab from the course materials location

Note: Many of the following pages use animation to first give you a task and then show you how to do it. Page Down *does not always go to the next page!*

Initialization

- Build a test harness for the project

Initialization

- Build a test harness for the project
- One possible command

```
gnattest -P default.gpr --harness-dir=my_test
```

  - If you do not specify `--harness-dir=<dir>` the harness goes in `obj/gnattest/harness`
- Build and run the test driver

Initialization

- Build a test harness for the project
- One possible command

  ```
  gnattest -P default.gpr --harness-dir=my_test
  ```

  - If you do not specify `--harness-dir=<dir>` the harness goes in `obj/gnattest/harness`
- Build and run the test driver

```
cd obj/my_test
gprbuild -P test_driver
test_runner
```

For each subprogram in `Stack`, you should get a line like

```
simple_stack.ads:3:4: error: corresponding test FAILED:
Test not implemented.
(simple_stack-test_data-tests.adb:44)
```

With a summary line like

```
7 tests run: 0 passed; 7 failed; 0 crashed.
```

Build Your First Test

- Build a test to prove that `Push` works
  - Criteria would be that, after the call:
    - `Empty` should be `False`
    - `Count` should be `1`
    - `Top` should be whatever was pushed
  - Hint: the filename you're looking for is in the `Test not implemented` message

  Next page for example solutions

- Build and run the test harness to verify your test passes

Build Your First Test

- Build a test to prove that `Push` works
  - Criteria would be that, after the call:
    - `Empty` should be `False`
    - `Count` should be `1`
    - `Top` should be whatever was pushed
  - Hint: the filename you're looking for is in the `Test not implemented` message

  Next page for example solutions

- Build and run the test harness to verify your test passes

```
gprbuild -P test_driver.gpr
test_runner
```

Note indication that test passed

Example Tests

- Solution 1 - one check

```
declare
   Pushed : constant integer := 123;
begin
   Push (Pushed);
   AUnit.Assertions.Assert ((not Empty) and then Top = Pushed and then Count = 1,
                            "Push test failed");
end;
```

- Solution 2 - multiple checks

```
declare
   Pushed : constant integer := 123;
begin
   Push (Pushed);
   AUnit.Assertions.Assert ( not Empty,
                             "Test failed - stack empty");
   AUnit.Assertions.Assert ( Top = Pushed,
                             "Test failed - Top /= pushed value");
   AUnit.Assertions.Assert ( Count = 1,
                             "Test failed - count incorrect");
end;
```

Note that when multiple assertions are used, the test stops on the first failed assertion

Improve First Test

- We want to know what happens when `Push` pushes to a full stack
- Add a second part of the testcase to test this
  - `Push` inside a loop is easiest

Improve First Test

- We want to know what happens when `Push` pushes to a full stack
- Add a second part of the testcase to test this
  - `Push` inside a loop is easiest

```
while not Full loop
   Push (234);
end loop;
Push (345);
AUnit.Assertions.Assert (Full and then Top = 234,
                         "Push to a full stack failed");
```

Test Remaining Subprograms

- Test all remaining subprograms
  - Criteria should be based on what **should** happen
    - Not what **does** happen
- Remember - there is a bug in the code!
  - If a test fails - recheck your assertions
  - If your assertions are correct - then check the code
  - Feel free to fix the code or leave the failure
    - Both are common practices

Test Remaining Subprograms

- Test all remaining subprograms
  - Criteria should be based on what **should** happen
    - Not what **does** happen
- Remember - there is a bug in the code!
  - If a test fails - recheck your assertions
  - If your assertions are correct - then check the code
  - Feel free to fix the code or leave the failure
    - Both are common practices

Hint: Only one execution, so global state is remembered

Test Remaining Subprograms

- Test all remaining subprograms
  - Criteria should be based on what **should** happen
    - Not what **does** happen
- Remember - there is a bug in the code!
  - If a test fails - recheck your assertions
  - If your assertions are correct - then check the code
  - Feel free to fix the code or leave the failure
    - Both are common practices

