GNAT SAS User's Guide

Release 27.0w

AdaCore

Apr 28, 2026

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 About GNAT SAS

GNAT SAS is the GNAT Static Analysis Suite. It is a set of analysis engines with complementary capabilities, that are able to detect a range of issues spanning from breaking coding style standards to deep logic errors.

GNAT SAS is designed to support large systems and to detect a wide range of programming errors such as misuse of pointers, indexing out of arrays, buffer overflows (a prevalent source of security storage leaks), numeric overflows, numeric wraparounds, and improper use of Application Programming Interfaces (APIs). It pinpoints the root cause of each error down to the source line of code.

GNAT SAS enables finding defects early in the development process in conjunction with compilation when they are least costly to repair. In addition, it should be run on existing code to find latent defects before they become a problem in the field.

One of the strengths of GNAT SAS is its ability to analyze partial systems (e.g. libraries, incomplete set of sources, etc.) and make reasonable assumptions about functions that are not visible.

Another strength is GNAT SAS' capability to compare each new run to a selected baseline run. This allows users to ignore initial messages on an existing code base, and concentrate on new messages only detected as part of code changes. GNAT SAS also provides advanced capabilities to filter and review generated messages.

In addition, GNAT SAS supports analysis of Ada code written for many different Ada compilers via the configuration of a project file, see *Ensuring compatibility with GNAT SAS*.

An analysis may be launched *from the command line* for easy scripting and automation, or interactively *using the GNAT SAS integration in GNAT Studio*. The analysis results can be output in various formats (e.g. text, CSV, Code Climate, SARIF, HTML).

GNAT SAS stores its analysis results in *Message files* with a `.sam` extension, and user reviews in a *Review file* with a `.sar` extension, as explained in *GNAT SAS Files Reference*.

See also:

- Refer to *Getting Started* for a quick introduction to GNAT SAS.
- Refer to *Workflows* for suggested workflows and recommended usage scenarios.
- Refer to *FAQ* for frequent questions and migration from previous CodePeer versions.

1.2 Features Overview

- Integration of multiple analysis engines

  GNAT SAS currently integrates four *analysis engines*, providing a rich set of complementary checks, and lets the user baseline and review their messages in a uniform way. Each engine can be individually configured for more precise analysis.

- Analysis modes

  GNAT SAS provides two *analysis modes*, *fast* and *deep*. Fast mode offers the best precision/performance trade-off, with fewer messages but less false positives. It is suitable for quick development workflows. Deep mode will output more precise messages but at the expense of reporting more false positives and taking more execution time.

- Incrementality

  The GNAT SAS analysis is fully incremental when run in *fast* mode. This means that during the first run all files will be analyzed, while only modified source files will be analyzed during subsequent runs. In *deep* mode, the Inspector engine will lose incrementality, but other engines will still run incrementally.

- Configurable analysis scope

  The GNAT SAS analysis can be easily configured to target various scopes, such as the full closure of a project, all sources in a project tree, or individual files. See *Partial Analysis* for more details.

- Advanced reporting capabilities

  GNAT SAS provides multiple report formats, from simple text or Comma-Separated Values (CSV) to standard formats such as CodeClimate or SARIF, enabling advanced CI integration workflows. GNAT SAS also provides comparison between subsequent runs or arbitrary ones. All this is also available within GNAT Studio. See *Viewing Results* for more details.

- Review capabilities

  GNAT SAS provides several ways to review the messages, graphically in GNAT Studio, using CSV files or using source code annotations. Those reviews can then be easily versioned and shared between multiple users when necessary. See *Reviewing Messages* for more details.

- Full integration with GNAT Studio

  GNAT SAS has full integration support within the GNAT Studio IDE, to run analyses, display reports, compare runs and review results. This is documented in *Using GNAT SAS in GNAT Studio*.

- Metric computations for Ada

  GNAT SAS integrates the GNAT Metrics tool that computes advanced metrics data for Ada source files and reports them in various formats. The tool can be launched with the `gnatsas metrics` subcommand. Refer to the GNAT Metrics Documentation for more details about specific usage and to *GNAT SAS CLI Reference* for more information about the command-line interface.

1.3 Integrated Analysis Engines

GNAT SAS includes several analysis engines providing together a rich set of checks. This section describes each engine at a high-level.

1.3.1 Inspector

Inspector is designed to follow numerical computations in a very precise way and thus excels in detecting possibly failing run-time checks as well as wide range of logical errors. Inspector guarantees full path coverage.

Even in the absence of explicit errors, Inspector provides a thorough characterization of every component of the system in terms of the *preconditions* on the inputs necessary to preclude run-time failures, the *presumptions* about return values of external subprograms, the *postconditions* that characterize the range of outputs, and heap object creations.

Inspector uses partitioning in its analysis. It usually gains in precision when using bigger partitions, including when switching from *fast* to *deep* mode. The downside of bigger partitions is an increase in Inspector's analysis time. Furthermore, messages are aggressively filtered in fast mode. This means that deep mode will display more messages, but they may contain more false positives.

Importantly, the same kinds of errors are searched in both modes.

Fast mode

- Inspector analyzes each library unit separately. This allows Inspector's analysis to be incremental: when re-analyzing a project, only the units that have changed will need to be re-inspected.
- Inspector is able to precisely follow function calls when the caller and the callee are in the same file. For other function calls, the effects of the call are approximated.

Deep mode

- Inspector performs the analysis by group of units, whose size depend on factors depending on *Partitioning of Analysis*. The analysis will start from scratch each time, losing incrementality, but it will be more thorough overall.
- In *deep mode*, Inspector is able to precisely follow all function calls in the same partitions.
- In addition, Inspector also provides detection of race conditions.

For information about how Inspector works internally, see *How Does Inspector Work?*.

1.3.2 Infer

The Infer framework, specialized to Ada by AdaCore, is designed to provide fast analyses with a very low false positive rate. Infer analyses can precisely follow different paths in the program even across function calls including calls to functions in different files. Infer is thus especially good in detecting problems occurring on certain program paths, such as null-pointer dereferences or memory leaks, and providing precise and comprehensible explanations of these problems.

Infer analysis is fully incremental with respect to previous runs.

1.3.3 GNAT Warnings

GNAT Warnings provides warnings issued by the GNAT compiler frontend. This includes warnings about suspicious constructs and warnings about situations when the compiler is sure an exception will be raised at run-time. This can be especially useful if you are using a different compiler for compilation of your code. Even if you are using GNAT, you can activate more warnings when running GNAT SAS and then use GNAT SAS' filtering and reviewing capabilities to select interesting messages.

1.3.4 GNATcheck

GNATcheck provides automated checking for suspicious code constructs and compliance with organizational and project-specific coding standard rules. This includes coding style rules, feature restriction rules, and metric compliance rules. GNATcheck provides a wide range of predefined rules and also makes it possible to define new rules.

1.4 Installation

1.4.1 Prerequisites

System Requirements

The operating systems GNAT SAS is supported on are listed in the README file that can be downloaded from GNAT Tracker below the GNAT SAS downloads, e.g. for the 25.2 release, README-gnatsas-25.2.txt.

Static error detection is complex and time consuming, so GNAT SAS will benefit from all the speed (CPU) and memory (RAM) that can be made available.

Even though any machine can run GNAT SAS these days, in order to get the most out of GNAT SAS, you will need to use a powerful machine with many cores and memory. A comfortable setup for a GNAT SAS server is a machine with 16 physical cores or more, with at least 64GB of memory (4GB per core) running a 64bits GNAT SAS.

For a desktop machine a minimum of 4 cores is recommended (8 preferred) with at least 4GB per core (so 16 to 32GB).

Note that using local and fast drives will also make a difference in terms of analysis and storing results. Network drives such as NFS, SMB or worse, configuration management filesystems (such as ClearCase dynamic views) should be avoided as much as possible and will produce very degraded performance (typically 2 to 3 times slower than on local fast drives). If such slow drives cannot be avoided for accessing the source code, you can still configure your project file so that the *object directory* for GNAT SAS' analysis is stored on a drive local to the machine performing the run. This can be achieved by setting the `Object_Dir` project attribute, see *Analyzer package attributes* for more details.

Installing GNAT Studio (optional)

The GNAT Studio IDE provides great integration of the GNAT SAS tool, allowing users to easily analyze their projects, display reports and review the results as described in *Using GNAT SAS in GNAT Studio*. Please refer to the GNAT Studio documentation for installation and use instructions.

Note that using GNAT Studio is optional as GNAT SAS is fully usable through the Command-Line Interface and provides various means and formats to output the results and review them.

1.4.2 Installation under Windows

Run the GNAT SAS installer, e.g. by double clicking on `gnatsas-VERSION-x86_64-windows-bin.exe`.

You should have sufficient rights for installing the package (administrator or normal rights depending on whether it is installed for all users or a single user).

The GNAT SAS installer can also be run from the Windows command line with the switches */D*, to select the installation directory, and */S*, to perform unattended installation.

1.4.3 Installation under GNU/Linux

Extract the GNAT SAS installer package and run the *.doinstall* script as below:

```
$ tar -xvf gnatsas-{VERSION}-x86_64-linux-bin.tar.gz $ cd
gnatsas-{VERSION}-x86_64-linux-bin $ ./doinstall
```

Note that you need to have sufficient rights for installing the package at the chosen location (e.g. root rights for installing under `/opt/gnatsas`).

CHAPTER

TWO

GETTING STARTED

This chapter provides a quick introduction to GNAT SAS by illustrating the main concepts: analysis, report and review. The user is guided through a simple command-line example which does not require any particular configuration.

Note: This chapter assumes that:

- the user environment is a *bash* terminal;
- GNAT SAS has been installed according to *Installation*.

It also provides links to more advanced resources in *More Resources*.

The chapter does not try to cover all features or more advanced use cases. Refer to other chapters for more comprehensive documentation.

2.1 Quick Start example

This section will illustrate how to run an analysis with GNAT SAS and view the analysis results, before showing how results can be reviewed and the code fixed.

Let's first change the directory to the directory containing our example project `<INSTALL_DIR>/share/examples/gnatsas/uninitialized/`:

```
$ cd <INSTALL_DIR>/share/examples/gnatsas/uninitialized/
```

This example namely contains:

- a README file
- a GPR project file `uninitialized.gpr` (if that is new to you, refer to *Creating a Project File*).
- Ada source code (`*.adb`/`*.ads` files).

The project file is configured to use `uninit.adb` as the main unit, and to use GNAT SAS in *deep mode*.

See also:

Refer to *Analysis modes* for a description of the *fast* and *deep* modes provided by GNAT SAS.

2.1.1 Analyzing the project

To analyze the project, run the `gnatsas analyze` command by specifying the name of the project file (the extension is optional):

```
$ gnatsas analyze -P uninitialized

[gnatsas]: launching GNATcheck
[gnatsas]: launching Infer
[gnatsas]: launching Inspector
Compile
   [Ada]          uninit.adb
   [Ada]          adt.adb
Bind
   [gprbind]      uninit.bexch
   [Ada]          uninit.ali
no partitioning needed.
starting the analysis of adt.scil
analyzed adt.scil in 0.00 seconds
starting the analysis of adt__body.scil
analyzed adt__body.scil in 0.00 seconds
re-starting the analysis of adt.scil
re-analyzed adt.scil in 0.00 seconds
starting the analysis of uninit__body.scil
analyzed uninit__body.scil in 0.00 seconds
starting the analysis of uninit.scil
analyzed uninit.scil in 0.00 seconds
starting the analysis of b__uninit.scil
analyzed b__uninit.scil in 0.00 seconds
starting the analysis of b__uninit__body.scil
analyzed b__uninit__body.scil in 0.00 seconds
re-starting the analysis of b__uninit.scil
re-analyzed b__uninit.scil in 0.00 seconds
analysis complete.
6 .scil files processed; 14 subprograms analyzed.
```

In the above output, we can see that GNAT SAS launched the following integrated engines and returned successfully (with exit code `0`):

- GNATcheck
- Infer
- Inspector

There is some verbose output, in particular about the Inspector analysis, telling us that some files were compiled, and 14 subprograms were analyzed in total.

Upon analysis GNAT SAS analysis artifacts were generated and stored in the *gnatsas directory* (here named `gnatsas`), which in particular contains the *output directory* `gnatsas/uninitialized.outputs` (hereafter referred to as <output_dir>). That directory contains important files among others:

- a *Message file* containing messages found during the current analysis run: `uninitialized.deep.sam`
- the baseline for this run: `uninitialized.deep.baseline.sam`.

However, `gnatsas analyze` does not directly display the results. That is done by the `gnatsas report` command as we will see next.

Note: The version of engines in the output (e.g. `gnatcheck 25.0w (20231112)` may vary depending on the GNAT SAS version).

See also:

- Refer to *Analyzing Code* for a comprehensive description of GNAT SAS analysis.
- Refer to *GNAT SAS Files Reference* for a description of GNAT SAS artifacts.

2.1.2 Viewing the results

Printing the text report

The resuts from the Message file `<output_dir>/uninitialized.deep.sam` can be displayed with the `gnatsas report` command that is run similarly by specifying the project file name:

```
$ gnatsas report -P uninitialized
```

which outputs the following list of messages in text format on standard output:

```
uninit.adb:7:4: high: validity check [CWE 457] (Infer): `X.B1` is read without␣
↪initialization during the call to `adt.update`
uninit.adb:7:4: medium: precondition <overflow check> [CWE 190] (Inspector):␣
↪precondition might fail on call to adt.update; requires not X.B1 or X.DZ /= Integer_32
↪'Last
```

The messages are formatted such as:

```
filename:line:column: rank: kind (analysis engine name): message contents
```

Here, for instance, the first message is emitted by Infer, has a *high* ranking, and is reported on `uninit.adb` at line 7, column 4:

```
1 with ADT; use ADT;
2
3 procedure Uninit is
4    T : ADT_Type;
5 begin
6    Initialise (T);
7    Update (T); -- high validity check reported here
8 end Uninit;
```

The *kind* of the message is `validity check [CWE 457]`, which means that GNAT SAS reports a potential error due to accessing an uninitialized variable. The `[CWE 457]` part indicates that this kind of message has the identifier `457` in the Common Weakness Enumeration (CWE); it is optional, not all messages can be linked to a CWE weakness.

The description of the specific potential violation corresponding to this kind is detailed in the *message contents* that lets us know that `X.B1` is read without initialization when calling `adt.update`. Note that this is an example of performing analysis across multiple units.

Note: Such check requiring cross-unit analysis can only be done in deep mode with the Inspector engine (the analysis mode only influences the results of Inspector and the result of Infer is not impacted). If you want to compare to the results in fast mode, you can quickly run:

```
$ gnatsas analyze -P uninitialized --mode=fast
$ gnatsas report -P uninitialized
```

That will only report the Infer message.

Indeed, if we look at how the body of `adt.adb`, we can see that `X.B1` is never initialized:

```
1  package body ADT is
2
3     ----------------
4     -- Initialise --
5     ----------------
6
7     procedure Initialise (X :  out ADT_Type) is
8     begin
9        X.V1 := 1;  --  we didn't initialize X.B1 nor X.DZ
10    end Initialise;
11
12    ------------
13    -- Update --
14    ------------
15
16    procedure Update (X : in out ADT_Type) is
17    begin
18       if X.B1 then          --  read uninitialized field X.B1
19          X.V1 := X.V1 +2;
20          X.DZ := X.DZ +1;   --  read uninitialized field X.DZ
21       end if;
22    end Update;
23
24 end ADT;
```

See also:

- We will not describe other messages here; refer to *GNAT SAS Messages Reference* for more information about all supported checks and related message kinds.
- For more information about the CWEs supported by GNAT SAS, refer to *CWE Categorization of Messages*.
- In this example the results are reported as text, but GNAT SAS supports various other formats (e.g. CSV, SARIF...). Refer to *Viewing Results* for a comprehensive description of GNAT SAS reporting capabilities.

Looking at backtraces

Messages reported by GNAT SAS may contain even more details, by showing us the backtraces associated to the potential issues. To enable displaying backtraces, simply run with `--show-backtraces`:

```
$ gnatsas report -P uninitialized --show-backtraces
```

The first message is now reported as:

```
uninit.adb:7:4: high: validity check [CWE 457] (Infer): `X.B1` is read without␣
↪initialization during the call to `adt.update`
```

(continues on next page)

(continued from previous page)

```
   struct field address `B1` created at uninit.adb:4:4
   when calling `adt.update` here at uninit.adb:7:4
      parameter `X` of adt.update at adt.adb:16:4
      read to uninitialized value occurs here at adt.adb:18:10
```

The backtrace provides more information:

- first, `X.B1` is created as a record field when declaring the variable `T` with type `ADT_Type` in `uninit.adb`:4:4
- then `T` is passed as argument to `ADT.Update` where the corresponding formal parameter `X` has its field `X.B1` read without initialization in `adt.adb`:18:10.

See also:

Refer to *GNAT SAS CLI Reference* for more information about command-line switches.

Filtering messages

The `gnatsas report` command also supports advanced filtering of messages. For example, say that as a first step, we are only interested in seeing messages of *high* ranking. This is specified with `--show rank=high`:

```
$ gnatsas report -P uninitialized --show rank=high
```

Only *high* ranked messages are reported (one in this case):

```
uninit.adb:7:4: high: validity check [CWE 457] (Infer): `X.B1` is read without␣
↪initialization during the call to `adt.update`
```

See also:

- The `--show` switch provides much flexibility. See *Filtering Messages* for a full description.
- The ranking described above corresponds to a message categorization done by GNAT SAS, depending on how interesting and likely to happen they are. See *Categorization of Messages*.

2.1.3 Reviewing the results

Triaging

The first step of a review is to "triage" the message depending on its nature: false positive, intentional violation, or actual bug. Then, depending on the use cases, the review information may be shared with other developers, validated, before the code is fixed if needed. Or you may want to fix the code directly if the error is introduced in your development branch.

See also:

Refer to *Workflows* for example use cases and recommended workflows.

In this example, we'll consider the case where we want to store the review information. We will use the *review through CSV file* method to review the first message described above, which we already identified as being a bug. Following this method we will:

- output messages in CSV format
- add the review information to the CSV file
- tell GNAT SAS to import the review information.

First, let's output only the Infer messages with *high* ranking, this time in a CSV format using `gnatsas report csv`, and store them in a file `messages.csv` with the `-o` switch:

```
$ gnatsas report csv -P uninitialized --show rank=high,tool=infer -o messages.csv
```

`messages.csv` contents is as below, containing exactly one message in this example:

```
project,basename,path,subp,line,column,category,history,ranking,tool,message,cwe,kind,
↪related_checks,key,key_seq,review_from_source,review_kind,review_date,review_status,
↪review_author,review_text
uninitialized.gpr,uninit.adb,uninit.adb,uninit,7,4,check,unchanged,high,Infer,`X.B1` is␣
↪read without initialization during the call to `adt.update`,457,validity check,,
↪6c2d263aed531f7fcb81e1f2c3a864c6,1,false,uncategorized,,,,
```

Now, let's modify the fields, then save the modified file as `reviews.csv`:

- `review_status` with `bug`
- `review_author` with your name (e.g. `Jules Verne`)
- `review_text` with a comment (e.g. `Missing initialization of ADT_Type.B1 field`).

which now contains:

```
project,basename,path,subp,line,column,category,history,ranking,tool,message,cwe,kind,
↪related_checks,key,key_seq,review_from_source,review_kind,review_date,review_status,
↪review_author,review_text
uninitialized.gpr,uninit.adb,uninit.adb,uninit,7,4,check,unchanged,high,Infer,`X.B1` is␣
↪read without initialization during the call to `adt.update`,457,validity check,,
↪6c2d263aed531f7fcb81e1f2c3a864c6,1,false,uncategorized,,bug,Jules Verne,Missing␣
↪initialization of ADT_Type.B1 field
```

Finally, let's inform GNAT SAS about this new review information with the `gnatsas review` command:

```
$ gnatsas review -P uninitialized --from-csv reviews.csv

[gnatsas | review]: reviews imported
```

The *output directory* should now contain a *review file* named `<output_dir>/uninitialized.sar` containing the review information in a GNAT SAS format.

Let's display again the messages (e.g. as text with `--show-reviews`) to confirm that the review has been associated to the corresponding message:

```
$ gnatsas report -P uninitialized --show-reviews
```

```
uninit.adb:7:4: high: validity check [CWE 457] (Infer): `X.B1` is read without␣
↪initialization during the call to `adt.update`. review: 2023-11-30 13:05:42: Bug, Bug,␣
↪approved by Jules Verne: Missing initialization of ADT_Type.B1 field
uninit.adb:7:4: medium: precondition <overflow check> [CWE 190] (Inspector):␣
↪precondition might fail on call to adt.update; requires not X.B1 or X.DZ /= Integer_32
↪'Last
```

As you can see above, the first message now has review information available.

See also:

Refer to *Reviewing Results and Improving Code* for a comprehensive description of the reviewing capabilities in GNAT

SAS.