Hint: Only one execution, so global state is remembered

Call `Reset` to reset the stack data

Sample Answers

These answers assume the bug in the code is fixed

Answers on next pages

Sample Answers

These answers assume the bug in the code is fixed

Bug is in Pop - should be

```
function Pop return Integer is
begin
   if not Empty then
      Next_Available := Next_Available - 1;
   end if;
   return Stack (Next_Available);
end Pop;
```

Answers on next pages

Answers (1/2)

```
-- Push
Reset;
declare
   Pushed : constant integer := 123;
begin
   Push (Pushed);
   AUnit.Assertions.Assert ((not Empty) and then Top = Pushed and then Count = 1,
                            "Push test failed");
end;

while not Full loop
   Push (234);
end loop;
Push (345);
AUnit.Assertions.Assert (Full and then Top = 234,
                         "Push to a full stack failed");

-- Pop
Reset;
declare
   Pushed : constant integer := 234;
  Popped : integer;
begin
   Push (Pushed);
  Popped := Pop;
   AUnit.Assertions.Assert (Pushed = Popped and then Empty and then Count = 0,
                            "Pop test failed");
end;
```

Answers (2/2)

```
-- Empty
Reset;
AUnit.Assertions.Assert (Empty, "Stack not empty");

-- Full
while not Full loop
   Push (567);
end loop;
Push (999);
AUnit.Assertions.Assert (Full and then Top = 567,
                         "Full check failed");

-- Top
Reset;
declare
   Pushed : constant integer := 234;
begin
   Push (Pushed);
   AUnit.Assertions.Assert (Pushed = Top,
                            "Top test failed");
end;

-- Count
Reset;
Push (111);
AUnit.Assertions.Assert (Count = 1,
                         "Count test failed");

-- Reset
Reset;
AUnit.Assertions.Assert (Count = 0 and then Empty,
                         "Reset test failed");
```

Controlling GNATtest

Overview

Controlling Test Behavior

- Two ways to affect test behavior
  - Internally
  - Externally
- Internal control comes from modifying the original source or the test driver
- External control comes from modifying the switches used to create or run the harness

Source-based Test Control

Global Data

- Many subprograms require global data initialization
    - Memory allocation
    - State values
- Test pass/fail criteria can depend on global state values
- Could build these into each individual test
    - But what if the values are common across multiple tests?

Common Pre-/Post-Test Behavior

- Unit's test data package (e.g. `Simple.Test_Data`) contains two visible subprograms
  - `Set_Up` is called before every test case is run
    - Allows initialization of global state
  - `Tear_Down` is called after every test case is run
    - Allows adding checks for global state
- Found in `<unit>-test_data.adb`

```
procedure Set_Up (Gnattest_T : in out Test) is
   pragma Unreferenced (Gnattest_T);
begin
   --  Clear stack before running test
   Simple_Stack.Reset;
end Set_Up;

procedure Tear_Down (Gnattest_T : in out Test) is
   pragma Unreferenced (Gnattest_T);
begin
   Ada.Text_IO.Put_Line ("Count:" & Simple.Stack.Count'Image);
end Tear_Down;
```

Passing Data Between Tests

- Notice that `Set_Up` and `Tear_Down` (in addition to each **Test** procedure) pass parameter `Gnattest_T` of type `Test`
  - Defined in **<unit>.Test_Data**

```
package Simple_Stack.Test_Data is

--  begin read only
   type Test is new AUnit.Test_Fixtures.Test_Fixture
--  end read only
   with null record;
```

  - Note that the completion of the record type is outside of the *read only* block allowing you to modify it as you see fit
- Parameter of type `Test` is passed to `Set_Up` and `Tear_Down` and every test
  - Allows passing of any user-defined data

Changes to Original Source Code

- What happens when testing finds a bug?
  - Your source code needs to be modified
  - But does the test infrastructure need to be updated?
- GNATTEST can be run multiple times on a project
  - Any existing test will not be modified as long as
    - Subprogram name is the same
    - Full Ada names and order of parameters are the same
    - Test's *begin/end read only* comments are intact
  - Any added subprogram will get a new driver