Fixing the code

Now that we have examined the first message reported and triaged it as a bug with relevant review information, let's fix it by editing the `Initialize` procedure in `adt.adb` to initialize `X.B1`:

```
package body ADT is

   ----------------
   -- Initialise --
   ----------------

   procedure Initialise (X :  out ADT_Type) is
   begin
      X.V1 := 1;
      X.B1 := True;  -- fixed
   end Initialise;
```

Let's run the analysis again and display the results (filtering Infer messages only for the example) to confirm that the issue has been fixed:

```
$ gnatsas analyze -P uninitialized
$ gnatsas report -P uninitialized --show tool=infer
```

```
uninit.adb:7:4: [added] high: validity check [CWE 457] (Infer): `X.DZ` is read without␣
↪initialization during the call to `adt.update`
```

The first message related to the missing initialization of `X.B1` has disappeared from the output and we now have another message related to a similar issue with `X.DZ`, showing as `[added]`.

Comparing runs

Another useful GNAT SAS feature is the capability to compare analysis runs. By default, `gnatsas report` will automatically compare the results of the analysis with the results of the previous run *(of the same timeline, see Timelines for more details)*. This is why the new message was tagged as `[added]`.

We can also force the display of removed messages with `--show age+removed`:

```
$ gnatsas report -P uninitialized --show tool=infer,age+removed --show-reviews
```

The list of messages now shows the previously fixed message as `[removed]`:

```
uninit.adb:7:4: [removed] high: validity check [CWE 457] (Infer): `X.B1` is read without␣
↪initialization during the call to `adt.update`. review: 2023-11-30 13:05:42: Bug, Bug,␣
↪approved by Jules Verne: Missing initialization of ADT_Type.B1 field
uninit.adb:7:4: [added] high: validity check [CWE 457] (Infer): `X.DZ` is read without␣
↪initialization during the call to `adt.update`
```

See also:

- See *Filtering Messages* for a full description of the `--show` switch.
- You may also compare arbitrary runs; refer to *Comparing GNAT SAS Runs* for more details.

2.2 More Resources

- Several other example projects are available:
  - under *<GNAT_SAS_INSTALL_DIR>/share/examples/gnatsas*
  - or via the *Help* → *GNATSAS* → *Examples* menu in GNAT Studio.
- For an advanced tutorial focusing particularly on the use of GNAT SAS in GNAT Studio and on the Inspector analysis engine, refer to the GNAT SAS tutorial available:
  - under *<GNAT_SAS_INSTALL_DIR>/share/doc/gnatsas/tutorial*
  - or via the *Help* → *GNATSAS* → *GNATSAS Tutorial* menu in GNAT Studio.

CHAPTER

THREE

PROJECT SETUP

GNAT SAS requires a GNAT project file (a.k.a. GPR project file) to run. If you are using a third party compiler, this chapter starts with *a quick walk-through* for the creation of an initial gpr file, as well as common configuration for this setup. If you already have a project file that you use to compile with gprbuild, you should look at *the second section* to ensure that it works well with GNAT SAS.

Finally, *the last section* explains how to configure the GNAT SAS analysis and goes on with describing specific configurations of integrated engines.

See also:

Refer to the GNAT Project Manager section of the GPR Tool's User Guide for general documentation about projects.

3.1 Configuration for the GNAT Pro compiler

GNAT SAS is able to analyze code bases built with any compiler and targeting any architecture/runtime. However, GNAT SAS is easiest to use when analyzing a code base built with a GNAT Pro of the same version. When using GNAT SAS to analyze a cross-target project built with an older GNAT Pro version, the GNAT Warnings and Inspector engines may lose precision and analyze less code. The GNATcheck and Infer engines will however be unaffected.

For the analysis of a project that requires building with an older GNAT Pro version (e.g. running an analysis with a GNAT SAS wavefront on a project built with GNAT Pro 21.8), we strongly recommend keeping your project compatible with more recent versions of GNAT Pro.

When keeping projects compatible with more recent GNAT Pro versions is not possible, we recommend keeping said projects compatible with an x86_64-linux or x86_64-windows target and runtime instead.

3.2 Configuration for Third Party Compilers

This section guides you through the typical steps to setup your project in order to use GNAT SAS when you do not use GNAT to compile your sources.

1. First, you need to *create a GPR project file*. In order to do so we recommend using *codepeer-gprname* as described *here*. Indeed, while the resulting project file needs to be updated each time you modify the sources included in the project, *codepeer-gprname* will highlight early problems that the analysis would eventually encounter.

   At this stage, the project file should contain enough information to identify all the sources of the project (if you used *codeper-gprname*, they are listed in the `prj_source_list.txt` file). By default, GNAT SAS will analyze all of them. However, if there are files that you do not wish to analyze, these should be excluded from the analysis in a second stage (see *this section*). Indeed, the project should be as "compilable" as possible for the analysis to perform correctly.

> **Warning:** Your project should not include the sources of your run-time.

2. You should set your target to "gnatsas" with the *Target* attribute:

```
project <project_name> is

  for Target use "gnatsas";

  [...]
```

   You should **not** use the *Runtime* attribute, in particular Section *Using the GNAT Target Runtime Directory* does not apply since it only works with a runtime from GNAT.

3. You may want to specify a *Target Configuration File* in order to provide target dependent values, such as endianness, type sizes, etc.

4. If you use supported compiler-specific packages, you need to set your project file to *use them*.

Once done, you should be able to confirm that the project is properly set up by running a first analysis with the command:

```
gnatsas analyze -P <project_name>.gpr
```

You can then look at how to run the tool *in this section*, and complete your project file by *Configuring the Analysis*.

3.3 Creating a Project File

In this section, we present different ways to create a project file if your project does not yet contain one:

1. Create a project file interactively using the Project Wizard in GNAT Studio;
2. Create a project file manually;
3. Use *codepeer-gprname* to automatically generate it.

We recommend taking the time to follow either option 1 or 2, as the generated project files in option 3 may be hard to understand, modify and hence maintain.

3.3.1 Creating a Project File with GNAT Studio

Please refer to GNAT Studio's documentation to discover how to generate project files from GNAT Studio using the Project Wizard.

3.3.2 Creating a Project File Manually

If your source code follows the default naming scheme (`.ads`/ `.adb`), then you can create a project file that will look like:

```
project My_Project is
   for Source_Dirs use ("./**");
   for Target use "gnatsas";
end My_Project;
```

saved in a file called `my_project.gpr` located in the root of the project tree. Note that you can specify as many source directories as needed as part of the `Source_Dirs` setting. The special syntax `dir/**` represents the directory `dir` and all its subdirectories recursively (in the above example, the current directory represented by `.` and all its subdirectories). Specifying `gnatsas` for the target will allow the analysis engines to find the Ada runtime bundled with GNAT SAS.

3.3.3 Generating a Project File

A project file can be generated from an existing directory structure using the `codepeer-gprname` utility. Whenever a new source file is added to the project or an existing source file is removed, `codepeer-gprname` needs to be re-run to update the project.

Here is the typical usage of `codepeer-gprname`:

```
codepeer-gprname -Pmy_project "-d./**" "*.ad?" "*.spc" "*.bdy"
```

It will create or update a project file called `my_project.gpr` listing all the Ada source files found under the current directory and all its subdirectories, whose name follow the given patterns (all files ending with `.ad<any character>`, all files ending with `.spc` and all files ending with `.bdy`).

Note that `codepeer-gprname` requires a writable temporary directory. If the project file generated is empty, you might need to set the environment variable `TMPDIR` to a writable directory.

See the gprname documentation in the GNAT User's Guide for more details on how to use `gprname`.

3.4 Ensuring compatibility with GNAT SAS

Note: This section assumes that a project file has been defined for the codebase to analyse. If that is not the case, refer to the instructions in *Creating a Project File*.

This section describes how to configure the project file so that:

- specificities of the code to analyze and of the target are taken into account;
- general aspects related to the analysis of Ada packages are considered;
- specific aspects related to GNAT SAS analysis engines that rely on the GNAT compiler are taken into account;
- ultimately, it can be correctly understood by GNAT SAS.

Warning: This section *does not* describe the configuration of the GNAT SAS analysis. That is described in *Configuring the Analysis*.

3.4.1 Using GNAT SAS-Specific Project Settings

If you are using project files also to build with GNAT and need to use e.g. different builder switches between GNAT builds and GNAT SAS analysis, you can then use the predefined `GPR_TOOL` variable to differentiate the two modes in your project file:

```
Tool := External ("GPR_TOOL", "");

package Builder is
   case Tool is
      when "gnatsas" =>
         for Global_Compilation_Switches ("Ada") use ("-gnatI");
         -- -gnatI is only relevant for GNAT SAS
      when others =>
         for Global_Compilation_Switches ("Ada") use ("-O", "-g");
         -- Switches only relevant when building
   end case;
end Builder;
```

In GNAT SAS 24, `GPR_TOOL` is set automatically by GNAT SAS to `gnatsas`, so you do not need any extra configuration or command line switches to use the above.

3.4.2 Support for UTF-8 encoding

GNAT SAS is able to analyze source files that have UTF-8 encoding. Simply add the GNAT switch `-gnatW8` to the *Compiler* package as below. GNAT SAS will report messages from Infer, GNAT Warnings and GNATcheck with UTF-8 encoding. See the Wide_Character Encodings section in the GNAT User's Guide for more details.

```
package Compiler is
   for Switches ("Ada") use ("-gnatW8");
end Compiler;
```

> **Warning:** Concatenation of `-gnatW8` with other switches is not supported. Please specify `-gnatW8` independently from other switches. E.g. use `-gnatiw -gnatW8` instead of `-gnatiwW8`.

Note: For Inspector messages, UTF-8 encoding is not fully supported. UTF-8 characters are converted to a different representation and the GNAT SAS output may become hard to read.

To be able to use identifiers with characters outside of the Latin-1 character set, typically UTF-8 characters, one may also use the `-gnatiw` switch as described below. See the Character Set Control section in the GNAT User's Guide for more details.

```
package Compiler is
   for Switches ("Ada") use ("-gnatW8", "-gnatiw");
end Compiler;
```

3.4.3 Use of Libraries Installed with GNAT

If your project uses some of the libraries installed with GNAT (e.g. GNATcoll, XML/Ada, AWS...) and, when launching GNAT SAS, you encounter a message similar to:

```
gnatcoll.gpr (4 items)
  unknown project file: "gpr"
```

Then add the following section to your project file to use the `"gnatsas"` runtime packaged with the GNAT SAS installation:

```
for Target use "gnatsas";
```

3.4.4 Use of Predefined Ada Packages

The predefined Ada packages (in particular, children of Ada, System, and Interfaces) come packaged with GNAT SAS. Make sure your project file does not include alternative predefined Ada source packages that may be provided by some Ada compilers, since these predefined packages may contain non-portable constructs that GNAT SAS may not recognize.

For compiler-specific packages, see *Use of Compiler-Specific Packages* for accessing some of these. If this isn't sufficient, you will need to include at least the package specs from your target compiler as part of your project file as for any other source file.

3.4.5 Use of Compiler-Specific Packages

If your Ada code base uses compiler-specific packages, GNAT SAS provides some of these packages pre-installed under `install/share/gpr` and accessible as pre-configured project files.

In particular you will find the following project files:

- `aa.gpr`: ApexAda specific packages.
- `am.gpr`: AdaMulti and AdaMagic specific packages.
- `oa.gpr`: ObjectAda specific packages.
- `vads.gpr`: VADS specific packages.
- `xgc.gpr`: XGC specific packages.

To use these project files, add for example the following line to your project file (adjust the name of the imported project as needed):

```
with "am";

project My_Project is
   --  other declarations...
end My_Project;
```

If some compiler-specific packages are missing, do not hesitate to contact your GNAT SAS support.

3.4.6 Using Preprocessing

In order to enable preprocessing, you can use the `-gnateD` and `-gnatep` switches, e.g.:

```
package Compiler is
   for Switches ("Ada") use ("-gnateDVAR1=xxx", "-gnateDVAR2=yyy");
end Compiler;
```

Note the use of the package *Compiler* here. The `-gnateD` switches in the *Analyzer package* are ignored. All the `-gnateD` switches are accumulated to find the defined symbols. If `-gnatep` is used, it should appear only once and the given file is used to specify the preprocessing data file. See also the section Form of Input Text for gnatprep in chapter *Preprocessing with gnatprep* for more details on the syntax supported by the preprocessor.

3.4.7 Arbitrary Naming Scheme

If you are using a different naming scheme than the default `.ads/ .adb` for your source files, you can define the naming scheme adding a package `Naming` if a single and consistent naming scheme is used for all your source files.

For instance:

```
project My_Project is
   ...

   package Naming is
      for Spec_Suffix ("Ada") use ".1.ada";
      for Body_Suffix ("Ada") use ".2.ada";
      for Separate_Suffix use ".2.ada";
      for Dot_Replacement use ".";
   end Naming;
end My_Project;
```

or:

```
project My_Project is
   ...

   package Naming is
      for Spec_Suffix ("ada") use "_.ada";
      for Body_Suffix ("ada") use ".ada";
      for Separate_Suffix use ".ada";
      for Dot_Replacement use "__";
   end Naming;
end My_Project;
```

If more complex naming schemes are used (e.g. files with mixed casing, no automatic mapping between unit names and file names, use of multiple units in a single file, etc...) then you have three options:

- In GNAT Studio: use the *File → Project → Add Complex File Naming Conventions...* menu and follow the dialog. This will call the underlying command line tool `codepeer-gprname` and modify the project file for you. See the GNAT Studio documentation for more details.
- Use the `codepeer-gprname` tool to generate the custom naming scheme automatically. See the section *Generating a Project File* for more details.
- Use the `codepeer-gnatchop` tool:

Your sources can be preprocessed to generate `.ads/.adb` files by running `codepeer-gnatchop` on all your Ada sources and use a simple project file on the generated sources, for instance under Linux:

```
cd <root source directory>
mkdir sources
find . -name sources -prune -o -name '*.ad?' -exec codepeer-gnatchop {}␣
↪sources \;
```

This command will take all files ending with `.ad<any character>` under the current source tree, and generate a corresponding `.ads/.adb` file under a new directory called `sources`. You can then use in your project file:

```
for Source_Dirs use ("sources");
```

3.4.8 Ada Language Version

Note: This section only impacts GNAT-based engines (i.e. Inspector and GNAT Warnings), and has no impact on other engines.

By default, GNAT Warnings and Inspector assume that sources use the Ada 2012 language. If your sources use an older version of Ada, then you will need to add the following to your project file:

```
package Builder is
   for Global_Compilation_Switches ("Ada") use ("-gnat95");
end Builder;
```

The switches corresponding to the various Ada versions are: `-gnat83`, `-gnat95`, `-gnat2005`, `-gnat2012`, `-gnat2022`.

Note that some Ada 95 compilers support the *overriding* keyword but no other Ada 2005 features. In order to analyze such code, you need to add the `-gnatd.D` switch in your project file, in addition to the `-gnat95` switch:

```
package Builder is
   for Global_Compilation_Switches ("Ada") use ("-gnat95", "-gnatd.D");
end Builder;
```

Note: Using `Global_Compilation_Switches` in the `Builder` package should be preferred over defining the switches in the `Compiler` package, since the former applies to the whole project hierarchy, while the latter only applies to the current project and not to subprojects.

3.4.9 Representation Clauses

Note: This section only impacts GNAT-based engines (i.e. Inspector and GNAT Warnings), and has no impact on other engines.

If GNAT Warnings or Inspector generate errors related to Ada representation clauses, a simple way to address these errors is to add the `-gnatI` compiler switch to your project file:

```
package Compiler is
   for Switches ("Ada") use ("-gnatI");
end Compiler;
```

`-gnatI` instructs GNAT Warnings and Inspector to ignore representation clauses, as if they had been stripped from the source code.

3.4.10 Target Configuration File

Usually, using either the default GNAT SAS settings, the GNAT SAS `--bits=32` switch (see *32-bits mode*) or the *Compiler* package Ada's `-gnatI` switch (see *Representation Clauses*), is sufficient to analyze code for most targets.

If you need to configure GNAT SAS to have a deeper knowledge and understanding of the target compiler, and in particular setting target dependent values such as endianness or sizes and alignments of standard types, then you might need to add the following to your project file:

```
package Compiler is
   for Switches ("Ada") use ("-gnateT=" & project'Project_Dir & "/target.atp");
end Compiler;
```

where `target.atp` is a file stored here in the same directory as the project file `my_project.gpr`, which contains the target parametrization. This file can be generated by calling the GNAT compiler for your target with the switch `-gnatet=target.atp`. The format of this file is described in the GNAT User's Guide as part of the `-gnateT` switch description.

Here is an example of a configuration file for a bare board PowerPC 750 processor configured as big-endian:

```
Bits_BE                       1
Bits_Per_Unit                 8
Bits_Per_Word                32
Bytes_BE                      1
Char_Size                     8
Double_Float_Alignment        0
Double_Scalar_Alignment       0
Double_Size                  64
Float_Size                   32
Float_Words_BE                1
Int_Size                     32
Long_Double_Size             64
Long_Long_Long_Size          64
Long_Long_Size               64
Long_Size                    32
Maximum_Alignment            16
Max_Unaligned_Field          64
Pointer_Size                 32
Short_Enums                   0
Short_Size                   16
Strict_Alignment              1
System_Allocator_Alignment    8
Wchar_T_Size                 32
Words_BE                      1

float          6  I  32  32
```

(continues on next page)

(continued from previous page)

```
double         15  I  64  64
long double    15  I  64  64
```

If your target compiler is not GNAT, or an old version of GNAT, then you can also build and run the small utility called `generate_target` which can be found under `<gnatsas install>/share/gnatsas/target`. See comments inside this file for instructions on how to use it.

Note that, depending on the Ada constructs you are using and the values specified in the target configuration file, some combinations will not be properly supported and will lead to errors. In this case, consider using the `-gnatI` switch instead, or contact GNAT SAS support for alternatives.

3.4.11 Using the GNAT Target Runtime Directory

Note: This section only impacts GNAT-based engines (i.e. Inspector and GNAT Warnings), and has no impact on other engines.

As a general rule, we strongly recommend using `for Target use "gnatsas";` in your project file (see *Use of Libraries Installed with GNAT*). If however you want to use a specific GNAT runtime, you can do so by following the explanations given in this section.

Warning: The runtime you intend to use *must* come from a version of GNAT identical to GNATSAS' one. A runtime coming from GNAT 22.2 will *not* be compatible with GNAT SAS 24.2. In such a case you will be forced to specify `for Target use "gnatsas";`

Let's assume you are using GNAT as your target compiler, and explicitly specify a runtime and target to use in your project:

```
for Target use "arm-eabi";
for Runtime ("Ada") use "ravenscar-sfp-stm32f4";
```

Inspector and GNAT Warnings will take such setting into account and will use the GNAT runtime directory, as long as your target compiler is found in your PATH environment variable. As mentioned above, you will need to use a matching version of GNAT and GNAT SAS (e.g. GNAT 24.0 and GNAT SAS 24.0).

The handling of runtimes of Inspector and GNAT Warnings is in fact unified with that of the GNAT compiler. For details, see "GNAT User's Guide Supplement for Cross Platforms", Section 3. If you specify a target, note that Inspector may use additional configuration, see the section *Target Configuration File*.

If for some reason you do not have access to the GNAT compiler on the same machine as GNAT SAS (e.g. the GNAT cross compiler is installed on a different host than GNAT SAS) you will first need to copy the GNAT runtime directory in the GNAT SAS installation, under `<gnatsas-install-dir>/libexec/codepeer/lib/gcc/<gnatsas-platform>/<gnatsas-toolchain-version>/` where *<gnatsas-platform>* is either *x86_64-pc-linux-gnu* or *x86_64-w64-mingw32* and *<gnatsas-toolchain-version>* is the version of GNAT SAS toolchain.

Note: This *does not wave aside* the requirement of having matching versions of GNAT SAS and GNAT.

For example, assuming you want to use a runtime called `ravenscar-sfp-stm32f4`, then first locate this runtime on the machine with the GNAT installation, e.g.:

```
$ arm-eabi-gnatls -v --RTS=ravenscar-sfp-stm32f4 | grep adalib
```

This command gives the path to <ravenscar-sfp-stm32f4 runtime>/adalib.

You then need to copy/transfer the <ravenscar-sfp-stm32f4 runtime> directory to the GNAT SAS installation, under <gnatsas-install-dir>/libexec/codepeer/lib/gcc/<gnatsas-platform>/<gnatsas-toolchain-version>/, for example using *bash* syntax:

```
$ scp -pr $(dirname $(arm-eabi-gnatls -v --RTS=ravenscar-sfp-stm32f4 | grep adalib)) \
  <gnat-sas-machine>:<gnatsas-install-dir>/libexec/codepeer/lib/gcc/<gnatsas-platform>/
↪<gnatsas-toolchain-version>
```

Note: If you don't have access to the GNAT compiler and you copied the GNAT runtime directory in the GNAT SAS installation, you need to use `for Target use "gnatsas";`. You can then take the specificities of your target into account separately as described in *Target Configuration File*.

Working with target-specific configurations

Even when a specific target is used for compilation, we recommend using gnatsas as a target for the analysis when possible. To achieve this, you can either create a dedicated project file for the analysis or use a scenario variable to set different targets for compilation and analysis:

```
case Enable_GNATSAS is
  when "True" =>
    for Target use "gnatsas";
  when "False" =>
    for Target use "x86_64-pc-linux-gnu";
end case;
```

If a target other than gnatsas is necessary for the analysis, be aware that GNAT SAS takes such a target into account, however, the project's `Target` attribute will still be interpreted as having the value `"gnatsas"` when processing the project file in the context of GNAT SAS. This means that the project's `'Target` attribute should not be directly referenced in other attributes (e.g. `for Object_Dir use Project'Target`) in parts of the project file that apply to GNAT SAS.

If the set of sources to analyze depends on the target, you can use a scenario variable to specify the set of sources for GNAT SAS:

```
Tool := External ("GPR_TOOL", "");
Target := project'Target;
Sources := (".");

case Tool is
  when "gnatsas" =>
    Sources := Sources & ("src/common/" & "x86_64-pc-linux-gnu");
  when others =>
    Sources := Sources & ("src/common/" & Target);
end case;

for Source_Dirs use Sources;
```

If you want to analyze a codebase with different targets and each target has specific sources, you can define a scenario variable defining such targets and use it instead of using `Target` directly. In this case, `--subdirs` or `for Subdirs` should be used to separate each target-specific analysis by creating a separate *gnatsas directory* for it. For example:

```
Tool := External ("GPR_TOOL", "");
type Gnatsas_Target_Type is ("x86_64-pc-linux-gnu", "aarch64-linux-gnu");
Gnatsas_Target : Gnatsas_Target_Type :=
  external("GNATSAS_TARGET", "x86_64-pc-linux-gnu");

Target := project'Target;

Sources := (".");

case Tool is
  when "gnatsas" =>
    Sources := Sources & ("src/common/" & Gnatsas_Target);
  when others =>
    Sources := Sources & ("src/common/" & Target);
end case;

for Source_Dirs use Sources;

package Analyzer is
  for Subdirs use Gnatsas_Target; -- will result in a gnatsas directory
                                  --  located in
                                   --  "<obj_dir>/<Gnatsas_Target>/gnatsas"
end Analyzer;
```

3.4.12 Configuration of System Package (system.ads)

Note: This section only impacts GNAT-based engines (i.e. Inspector and GNAT Warnings), and has no impact on other engines.

Some Ada compilers provide additional, non-portable definitions in the predefined package `System` which might lead to errors when running Inspector and GNAT Warnings analyses.

A solution to address this issue automatically is to use the `pragma Extend_System` in a pragma configuration file, either using one of the predefined GNAT extensions (e.g. `Aux_Dec` for the DEC Ada compiler or `VADS` for the VADS compiler, available via the project file `vads.gpr` as described in *Use of Compiler-Specific Packages*), or by providing a customized `s-auxcom.ads` file.

Note: This technique doesn't allow overriding existing definitions in the predefined `System` package. If the definitions in the `System` package are incompatible with the analyzed source code, make sure that the correct *runtime* is used. An alternative solution is to create new definitions in another package and use these definitions instead of the definitions in the `System` package. This solution, however, implies modifications to the analyzed code.

For example, you can create a file called `gnatsas.adc` located under the same directory where your project file is, which will contain:

```
pragma Extend_System (Aux_Dec);
```

or alternatively:

```
pragma Extend_System (Aux_VADS);
```

And configure your project file to take this file into account:

```
package Builder is
   for Global_Configuration_Pragmas use "gnatsas.adc";
end Builder;
```

or if using `Aux_VADS`:

```
with "vads";
project My_Project is

   package Builder is
      for Global_Configuration_Pragmas use "gnatsas.adc";
   end Builder;

end My_Project;
```

Similarly, if your target compiler provides additional definitions, you can manually create a `s-auxcom.ads` file containing these additional definitions, e.g.:

```
package System.Aux_Compiler is
   pragma Preelaborate;

   function To_Integer (Addr : Address) return Integer;
   function To_Address (Int  : Integer) return Address;

   No_Addr      : constant Address := Null_Address;
   Address_Zero : constant Address := Null_Address;

   Assertion_Error : exception;

   ...
end System.Aux_Compiler;
```

and then specify:

```
pragma Extend_System (Aux_Compiler);
```

in `gnatsas.adc`.

See the GNAT Reference Manual for more details on `Pragma Extend_System`.

Also note that you should never include the compiler System package in your project file (either in your source file list or even in your source directories), as Inspector needs to use its own version of `system.ads`.

3.4.13 Ignoring Pragmas

Note: This section only impacts GNAT-based engines (i.e. Inspector and GNAT Warnings), and has no impact on other engines.

In some cases, non-portable pragmas are used across Ada technologies in an incompatible way, generating some spurious error messages when Inspector or GNAT Warnings analyze these files. In order to address these issues, you can configure your project file to ignore a set of pragmas via the `Ignore_Pragma` pragma that you should place in a `gnatsas.adc` file as explained in *Configuration of System Package (system.ads)*. For example, if you want to ignore pragmas called `Import_Procedure` and `Import_Function` you can put in your `gnatsas.adc`:

```
pragma Ignore_Pragma (Import_Procedure);
pragma Ignore_Pragma (Import_Function);
```

Similarly if you want to ignore pragma `Global`:

```
pragma Ignore_Pragma (Global);
```

If not already done, you need to configure your project file to take this file into account:

```
package Builder is
   for Global_Configuration_Pragmas use "gnatsas.adc";
end Builder;
```

3.4.14 Providing Stubs for Missing Generic Bodies

Note: This section only impacts GNAT-based engines (i.e. Inspector and GNAT Warnings), and has no impact on other engines.

GNAT Warnings and Inspector will ignore missing bodies and will make reasonable presumptions about calls to such unknown subprograms (see *Inspector Presumptions*).

However they will generate an error when they encounter a generic package whose body is missing at instantiation time. In order to work around this error, you can provide a skeleton for the generic package via the `codepeer-gnatstub` utility, e.g.:

```
codepeer-gnatstub -Pmy_project generic_package.ads
```

will generate a file `generic_package.adb` which can be used by GNAT Warnings and Inspector.

3.5 Configuring the Analysis

Note: This section assumes that a project file has been defined for the codebase to analyse and configured to be compatible with GNAT SAS. If that is not the case, refer to the instructions in *Creating a Project File* and *Ensuring compatibility with GNAT SAS*.

This section describes how the GNAT SAS analysis can be configured by users to adjust the scope of the analysis, its precision, control where files are generated, etc. It also explains the available configuration for each integrated analysis engine.

All analysis engines are enabled by default. It is possible to individually enable or disable any engine (see *gnatsas analyze switches*).

3.5.1 *Analyzer* package attributes

In addition to the project attributes described in the GPRbuild and GPR Companion Tools User's Guide and described in *Project Setup* (e.g. `Source_Dirs`, `Object_Dir` attributes, *Naming* and *IDE* packages, ...), the optional project attributes described below are available in the *Analyzer package*.

- `Subdirs`

  Specify the relative path from the *object directory* (if specified, project directory otherwise) to the *gnatsas directory*. See *subdirs*.

- `Output_Dir`

  Specify the *output directory* to use for this project. If you specify a relative directory, that directory is interpreted as relative to the directory that contains the project file.

- `Review_File`

  The *review file* to use for this project. If you specify a relative path, that path is interpreted as relative to the directory that contains the project file.

- `Excluded_Source_Files`

  List of project source files (as base names) which should be excluded from GNAT SAS' analyses. See *Excluding Files from Analysis* for more info.

- `Excluded_Source_Dirs`

  List of project source directories which should be excluded from GNAT SAS' analyses. See *Excluding Directories from Analysis* for more info.

- `Switches`

  An associative array representing the additional switches that should be used for each GNAT SAS subcommand and analysis engines. Possible keys are:

  - `"analyze"`,
  - `"inspector"`, `"infer"`, `"gnat"` and `"gnatcheck"` (see the warning below),
  - `"report X"`, where X can be any format supported by the `gnatsas report` command (e.g. `"report text"` or `"report csv"`),
  - `"baseline"`.

  Warning: Using the `Switches` attribute within the *Analyzer package* to configure GNATcheck or GNAT only affects these tools when they are launched through GNAT SAS. These settings are not recognized when the tools are run standalone.

  Consider whether a configuration needs to apply systematically to the tool in all contexts. If so, the setup should be placed in the corresponding package: use the `Check` package for GNATcheck rules, and the `Compiler` or `Builder` packages for GNAT Warnings. **If the tool is not intended for standalone use, we recommend configuring the corresponding package.** Refer to *Configuring GNATcheck* and *Configuring GNAT Warnings* for more details on their configuration.

```
project Prj is
   package Analyzer is
      -- Applied only when launched through gnatsas
      for Switches ("gnat") use ("-gnatwa");
   end Analyzer;

   package Check is
      -- systematically applied (standalone and gnatsas)
      for Default_Switches ("ada") use ("--rule", "Local_Packages");
   end Check;

   package Compiler is
      -- systematically applied (standalone and gnatsas)
      for Default_Switches ("ada") use ("-gnat2022");
   end Compiler;
end Prj;
```

- `Pending_Status`, `Not_A_Bug_Status`, `Bug_Status`

  Lists of custom review status definitions, see *Custom Review Status*.

  See also:

  Refer to the *GNAT SAS CLI Reference* for more information on supported switches.

- `Trusted_Taint_Sources`, `Distrusted_Taint_Sources`

  Lists of *Kinds of Sources* that should be trusted or distrusted for *Taint Analysis*. All taint sources are distrusted by default.

  See also:

  Trusting Sources has examples describing how to use these attributes.

32-bits mode

By default GNAT SAS assumes that the compilation target is the same as the host on which it is run. This is often a sufficient default, although one typical case where this is not sufficient is when using GNAT SAS in 64-bits mode and analyzing code designed for 32-bits architectures. In this case, you can run GNAT SAS with the `--bits=32` switch so that GNAT SAS will analyze the code assuming the same architecture, but in 32-bits mode instead of 64-bits. You can use this switch either from the command line, or specify it in the project file:

```
package Analyzer is
   for Switches ("analyze") use ("--bits=32");
end Analyzer;
```

See also *Target Configuration File* for advanced configuration of the target.

Warning: This is currently not supported for Infer and GNATcheck.

3.5.2 Partial Analysis

This section describes how to perform a partial analysis, and in particular, how to exclude entities (projects, directories, files, packages, subprograms) from analysis.

Selecting the Analysis Scope

GNAT SAS provides several switches allowing users to control the analysis scope, by passing them to the `gnatsas analyze` command line or specifying them in the project file:

- The default behavior when none of the switches below are specified is to analyze the closure of the main units specified in the project file for the full project tree, except externally built units.
- `-U`: analyze all files in the full project tree, except externally built ones.
- `-U FILE`: analyze files contained in the closure of the FILE unit.
- `--no-subprojects`: analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.
- `--file FILE`: analyze a single file FILE.
- `--files-from FILE`: analyze files listed in FILE.

See also:

Refer to *GNAT SAS CLI Reference* for more information about GNAT SAS command-line usage.

In addition, it is possible to exclude entities from the analysis as explained in the following sections.

Excluding Subprograms or Packages from Analysis

Note: This section only applies to Infer and Inspector engines.

To selectively disable the analysis of a particular subprogram or set of subprograms for Infer and Inspector, you can add in your Ada source code a pragma Annotate of the form:

```
pragma Annotate (GNATSAS, Skip_Analysis);
```

When used in this way, an Annotate pragma takes exactly these two arguments. The pragma may be placed at the beginning (possibly preceded only by different Annotate pragmas) of the declaration list of a subprogram body, a package body, or package visible part.

In the case of a subprogram body, Infer and Inspector's analysis of that subprogram body (and anything declared therein, including nested subprograms) will not be performed. In the case of a package body, Infer and Inspector's analysis of any subprogram body occurring (directly or indirectly) within the package body will not be performed. In the case of a package visible part, analysis of subprogram bodies occurring in the package specification (as well as the package body) will not be performed; note that a subprogram body can occur in a package specification as part of an instantiation of a generic unit.

The subprogram or package in question may be a generic unit or may occur within a generic unit; in this case, the pragma will be replicated (as described in Ada RM 12.3(13)) when an instance of the generic unit is declared and will affect Inspector's analysis of the instance as described in this section.

When an Annotate pragma is used in this way to prevent Inspector and Infer's analysis of a subprogram body, the analysis of clients of the subprogram will be affected in the same way as if the body of the subprogram were unavailable for some other reason (e.g. a missing .adb file).

Here is an example of usage to disable analysis of a specific subprogram:

```
procedure Complex_Subprogram (...) is
   pragma Annotate (GNATSAS, Skip_Analysis);
begin
   ...
end Complex_Subprogram;
```

And similarly for a package:

```
package Complex_Package is
   pragma Annotate (GNATSAS, Skip_Analysis);

   ...
end Complex_Package;
```

Excluding Files from Analysis

If you want to exclude some files from the analysis (because e.g. GNAT SAS is taking lots of time analyzing them, and these files do not contain useful subprograms for GNAT SAS to analyze), you can use the `Excluded_Source_Files` project attribute in the *Analyzer package*, for example:

```
package Analyzer is
   for Excluded_Source_Files use ("xxx.adb");
   -- Analysis of xxx.adb generates lots of timeouts, skip it for now
end Analyzer;
```

For the values, ensure to **always specify the basename** of the excluded files, with no path information. The path information is computed automatically by GNAT SAS based on other project properties.

Warning: Project attributes in the *Analyzer package* (in particular `Excluded_Source_Dirs` and `Excluded_Source_Files`) are ignored when executing **GNATcheck or GNAT Warnings as standalone tools**.

Note: Note that excluding a specification file from the analysis also excludes the corresponding body file, if present. On the contrary, excluding a body file does not exclude the corresponding specification file.

Inspector-specific handling

The Inspector engine has some limitations when it comes to excluding files from the analysis. In the general case, exclusion can be done using the *Analyzer* package as previously described. However, in some specific cases file exclusion needs to be done differently. This section describes such cases and their impact.

Excluding a file containing a generic

When a file excluded from the analysis contains a generic, the generic will still be analyzed by Inspector if it is instantiated in at least one file that is not excluded. This is because Inspector doesn't analyze generics per se, but analyzes each instantiation of the generic as explained in *Handling of Generic Units*.

In such case, it is possible to exclude all instantiations of a generic package from the analysis as described in *Excluding Subprograms or Packages from Analysis*. Inspector will no longer attempt to analyze the excluded file (note, however, that it will still compile it and generate a SCIL for it; if that is not desirable, see the next paragraph).

Excluding a file from compilation and SCIL generation

If you want to not only exclude files from the analysis, but go further and also exclude it from compilation and SCIL generation; and if these files are not in the transitive closure of other files (e.g. a body file, as opposed to a specification file used by other Ada units); then the project-level `Excluded_Source_Files` attribute can be used instead of the *Analyzer* attribute:

```
project My_Project is
   for Excluded_Source_Files use ("xxx.adb");
end My_Project;
```

Warning:

- Unlike excluding a body file, excluding a specification file at the project level **may cause compilation errors** on the files depending on this spec and as a result prevent the analysis of those files.
- If a specification or a body file of a module is excluded at the project level, Inspector will not be able to perform an incremental analysis of the modules depending on that module. This means that **those modules will be fully re-analyzed** by Inspector (even in fast mode).

Those effects do not occur if files are excluded using the *Analyzer* package.

See also:

This mechanism may also be used to address some compilation errors as explained in *Compilation Errors*.

Excluding Directories from Analysis

If you want to exclude from analysis all source files from a given set of directories, you can use the `Excluded_Source_Dirs` project attribute in the *Analyzer package*, for example:

```
package Analyzer is
   for Excluded_Source_Dirs use ("directory1", "directory2");
end Analyzer;
```

Warning: Project attributes in the *Analyzer package* (in particular `Excluded_Source_Dirs` and `Excluded_Source_Files`) are ignored when executing **GNATcheck or GNAT Warnings as standalone tools**.

You can specify absolute directories or directories relative to the location of the project file (same as for source directories).

As explained in the section *Excluding Files from Analysis*, generic instantiations made in files not excluded from the analysis will be still analyzed. Note that it is possible to exclude all instantiations of a generic package from the analysis as described in *Excluding Subprograms or Packages from Analysis*.

Excluding Projects from Analysis

If you have a tree of project files (`.gpr`) and want to exclude some entire projects (aka subsystems) from GNAT SAS analysis, you can do it by adding the following project attributes to each of the projects involved:

```
for Externally_Built use "True";
```

This attribute tells GNAT SAS to disable analysis on all source files for this project.

Projects using files (e.g. spec) from this project will still be analyzed, as well as generic instantiations made in other projects.

3.5.3 Configuring GNATcheck

GNAT SAS uses a small selection of GNATcheck rules by default as listed in *GNATcheck Messages*. GNATcheck is enabled by default and can be disabled with the `--no-gnatcheck` switch. More rules can be checked by configuring the *Check* package of the project file.

Note: The default rules will not be enabled by GNAT SAS if any of the following is true:

- an LKQL rule file is specified in the *Check* package (through the `Rule_File` attribute)
- at least one rule is specified in the *Check* package (through the `Rule` attribute or the `-rules` switch)
- a default `rules.lkql` file exists in the directory of the project file.

In those cases the tool emit the following messages:

```
[gnatsas]: found GNATcheck configuration in Check package, disabling default GNATcheck␣
↪rules
```

See also:

For more detailed information about configuring GNATcheck, see the GNATcheck Reference Manual.

Configuring the `Check` package within the project file ensures that the GNATcheck setup is applied consistently, whether the tool is invoked via GNAT SAS or as a standalone tool. Users should determine if a configuration needs to apply systematically to GNATcheck; in such cases, the settings should be defined in the `Check` package rather than the `Analyzer` package (with the `Switches` attribute, see below), as the latter only affects GNAT SAS execution. **If GNATcheck is not intended for standalone use, we recommend configuring the `Check` package.**

```
project Prj is
   package Analyzer is
      -- Applied only when launched through gnatsas
      for Switches ("gnatcheck") use ("--rule-file", "gnatsas_rules.lkql");
   end Analyzer;

   package Check is
      -- Applies to GNATcheck in all contexts (standalone and gnatsas)
      for Default_Switches ("ada") use ("--rules-dir", "<path_to_gnatcheck_rule_
```

(continues on next page)

(continued from previous page)

```
↪directory>");
   end Check;
end Prj;
```

Some examples of configuration are provided below.

Differences with standalone GNATcheck tool

There exist noteworthy differences between using GNATcheck integrated in GNAT SAS and using GNATcheck as a standalone tool.

Note: In GNAT Studio, GNAT SAS with GNATcheck is available from the top-menu *GNATSAS -> Analyze...* (see *GNAT SAS menu*), while GNATcheck standalone is available from the menu *Analyze -> Check Coding Standard*.

In particular:

- GNAT SAS runs five GNATcheck rules by default (see *GNATcheck Messages*), while standalone GNATcheck runs none.
- **The output of both tools is not stored and displayed in the same manner**:
  - Standalone GNATcheck's default output directory is the directory specified by the `Object_Dir` of the project, or the current directory if no `Object_Dir` is specified. GNATcheck will generate a text report `gnatcheck.out` and other files under that directory. This behavior can be overridden with multiple switches defined in the *Check* package, such as `--subdirs` or `-o`. See the GNATcheck Reference Manual. for more details.
  - However, the results of the GNATcheck analysis integrated in GNAT SAS will be generated in the *output directory* with the same format as messages from other engines (*Message Files*) and can be displayed in multiple ways (see *Viewing Results*). Internal GNATcheck output is also available under the `gnatsas/<prj>.gnatcheck` directory. Other settings from the *Check* package will be ignored.
- **The set of sources analyzed by GNATcheck may differ** (specified as `"analyze"` key switches in the *Analyzer package* or GNAT SAS command-line: `-U`, `--file`, `--files-from`, `--no-subprojects``or ``for Source_Files`; and specified as values of `for Default_Switches ("ada")`: `-U`, `-files=`, `--no-subprojects` in the *Check* package).
  - When running standalone GNATcheck, above settings from the *Analyzer package* will be ignored.
  - When running GNATcheck integrated in GNAT SAS, above settings from the *Check* package will be ignored.
- Projects marked as externally built with the `Externally_Built` project attribute will not be analyzed when GNATcheck is ran through GNAT SAS.
- The `Annotate` pragmas for GNAT SAS will only be taken into account when running GNATcheck through GNAT SAS. See *Improve Your Code Specification*.
- The `-j` switch for controlling the number of jobs, can be specified through `for Switches ("analyze")` in the *Analyzer package*, which will override the settings set through `for Default_Switches ("ada")` in the *Check* package.

3.5.4 Configuring Infer

GNAT SAS runs some analyses based on Infer by default. Enabled by default, can be disabled with `--no-infer`.

Some Infer analyses can be activated and/or parameterized using Infer-specific switches. These switches can be specified using the index *"infer"* of the project attribute *Switches* in the *Analyzer package*:

```
project My_Project is
   package Analyzer is
      for Switches ("infer") use ("--side-effects");
   end Analyzer;
end My_Project;
```

This example will enable emitting a warning when a function has side effects.

The section *Infer Messages* contains a list of all Infer messages, with their description and detailed configuration parameters.

Infer also performs *Taint Analysis* by default. It can be disabled with `--no-taint` and re-enabled with `--taint`.

3.5.5 Configuring GNAT Warnings

The GNAT front-end warnings. Enabled by default, can be disabled with `--no-gnat`.

Specifying warnings to check

When enabled, GNAT SAS invokes the GNAT front-end using the configuration defined in the `Compiler` and `Builder` packages of the gpr file, alongside a default selection of checks managed by GNAT SAS (see the *table listing GNAT Warnings*).

To modify this selection specifically for GNAT SAS, additional switches can be added to the *Analyzer package*. However, these settings only apply when the tools are launched via GNAT SAS. If the warning configuration should apply systematically—including during standard compilation—it is recommended to define them in the `Compiler` packages instead.