Test Harness File Structure

Default File Structure

- By default, two folders are created in project's object directory
  - `driver` contains the main driver for the test runner
    - Not for modification by the user
  - `gnattest` contains the modifiable test harness
    - But do not edit inside the *begin/end read only* comments!
- GNAT typically puts all files it generates in the project's object directory
  - So we tend to set up source code control to ignore the object directory
- But we **do** want to control the tests we've created

Controlling Test Harness Location

- `driver` folder is always auto-generated - do **not** want to save it
- `gnattest` folder contains our test cases - **do** want to save it
- Three (mutually exclusive) ways to control this
  - `--tests-dir=dirname`
    - Put all tests in `dirname`
  - `--tests-root=dirname`
    - `dirname` will mirror the source directory hierarchy
    - Tests for units in each source directory go in the corresponding directory within `dirname`
  - `--subdirs=dirname`
    - `dirname` will be created *inside* each appropriate source directory
    - Tests for units in source directory go in `dirname` subdirectory
- Notes
  - If `dirname` is relative, it will be relative to the object directory
  - If your GPR file uses `source_dir/**`, you should not use `subdirs`
    - And if using the other options, do not put them in your source folders

External Test Control (Switches)

Default Test Behavior

- Default test behavior is to fail on an unimplemented test
  - Value of `Default_Assert_Value` is set to `True`
- Can be controlled at generation or execution

`--skeleton-default=xxxx` (where **xxxx** is *pass* or *fail*)

  - When used during build (`gnattest --skeleton-default=pass`) `Default_Assert_Value` is initialized based on value
  - When used during execution (`test_runner --skeleton-default=pass`) `Default_Assert_Value` is set based on value

Common Switches

`-U <source file>` Only build tests for *source file* and any of its dependents

`--no-subprojects` Only process base project

`--files=<filename>` Process files listed in *filename* (switch may appear multiple times)

`--ignore=<filename>` Ignore files listed in *filename*

`--passed-tests=val` *val* can be either *show* or *hide* to either display (or not display) passed tests

`--separate-drivers[=val]` Generate separate test driver for each unit or test. (*val* can be either *unit* or *test*, defaulting to *unit*)

Lab

Controlling GNATtest Lab

- We are going to use the same code as the previous lab
  - But clean up our test code
  - And try some GNATTEST switches
- Copy the `test_030_controlling_gnattest` lab from the course materials location
  - Put it in a new directory so you can refer back to the *Usage Lab* answers

Note: Many of the following pages use animation to first give you a task and then show you how to do it. Page Down *does not always go to the next page!*

Build Harness for One Unit

- Build a test harness only for the `Simple_Stack` unit

Build Harness for One Unit

- Build a test harness only for the `Simple_Stack` unit

```
gnattest -P default.gpr --harness-dir=my_test -U simple_stack.ads
```

Note the unit specifier is a filename, not Ada name. (Also, spec or body filename is allowed)

- Run all the tests to get the *not implemented* message

Build Harness for One Unit

- Build a test harness only for the `Simple_Stack` unit

```
gnattest -P default.gpr --harness-dir=my_test -U simple_stack.ads
```

Note the unit specifier is a filename, not Ada name. (Also, spec or body filename is allowed)

- Run all the tests to get the *not implemented* message

```
cd obj/my_test
gprbuild -P test_driver
test_runner
...
7 tests run: 0 passed; 7 failed; 0 crashed.
```

- Now run the tests with *not implemented* tests indicating *passed*

Build Harness for One Unit

- Build a test harness only for the `Simple_Stack` unit

```
gnattest -P default.gpr --harness-dir=my_test -U simple_stack.ads
```

Note the unit specifier is a filename, not Ada name. (Also, spec or body filename is allowed)

- Run all the tests to get the *not implemented* message

```
cd obj/my_test
gprbuild -P test_driver
test_runner
...
7 tests run: 0 passed; 7 failed; 0 crashed.
```

- Now run the tests with *not implemented* tests indicating *passed*

```
cd obj/my_test
gprbuild -P test_driver
test_runner --skeleton-default=pass
...
7 tests run: 7 passed; 0 failed; 0 crashed.
```