For example:

```
project Prj is
   package Analyzer is
      -- These switches only apply during gnatsas analysis
      for Switches ("gnat") use ("-gnatwbz.a");

      -- To disable warnings in gnatsas:
      -- for Switches ("gnat") use ("-gnatwBZ.A");
   end Analyzer;

   package Compiler is
      -- Recommended for consistent warnings during both build and analysis
      for Default_Switches ("ada") use ("-gnatwa");
   end Compiler;
end Prj;
```

Ignoring Source File Timestamp Mismatch

If your environment changes the timestamps of your source files between GNAT SAS runs (with no actual changes in source files contents), then in order to minimize the regeneration of Inspector SCIL files or recomputation of GNAT Warnings, you can specify the `--incrementality-method=checksums` switch in the *Analyzer package* of your project file, so that checksums rather than timestamps should be used for determining whether source files have changed since the previous run:

```
package Analyzer is
   for Switches ("analyze") use ("--incrementality-method=checksums");
end Analyzer;
```

Note: This switch only impacts GNAT based engines, i.e., Inspector and GNAT Warnings.

Warning: When incrementality is based on checksums, some messages may end up referring to wrong source locations. Indeed:

Incrementality based on checksums internally relies on GPRbuild checksum computation. GPRbuild ignores comments when computing the checksum of a file. Hence, a file were only comments are modified between two gnatsas runs is not re-analyzed. While the messages on the file are still valid, they refer to source locations prior to the comments modifications.

3.5.6 Configuring Inspector

Inspector is the historical analysis engine of GNAT SAS. Enabled by default, can be disabled with `--no-inspector`.

See also:

See *Inspector Reference* for more advanced Inspector configuration.

Example project file

Here is an example of a project file using some of the Inspector attributes described in the following sections:

```
project Prj1 is
   ...

   package Analyzer is
      for Excluded_Source_Files use ("file1.ads", "file2.adb");
      for Output_Dir use "/work/project1.outputs";
      for Switches ("analyze") use ("--keep-going");
      for Switches ("report text") use ("--show-backtraces");
      for Additional_Patterns use "ExtraMessagePatterns.xml";
   end Analyzer;
end Prj1;
```

Analyzer package attributes

- `Message_Patterns`

  Alternate `MessagePatterns.xml` file that Inspector should use for this project.

- `Additional_Patterns`

  Extra `MessagePatterns.xml` file that Inspector should use in addition to the default patterns file. This can be either a path relative to the project's directory, or an absolute path.

Ignoring Source File Timestamp Mismatch

See *Ignoring Source File Timestamp Mismatch*.

Ignoring Exception Handlers

By default Inspector tries to take exception handlers into account in its analysis, although by doing so this can lead to a loss of precision in the detection of possible (or certain) run-time errors. If you want Inspector to analyze the code by completely ignoring exception handlers (behave as if the exception handlers were stripped from the code) then you can add the `-gnatd.x` switch, e.g.:

```
package Analyzer is
   for Switches ("inspector") use ("-gnatd.x");
end Analyzer;
```

Handling Enumeration Representation Clauses

Even without `-gnatI`, Inspector by default ignores enumeration representation clauses, to reduce the various implicit representation transformations required by such clauses, which can produce confusing output. If your application uses Enum_Rep, Enum_Val, or Unchecked_Conversion to manipulate the underlying codes specified by an enumeration representation clause, then you may want to override this Inspector default by using the compiler switch `-gnatd.I`. This will cause Inspector to obey the enumeration representation clause and ensure that Enum_Rep, Enum_Val, and Unchecked_Conversion are interpreted correctly for this enumeration type. This switch can be added to your project file, as follows:

```
package Analyzer is
   for Switches ("inspector") use ("-gnatd.I");
end Analyzer;
```

Detection of Floating Point Overflow

Inspector assumes that floating point operations are carried out in single precision (binary32) or double precision (binary64) as defined in the IEEE-754 standard for floating point arithmetic. You should make sure that this is the case on your platform. For example, on x86 platforms, by default some intermediate computations may be carried out in extended precision, leading to unexpected results. With GNAT, you can specify the use of SSE arithmetic by using the compilation switches "-msse2 -mfpmath=sse" which cause all arithmetic to be done using the SSE instruction set which only provides 32-bit and 64-bit IEEE types, and does not provide extended precision. SSE arithmetic is also more efficient. Note that the ABI allows free mixing of units using the two types of floating-point, so it is not necessary to force all units in a program to use SSE arithmetic.

Inspector considers the floating point values which represent positive, negative infinity or NaN as invalid, and it checks that such values cannot occur.

By default, Inspector will only detect potential overflows on floating point operations on constrained floating point types (user floating point types with explicit ranges) and not on unconstrained types (e.g. predefined float types).

You can ask Inspector to check for possible floating point overflows on unconstrained types (which will then lead to infinite values) via the `-gnateF` compiler switch, e.g.:

```
package Compiler is
   for Switches ("Ada") use ("-gnateF");
end Compiler;
```

CHAPTER

FOUR

ANALYZING CODE

Note: This section assumes that a project file exists for your codebase and has been configured to be compatible with GNAT SAS according to *Project Setup*. If there is no project file defined, follow *Creating a Project File*.

This section describes how to run an analysis with GNAT SAS and presents in more details some analysis features such as *Analysis modes* and *Timelines*.

It also provides recommendations to get the best out of the analysis depending on your needs, see *Getting the Right GNAT SAS Settings*.

See also:

The analysis can be configured in various ways (analysis scope, precision, *output directory*...). This is described in details in *Configuring the Analysis*.

4.1 Running GNAT SAS Analysis

4.1.1 Running GNAT SAS from the Command Line

GNAT SAS can be run from the command line, when doing simple experiments, development or for running jobs automatically. A typical invocation of GNAT SAS will look like:

```
$ gnatsas analyze -Pmy_project
```

Once done, GNAT SAS generates a *SAM file* that contains the results of the analysis (see *GNAT SAS Files Reference*). You can then view the results in multiple formats or IDEs (e.g. GNAT Studio, a Web browser, a SARIF viewer...), or directly in text format from your terminal with:

```
$ gnatsas report -Pmy_project
alias.adb:10:11: high: divide by zero fails here: requires Page_Size /= 0
```

See also:

- See *GNAT SAS CLI Reference* for more details about the GNAT SAS command line.
- See *Viewing Results* for more details about the ways to view the GNAT SAS output.

4.1.2 Running GNAT SAS from GNAT Studio

GNAT SAS can also be run from GNAT Studio where it is fully integrated. See *Running GNAT SAS Analysis*.

4.2 Analysis modes

You can vary the thorougness of GNAT SAS' analysis by changing its *mode*. Two modes are available, and can be selected from the command-line or from GNAT Studio: *fast* and *deep*. Analyses are done in fast mode by default.

Command-line examples:

Fast mode

```
$ gnatsas analyze -Pmy_project
or
$ gnatsas analyze -Pmy_project --mode fast
```

Deep mode

```
$ gnatsas analyze -Pmy_project --mode deep
```

For selecting the mode in GNAT Studio, see *GNAT SAS menu*.

We recommend developers run fast mode, as it offers the best precision/performance trade-off. Also, incrementality is very important to get fast analysis times e.g. when running from GNAT Studio. Deep mode should be reserved when a more precise analysis is meaningful, e.g. during nightly testing.

Note: Note that changing the mode currently only impacts Inspector's results and has no impact on other engines. The impact on Inspector is as below.

- In fast mode, Inspector analyzes each library unit separately. This allows Inspector's analysis to be *incremental*: when re-analyzing a project, only the units that have changed will need to be re-inspected.
- In deep mode, Inspector performs the analysis by group of units, whose size depend on factors depending in *Partitioning of Analysis*. The analysis will start from scratch each time, and will be more thorough overall.

Importantly, for each analysis mode, GNAT SAS generates a separate baseline. Each new run in a given mode is automatically compared to the associated baseline. The implications are described in the next section.

4.3 Timelines

The fact that GNAT SAS uses a separate baseline for each *analysis mode*, and is able to compare runs in a given mode to the associated baseline, is due to the existence of *timelines*. A timeline is exactly that: a name associated to a baseline run, and the automatic comparison of each new run to that baseline. As such, timelines are separate from one another.

Both fast and deep modes, as well as combinations of an analysis mode and switches that have an influence on the scope of the analysis (detailed below), are automatically associated to a separate timeline.

More specifically, the timeline name is computed as follows:

- The timeline for the default mode (fast mode) of a given project `MyProject.gpr` when none of the switches below are specified is `"fast"`, and the timeline of a deep analysis is `"deep"`.
- with `--file FILE_NAME` or `--file FILE_PATH`: `"<FILE_NAME>[.deep]"`

- with `--files-from FILE_NAME` or `--files-from FILE_PATH`: "<FILE_NAME>[.deep]"
- with `-U`: "_U[.deep]"
- with `-U FILE_NAME` or `-U FILE_PATH`: "_U_<FILE_NAME>[.deep]"
- with `--no-subprojects`: "_no_sub[.deep]"

where `.deep` is appended in deep mode.

See also:

- See *GNAT SAS CLI Reference* for more details about the GNAT SAS command line and supported switches.
- See also *Migrating Away from the Historical CodePeer Database* for timelines generated by a CodePeer database importation.

4.3.1 Using custom timelines

In some cases, it may be desirable to manipulate specific timelines different from the default generated ones. For example:

- When you want to perform an analysis in a given mode, but due to the settings large discrepancies are expected. Typically, when using different scenario variables values in the project file. As an example, consider a case where you want to have a finer control over the scope of the analysis (typically excluding some directories or projects), and you do not want to mix the results with the results from an analysis of the whole project tree. The analysis can be run on the whole project tree in one timeline, and then on a more restricted project tree in another.
- You may also want to do that in order to later merge otherwise separated timelines (e.g. fast and deep mode).

To support such use cases, GNAT SAS provides the capability to create and use new timelines with the `--timeline` switch.

The `--timeline <name>` switch can be set to the `gnatsas analyze` subcommand, with the following effects:

- The first analysis of a timeline becomes the baseline of the said timeline *(unless a baseline was specifically set prior to it with* `gnatsas baseline --timeline <name> --set-baseline <baseline>` *, see Comparing GNAT SAS Runs)*.
- All analyses are automatically compared to the timeline's baseline.

The `--timeline <name>` switch can be set for the `gnatsas report` subcommand too. The effect is to display the results of the specified timeline's last run (instead of the actual last run's results).

The `--timeline <name>` switch can be specified in the `gnatsas baseline` subcommand in order to manipulate the specified timeline's baseline (see *Comparing GNAT SAS Runs*), or current run (see use cases in *Importing GNAT SAS Results* and *sam-from-db*).

4.3.2 Displaying timeline information

Information about available timelines can be displayed using the `--list-timelines` switch. This switch will display all timelines that were automatically created by the tool or explicitly specified by users for the specified project.

For example:

```
$ gnatsas analyze -Pprj
$ gnatsas analyze -Pprj --mode=deep
$ gnatsas analyze -Pprj --list-timelines
```

will display the list of timelines (fast, deep) and related information, such as the date of last run, command-line and project switches used for the analysis, paths to SAM file and baseline file.

See also:

See *GNAT SAS CLI Reference* for more details about the GNAT SAS command line and supported switches.

4.4 Getting the Right GNAT SAS Settings

We recommend that you first get familiar with *System Requirements* for a proper recommended configuration to run GNAT SAS.

4.4.1 Analyze Messages

If a given run contains many messages, start analyzing them and identify if there are groups of uninteresting or incorrect messages. You can disable a category of messages producing uninteresting results in text output using the `--show` switch (see *Filtering Messages* for more details).

For example if messages related to *access checks* are not producing interesting results in your code base, you can disable these messages in text output using `--show kind-access_check` with `gnatsas report`.

If many uninteresting or incorrect messages are issued for a file, you can exclude this file from analysis via the `Analyzer'Excluded_Source_Files` project attribute or via `pragma Annotate (GNATSAS, Skip_Analysis)` as explained in *Partial Analysis*.

In the case of taint error messages, you may try *Trusting Sources* in order to focus on the taint errors you truly care about.

Also choose the relevant messages based on ranking: GNAT SAS ranks messages in categories depending on how interesting and likely to happen they are (see *Categorization of Messages*).

While analyzing existing code, you should start looking at *High* messages first, then *Medium*, and finally if it makes sense, *Low* messages.

A recommended setting is to consider *High* and *Medium* messages, which is what is displayed by default in the GNAT Studio report.

If the run is mostly clean or contains mainly interesting messages, then try to see if there are messages disabled by default which may be interesting in your context. These are some *Infer Messages*, some *GNAT Warnings Messages*, some *GNATcheck Messages*, and *Race Condition Messages*.

See also:

During this analysis of the results, you may want to start triaging and reviewing some messages. Refer to *Reviewing Results and Improving Code* for more details about the review process.

4.4.2 Run GNAT SAS Faster

If runs are too slow for your needs (e.g. quick user feedback as opposed to nightly runs) then you should consider disabling some analysis engines (see *gnatsas analyze switches*), making sure you are using the `-j0` switch (enabled by default unless you are using another `-jxx` switch explicitly) and using a machine with as many cores as possible as well as using local fast drives, as explained in *System Requirements*.

If this isn't sufficient, or you don't want to disable an analysis engine that takes a lot of time, the next step is to identify the files that are taking a long time to analyze and exclude them from the analysis as per *Partial Analysis*.

Identifying files that take a long time to analyze by Inspector

To identify the files that are taking a long time to analyze by Inspector, you can save GNAT SAS' default output and sort it on the 4th column (timing info):

```
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 example using the sort utility:

```
$ gnatsas analyze -Pprj |& grep "^analyzed " | tee gnatsas-output.txt
sort -r -g -k 4 gnatsas-output.txt
```

You'll get:

```
analyzed main__body.scil in 620.31 seconds
analyzed pack1__body.scil in 20.02 seconds
analyzed pack2__body.scil in 5.13 seconds
analyzed main.scil in 0.05 seconds
```

In this simple example, skipping the analysis of the body of unit main (e.g. `main.adb`) will produce a 25 seconds analysis instead of taking more than 10 minutes.

If it is not acceptable to exclude the whole file from the analysis, it is possible to use the switch `--dbg-on dbg_slow_analyses` to get Inspector timings for individual subprograms and *exclude only particular subprograms from the analysis*, like this:

```
package Analyzer is
   for Switches ("inspector") use ("--dbg-on", "dbg_slow_analyses");
end Analyzer;
```

Identifying files that take a long time to analyze by Infer

> **Warning:** This section is specific to Unix systems. In particular, it does not apply to Windows systems.

For Infer, you can use an interactive progress bar to get the information where Infer spent long time. You can enable this progress-bar by using `-Q --progress-bar-style multiline`, like this:

```
package Analyzer is
   for Switches ("infer") use ("-Q", "--progress-bar-style", "multiline");
end Analyzer;
```

CHAPTER

FIVE

VIEWING RESULTS

5.1 GNAT SAS Report Command

The `gnatsas analyze` command does not print any results. Instead, the `gnatsas report` command should be used, along with a third argument representing the desired output format. If no format is specified, `text` is used as default.

The following switch can be used to save results to a specific file:

`--out FILE`
Output the results to FILE instead of the standard output.

See also:

See *GNAT SAS CLI Reference* for more information about available command-line switches.

Examples

```
$ gnatsas report -P prj.gpr
```

will report results for the analysis of project `prj.gpr` in text format to standard output.

```
$ gnatsas report csv -P prj.gpr --out output_file.csv
```

will output results in CSV format to the file `output_file.csv`.

5.1.1 Supported Report Formats

GNAT SAS supports multiple report formats:

- `text` (see *Text Output*)
- `security` (see *Security Report*)
- `html` (see *HTML Output*)
- `csv` (see *CSV Output*)
- `code-climate` (see *CodeClimate Export*)
- `SARIF` (see *SARIF Export*)
- `exit-code` (see *Exit Code*)
- `gnat-studio` (see *Viewing GNAT SAS Output in GNAT Studio*)
- `gnathub`

The `text`, `security`, `html` formats are the main outputs formats, and are intended for "immediate use". The HTML format allows the user to browse the messages emitted by GNAT SAS, in a similar way as in GNAT Studio (see *Viewing GNAT SAS Output in GNAT Studio*).

The `csv` format is suitable for importation in a spreadsheet, typically to generate statistics. The `code-climate` format can be used to print reports in a format that can be ingested by Gitlab. The `sarif` format should be used when running analyses for viewers that accept inputs in the SARIF OASIS format..

The `exit-code` format has a very simple output: the exit code returned by this command will be the number of messages emitted, up to 255 (the restriction comes from a Linux restriction on the return code size). Its sole purpose is to be used in automation pipelines, e.g. to detect new messages.

The `gnat-studio` format will be automatically used when running an analysis from GNAT Studio, and so will be the `gnathub` format. Those are internal formats, not meant to be required by users directly.

See also:

- More information about supported formats is available in *Report Formats in detail*.
- See *Workflows* for examples of using those formats in integration workflows.

5.1.2 Selecting the Results to Display

Running the default report command `gnatsas report <format> -P <prj>` displays the results of the last `gnatsas analyze -P <prj>` run that was performed, regardless of the switches used for that run.

In order to generate a report for another run, you may use any of the following options:

- Specify a timeline, to display the results of the last run made in that timeline (see *Timelines*):

```
gnatsas report <format> -P <prj> --timeline <timeline>
```

- Directly generate a report from a SAM file, with the command:

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

  Note that you do not have to specify a project file in this case.

As seen in *Comparing GNAT SAS Runs*, you can add the `--compare-with <sam-file>` switch to compare the selected run with any other run.

5.1.3 Filtering Messages

Messages can be filtered using the `--show` switch. The switch is fully documented in `gnatsas report --help` in the section *SHOW OPTION FORMATTING*, see also the section *SHOW OPTION DEFAULT VALUE* and *SHOW OPTION EXAMPLES*. We present here a few use cases:

- `--show all`, to display even the messages that would be hidden by default.
- `--show age=removed`, to display only the messages that disappeared since the previous analysis.
- `--show rank+low`, to add low messages to the messages displayed by default.
- `--show kind=annotation`, to display only Inspector's annotations.
- `--show kind=info`, to display only Inspector's info messages

The basic format of the switch is either:

- `<category>=<constraint>` to *restrict* the report to the messages for which *category* (e.g. *age*) satisfies *constraint* (e.g. *removed*).

- `<category>+<constraint>` to *add* to the report the messages for which *category* (e.g. *rank*) satisfies *constraint* (e.g. *low*).
- `<category>-<constraint>` to *remove* from the report the messages for which *category* (e.g. *kind*) satifies *constraint* (e.g. *precondition*).

You can add multiple *constraint* to one category, as in `--show age=added+unchanged` which constrains the messages to be either added or unchanged (this is the default for text output).

The full list of *categories* and their associated *constraints* is shown in `gnatsas report --help`.

Multiple such filters can be combined as in:

```
gnatsas report -P<prj> --show all,age=added,rank=high,review_status=none
```

The command above makes GNAT SAS only report added high messages with no user review.

Specifying `--show` multiple times performs a union of the results of each filter. For example, the following switch will only show Infer and high Inspector messages:

```
gnatsas report -P<prj> --show tool=inspector,rank=high --show tool=infer
```

Default filtering per report format

Some messages are hidden by default in the output depending on the report format. This is because some formats are expected to be used in environments providing advanced filtering capabilities compared to others (e.g. it is easier to sort and filter messages in CSV in a spreadsheet software, compared to text output).

Note: It is possible to deactivate the default filtering of any format (except for the security report) and print all messages, with the command `--show all`.

The applicable default filtering settings are described in the table below for all report formats.

Format	Hidden messages by default	Displaying hidden messages
Text	All messages with any of the following conditions: • `age` is `removed`, or • `kind` is `annotation` or `info`, • `review_kind` is `not_a_bug`	Hidden messages can be displayed back using the corresponding re-enabling filter: • `--show=age+removed` • `--show=kind+info` (or/and `+annotation`) • `--show=review_kind+not_a_bug` or a combination of those.
Code-Climate	Same as text	
Exit Code	Same as text	

continues on next page

Table 1 – continued from previous page

Format	Hidden messages by default	Displaying hidden messages
Security	Same as text. Additionally, only messages with associated potential security vulnerabilities (i.e., messages corresponding to a *CWE weakness*) are displayed. Among those, potential dynamic errors such as access checks which are mitigated by Ada's built-in run-time checks and exceptions are also hidden by default.	Potential dynamic errors mitigated by Ada's built-in run-time checks and exceptions can be reported using the `--show-mitigated-cwes` switch. Other hidden messages cannot be displayed and other report formats should be used instead.
CSV	All messages with the following condition: • `kind` is `annotation` or `info`,	Hidden messages can be displayed back using the corresponding re-enabling filter: • `--show=kind+info` (or/and `+annotation`)
SARIF	Same as CSV	
HTML	All messages	

See also:

See *Report Formats in detail* for detailed information about each report format.

Filtering by message kind

When filtering messages by their *kinds* (see section *GNAT SAS Messages Reference* for their list and descpritions) you need to replace any space with an underscore. For example, to reference the *conditional check* category, use *conditional_check*; similarly for *useless reassignment*, use *useless_reassignment*.

In addition, you can use kinds *groups*. The groups of messages for `--show kind=` are:

Kind	**Effect**
check	References all *Run-Time Checks*, *User Checks* and *Uninitialized and Invalid Variables* *(Specific to Inspector and Infer)*
warning	References all *Warning Messages* *(all engines)*
race_condition	References all *Race Condition Messages* *(Inspector-specific)*
info	References all *Information Messages* *(Inspector-specific)*
annotation	References all *Annotations* *(Inspector-specific)*

To tell GNAT SAS to only emit checks, except for *precondition* checks, use a dash character (interpreted as a minus):

```
--show kind=check-precondition
```

5.2 Comparing GNAT SAS Runs

For each *analysis mode*, GNAT SAS keeps the results of a *baseline run*, set by default to the first run performed at the corresponding mode (refer to *Timelines* for more details). This allows GNAT SAS to specify, for each subsequent run, if a message is *new*, *unchanged* or *removed*, in comparison to the baseline.

This behavior also extends to runs that apply to one specific file with the `--file <file>` switch, or the `--files-from <file>` switch. In both cases a specific baseline is stored so that successive runs made on a file are compared to a baseline made on the same file.

The baseline stays the same from one run to another, unless the `gnatsas baseline` command is ran. This command accepts the following switches:

`--bump-baseline`
: Makes the last run of the selected mode the new baseline for the following runs.

`--set-baseline <sam_file>`
: Sets the baseline to *sam_file*, starting from the current run. GNAT Studio displays the name of the current run and its associated baseline at the top of the "GNAT SAS report" tab (see *Viewing GNAT SAS Output in GNAT Studio*).

> **Warning:** Registering a new baseline overrides any pre-existing baseline's SAM file. In case of a mistake, GNAT SAS makes one backup of the previous baseline before doing so.
>
> You can use the switch `--compare-with` of the `gnatsas report` command to temporarily test a potential baseline before setting it up (see below). If you want to save the previous baseline before setting a new one, you have to do so manually, see *GNAT SAS Files Reference* to locate the file to backup.

In addition, the user can temporarily ignore the baseline and do a one-off `gnatsas report` comparison with the switch `--compare-with <sam_file>`.

`--compare-with <sam_file>`
: The current run is compared to the specified run, without impacting the baseline.

In GNAT Studio you can bump the baseline with the menu *GNATSAS → Baseline → Bump Baseline to Current Run*. The baseline can be set to a specific file with the *GNATSAS → Baseline → Set Baseline to Run...*. See *GNAT SAS menu*.

5.2.1 Classifying Messages as Unchanged, Added, and Removed

In order to classify messages from the current run as `unchanged`, `added`, and `removed` relatively to the baseline run, for each message from the current run, GNAT SAS decides whether it matches an existing message in the baseline SAM file. GNAT SAS tries to match two messages even if the surrounding source code changes, but it can happen that GNAT SAS fails to do so and a message from the current run is marked as added even if there exists a corresponding message in the baseline run (`false added`). Indeed, GNAT SAS needs to be careful and not match an added message with an existing message which does not correspond to it (`false existing`) since such message would be "lost".

To decide whether a message from the current run matches a message in existing one, the following information is used:

- full name of the procedure in which the message is emitted

- analysis engine emitting the message
- kind of the message
- selected parts of the message text depending on the message kind
- if all the above matches multiple messages, GNAT SAS uses their order of appearance in the code.

Note: By default, GNAT SAS reports messages with the following settings:

- Only added and unchanged messages are reported.
- Unchanged messages do not have a specific mention; whereas added messages are explicitly marked as such.

In order to display removed messages, you can tell GNAT SAS to display them via the `--show age+removed` report switch. See `gnatsas report --help` for a complete list of supported `--show` values.

5.3 Viewing GNAT SAS Output in IDEs

5.3.1 In GNAT Studio

GNAT SAS output can be displayed in a user-friendly report interface in the GNAT Studio IDE. The GNAT SAS Report Window lists the messages for a given run and enables interactive filtering and source navigation. Comparison between runs is also supported, amongst other features.

See also:

Refer to *Viewing GNAT SAS Output in GNAT Studio* for a complete guide.

5.3.2 In Visual Studio Code

GNAT SAS output generated in SARIF format can be displayed by any SARIF-viewing capabilities tool, in particular in Visual Studio Code with SARIF dedicated extensions.

See also:

Refer to *Using GNAT SAS in Visual Studio Code* for more details.

5.4 Report Formats in detail

5.4.1 Text Output

You can get a compiler-like listing of messages generated by GNAT SAS with the `gnatsas report text` command (used by default when simply running `gnatsas report` without a format), e.g.:

```
$ gnatsas analyze -Pmy_project
gnatcheck 24.0w (20230603)
[...]
Compiler
   [Ada]          alias.adb
$ gnatsas report -Pmy_project
alias.adb:10:11: high: validity check (Inspector): Int is uninitialized here
```

The messages are formatted such as:

```
filename:line:column: rank: kind (analysis engine name): message contents
```

The analysis engine name is one of:

- `Inspector`
- `Infer`
- `GNATcheck`
- `-gnatw<X>` for GNAT Warnings, where `<X>` is the GNAT compiler switch corresponding to the warning message. Refer to *GNAT Warnings Messages* for more details.

You can add more switches to affect text output, e.g.:

`--show-backtraces`
: Show backtrace information

`--show-reviews`
: Show review associated to messages

See also:

See *GNAT SAS CLI Reference* for more information about available command-line switches.

5.4.2 HTML Output

In order to generate HTML output, you need to use the `gnatsas report html` command after having run a `gnatsas analyze` command.

For example, to perform both a GNAT SAS analysis and generate HTML, you can use:

```
$ gnatsas analyze -Pprj
$ gnatsas report html -Pprj
```

By default, the HTML report is generated under the *output directory* as `<output_dir>/html-report`. The location of that directory can be controlled with the `--out` or `-o` switch, e.g. `gnatsas report html -Pprj --out my_report`.

Once the report has been generated, you can visualize it by opening the file named `index.html` in a web browser.

For more details on the HTML interface, see the documentation of GNATdashboard web interface.

5.4.3 CSV Output

You can use the `gnatsas report csv` command to generate messages in a *CSV* format, suitable for use by other tools such as spreadsheets.

When the switch `--out` is used to specify an output file, the CSV output is emitted in that file. Otherwise, the report is printed on the standard output.

For example:

```
$ gnatsas report csv -Pprj --out messages.csv
```

will generate a file `messages.csv` with the following contents: first a heading listing all columns:

```
project,basename,path,subp,line,column,category,history,ranking,tool,message,cwe,kind,
↪related_checks,key,key_seq,review_from_source,review_kind,review_date,review_status,
↪review_author,review_text
```

then one line for each message, with each field separated by a comma, e.g:

```
prj.gpr,reprod.adb,reprod.adb,reprod.test,7,19,check,unchanged,high,Inspector,"overflow␣
↪check fails here: requires I /= Integer_32'Last",,overflow check,,
↪52218f775ac40790bcc25156dc7536d3,1,false,uncategorized,,,,
```

CSV format specification

The available columns are listed in the table below.

- Except for the `review_*` fields, all fields are emitted as output of the GNAT SAS report and are not meant to be edited by users when importing CSV files.
- Regarding `review_*` fields:
  - They are emitted when GNAT SAS finds review data associated to a message, in which case the last review is displayed.
  - They can also be used to import user reviews as described in *Through a CSV File*.

Field name	Description
project	GPR project name.
basename	Basename of the file the message is coming from.
path	Relative path of the file from the GPR project.
subp	Fully qualified name of the enclosing subprogram.
line	Line number of the message.
column	Column number of the message.
category	Category of the message, as listed in *GNAT SAS Messages Reference*.
history	*added* if message is new relative to the baseline, *removed* if message has been removed relative to the baseline, and *unchanged* otherwise, see also *Comparing GNAT SAS Runs*.
ranking	Ranking of the message (info, low, medium, high, suppressed).
tool	Tool that emitted the message.
message	Text of the message, surrounded by double quotes.
cwe	List of relevant Common Weakness Enumeration ids for the given message. See *GNAT SAS Messages Reference* more information about CWE.
kind	Message kind (check, warning, info, annotation).
related_checks	Only relevant for Precondition_Checks. This is a list of the checks that contributed to the associated precondition (that is, checks that might fail if the precondition is violated by a caller).
key	Internal field to uniquely identify a message.
key_seq	Related to Key to uniquely identify a message.
review_from_source	Whether the review message comes from the source.
review_kind	Kind of the review associated to the review status.
review_date	Date of the latest review, if relevant.
review_status	Status of the review (uncategorized, pending, false_positive, not_a_bug, bug).
review_author	Name of the review author, if relevant.
review_text	Last review comment, if relevant.

The statements below also apply to the CSV format supported by GNAT SAS.

- CSV files are exported and imported in the UTF-8 encoding.
- The separator is always expected to be a comma `,`, regardless of the locale.
- By default, double-quotes surrounding field values are optional. Unless the value contains a double-quote, in which case:
  - the value may be surrounded by double-quotes and the inner double-quotes escaped by doubling them. E.g. `"Some value with ""inner double-quoted text""."`, or
  - the value may be surrounded by single-quotes and the inner double-quotes do not need escaping. E.g. `'Some value with "inner double-quoted text".'`.

5.4.4 Security Report

GNAT SAS can emit a security-oriented analysis report in HTML format via the `--security-report` switch, which can be combined with other switches. For example:

```
$ gnatsas report security -Pprj
```

This command generates the file `security-report.html` in the GNAT SAS *output directory*. The location of the report can be controlled with the `--out` switch.

The security report contains the following potential vulnerabilities (see *GNAT SAS Messages Reference* for more details):

- unprotected access
- unprotected shared access
- mismatched protected access
- validity check
- dead code
- test always false
- test always true
- unused assignment
- unused out parameter
- useless reassignment
- loop does not complete normally

Potential dynamic errors such as access checks are mitigated by Ada's built-in run-time checks and exception and are not reported by default. They can be shown as well using the `--show-mitigated-cwes` switch.

This HTML file contains the following sections:

1. Potential Vulnerabilities Detected by GNAT SAS
2. Security Vulnerabilities Not Present
3. List of Ada Source Files Analyzed by GNAT SAS

A `sample security report` is available here or in the directory `share/examples/gnatsas/gnat_sas_by_example` of your installation.

5.4.5 CodeClimate Export

GNAT SAS can emit a report in the Code Climate format with the `gnatsas report code-climate` switch. The Code Climate format enables integrating GNAT SAS' output with various tools such as BitBucket, Github or Gitlab. The paths in the Code Climate report will be relative to the analyzed GPR file by default. You can make the paths relative to any other directory by using the `--root` switch. For example,

```
$ gnatsas report code-climate -Pprj --root .
```

Will make all paths relative to the directory GNAT SAS is being launched from and

```
$ gnatsas report code-climate -Pprj --root /
```

Will make all paths relative to the root of the filesystem.

5.4.6 SARIF Export

GNAT SAS can emit a report in the SARIF format with the `gnatsas report sarif` command. The SARIF format enables integrating GNAT SAS's output with any SARIF viewer tool. In particular, it is the format used for *GNAT SAS integration with Visual Studio Code*.

```
$ gnatsas report sarif -Pprj
```

By default, the URIs in file locations in the generated SARIF file will be relative to the project directory. The directory to which those URIs should be relative can be controlled using the `--root DIR` switch, to ensure portability of the generated report.

See also:

Refer to *GNAT SAS CLI Reference* for more information about command-line switches.

5.4.7 Exit Code

The `exit-code` format has a very simple output: the exit code returned by this command will be the number of messages emitted, up to 255 (the restriction comes from a Linux restriction on the return code size). Its sole purpose is to be used in automation pipelines, e.g. to detect new messages.

```
$ gnatsas analyze -Pmy_project
[...]
$ gnatsas report exit-code -Pmy_project
$ echo $?
6
```

indicates that six messages were reported by GNAT SAS.

CHAPTER

SIX

REVIEWING RESULTS AND IMPROVING CODE

A message generated by GNAT SAS may be a false positive (i.e. the error situation GNAT SAS is warning about cannot actually occur) or point out an intended behavior (e.g., numeric overflow might be intentionally raised in some situations). In either case, such a message does not indicate a potential error in the code that requires further investigation. It is often useful to preserve the results of such review. The following sections describe two ways to deal with such a message once identified, namely:

- tell GNAT SAS about the status of this message with the review mechanism described in *Reviewing Messages*. This way, GNAT SAS knows to either hide and/or attach information to said message for the current and future runs. Or,
- improve the code specification in order to clarify the intent of the code, and at the same time eliminate false alarms, as seen in *Improve Your Code Specification*.

6.1 Reviewing Messages

GNAT SAS provides two different mechanisms for capturing review information and associating it with the corresponding messages.

The first mechanism allows interactively reviewing messages via either:

- The GNAT Studio IDE (see *Reviewing messages through GNAT Studio*)
- The CSV output (see *Through a CSV File*)

The second mechanism consists of adding annotations in the form of pragmas to the Ada source code being analyzed as described in *Through Pragma Annotate / Justification in Source Code*.

Each approach has its pros and cons.

Advantages of interactive review include:

- No source code modifications are required; frozen source code can be reviewed.
- Review does not perturb line numbering of sources, which in turn can affect the text of other messages.
- Review can be performed by people not familiar with modifying Ada source code.
- Review status values other than False_Positive and Intentional are available (e.g., Pending as well as *Custom Review Status*).
- Review affects the messages of the current run.

Advantages of adding pragmas to the source include:

- Review is integrated with the sources and easier to relate to the sources.

- Review is less likely to be invalidated by other source changes; the mapping from the review to the message being reviewed is more straightforward.
- Existing editing and version control tools can be used to create and manage reviews.

The different techniques can be mixed, even within a single Ada unit.

In addition, it is also possible and sometimes desirable to improve the code specification to clarify the intent of the code and at the same time eliminate false alarms, as explained in *Improve Your Code Specification*.

When reanalyzing a source code, GNAT SAS classifies messages as unchanged, added, or removed with respect to the *baseline* run and it will preserve the existing reviews for unchanged messages. See *Classifying Messages as Unchanged, Added, and Removed* for more details.

6.1.1 Through GNAT Studio

See *Reviewing messages through GNAT Studio* for a detailed guide.

6.1.2 Through a CSV File

Note: This section assumes that the analysis has already been run on the project and that it found messages.

Another possibility for users to review messages is to export the messages via the *CSV Output* and then edit in e.g. a spreadsheet the CSV file to add manual reviews.

In order to input a review from a CSV file, you need to fill the `review_status` column (field) for the row corresponding to the message that you want to review. The value should be either `pending`, `false_positive`, `not_a_bug`, `intentional`, `bug`, or any *Custom Review Status*. Note that the `review_kind` field is automatically computed from the `review_status` field and should not be modified.

Additionally, you can fill the `review_author` field, and the `review_text` field, or leave them empty. The `review_date` field can be left empty, in which case it will be set to the time of the import by `csv-review`. If `review_author` is left empty, the tool will automatically add one using your login: `<login> (from csv)`.

Any other field should be left as set by the CSV export.

See also:

Refer to *CSV Output* for more details on the CSV format specification.

Once the CSV file has been modified and saved (in the same CSV format), use `gnatsas review` with the `--from-csv FILE` switch to import the reviews. The command:

```
$ gnatsas review -P<proj>.gpr --from-csv <file.csv>
```

will import the reviews manually added in `file.csv` to the *review file* corresponding to the project.

Note: The reviews will not be actually saved into the *Message Files* until another analysis or report action is executed.

6.1.3 Through Pragma Annotate / Justification in Source Code

To mark a message as reviewed in GNATSAS reports with a source annotation, insert an Annotate pragma directly below the line that triggers the message. (Refer to the section below for details on correct pragma placement.) The Annotate pragma uses the following format:

```
pragma Annotate (GNATSAS, False_Positive|Intentional, "<check name>", "<review message>
↪");
```

Reports of analysis made with this annotation no longer contain the corresponding message by default and, instead, list a manual review in the SAM file.

Warning: Since annotations are part of the code, they need to be written prior to an analysis to have an effect on the analysis' report: When writing a new annotation to review a message from the current analysis, you need to make a second analysis for the annotation to have an effect on the report.

Note that this pragma is ignored by the compiler when generating code, it only affects GNAT SAS' handling of generated messages, so you can safely add it to your source code without impacting compilation or run-time behavior. If for some reason pragma Annotate would be both recognized and not handled the same way by your Ada compiler, then you can safely replace `pragma Annotate` by `pragma Gnat_Annotate`.

When used in this way, an Annotate pragma takes exactly four arguments:

1. The identifier *GNATSAS* (for backward compatibility, *CodePeer* is also accepted).
2. One of two identifiers: *False_Positive* or *Intentional*.
   - *False_Positive* indicates a situation where the condition in question cannot occur but GNAT SAS was unable to deduce this
   - *Intentional* indicates that the condition can occur but is not considered to be a bug.
3. A string literal matching one of the message kinds listed in the tables presented in *GNAT SAS Messages Reference*.
4. A string literal which is used as the comment associated with the review of this message in the database.

The placement of the pragma in the source determines the messages (of the kind specified by the third argument) that it applies to. The pragma applies to messages associated with the preceding item in a statement or declaration list (ignoring other Annotate pragmas); if no such preceding item exists, then the pragma applies to messages associated with the immediately enclosing construct (excluding any portion of that construct which occurs after the pragma).

For a message saying that a subprogram always fails, the pragma can be placed either after the definition of the subprogram or at the start of the declaration part of the subprogram.

If unsure, you can use GNAT Studio to place the pragma at the correct place for you, see *Annotating the source*.

For the following example:

```
procedure Throw_Exception (Progr_Error : Boolean) is
begin
   if Progr_Error then
      raise Program_Error;
   else
      raise Constraint_Error;
   end if;
end Throw_Exception;
```

GNAT SAS generates the following message:

```
throw_exception.adb:1:1: high warning: subp always fails (Inspector): throw_exception␣
↪always ends with an exception or a non-returning call
```

One way to handle this situation is to justify the message by adding an Annotate pragma as follows:

```
procedure Throw_Exception (Progr_Error : Boolean) is
   pragma Annotate (GNATSAS, Intentional, "subp always fails", "reviewed by John Smith");
begin
   if Progr_Error then
      raise Program_Error;
   else
      raise Constraint_Error;
   end if;
end Throw_Exception;
```

A better solution to this problem would be to use the pragma No_Return. Applying this pragma to the procedure Throw_Exception will prevent the display of the "subprogram always fails" message and in addition to this, it will provide a compiler check that there is no control-flow path that can reach the "end" of the procedure.

The message saying that a subprogram always fails may be emitted because of one or more messages saying that some errors always happen in the subprogram. Note that in this case, the "subprogram always fails" message must be explicitly justified by a dedicated *pragma Annotate*. The following example shows that to justify the message that a subprogram always fails, it is not enough to just justify the message about error(s) in the subprogram:

```
procedure Justified_Error_Inside (Progr_Error : Boolean) is
begin
   if Progr_Error then
      raise Program_Error;
   end if;
   pragma Assert (Progr_Error);  --  Justified error inside subprogram
   pragma Annotate (GNATSAS, Intentional, "assertion", "reviewed by John Smith");
end Justified_Error_Inside;
```

GNAT SAS still generates the message indicating the subprogram always fails:

```
justified_error_inside.adb:1:1: high warning: subp always fails (Inspector): justified_
↪error_inside fails for all possible inputs
```

Similarly, for the following example:

```
function Func return Integer is
   X, Y : Integer range 1 .. 10000 := 1;
begin
   for I in 1 .. 123 loop
      X := X + ((3 * I) mod 7);
      Y := Y + ((4 * I) mod 11);
   end loop;
   return (X + Y) / (X - Y);
end Func;
```

GNAT SAS generates the following message:

```
func.adb:8:19: medium: divide by zero (Inspector): requires X /= Y
```

As it happens, this message is a false positive; the function will always safely return -4, but GNAT SAS is unable to deduce this fact.

One way to handle this situation is to justify the message by adding an Annotate pragma.

Consider adding a pragma as follows:

```
function Func return Integer is
   X, Y : Integer range 1 .. 10000 := 1;
begin
   for I in 1 .. 123 loop
      X := X + ((3 * I) mod 7);
      Y := Y + ((4 * I) mod 11);
   end loop;
   return (X + Y) / (X - Y);
   pragma Annotate (GNATSAS, False_Positive,
                    "Divide By Zero", "reviewed by John Smith");
end Func;
```

With this modification, GNAT SAS displays no message for this example.

In addition, if manual reviews are displayed (for example, if GNAT SAS is invoked with the `--show review_kind+not_a_bug --show-reviews` switches), the following is displayed:

```
func.adb:8:19: medium: divide by zero (Inspector): requires X /= Y. review: False␣
↪Positive,
Not A Bug, approved by Annotate pragma at 9:4: reviewed by John Smith
```

Note: During analysis, GNAT SAS emits a warning when an Annotate pragma does not have any effect. It is possible to hide these warnings with the switch `--no-unused-annotate-warning`.

6.1.4 Viewing Reviews in the HTML Interface

Starting from CodePeer 23, the HTML interface allows you to **browse** the reviews, but no longer offers the possibility of **adding** reviews. The review status is shown in a dedicated column of the Messages panels, and the review history can be displayed by clicking on the icon in the rightmost column of the message line.

6.1.5 Custom Review Status

It is possible to define your own review statuses for special needs. In order to define one you first need to select a review status kind. There are three kinds: `Pending`, `Not_A_Bug` and `Bug`. The review status kind impacts the behavior of the user interface: by default, GNAT SAS hides (in textual and HTML output as well as in GNAT Studio) any messages that have a review with a review status of kind `Not_A_Bug`. In addition, in the Locations View GNAT Studio highlights the messages with a review, using one particular color per review status kind.

Once you have selected a kind, you can define your custom review status in your project file with the project attribute corresponding to the selected kind among: `Pending_Status`, `Not_A_Bug_Status` and `Bug_Status`, e.g.:

```
project Proj is
   package Analyzer is
      for Pending_Status use ("To Do", "Don't know");
      for Not_A_Bug_Status use ("To Be Dealt With Later", "Don't care");
```

(continues on next page)

(continued from previous page)

```
      for Bug_Status use ("To be fixed ASAP", "Problem");
   end Analyzer;
end Proj;
```

Default review statuses map to the following review status kinds (status -> kind):

- `pending` -> `pending`
- `bug` -> `bug`
- `not_a_bug`, `false_positive`, `intentional` -> `not_a_bug`

6.2 Improve Your Code Specification

In some cases, it may be better to annotate the Ada code directly, providing additional information to help going further in the analysis and help also clarify the intent of the code.

Ada provides various ways to improve the specification of the code and thus allowing static-analysis tools to reason on additional information. For example:

- explicit range given on types, subtypes and variables
- explicit assertions written in the code via `pragma Assert` or `pragma Assume`
- explicit annotations written in the code via `pragma Annotate`
- pre and post conditions
- predicates and invariants

Note: Note that pragma Annotate is ignored by the compiler when generating code, so is only taken into account by GNAT SAS.

6.2.1 Using Pragmas Assert and Assume

Note:

- `pragma Assert` is supported by both Inspector and Infer, and has no effect on the analysis of other engines.
- `pragma Assume` is only supported by Inspector.

For both pragmas Assert and Assume (as with any run-time check), analysis of subsequent code assumes the success of the preceding run-time check. `pragma Assume` can, therefore, be used to prevent generation of unwanted messages about subsequent code. No message will be generated about a possible division by zero in the following example (provided that GNAT SAS is unable to deduce that failure of the assumption is a certainty):

```
declare
   Num : Natural := Some_Function;
   Den : Natural := Some_Other_Function;
   pragma Assume (Den /= 0);
   Quo : Natural := Num / Den;
begin
```

(continues on next page)

(continued from previous page)

```
   ...
end;
```

In this respect, `pragma Assume` can be viewed as an alternative to `pragma Annotate (CodePeer, False_Positive, ...);`. It is important to be aware of two important differences between these two approaches to avoiding unwanted messages. First, an Assume pragma (unlike an Annotate pragma) can affect the run-time behavior of a program - an Annotate pragma has no effect on run-time behavior. Second, an incorrect Assume pragma can invalidate GNAT SAS' analysis, possibly to an extent that is not obvious, while an Annotate pragma has no effect on GNAT SAS' analysis of subsequent constructs. The user-visible effects (with respect to both run-time behavior and to GNAT SAS' analysis) of an Assume pragma are more like those of an Assert pragma followed by an Annotate pragma that suppresses any message about the possibility that the assertion might fail.

6.2.2 Pragma Annotate / Modified in Source Code

Note: The approach described in this section is only supported in the Inspector and Infer engines and will only affect messages reported by those.

Pragma Annotate can also be used in another way to help cope with the problem of incorrect message generation. Suppose, for example, that the analysis engine somehow becomes confused about the value of a variable at some point in a program and consequently generates incorrect messages. Pragma Annotate / Modified can be used to tell the engine to assume that (at the point where the pragma is placed) a given object is fully initialized and that nothing else should be assumed about the value of that object; any previous deductions that have been made about the value of the object at the point of the pragma should be discarded. Note that this means that Inspector and Infer can issue false positive messages stemming from the fact that they assume that the object can have any value after the location of the pragma.

This is illustrated by the following examples:

```
procedure Proc1 is
begin
   -- Before the call to Initialize_Global_Variable_1,
   -- Global_Variable_1 is correctly known to be uninitialized.
   Initialize_Global_Variable_1;
   -- Suppose that Global_Variable_1 is initialized by the preceding
   -- call, but for some reason Inspector fails to realize that and
   -- therefore incorrectly assumes that Global_Variable_1 remains
   -- uninitialized after the call. The pragma corrects this mistaken
   -- assumption.
   pragma Annotate (GNATSAS, Modified, Global_Variable_1);
   -- Because of the pragma, no false positive message about reading
   -- an invalid value will be generated when Global_Variable_1 is
   -- passed as a parameter in the following call.
   Some_Procedure (In_Mode_Parameter => Global_Variable_1);
end Proc1;
```

```
procedure Proc2 (Count : in out Natural) is
begin
   Global_Variable_2 := 0;
   Update_Global_Variable_2;
   -- Suppose that Global_Variable_2 is updated by the preceding call,
   -- but for some reason Inspector fails to realize that and
```

(continues on next page)

(continued from previous page)

```
   -- therefore incorrectly assumes that Global_Variable_2 still has the
   -- value zero after the call.
   pragma Annotate (GNATSAS, Modified, Global_Variable_2);
   -- Because of the pragma, the assignment to Count will not be
   -- incorrectly identified as dead code.
   if Global_Variable_2 /= 0 then
      Count := Count + 1;
   end if;
end Proc2;
```

When used in this way, an Annotate pragma takes exactly three arguments:

1. The identifier *GNATSAS* (for backward compatibility, *CodePeer* is also accepted).
2. The identifier *Modified*.
3. A name denoting a stand-alone object or a parameter.

Such an Annotate pragma is subject to the same placement restrictions as an Assert pragma (roughly speaking, such a pragma shall only occur within a statement list or a declaration list).

6.2.3 Pragma Annotate / Identifying Race conditions

Note: The option described in this section is only supported in the Inspector engine and will only affect its messages.

In *deep analysis mode*, the pragma Annotate can also be used to aid Inspector in identifying tasks and race conditions with one of the following pragmas:

```
pragma Annotate (GNATSAS, Single_Thread_Entry_Point|Multiple_Thread_Entry_Point, "entry␣
↪point");
```

```
pragma Annotate (GNATSAS, Mutex, "lock subprogram", "unlock subprogram");
```

See also:

See *Identify Possible Race Conditions* for more details.

6.2.4 Using Pre and Post conditions

Note: Using pre/post conditions as described in this section will only impact Inspector's analysis. Other engines will ignore them.

Taking the GNAT Studio demo located under `<GNAT Studio install>/share/examples/gnatstudio/demo` as an example, and running it under GNAT SAS via:

```
$ gnatsas analyze -P demo --mode deep
$ gnatsas report text -P demo --show-backtraces
```

will generate in particular a potential array index check failure.

```
input.adb:258:31: low: precondition <array index check> [CWE 120] (Inspector):␣
↪precondition might fail on call to input.get_char; requires (if First_Char <= Last_
↪Char then First_Char else 1) <= 1_024
   array index check at input.adb:110:12
```

Instead of hiding the error, we're going to extend the contract of the subprogram `Get_Char` so that we can make sure the callers are correct.

First, let's look more closely at the subprogram itself:

```
100 function Get_Char return Character is
101    C : Character;
102
103 begin
104    --  First check if the line is empty or has been all read.
105
106    if End_Line then
107       Read_New_Line;
108    end if;
109
110    C := Line (First_Char);
111    First_Char := First_Char + 1;
112
113    return C;
114 end Get_Char;
```

The potential error occurs on the access to the element of `Line` at line 110, where `First_Char` might not be smaller or equal to 1024. So instead of removing this error, we're going to provide as a precondition of `Get_Char` the fact that the value has to be within expected range:

```
function Get_Char return Character;
pragma Precondition (First_Char <= 1_024);
```

Note: Note that the above is using a pragma so that this syntax can be used with any Ada compiler (the pragma will basically be ignored by non-GNAT compilers).

If you are using an Ada 2012 compiler then the following syntax is preferred:

```
function Get_Char return Character
  when Pre => First_Char <= 1_024;
```

Re-running GNAT SAS will remove the error. Now `First_Char` is assumed to be below 1024 and thus there is no potential error.

Note: Note that with GNAT, this pragma does not have any effect on the executable code by default - it is just here for documentation purposes and used by the additional tools. However, when using the GNAT compiler, a corresponding dynamic check can be added when e.g. compiling with the `-gnata` switch.

Warning: Although the capability of writing assertions or pre/post conditions may generally sound promising to users, please be aware that there are tool limitations and such complex conditions may not always help the

analysis. For example, the use of quantified expressions such as `for all I in F'Range => F(I) in 1..100` might not be understood by Inspector in all cases, but simple preconditions about some elementary-type global variable or parameter being in a certain range, or being non-null, would be fully understood. Combining such simple preconditions with `and` would also be fully understood.

The way pre/post conditions are being used by Inspector can be viewed by displaying annotations in the GNAT SAS report:

```
$ gnatsas report text -P demo --show kind+annotation

input.adb:101: (pre)- input.get_char:(user precondition, overflow check) First_Char <= 1_
↪024
```

or in context within the source file, using GNAT Studio:

```
--
--  Subprogram: input.get_char
--
--  Global_outputs:
--    First_Char, Last_Char, Line(1..1_024), Line_Num
--
--  Pre:
--    First_Char <= Last_Char or Line_Num /= 2_147_483_647
--    First_Char <= 1_024
--
--  Post:
--    possibly_updated(Line(1..1_024))
--    input.get_char'Result'Initialized
--    Line_Num'Initialized
--    Line_Num = One-of{Line_Num'Old, Line_Num'Old + 1}
--    Last_Char /= 0
--    First_Char in 2..1_025
--    First_Char = One-of{First_Char'Old, 1} + 1
--
--  Global_inputs:
--    First_Char, Last_Char, Line, Line_Num
--
function Get_Char return Character is
   C : Character;

begin
   --  First check if the line is empty or has been all read.

   if End_Line then
      Read_New_Line;
   end if;

   C := Line (First_Char);
   First_Char := First_Char + 1;

   return C;
end Get_Char;
```

CHAPTER

SEVEN

WORKFLOWS

There are many different ways to take advantage of GNAT SAS capabilities, depending on the stage of the project's lifecycle. During development phases, GNAT SAS can either be run every day or as often as there is a build of some part of the system. Using GNAT SAS to find defects before a project enters run time testing minimizes the number of crashes encountered during testing, allowing functional tests to run to completion and thereby providing more usable output for test engineers. On existing code being maintained, it can be used to find potential problems that are not revealed by testing and which can be extremely difficult to track under deployment. It can also be used to perform a change impact analysis. In high-integrity and certification-related environment, it can be used to demonstrate the absence of certain kind of errors or improve the quality of code review activities.

In all cases, the source code should not be shared directly (say, on a shared drive) between developers, as this is bound to cause problems with file access rights and concurrent accesses. Rather, the typical usage is for each user to do a check-out of the sources/environment, and use therefore their own version/copy of sources and project files, instead of physically sharing sources across all users.

This section will describe in detail various ways to put GNAT SAS in production and give it to the hand of all team members, or only a few selected ones.

Note: In many workflows below where GNAT SAS is to be run as part of a script, you might want to use the `--quiet` switch to suppress info output.

See also:

These workflows assume familiarity with *Getting the Right GNAT SAS Settings* which explains how to get the proper setting for your specific needs.

7.1 Use on a developer machine

This is the most basic usage: each developer runs GNAT SAS on their own machine, in `fast` mode (see *Analysis modes*). Reviews are versioned along the sources, and are updated by developers when new messages appear.

Thanks to fast mode's incrementality, the developer's machine is not required to be overly powerful (once an initial analysis has been performed).

7.2 Nightly Runs on a Server

In this workflow, GNAT SAS is run nightly on a dedicated server with lots of resources available (e.g. 16 cores as per *System Requirements*), in `deep` mode (set with the switch `--mode=deep`).

These runs will typically be run nightly to take into account all commits of the day, and provide results to users the next morning.

The next day, developers can analyze the results, either by checking them out if they are versioned, or by retrieving them as described in *Importing GNAT SAS Results*.

Developers may then review the messages and depending on the outcome, decide to fix the code or justify the relevant messages, directly in the code or using a *review file* to be committed. Refer to *Reviewing Messages* for the message review methods, and to *Importing and Sharing GNAT SAS User Reviews* for managing reviews from multiple users.

7.3 Continuous Runs on a Server after Each Change

In this workflow, GNAT SAS is run on a dedicated server with lots of resources available (e.g. 16 cores as per *System Requirements*) for performing runs sufficiently rapidly as to provide quick results to users after each commit. The idea of these runs is not to be exhaustive, but to focus on the differences from the previous run.

These continuous runs will trigger on a new repository change, typically integrated with a continuous integration framework such as Jenkins.

At the end of a run, a summary is sent to developers via email or a web interface, or even directly through GNAT SAS' report formats that are compatible with forges (e.g. SARIF, or Code Climate for Gitlab).

Once a first analysis has been performed, it is possible to filter the report of the subsequent analyses in a way that only new messages are shown, for example with the following command:

```
$ gnatsas report text -Pprj --show age=added
```

Developers may then review the messages and depending on the outcome, decide to fix the code or justify the relevant messages, directly in the code or using a *review file* to be committed. Refer to *Reviewing Messages* for the message review methods, and to *Importing and Sharing GNAT SAS User Reviews* for managing reviews from multiple users.

7.4 Importing GNAT SAS Results

As stated in *Message Files*, GNAT SAS stores the results of a run in a SAM file, in the *output directory*. Depending on the use case, you may only have to retrieve this SAM file, or you may need additional elements. We describe here a few use cases:

- Assuming all project dependencies are available, simply use:

  ```
  gnatsas report <format> -P <project>
  ```

- When all dependencies required in the GPR project are not available in the environment, the above command may fail with an explicit error message. It is still possible to generate a report from a SAM file without specifying the project file by running the command below.

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

Warning:

- The <format> argument is mandatory.
- Only the following formats are supported: `text`, `csv`, `exit-code`, `security`, `gnathub`. Other formats require an explicit `-P <project>` switch.

- In order to use the results of a distant run as a baseline, or to compare local runs to it, retrieve the distant SAM file and use it as argument to, either

```
gnatsas baseline -P<prj> --set-baseline <sam-file>
```

or

```
gnatsas report -P<prj> --compare-with <sam-file>
```

You can specify `--mode <mode>` on both commands to change the affected mode (see *Analysis modes*).

- In order to handle a distant analysis as if it was run locally, retrieve the distant SAM file and use it as argument to

```
gnatsas baseline -P<prj> --set-current <sam-file>
```

This way, the imported run is set as the current run (specify `--mode <mode>` to change the affected mode). The local baseline and reviews are applied. You can then display the results with, e.g. `gnatsas report -P<prj>`, add new reviews, etc...

- In order to duplicate the whole state of a distant machine (comprising all analysis, baselines and reviews), replace your local *output directory* and *review file* with the distant ones. Note that you then lose your local state.

Note: You can identify the SAM file of a run with:

```
gnatsas report -P<prj> --show-header
```

See also:

- Refer to *Workflows* for more information about versioning files.
- Refer to *GNAT SAS CLI Reference* for more information about GNAT SAS command-line switches.

7.5 Importing and Sharing GNAT SAS User Reviews

When multiple users are reviewing messages locally, they can export their manual reviews (marking messages as e.g. false positives), in order to import or share them.

To export manual reviews, the user simply needs to copy the *review file*.

One can then directly incorporate reviews from an external review file with the command

```
gnatsas review -P<proj>.gpr --import <external_review_file>.sar
```

It is also possible to fuse multiple review files into one with the tool `merge-reviews`:

```
merge-reviews -o merged.sar <file1.sar> [<fileN.sar>]...
```

Note: In case of multiple users reviewing the same message, the review that applies to the message is always the one with the newest review date (but all reviews are kept in the review file).

See also:

See the *GNAT SAS CLI Reference* for more details about GNAT SAS command line.

7.6 Using GNAT SAS with a Jenkins Automation Server

The recommended way of integrating GNAT SAS with the Jenkins Automation Server is using a pipeline script. The script can be source controlled alongside the code, or saved in a pipeline project's configuration.

7.6.1 Jenkins Agent Setup

To set-up a Jenkins Agent with the capability to run a GNAT SAS analysis follow the installation instructions corresponding to the agent's operating system; the instructions for installing GNAT Studio can be omitted.

7.6.2 Running the Analysis

To run the analysis add a `gnatsas-analyze` stage to your project's pipeline after any stages which make the code available on the agent's filesystem (the checkout stage in the *Example Pipeline script*).

Within this stage add a step that executes a host shell command invoking `gnatsas analyze` with the desired command line switches.

7.6.3 Getting Analysis Results

To retrieve the results of a GNAT SAS run outside of the Jenkins log console, you will need to setup a workflow following section *Importing GNAT SAS Results*.

You can add a step to the following pipeline that stores the SAM/SAR file(s) in a specific location where you can easily download it.

7.6.4 Example Pipeline script

An example of pipeline script which accesses the internet to clone a git repository from GitHub and run a GNAT SAS analysis on the checkout is:

```
pipeline {
   agent any
   stages {
      stage('checkout') {
         steps {
            url: 'https://github.com/...'
         }
      }
      stage('gnatsas-analyze') {
         steps {
```

(continues on next page)

(continued from previous page)

```
            sh 'gnatsas analyze -P my_project.gpr'
         }
      }
   }
}
```

7.6.5 Warnings Next Generation usage

Warnings Next Generation is a Jenkins plugin that displays analysis results within Jenkins itself. See https://plugins.Jenkins.io/warnings-ng/ for more details on this plugin. This plugin supports SARIF, thus you can emit the messages in this format to import them in warnings ng.

Here is what the pipeline would look like using this plugin to publish the results.

```
pipeline {
   agent any
   stages {
      stage('checkout') {
         steps {
            url: 'https://github.com/...'
         }
      }
      stage('gnatsas-analyze') {
         steps {
            sh 'gnatsas analyze -P my_project.gpr'
         }
      }
   }
   post {
      always {
         sh 'gnatsas report sarif -P my_project.gpr -o output.sarif --show age-removed'
         recordIssues (
             enabledForFailure: true,
             tool: sarif(pattern: 'output.sarif')
         )
      }
   }
}
```

7.7 Easier Review of SAM files stored in Git

In order to make reviewing the *SAM files* you commit in your Git repositories easier, you can configure Git to use GNAT SAS to display SAM files as a textual report.

First, in your Git repository, define a Git textconv option for SAM files, like this:

```
git config --local diff.sam.textconv 'gnatsas report text'
```

Then, create a `.gitattributes` file whose content is as follows:

```
*.sam diff=sam
```

This will result in Git displaying diffs on SAM files as a textual format:

```
diff --git a/results.sam b/results.sam
--- a/results.sam
+++ b/results.sam
@@ -1,3 +0,0 @@
-file.adb:9:6: medium warning: unused assignment [CWE 563] (Inspector): into X
+file.adb:9:11: high: overflow check [CWE 190] (Inspector): fails here: requires X > 0
```

Instead of a JSON file.

Note: The displayed diff will use the default filtering of the text format. This can be changed using the `--show` option as described in *Filtering Messages*.

7.8 Using GNAT SAS in a GitLab Pipeline

The recommended way of integrating GNAT SAS in a GitLab Pipeline is using a pipeline script. The script can be source controlled alongside the code, or saved in a pipeline project's configuration.

7.8.1 Pre-requisites

In order to enable running GNAT SAS in a CI, one requirement is to make it available to the GitLab Pipeline. To do so, one simple approach is to embed it in a Docker image. This is not detailed here and the next section assumes that such an image has been created and published to your GitLab's Containing Registry, and is available as `$REGISTRY_URL/$PROJECT_PATH/gnatsas:24.0w`.

See also:

A detailed example, starting from making GNAT SAS available in a Docker image, running that image and displaying the results in GitLab is available in the blog post: Building a GNAT SAS analysis pipeline on GitLab.

7.8.2 Running the Analysis

To run the analysis, simply run the `gnatsas analyze` command on the project's GPR file:

```
script:
   - gnatsas analyze -P my_project.gpr
```

7.8.3 Getting Analysis Results

To display the results of the analysis in text format in the CI log, run the following command:

```
script:
   - gnatsas report -P my_project.gpr
```

Furthermore, GNAT SAS provides the *CodeClimate Export* report format which enables displaying results in the GitLab UI. One needs to declare the report's path (here, `code_quality_report.json`) so that GitLab takes it into account:

```
script:
   - gnatsas
      report
      code-climate
      -P my_project.gpr
      --out $CI_PROJECT_DIR/gnatsas/code_quality_report.json
      --root $CI_PROJECT_DIR
      --long-desc

artifacts:
   - paths: gnatsas/
reports:
   codequality: gnatsas/code_quality_report.json
```

Note the `--long-desc` switch specified to the CodeClimate report format. This switch generates messages with a description that contains more information than usual, such as check name, analysis engine, CWE... as such information is not displayed by default in the CodeQuality widget integrated in GitLab.

7.8.4 Example Pipeline script

Here is a full example combining the steps described above. The script below uses a Docker image with GNAT SAS available to run a GNAT SAS analysis and report the results in the "Code Quality" tab of GitLab's interface.

This script may be committed to the root of the project's repository as `.gitlab-ci.yml`, for example.