Create Tests

- Re-write or copy the test answers from the *Usage* lab (or use these)

```
-- Push
Reset;
declare
   Pushed : constant integer := 123;
begin
   Push (Pushed);
   AUnit.Assertions.Assert ((not Empty) and then Top = Pushed and then Count = 1,
                            "Push test failed");
end;

while not Full loop
   Push (234);
end loop;
Push (345);
AUnit.Assertions.Assert (Full and then Top = 234,
                         "Push to a full stack failed");

-- Pop
Reset;
declare
   Pushed : constant integer := 234;
  Popped : integer;
begin
   Push (Pushed);
  Popped := Pop;
   AUnit.Assertions.Assert (Pushed = Popped and then Empty and then Count = 0,
                            "Pop test failed");
end;

-- Empty
Reset;
AUnit.Assertions.Assert (Empty, "Stack not empty");

-- Full
while not Full loop
   Push (567);
end loop;
Push (999);
AUnit.Assertions.Assert (Full and then Top = 567,
                         "Full check failed");

-- Top
Reset;
declare
   Pushed : constant integer := 234;
begin
   Push (Pushed);
   AUnit.Assertions.Assert (Pushed = Top,
                            "Top test failed");
end;

-- Count
Reset;
Push (111);
AUnit.Assertions.Assert (Count = 1,
                         "Count test failed");

-- Reset
Reset;
AUnit.Assertions.Assert (Count = 0 and then Empty,
                         "Reset test failed");
```

Ensure Every Test Starts the Same

- Previously, every test called `Simple_Stack.Reset` to ensure the stack was initialized
  - Lots of redundant code
- Remove calls to `Simple_Stack.Reset` and (re)run the tests

Ensure Every Test Starts the Same

- Previously, every test called `Simple_Stack.Reset` to ensure the stack was initialized
  - Lots of redundant code
- Remove calls to `Simple_Stack.Reset` and (re)run the tests
- Answer

```
cd obj/my_test
gprbuild -P test_driver
test_runner
...
7 tests run: 2 passed; 5 failed; 0 crashed.
```

- Rerun the tests but do not display the passed tests

Ensure Every Test Starts the Same

- Previously, every test called `Simple_Stack.Reset` to ensure the stack was initialized
    - Lots of redundant code
- Remove calls to `Simple_Stack.Reset` and (re)run the tests
- Answer

```
cd obj/my_test
gprbuild -P test_driver
test_runner
...
7 tests run: 2 passed; 5 failed; 0 crashed.
```

- Rerun the tests but do not display the passed tests
- Answer

```
test_runner --passed-tests=hide
...
7 tests run: 2 passed; 5 failed; 0 crashed.
```

Status is the same, we just do not see individual passed tests

Add "Global" Code

- Add code to call `Simple_Stack.Reset` before every test case

Add "Global" Code

- Add code to call `Simple_Stack.Reset` before every test case

simple_stack-test_data.adb

```
procedure Set_Up (Gnattest_T : in out Test) is
   pragma Unreferenced (Gnattest_T);
begin
   Reset;
end Set_Up;
```

- For extra credit, add code to clear global data

Add "Global" Code

- Add code to call `Simple_Stack.Reset` before every test case

simple_stack-test_data.adb

```
procedure Set_Up (Gnattest_T : in out Test) is
   pragma Unreferenced (Gnattest_T);
begin
   Reset;
end Set_Up;
```

- For extra credit, add code to clear global data

simple_stack-test_data.adb

```
procedure Tear_Down (Gnattest_T : in out Test) is
   pragma Unreferenced (Gnattest_T);
begin
   Reset;
end Tear_Down;
```

This ensures the stack is reset when tests for other units are run

Advanced Testing Techniques

Overview

Improving Test Execution

- By default, GNATTEST builds a monolithic test driver
  - One executable to run all tests
  - Suitable for small to medium projects
- But that has limitations
  - Every test runs in succession
    - No way to run multiple tests at once
  - Larger projects can create a massive test executable
- GNATTEST has a mechanism to build multiple executables
  - Tests grouped by unit or test

Control Over Dependent Units

- When testing a unit, sometimes it is easier to test in isolation
  - Control over dependent unit calls
    - Verify data passed in
    - Control data being returned
- GNATTEST allows *stubs* to be created for dependent units
  - Add verification process to data passed in
  - Set output or return values
- Stubs are used for all dependents of units being tested

Individual Test Drivers

Example Code

- Main unit

```
package Simple is
   function Inc (X : Integer) return Integer;
   function Dec (X : Integer) return Integer;
end Simple;
```

- Which depends on

```
package Dependent is
   procedure Yes_Or_No (Aaa : in     Integer;
                        Bbb : in out Integer;
                        Ccc :    out Boolean);
end Dependent;
```

Building Multiple Test Harnesses

- GNATTEST can build multiple test harnesses
  - `gnattest --separate-drivers=[unit|test]`

    unit Builds an executable for every package (for our example, `Simple` and `Dependent`)

    test Builds an executable for every test (for our example, `Inc`, `Dec`, `Yes_Or_No`)
- Then build the individual test drivers
  - `gprbuild -P obj/gnattest/harness/test_drivers.gpr`

Running Multiple Test Harnesses

```
gnattest <test_drivers.list>
```

- Where `test_drivers.list` is a file containing a list of executables
- Default version of list is in `obj/gnattest/harness/test_drivers.list`
  - Can be edited in-place or copied

```
dependent.ads:2:4: error: corresponding test FAILED: Test not implemented. (dependent-test_data-tests.adb:44
simple.ads:7:4: error: corresponding test FAILED: Test not implemented. (simple-test_data-tests.adb:65)
simple.ads:3:4: error: corresponding test FAILED: Test not implemented. (simple-test_data-tests.adb:44)
3 tests run: 0 passed; 3 failed; 0 crashed.
```

Test Stubs

What Is a Stub?

- Stub is a piece of code that replaces the actual body of a unit if
  - Unit has not been implemented yet
  - Unit is hardware-dependent and hardware is not available
  - Specific unit results are difficult to control
    - For when you need a specific value to test your code
- Useful when you need to test one module without worrying about dependencies

Creating Stubs

```
gnattest --stub -P default.gpr
```

- Creates stubs and drivers for all units
- Every dependent of unit being tested is stubbed
  - Including generics
- Stub harnesses are in `gnattest_stub`
  - Rather than `gnattest`
  - Both folders can exist!
- Stubs are common across units
  - Mutiple test drivers call the same stub
  - Stub control handled by test

Controlling Stubs

- Setter routines for setting output/return values
  - Manipulate a global object containing stub information
  - Reside in package `Dependent.Stub_Data`
  - Typically called from test driver
- Can edit stub implementation directly
  - Add assertions to verify data passed in is correct
  - In `stubs` subfolder in folder named for project
  - Can add your own processing
    - e.g. Raise an exception on a specific input or after some number of calls

Example

Code to Be Tested

```
with Sensor;
package Simple is
   procedure Check (Which  :        Sensor.Sensor_T;
                    Value  : in out Integer;
                    Status :    out Boolean);
end Simple;

with Logger;
package body Simple is
   procedure Check (Which  :        Sensor.Sensor_T;
                    Value  : in out Integer;
                    Status :    out Boolean) is
   begin
      Value   := Sensor.Read (Which);
      Status := True;
      case Which is
         when Sensor.Speed =>
            if Value < 0 or Value > 99 then
               Status := False;
               Logger.Log_Error ("Invalid Speed");
            end if;
         when others =>
            null;
      end case;
end Simple;
```

Dependent Units

```
package Logger is
   procedure Log_Error (Message : String);
end Logger;

package Sensor is
   type Sensor_T is (Speed, Heading, Altitude);
   function Read (Which : Sensor_T) return Integer;
end Sensor;
```

Implementation of these units is unimportant

Building Tests

- No matter how the dependent units are implemented, the tests should be the same

simple-test_data-tests.adb

```
--  begin read only
   procedure Test_Check (Gnattest_T : in out Test);
   procedure Test_Check_0265af (Gnattest_T : in out Test) renames Test_Check;
--  id:2.2/0265af9a17cc096e/Check/1/0/
   procedure Test_Check (Gnattest_T : in out Test) is
   --  simple.ads:3:4:Check
--  end read only

      pragma Unreferenced (Gnattest_T);

      Value :  Integer := 0;
      Status : Boolean;

   begin

      -- Test 1
      Check (Sensor.Speed, Value, Status);
      AUnit.Assertions.Assert
        (Value in 0..99 and Status,
         "Valid speed not detected");

      -- Test 2
      Check (Sensor.Speed, Value, Status);
      AUnit.Assertions.Assert
        (not (Value in 0..99) and not Status,
         "Invalid speed not detected");

--  begin read only
   end Test_Check;
--  end read only
```

Setting Stub Return Data

- To make sure Check passes each test, we should stub Sensor
  - To control the value returned by Sensor.Read:

```
gnattest -P default.gpr --stub
gprbuild -P obj/gnattest_stub/harness/test_drivers.gpr
```

- Method 1 - use the setter function with Test_Check

```
-- Test 1
Set_Stub_Read_cac9ed_9101fc (Read_Result => 12);
Check (Sensor.Speed, Value, Status);
AUnit.Assertions.Assert
  (Value in 0..99 and Status,
   "Valid speed not detected" & value'Image & " " & status'Image);

-- Test 2
Set_Stub_Read_cac9ed_9101fc (Read_Result => 234);
Check (Sensor.Speed, Value, Status);
AUnit.Assertions.Assert
  (not (Value in 0..99) and not Status,
   "Invalid speed not detected");
```

- Method 2 - edit the stub directly

obj/gnattest_stub/stubs/default/sensor.adb

```
--  begin read only
function Read
  (Which : Sensor_T) return Integer is
--  end read only
begin
   Stub_Data_Read_cac9ed_9101fc.Stub_Counter := Stub_Data_Read_cac9ed_9101fc.Stub_Counter + 1;
   if Stub_Data_Read_cac9ed_9101fc.Stub_Counter > 1 then
      return -1;
   else
      return Stub_Data.Stub_Data_Read_cac9ed_9101fc.Read_Result;
   end if;
--  begin read only
end Read;
--  end read only
```

Lab

Advanced Testing Lab

- We will test a simplistic sensor read/write capability
  - `Simple.Read` reads a sensor and determines if the value is in range
  - `Simple.Write` writes to a sensor and reports if the write failed
  - Error messages are sent to an error logger
- Copy the **`test_040_advanced_testing`** lab from the course materials location

Note: Many of the following pages use animation to first give you a task and then show you how to do it. Page Down *does not always go to the next page!*

Create Test Harness

- Build a test harness that enables stubbing and allows each test to be run individually

Create Test Harness

- Build a test harness that enables stubbing and allows each test to be run individually

```
gnattest --stub -P default.gpr --separate-drivers=test
--harness-dir=my_test
```

- `--stub` enables stubbing
- `--separate-drivers=test` builds an executable for each test
    - Stubbing always requires separate drivers
    - If not specified, an executable is built for each unit
- `--harness-dir=my_test` puts the harness code in folder `my_test`

Build and Execute Test Harness

- Built the test harness
  - Hint: This is slightly different than earlier labs due to `separate-drivers`

Build and Execute Test Harness

- Built the test harness
  - Hint: This is slightly different than earlier labs due to `separate-drivers`

```
cd obj/my_test
gprbuild -P test_drivers.gpr
```

Note the **s** *at the end of* **driver**

- Now execute the test harness
  - Hint: This is quite different than earlier labs!

Build and Execute Test Harness

- Built the test harness
  - Hint: This is slightly different than earlier labs due to `separate-drivers`

```
cd obj/my_test
gprbuild -P test_drivers.gpr
```

Note the **s** *at the end of* **driver**

- Now execute the test harness
  - Hint: This is quite different than earlier labs!

```
gnattest test_drivers.list
```

- When running multiple test drivers, pass the list of drivers into GNATTEST
  - `test_drivers.list` is automatically created in `my_test`
  - Copy and/or edit it to control what tests get run
  - Unlike the monolithic driver, skipped tests are not reported

Build One Test with Stubs

- Build and run test for `Simple.Read` that
  - Receives a good value from `Sensor.Read`
  - Verifies output parameter `Value` has the previously set result
  - Verifies output parameter `Status` is True
  - Hint: one mechanism to set stub return data is in the test case skeleton

Build One Test with Stubs

- Build and run test for `Simple.Read` that
  - Receives a good value from `Sensor.Read`
  - Verifies output parameter `Value` has the previously set result
  - Verifies output parameter `Status` is True
  - Hint: one mechanism to set stub return data is in the test case skeleton
- Test code inserted into **simple-test_data-tests.adb**

```
declare
   Sensor_Value : constant := 12;
   Result : Integer := 0;
   Status : Boolean;
begin
   Sensor.Stub_Data.Set_Stub_Read_cac9ed_9101fc(Read_Result => Sensor_Value);
   Read (Sensor.Speed, Result, Status);
   AUnit.Assertions.Assert
     (Result = Sensor_Value and then Status,
      "Read positive test failed");
end;
```

- Execution command

```
gnattest test_drivers.list
```

Build More Advanced Test

- We want to test `Simple.Read` creates an error message. Criteria would be that it
  - Receives a bad value from `Sensor.Read`
  - Verifies output parameter `Status` is False
  - `Logger.Log_Error` receives the appropriate message
  - Hint: You need to modify `Logger` to check for the error message
    - No mechanism to retrieve input to a stub

Build More Advanced Test

- We want to test `Simple.Read` creates an error message. Criteria would be that it
  - Receives a bad value from `Sensor.Read`
  - Verifies output parameter `Status` is False
  - `Logger.Log_Error` receives the appropriate message
  - Hint: You need to modify `Logger` to check for the error message
    - No mechanism to retrieve input to a stub
- Test code inserted into **simple-test_data-tests.adb**

```
declare
   Result : Integer := 0;
   Status : Boolean;
begin
   Sensor.Stub_Data.Set_Stub_Read_cac9ed_9101fc(Read_Result => 1234);
   Read (Sensor.Speed, Result, Status);
   AUnit.Assertions.Assert
     ( not Status,
      "Read negative failed - status");
end;
```

- Test code inserted into **logger.adb**
  - In **/gnattest_stub/stubs/default** folder

```
if Stub_Data_Log_Error_e35760_8432c2.Stub_Counter = 1 then
   AUnit.Assertions.Assert
     ( Message = "Invalid Speed",
      "Read negative failed - Log_Error");
end if;
```

  - There are more advanced ways of ensuring stub is checked for appropriate text, but they're outside the scope of this class

Finish Testing

- Build as many more tests as you can in the remaining time
  - Experiment with both methods of setting return values
    - **Setter** subprogram
    - Edit stub directly
- Extra credit: figure out a better way of checking which test case called the stub

Extra Credit Answer

- `gnattest_stub/stubs/default/logger-stub_data.ads`

```
type Caller_T is (Speed, Heading, Altitude, Unknown);
type Stub_Data_Type_Log_Error_e35760_8432c2 is record
   Caller : Caller_T := Unknown;
   Stub_Counter : Natural := 0;
end record;
Stub_Data_Log_Error_e35760_8432c2 : Stub_Data_Type_Log_Error_e35760_8432c2;
```

- `gnattest_stub/stubs/default/logger.adb`

```
Stub_Data_Log_Error_e35760_8432c2.Stub_Counter :=
      Stub_Data_Log_Error_e35760_8432c2.Stub_Counter + 1;
if Stub_Data_Log_Error_e35760_8432c2.Caller = Speed then
   AUnit.Assertions.Assert
     ( Message = "Invalid Speed",
      "Read negative failed - Log_Error");
end if;
```

- `simple-test_data-tests.adb`

```
Sensor.Stub_Data.Set_Stub_Read_cac9ed_9101fc(Read_Result => 1234);
Logger.Stub_Data.Stub_Data_Log_Error_e35760_8432c2.Caller :=
   Logger.Stub_Data.Speed;
```