```
default:
   image: $REGISTRY_URL/$PROJECT_PATH/gnatsas:24.0w

GNATSAS:
   script:
      - gnatsas analyze -P my_project.gpr
      - gnatsas report -P my_project.gpr
      - gnatsas
            report
            code-climate
            -P my_project.gpr
            --out $CI_PROJECT_DIR/gnatsas/code_quality_report.json
            --root $CI_PROJECT_DIR
            --long-desc

   artifacts:
      - paths: gnatsas/
```

(continues on next page)

(continued from previous page)

```
  reports:
    codequality: gnatsas/code_quality_report.json
```

7.8.5 Advanced integration

Effective Code Quality report comparison

GitLab Code Quality engine provides its own report comparison mechanism. The report in CodeClimate format generated for the source branch of a Merge Request (MR) is automatically compared to the report of the target branch, when it exists.

Filters can still be used (see *Filtering Messages*) to customize the messages that are reported. Note that the *default filtering* for CodeClimate format is that removed messages and messages reviewed as "Not a bug" are not displayed.

Therefore, our recommendation is to configure GNAT SAS to report all messages found for the source branch. In practice, this means that

- *Review files* should be committed so "Not a bug" messages are not reported
- it is not necessary to commit the *SAM files* for comparison; but you may do so to version them and keep a history, and bump baselines when appropriate.

Incremental analysis

To benefit from incremental analysis and speed up CI time, we recommend to use caching capabilities in GitLab runners. There are a few things to consider when setting up the project:

- GNAT SAS should be run in `fast` mode (see *Analysis modes*).
- Default recommended switches for `gnatsas analyze` are:
  - use `-j0`
  - use `--incrementality-method=minimal`. This switch will not trigger re-compilation if sources timestamps differ but checksums match.
  - If any pragma configuration files are used to compile the project (usually `.adc` files passed with the `-gnatec=` switch, or with the `Global_Configuration_Pragmas` of the *Builder* package), ensure that the switch `-gnateb` is specified as well to all projects with:

    ```
    for Global_Compilation_Switches ("Ada") use ("-gnateb");
    ```

    This way, those files are not recompiled when only their timestamps change.

The next step is to prepare for using caching in the GitLab CIs.

- The `gnatsas` directory under the object directory (defined with the `Object_Dir` switch in the GPR project file), or the object directory directly, should be cached.
- Importantly, as described in *Effective Code Quality report comparison*, the cache should not contain SAM files, otherwise they would be used as analysis baselines and reported messages may not be the ones expected for the branch in some cases if the cache is outdated. We thus recommend to define an output directory `Output_Dir` (see *Analyzer package attributes*) **outside** of the cached directories.
- The caching policy is up to you; for example, a pull-push policy can be used in MRs and after merging, with a fallback key allowing the GitLab runner to restore cache from the target branch when cache does not exist yet for the source branch.

- Finally, the CI job must do the following:
  - restore cache (automatically done by GitLab)
  - move/copy cache to the expected location with respect to the project (if locations differ)
  - analyze the project and generate a report in CodeClimate format with GNAT SAS
  - copy back the updated cached directories to the cache location (if locations differ)
  - save cache (automatically done by GitLab)
  - save artifacts: at least the report. More artifacts can be saved e.g. for debugging.
  - tell GitLab to use the report as a `codequality` report.

Aggregating multiple reports

It is possible to analyze multiple projects and aggregate their reports in a single pipeline. However, there is a GitLab limitation that only one `codequality` report can be defined for a given job. Therefore, it is necessary to define a job per report and GitLab will automatically aggregate them.

For example, one could define a single `gnatsas` job analyzing two projects, and two parallel report jobs reporting each `codequality` report:

```
# Define a gnatsas CI job doing the analysis
gnatsas:
   # ...
   stage: gnatsas
   script:
      - gnatsas analyze -P my_project.gpr
      - gnatsas
          report
          code-climate
          -P my_project.gpr
          --out $CI_PROJECT_DIR/gnatsas/my_project.json
          --root $CI_PROJECT_DIR
          --long-desc

      - gnatsas analyze -P another_project.gpr
      - gnatsas
          report
          code-climate
          -P another_project.gpr
          --out $CI_PROJECT_DIR/gnatsas/another_project.json
          --root $CI_PROJECT_DIR
          --long-desc

   artifacts:
      - paths: gnatsas/

# Only one code-quality report per job can be uploaded, so use a matrix of jobs
# to report each report. Artifacts from previous job are available in subsequent
# ones.
gnatsas-report:
   stage: gnatsas
   interruptible: true
```

(continues on next page)

(continued from previous page)

```
  needs:
     - job: gnatsas
       optional: true
  rules:
     ...
  parallel:
     matrix:
        - REPORT_PATH:
            [
               my_project.json,
               other_project.json
            ]
  script:
     # A non-empty script is required, otherwise artifacts is not recognized as a key
     - ls .
  artifacts:
     when: always
     reports:
        codequality: $CI_PROJECT_DIR/gnatsas/$REPORT_PATH
```

Alternatively, one could define two different analysis jobs, each analyzing a single project and outputting its report.

CHAPTER

EIGHT

USING GNAT SAS IN GNAT STUDIO

8.1 Prerequisites

We recommend using a matching version of GNAT SAS and GNAT Studio. GNAT Studio is also designed to be backward compatible with older versions of GNAT SAS. Using an older version of GNAT Studio with a newer version of GNAT SAS is not supported, but it can work in some cases.

If the compiler for your project is not GNAT, make sure you have properly setup your project file, as explained in *Use of Libraries Installed with GNAT*.

8.2 Running GNAT SAS Analysis

When GNAT SAS is installed and found on your PATH, GNAT Studio will automatically detect it and provide a new *GNATSAS* top level menu.

Analyze All launches the default analysis. *Analyze...* opens a dialog when you can select more precisely the kind of analysis performed and in particular to add or override GNAT SAS switches defined in your project file.

After an analysis finishes, GNAT Studio automatically opens the Report Window. The Report Window can be also opened manually by choosing *Display Code Review*.

See also:

See *GNAT SAS menu* for a detailed description of the *GNATSAS* menu.

8.3 Viewing GNAT SAS Output in GNAT Studio

8.3.1 Selecting the Results to Display

GNAT Studio has the following three behaviors:

- New results are automatically displayed in the *GNAT SAS Report Window* after each analysis done from GNAT Studio.
- You can display the results from the last run (even after having just opened GNAT Studio) by clicking on the menu *GNATSAS* → *Display Code Review*.
- With the menu *GNATSAS* → *Advanced* → *Regenerate Report...*, you may:
  - Display the results of an arbitrary run. The last run is displayed by default. To select another run, specify the `--timeline <TIMELINE>` switch to the command-line.

  - Compare that run to an arbitrary SAM file (ignoring the baseline), with the field `Compare with`. This maps to the `--compare-with <sam-file>` switch.
  - Customize report generation by passing arbitrary `gnatsas report` switches to the command-line.

  See also:

  Refer to *Timelines* for more information about timelines and to *GNAT SAS CLI Reference* for more details about the GNAT SAS command line usage.

- With the menu *GNATSAS* → *Advanced* → *Display Result File...*, you can select an arbitrary SAM file and display the results it contains.

8.3.2 GNAT SAS Report Window

When you open a GNAT SAS report (see *Selecting the Results to Display*), the Locations View is filled with messages from the GNAT SAS run, and the Report Window is displayed.

The top part of the Report Window shows information about the baseline run and the current run. For each of them, the following data is displayed:

- the name of the run's SAM file as `<sam-file-name>.sam`, or `<run_name> (<sam-file-name>.sam)` if a name was set with `--run-name`
- the absolute and relative date of the run
- the corresponding command-line, which can be useful to remind you of any specific switches used, like the analysis mode.

Note: The above information is also available in a structured format in the SAM file corresponding to the run.

Shortcut buttons to execute baseline actions (bumping the baseline, setting a new baseline or replacing the current run) are also available, corresponding to the Baseline submenu (see *GNAT SAS menu*).

Warning: The above actions will only apply to the timeline corresponding to the last run. To execute such actions for different timelines, either run an analysis with an explicit `--timeline TIMELINE` switch beforehand, or use the GNAT SAS command line. Refer to *Timelines* for more information about timelines and to *GNAT SAS CLI Reference* for more details about the GNAT SAS command line usage.

The Report Window has two tabs: **Messages** and **Race conditions**. The first tab provides a summary of all messages and filtering capabilities, and the second provides a summary of potential race conditions found. The race conditions tab is only meaningful when GNAT SAS is run in deep analysis mode.

On the left side of the messages window, there is a main area listing all files for which messages have been generated, with information organized hierarchically by project, file and subprograms. You can click on any file to display the first message on this file in the Locations View. See *Using the Locations View* for more details on the use of the Locations View.

Similarly, you can double click on any file or subprogram to jump to the corresponding source editor, which will be displayed with their corresponding Inspector annotations (if Inspector was enabled during the analysis).

For each of these entities, three columns displaying the number of *high*, *medium* and *low* messages corresponding to the current filter selection (see *Using Filters*).

Different kinds of filters are available on the right side of the window. Clicking on each of these filters will show/hide the corresponding messages in the Locations View. This way, you can easily concentrate on e.g. *high* ranking messages only, or on a specific category (e.g. *validity check*) during your review of the messages.

The filters are divided into four kinds:

- Message categories

  This section (the first column on the right of the file summary) lists all the message categories found in the current analysis in two groups: warnings and checks. In addition, it lists all CWEs found in the current analysis when displaying of CWEs is enabled. By default, all categories found are enabled, and you can select/unselect all categories in each group at once by clicking on the check box left to the *Warning categories* (or *Check categories*, or *CWE categories*) label, or individually by checking each item. See *GNAT SAS Messages Reference* for more details on the meaning of each category.

  CWE categories filter applies for all kinds of messages. When some CWE is selected in filter all messages with this CWE are displayed (even when not selected in *Warning categories* and *Check categories* filters). Precondition messages are displayed when related checks have the chosen CWE.

- Message history

  By default, *added* and *unchanged* messages are displayed (with respect to *baseline*). You can also select old *removed* messages that are no longer present in the last run, or concentrate on *added* messages only and ignore *unchanged* messages. This is particularly useful when using GNAT SAS on legacy code, without having to review previously found messages, and concentrate on messages found after new changes, to analyze the impact of these new changes.

- Message ranking

  By default, GNAT Studio only displays the most interesting messages (ranked *medium* and *high*). Once you have reviewed these messages, reviewing the *low* messages can be useful. It can also help understanding some messages to display the *low* and/or *info* messages, which can provide additional information.

- Message review status

  By default, GNAT Studio displays all the messages except for the ones that have a review of review status kind *Not_A_Bug* (i.e. with review status *not a bug*, *false positive*, *intentional*, or any *Custom Review Status* of this kind). You can change this default setting by checking or unchecking the corresponding boxes.

On the top of the race conditions window, there is a list of shared objects. Clicking on shared object will open a list of entry points in the bottom of the race conditions window and automatically scroll the Locations window to simplify access to locations where shared objects are used.

On the bottom side of the race conditions window, a list of entry points and kind of access is displayed for the currently selected shared object. Clicking on row of this view opens source editor at scroll it to point of access to shared object.

8.3.3 Using the Locations View

When you open a GNAT SAS report (see *Running GNAT SAS Analysis*), the Locations View at the bottom part of GNAT Studio is filled with messages from the GNAT SAS run.

You can click on any of these messages, and GNAT Studio will open a source editor containing the relevant source file, at the listed source location. You can also use the *Filter panel* available from the contextual menu in the Locations View in order to display only messages containing a specific text pattern.

For some message kinds, extra information may be provided by GNAT SAS as a *backtrace* associated with the message, giving source locations that are related to the context of the message. Examples include locations where side effects are happening for a *function with side effects* message, locations of calls forming a call stack, and actual checks for a *precondition* message, or locations of taint sources and taint propagations for a message related to taint analysis, such as a *sql injection* message. Each element in the backtrace can be clicked to see the corresponding source code and ease the understanding of these messages.

Warning: Note that the Locations View's filter applies to both messages and backtrace elements the same way. For example, using a file name as a filter will filter out backtrace elements from other files, even if they belong to a message emitted for that file.

For more details on how to use the Locations View, see the GNAT Studio documentation directly, which explains how this view is managed.

In addition, an *Edit* icon is displayed in front of each GNAT SAS message. Clicking on this icon allows you to post a manual review, or a source review (through a pragma Annotate), of the message. It is possible to review single or multiple messages at once. See *Reviewing Messages* for more information about use of message review dialogs.

8.3.4 Exploring Inspector Annotations

Inspector generates annotations as-built documentation for each subprogram that it analyzes. This documentation is presented in the form of virtual comments at the beginning of each subprogram in a source-editor window. This as-built documentation includes annotations in the form of Preconditions, Presumptions, Postconditions, etc. which characterize the functioning of the subprogram, as determined by a global static analysis of the subprogram and the routines that it calls, directly or indirectly.

See also:

- For more details on the form of these annotations, see *Inspector Annotations*.
- For more details on using these annotations as part of code review, see *Use Annotations generated by Inspector for Code Reviews*.

The annotations are not actually added to your source code; they are only visible when viewing the source in a GNAT Studio source-editor window. You may hide the annotations using the *GNATSAS* → *Hide annotations* item in the source editor's contextual menu. You may display them again using the contextual menu item *GNATSAS* → *Show annotations*.

Note: By default, Inspector's annotations will be imported and displayed in the source editor. This step can take a significant amount of time for large projects and can be disabled globally via the *Preferences and Project Properties*.

8.3.5 Source Navigation

Using the GNAT Studio source navigation can be very useful for understanding GNAT Studio messages, verifying that messages are indeed relevant, and modifying the source code.

In particular, using the *Find all references* and *Find all local references* contextual menu, as well as *Goto body* will help significantly with reviewing messages.

For more information on GNAT Studio source-navigation capabilities, see the GNAT Studio User's Guide..

8.4 Reviewing messages through GNAT Studio

This section explains how to review GNAT SAS messages directly from GNAT Studio.

See also:

For more information about the general review process, refer to *Reviewing Results and Improving Code*.

In GNAT Studio, when clicking the *Edit* icon at the left side of message in the Locations View (see *Using the Locations View*), you can choose between *Manual review* or *Annotate*. The default action to use can be configured through the "Default review action" in *Preferences and Project Properties*.

8.4.1 Manual review

When clicking on *Manual review*, you get two possible review dialogs: one to review a single message and another to review multiple messages. Which dialog will be used depends on how many messages are selected in the Locations View before clicking on the edit icon.

The **New Status** drop-down box allows selecting the review status of the message. The review status is initially set to `Uncategorized` and can be set to `Pending`, `Not A Bug`, `Intentional`, `False Positive`, `Bug` or any *Custom Review Status*.

Note: The review status impacts the behavior of GNAT Studio: by default, GNAT Studio hides any messages that have as review status either *Intentional* ` or *False Positive*. In addition, in the Locations View, GNAT Studio highlights messages with different colors depending on the review status.

The **Approved By** text box allows the name of the reviewer to be recorded.

The **Comment** text box allows the reviewer to enter an explanation/justification for the message.

The single-message review dialog displays a history of the changes related to that message, while the multiple-message review dialog displays a list of messages to be reviewed along with their ranking and review status.

8.4.2 Annotating the source

When selecting *Annotate*, GNAT Studio automatically inserts a *pragma Annotate* at the correct source position. You are left with updating its content accordingly to your review, see *Through Pragma Annotate / Justification in Source Code*.

8.5 GNAT SAS entry points

8.5.1 GNAT SAS menu

The main entry point to run GNAT SAS and display analysis results is the GNAT Studio menu *GNATSAS*. This section provides a complete description of this menu.

- Analyze All

  Launches the default analysis, and load the GNAT SAS report window. See *Viewing GNAT SAS Output in GNAT Studio* for more details on the Report Window. This menu item will use the GNAT SAS switches set in your project files, if any.

- Analyze...

  Open a dialog where you can choose the kind of analysis you want to do. This dialog lets you select graphically the options described below. If there is a conflict between the selected options and the ones set in your project file, the ones set in this menu take precedence.

Multiprocessing	Specify the number of processes to generate SCIL files and analyze files (0 means use as many cores as available on the machine). Maps to the `-j` switch.
Analysis mode	Set the analysis mode to either fast (the default) or deep (see *Analysis modes*). Maps to the `--mode` switch.
Root project only	GNAT SAS will only analyze the source files associated with the *root* project file (presuming your application uses a tree of project files to represent all of its constituents). Maps to the `--no-subprojects` switch.
Force analysis	Force generation of all SCIL files and analysis of all files (without incrementality). Maps to the `-f` switch.
Enable GNAT Warnings	Enable/disable launching GNAT front end and collecting its warnings, see *Configuring GNAT Warnings*. Maps to the `--gnat` switch.
Enable Infer	Enable/disable Infer analysis (enabled by default). Maps to the `--infer`/`--no-infer` switches.
Enable Inspector	Enable/disable Inspector analysis (enabled by default). Maps to the `--inspector`/`--no-inspector` switches.
Enable GNATcheck	Enable/disable GNATcheck analysis (enabled by default). Maps to the `--gnatcheck`/`--no-gnatcheck` switches.

  Once the selection is done, click on the *Execute* button to get a behavior similar to *Analyze All*.

- Analyze File

  Perform a default analysis of the current source file only. This menu is only available when a source file is selected in GNAT Studio.

  Note that this makes Inspector's analysis ignore effects of calls to functions outside the selected file (this only affects deep mode analysis).

  Note that analyzing a generic unit will have no effect: you need to analyze one of its instantiations to get messages on a generic unit. Analyzing a subunit (separate unit) will trigger an analysis on the enclosing unit.

  In order to set more switches when analyzing a single file, you can either set those switches in your project files, or use *Analyze...*, and append `--file %F` to the end of the command line.

- Display Code Review

  Load the GNAT SAS results and review information from the disk, and display a summary. See *Viewing GNAT SAS Output in GNAT Studio* for more details.

- Generate HTML Report...

  Generate a GNAT SAS report in HTML format. This menu must be run after a GNAT SAS analysis has completed. See *HTML Output* for more information. The following switch can be specified to refine the information generated, as part of an intermediate dialog:

Show info messages	Show messages of kind info (hidden by default).

- Generate CSV Report...

Generate a GNAT SAS report in CSV format, and open the report in a source editor. This menu must be run after a GNAT SAS analysis has completed. It will open an intermediate dialog with the following options.

Show annotations	Show Inspector's annotations in addition to messages (hidden by default).
Show info messages	Show messages of kind info (hidden by default).
Show removed	Show messages removed from the baseline run (hidden by default).
Hide low messages	Hide messages that have a low ranking (shown by default).

See *CSV Output* for more details on the CSV output.

- Baseline

  This submenu provides baseline manipulation actions (see *Comparing GNAT SAS Runs*).

  - Bump Baseline to Current Run

    Make the last run the new baseline for following runs in the current timeline.

  - Set Baseline to Run...

    Select a new baseline. This opens a menu to select a SAM file.

  - Replace Current Run with Run...

    Select a SAM file that will be set as the current run (see *Importing GNAT SAS Results*).

> **Warning:** The above actions will only apply to the timeline corresponding to the last run. To execute such actions for different timelines, either run an analysis with an explicit `--timeline TIMELINE` switch beforehand, or use the GNAT SAS command line. Refer to *Timelines* for more information about timelines and to *GNAT SAS CLI Reference* for more details about the GNAT SAS command line usage.

- Advanced

  This submenu provides advanced capabilities, less often needed:

  - Regenerate Report...

    Run GNAT SAS to regenerate a report without performing a new analysis. This can be used in order to:

    * Display the results of an arbitrary run. The last run is displayed by default. To select another run, specify the `--timeline <TIMELINE>` switch to the command-line.
    * Do a one-off comparison of that run with another arbitrary run using `Compare with`. If no file is selected, the baseline of the specified timeline (or the last run if none specified) is used for comparison.
    * Customize report generation by passing arbitrary `gnatsas report` switches to the command-line.

  - Display Result File...

    Select any SAM file and generate a report from it. You should ensure that the selected SAM file corresponds to a run on the project currently opened.

  - GNAT SAS Log

    Open a text editor with the low level log file produced by GNAT SAS. This can be useful for example to understand why GNAT SAS did not generate the expected output, or to send GNAT SAS bug reports.

Note that most of the menus will launch commands that can be configured or tuned using the *Edit → Preferences* menu, in the *Build Targets* section. See the GNAT Studio documentation for more details on the build targets.

8.5.2 Preferences and Project Properties

The GNAT SAS plug-in for GNAT Studio adds the following preferences in *Edit → Preferences...*:

- *GNATSAS*

  Import Inspector annotations
  : Controls whether to import and display GNAT SAS annotations generated by Inspector analysis such as generated contracts and values of variables in the source editor. Note that importing GNAT SAS annotations can take a significant amount of time for large projects.

  Default review action
  : Choose the default action performed when clicking on a review action. 'Review' to add a manual review; 'Annotate' to add a pragma Annotate to the code; or 'Both' to let the user choose the method for each review.

  Color for 'removed' messages
  : Color to use for the foreground of removed messages in the Locations view.

To set GNAT SAS options on a per-project basis, you can edit the project file directly, setting the relevant project attributes as described in *Analyzer package attributes*. Note that GNAT Studio provides completion, tooltips, outline and other common IDE features for project files through LSP and the Ada Language Server, helping you to customize your project more easily.

CHAPTER

NINE

USING GNAT SAS IN VISUAL STUDIO CODE

GNAT SAS can be used in Visual Studio Code by simply using its command line interface from the integrated terminal. In order to benefit from more advanced integrated GNAT SAS support in Visual Studio Code, we strongly recommend installing the Ada & SPARK Extension developed by AdaCore.

The extension provides automated tasks to run the analysis and generate reports in SARIF formats, which can then be displayed using any third-party extension that displays SARIF reports.

9.1 Prerequisites

- Make GNAT SAS available in the `PATH` before launching Visual Studio Code.
- Install the Ada & SPARK Extension.
- If the root directory of the Visual Studio Code workspace contains more than one project file, or if the project file you want to analyze is not in the workspace root directory, you need to set the `ada.projectFile` extension setting.

9.2 Running GNAT SAS in Visual Studio Code

9.2.1 Using the integrated terminal

The integrated terminal can be used to run arbitrary GNAT SAS commands. Refer to other chapters of this guide and the *GNAT SAS CLI Reference* for more information.

9.2.2 Using pre-defined tasks

The supported feature set is more limited than the features available in *GNAT Studio*. Support is based on Visual Studio Code Tasks. The extension provides pre-defined tasks for the following scenarios.

Tasks may be accessed by opening the action bar (*Cmd+Shift+P*) and selecting `Tasks: Run Task`.

Note: You cannot specify switches when calling GNAT SAS when using pre-defined tasks. To configure the analysis (mode, engines, ...) or the reporting, edit the `.gpr` project file as explained in *Configuring the Analysis*.

- `ada: Analyze the project with GNAT SAS`

  Analyze the project specified by the `ada.projectFile` extension setting, if specified, or the project located at the workspace's root if its `.gpr` file is the only one in that directory.

- `ada: Create a report after a GNAT SAS analysis`

  Generate a SARIF report for the previous analysis.

  If a third-party extension providing SARIF-viewing capabilities is available, it will automatically display the report (see *Viewing GNAT SAS Output in Visual Studio Code*).

- `ada: Analyze the project with GNAT SAS and produce a report`

  Combine the above project analysis and report tasks.

- `ada: Analyze the current file with GNAT SAS`

  Run an analysis on the file currently opened in the editor.

- `ada: Analyze the current file with GNAT SAS and produce a report`

  Combine the above current file analysis and report tasks.

See also:

Refer to Tasks for the complete list of tasks provided by the extension.

9.3 Viewing GNAT SAS Output in Visual Studio Code

Results from GNAT SAS generated in the SARIF format can be browsed in Visual Studio Code using any third-party extension with SARIF-viewing capabilities. Such third-party extensions may provide an interface to display the results interactively and inline messages and backtraces in the editor.

Note that SARIF-viewing features are specific to each particular extension and are not maintained by AdaCore.

Note: By default, file URIs in the report are made relative to the workspace's root directory. This is compatible with the third-party SARIF-viewing extensions that we tested. If needed, you can modify the file URIs' base directory with the `--root DIR` gnatsas report switch. This can be done directly in the `.gpr` file with:

```
project Prj is
    ...
    package Analyzer is
      for Switches ("report sarif") use ("--root=<PATH_TO_BASE_URI_DIR>");
    end Analyzer;
end Prj;
```

CHAPTER

TEN

GNAT SAS FILES REFERENCE

This chapter describes the files that GNAT SAS creates in order to store the analysis results, reviews, and information about runs.

The most important ones are *Message Files* and *Review File* which are usually the ones that need to be copied/versioned in order to share or back up your results.

See also:

For more information about versioning and sharing of GNAT SAS files, refer to *Workflows* that describes several practical use cases.

10.1 Message Files

During each analysis run, GNAT SAS creates a Static Analysis Messages (SAM) file that contains the messages generated by all its analysis engines that were enabled during the run. This file also contains the age of each message (see *Comparing GNAT SAS Runs*), as well as the last review that applies to it, if any.

The default name of the generated SAM file is `<prj>(.<file>).(fast|deep).sam`, where `prj` is the name of the project, `file` is the file specified with the switch `--file <file>` or `--files-from <file>`, and the suffix `(fast|deep)` is describes the analysis mode (note that it can also be described as `<prj>.<timeline>.sam`, as seen in *Timelines*). It is possible to change the name of a run's SAM file by setting the switch `--run-name <name>` (the `.sam` extension is automatically added if not present in `name`).

Warning: By default, any new analysis overrides the last run's SAM file, or, if `--run-name <name>` is set, any run with the same name. To avoid mistakes, GNAT SAS does make one backup before doing so (with the `.backup` extension). However, **we strongly encourage users to version the SAM file along with their sources**. This way, the results and sources are kept in sync: later on, when checking out a revision of the sources, you will get the analysis results associated with that revision.

The SAM files are generated in the *output directory*. That directory also contains the SAM files corresponding to baselines, named `<prj>(.<file>).(fast|deep).baseline.sam`. GNAT SAS takes care of copying, renaming, and setting to read-only mode, new baselines' SAM files (see *Comparing GNAT SAS Runs*).

10.2 Review File

User reviews are stored in a Static Analysis Reviews (SAR) file, or simply *review file*.

Importantly, this file is shared between all the *timelines* and therefore reviews will apply to the same message in all analyses results, independently of the analysis mode or other switches used for those analyses.

Note: As for SAM files, we strongly encourage users to version the review file.

See also:

See *Reviewing Results and Improving Code* for more information about reviews.

10.3 Additional artifacts

These artifacts are generally not relevant to users but may be useful for debugging purposes.

10.3.1 `runs_info` file

The *output directory* contains a file `<prj>.runs_info.json` that provides information about each timeline's last run and the path to the review file. It is updated each time GNAT SAS is called.

> **See also:**
>
> The switch `--list-timelines` should be used instead to get user-friendly information about timelines. See *Timelines* for reference.

10.3.2 GNAT SAS logs

GNAT SAS log file is generated by default under the *gnatsas directory* as `gnatsas/<prj>.gnatsas/gnatsas.log`, along with some GNAT SAS internal artifacts.

10.3.3 Engine-specific outputs

Integrated engines generate some internal files and logs in respective directories under the *gnatsas directory* as `gnatsas/<prj>.<engine>`.

10.3.4 Inspector annotations

If Inspector was enabled during the analysis, annotations from Inspector are generated and stored in subfolders of the *output directory*. Those artifacts can become quite large.

CHAPTER

ELEVEN

GNAT SAS MESSAGES REFERENCE

GNAT SAS outputs analysis results from multiple integrated engines as "messages" in a consistent, uniform way. The types of messages produced are described below. For an overview of GNAT SAS' output, refer to *Viewing Results*.

GNAT SAS uses static analysis to track possible run-time values of all variables, parameters, and components at each point in the source code. Using this information, it identifies conditions under which an Ada run-time check might fail (null pointer, array out of bounds, etc.), an exception might be raised, a user-written assertion might fail, or a numeric calculation might overflow or wraparound.

The messages that GNAT SAS emits, when it detects one of the above conditions, are referred to as *Check-Related Messages*.

Some messages that GNAT SAS emits do not fall into the *Check-Related* category and rather signal a suspicious coding (e.g., dead code, memory leak, unreferenced formal parameter, or code style violation).

The messages produced by GNAT SAS indicate particular lines of source code that could cause a crash, questionable behavior at run time or suspicious coding. The kind of check as well as the run-time expression that might not be satisfied (or that GNAT SAS is not able to prove statically) is given (such as *array index check (Inspector): check might fail; requires i in 1..10*), along with a ranking corresponding to a rough indication (based on heuristics) taking into account both the severity (likelihood that this message identifies a defect that could lead to incorrect results during execution), and the certainty of the message.

Note: *CWE* refers to the *Common Weakness Enumeration*, a community-developed dictionary of common software weaknesses, hosted on the web by The MITRE Corporation (https://cwe.mitre.org). This section contains links to and excerpts from that copyrighted material (see https://cwe.mitre.org/about/termsofuse.html). The current version of GNAT SAS is based on CWE version 4.19 (released on 2025-12-11). The numbers following *CWE* are the indices into the *CWE dictionary*, for weaknesses that correspond to the given GNAT SAS message. See *CWE to GNAT SAS Message Mapping* for a complete table of all CWE ids supported by GNAT SAS.

11.1 Categorization of Messages

GNAT SAS makes conservative assumptions in order to avoid missing potential problems. Because of these assumptions, GNAT SAS might generate messages that a programmer would decide are not true problems, or that are problems that would not occur during a normal execution of the program. For example, a given numeric overflow might occur only if the program ran continuously for decades. Messages that are not true problems are called *false positives*.

To help deal with false positives, GNAT SAS categorizes messages into three levels of ranking, according to the potential severity of the problem and to the likelihood that the message corresponds to a true problem.

For example, a *divide by zero* error might be categorized as **high**. An *overflow* that only occurs near the limits of the maximum long integer value, however, might be categorized as **medium**, especially if there are intermediate exit points in the loop that would prevent the overflow. Use of a global pointer variable that is not explicitly initialized might be

categorized as **low** because Ada provides well-defined default initial values for all pointers. These levels are determined by matching against patterns provided in a `MessagePatterns.xml` file.

Message	Description
High	High likelihood there is a defect on this line of code. It is likely to cause a problem on every execution, or the problem is very unlikely to be a false positive.
Medium	Moderate likelihood there is a bug on this line of code, or a definite occurrence of a code pattern that is in some contexts considered a bug. If a bug is present, it may cause a problem on some executions.
Low	There is a small chance there is a defect on the given line of code or the message is potentially not very interesting. It is worth investigating only when other (high, medium) messages have been reviewed, or when trying to eliminate any possibility of incorrect execution.
Info	Information about the code (e.g. subprogram not analyzed) or more details about the current message is provided via info messages.

11.2 CWE Categorization of Messages

GNAT SAS has been designated as CWE-Compatible by the MITRE Corporation's Common Weakness Enumeration (CWE) Compatibility and Effectiveness Program (cwe.mitre.org).

11.2.1 CWE to GNAT SAS Message Mapping

GNAT SAS will look for the following CWE weaknesses:

CWE weakness	Description	GNAT SAS Message
CWE-120, 124, 125, 126, 127, 129, 130, 131, 787	Buffer overflow/underflow	array index check
CWE-136, 137	Variant record field violation, Use of incorrect type in inheritance hierarchy	discriminant check, tag check
CWE-190, 191	Numeric overflow/underflow	overflow check
CWE-362, 366	Race condition	unprotected access, unprotected shared access
CWE-369	Division by zero	divide by zero
CWE-416	Use after free	use after free
CWE-457	Use of uninitialized variable	validity check
CWE-476	Null pointer dereference	access check
CWE-561	Dead (unreachable) code	dead code
CWE-563	Unused or redundant assignment	unused assignment, unused out parameter, useless reassignment
CWE-570	Expression is always false	test always false
CWE-571	Expression is always true	test always true
CWE-628	Incorrect arguments in call	precondition failure
CWE-667	Improper locking	unprotected access, unprotected shared access, mismatched protected access
CWE-682	Incorrect calculation	range check, postcondition failure
CWE-820	Missing synchronization	unprotected access, unprotected shared access
CWE-821	Incorrect synchronization	mismatched protected access
CWE-835	Infinite loop	loop does not complete normally
CWE-79	This CWE is detected by taint analysis, see *Taint Analysis* for more details.	cross site scripting
CWE-89	This CWE is detected by taint analysis, see *Taint Analysis* for more details.	sql injection
CWE-22	This CWE is detected by taint analysis, see *Taint Analysis* for more details.	path traversal
CWE-78	This CWE is detected by taint analysis, see *Taint Analysis* for more details.	os command injection
CWE-94	This CWE is detected by taint analysis, see *Taint Analysis* for more details.	code injection
CWE-77	This CWE is detected by taint analysis, see *Taint Analysis* for more details.	command injection
CWE-200	This CWE is detected by taint analysis, see *Taint Analysis* for more details.	information leak
CWE-20	This CWE is detected by taint analysis, see *Taint Analysis* for more details.	improper input validation
CWE-918	This CWE is detected by taint analysis, see *Taint Analysis* for more details.	server side request forgery
CWE-526	This CWE is detected by taint analysis, see *Taint Analysis* for more details.	cleartext storage

See *Inspector Messages* and *Infer Messages* for more details on the meaning of each GNAT SAS message.

11.2.2 CWE-676 Use of Potentially Dangerous Function

By design, most functions in Ada are safe to use. A notable exception are the predefined Ada functions `Ada.Unchecked_Conversion` and `Ada.Unchecked_Deallocation`.

GNAT SAS can check for the usage of these functions via the use of the following configuration pragmas:

```
pragma Restrictions (No_Unchecked_Conversion, No_Unchecked_Deallocation);
```

that you can put in a `gnatsas.adc` file and reference this file in your project file via the `Global_Configuration_Pragmas` project attribute, as described in *Configuration of System Package (system.ads)*.

11.2.3 CWE Top 25 Most Dangerous Software Errors

The Common Weakness Enumeration (CWE) Top 25 Most Dangerous Software Errors (CWE Top 25) list is continuously updated to maintain the latest trends in dangerous and widespread software weaknesses that exist in the wild. Please visit https://cwe.mitre.org/top25/ for the most up-to-date information.

Amongst the 2025 CWE Top 25, GNAT SAS supports the following CWEs:

Rank	ID	GNAT SAS Message
1	CWE-79	cross site scripting
2	CWE-89	sql injection
5	CWE-787	array index check
6	CWE-22	path traversal
7	CWE-416	use after free
8	CWE-125	array index check
9	CWE-78	os command injection
10	CWE-94	code injection
11	CWE-120	array index check
13	CWE-476	access check
18	CWE-20	improper input validation
20	CWE-200	information leak
22	CWE-918	server side request forgery
23	CWE-77	command injection

11.3 GNATcheck Messages

By default, GNAT SAS reports the messages issued by GNATcheck (unless GNATcheck is explicitly disabled). GNAT SAS enables five GNATcheck rules by default:

Message	Description
suspicious equalities	Flag "or" expressions whose left and right operands are inequalities referencing the same entity and a literal and "and" expressions whose left and right operands are equalities referencing the same entity and a literal.
duplicate branches	Flag a sequence of statements that is a component of an `if` statement or of a `case` statement alternative, if the same `if` or `case` statement contains another sequence of statements as its component (or a component of its `case` statement alternative) that is syntactically equivalent to the sequence of statements in question.

continues on next page

Table 2 – continued from previous page

Message	Description
same logic	Flags expressions that contain a chain of infix calls to the same boolean operator (`and`, `or`, `and then`, `or else`, `xor`) if an expression contains syntactically equivalent operands.
same operands	Flags infix calls to binary operators `/`, `=`, `/=`, `>`, `>=`, `<`, `<=`, `-`, `mod`, `rem` (except when operating on floating point types) if operands of a call are syntactically equivalent.
same tests	Flags condition expressions in `if` statements or `if` expressions if a statement or expression contains another condition expression that is syntactically equivalent to the first one.

The user may extend that list by specifying predefined rules, or custom rules as explained in *Configuring GNATcheck*.

The complete list of message kinds and their descriptions, along with examples, can be found in the GNATcheck reference manual in the Predefined Rules chapter.

11.4 Infer Messages

By default, GNAT SAS reports messages issued by Infer (unless Infer is explicitly disabled). See *Configuring Infer* for more details.

Certain limitations and heuristics can apply to infer messages. See *Infer's Limitations and Heuristics* for the list of these limitations and heuristics.

First, some messages correspond exactly to messages emitted by Inspector, and won't be described further: `validity check` (see *Uninitialized and Invalid Variables*) and `unassigned parameter` (see *Warning Messages*).

Second, the following checkers are specific to Infer:

<table>
<tr><th>Message</th><th>Description</th></tr>
<tr><td>access check</td><td>The main difference with Inspector messages is that Infer tries to find a path from the assignment to a null pointer (appearing as null in the source code), to its dereference. As a consequence, the example shown earlier is not flagged by Infer.
Here is an example that is flagged:

```
procedure P is
  type Int_Ptr is access Integer;

  Some_Cond : Boolean with Volatile;

  function Get_Ptr return Int_Ptr is
  begin
    if Some_Cond then
      return new Integer;
    else
      return null;
    end if;
  end Get_Ptr;

  X : Int_Ptr := Get_Ptr;
begin
  X.all := 42;
end P;
```

Infer will flag:

```
p.adb:17:12: high: access check (Infer): `X` could be null (from the␣
↪call to `p.get_ptr`
on line 15) and is dereferenced
```
</td></tr>
<tr><td>memory leak</td><td>Memory leaks are reported when a variable is dynamically allocated with new and not deallocated with an instance of Ada.Unchecked_Deallocation.

```
procedure P is
  X : access Integer := new Integer;
begin
  null;
end P;
```

Infer will flag:

```
p.adb:2:26: medium warning: memory leak (Infer): Memory dynamically␣
↪allocated by `new`
on line 2 is not freed after the last access at line 2, column 26
```
</td></tr>
</table>

continues on next page

Table 3 – continued from previous page

Message	Description

use after free	An error is emitted when a pointer is dereferenced after it has been freed.
	Here is a simple example:
	`with Ada.Unchecked_Deallocation;`
	
	`procedure P is`
	`  type Int_Ptr is access Integer;`
	
	`  procedure Free is new Ada.Unchecked_Deallocation (Integer, Int_Ptr);`
	
	`  X : Int_Ptr;`
	`  Y : Int_Ptr;`
	`begin`
	`  X := new Integer;`
	`  Y := X;`
	
	`  Free (X);`
	
	`  Y.all := 42;`
	`end P;`
	Infer will flag:
	`p.adb:16:13: high: use after free (Infer): accessing memory that was invalidated by call to`
	`` `p.free` on line 14``

double free	A medium error is emitted when a pointer pointing to an already deallocated memory location is passed to an instance of the 'Ada.Unchecked_Deallocation' generic procedure.
	Here is a simple example:
	`with Ada.Unchecked_Deallocation;`
	
	`procedure Test is`
	`  type Int_Access is access all Integer;`
	`  procedure Free is new Ada.Unchecked_Deallocation (Integer, Int_Access);`
	
	`  X : Int_Access := new Integer;`
	`  Y : Int_Access := X;`
	`begin`
	`  Free (X);`
	`  Free (Y);`
	`end;`
	Infer will flag:
	`test.adb:11:4: medium: double free [CWE 415] (Infer): Y points to memory already deallocated at test.adb:10:4`

continues on next page

Table 3 – continued from previous page

Message	Description

uninitialized value passed to call	A warning is emitted when an uninitialized value of a composite type is passed to a call in an in or in out mode. The checker considers a value of a composite type uninitialized when the composite type and any of its components don't have default initialization, it is a value of a local variable or an out parameter, and any component of the value isn't written before the call.
	Here is an example:
	`type Rec is record`
	`   F1 : Integer;`
	`   F2 : Integer;`
	`end record;`
	
	`function Compute (R : Rec) return Integer with Import;`
	
	`procedure Uninitialized_Value_Passed_To_Call (R : out Rec) is`
	`begin`
	`   R.F2 := Compute (R);`
	`end Uninitialized_Value_Passed_To_Call;`
	Infer will flag:
	`p.adb:10:15: medium warning: uninitialized value passed to call (Infer): R not written before being passed to call to Compute`
	This warning is not emitted for null records and for calls to primitives of Ada containers. In addition, as a heuristics, the warning is also not emitted for calls to primitives of tagged types.

access without open	A warning is emitted when a file handler which is not opened or opened in incorrect mode is used to access a file.
	Here is an example:
	`with Ada.Text_IO; use Ada.Text_IO;`
	
	`procedure Write_Without_Open is`
	`   F : File_Type;`
	`begin`
	`   Create (F, In_File, File_Name);`
	`   Put_Line (F, "Hello World");`
	`end Write_Without_Open;`
	Infer will flag:
	`a.adb:7:7: medium warning: access without open (Infer): file F is written and opened for read access`

continues on next page

Table 3 – continued from previous page

Message	Description

double open	A warning is emitted when an open operation is called on a file handler that has already been opened.
	Here is an example:
	`with Ada.Text_IO; use Ada.Text_IO;`
	
	`procedure Double_Open is`
	`  F : File_Type;`
	`begin`
	`  Create (F, Out_File, File_Name);`
	`  Create (F, Out_File, File_Name);`
	`end Double_Open;`
	Infer will flag:
	`a.adb:7:7: medium warning: double open (Infer): file F is already opened at a.adb:6:4`

close without open	A warning is emitted when a close operation is called on a file handler that hasn't been opened.
	Here is an example:
	`with Ada.Text_IO; use Ada.Text_IO;`
	
	`procedure Close_Without_Open is`
	`  F : File_Type;`
	`begin`
	`  Close (F);`
	`end Close_Without_Open;`
	Infer will flag:
	`a.adb:6:7: medium warning: close without open (Infer): file F is closed without being opened`

continues on next page

Table 3 – continued from previous page

Message	Description

function with side effects	A warning is emitted when a function has side effects.
	This checker has to be explicitly enabled using the Infer-specific switch `--side-effects`. By default, the checker looks for side effects on global variables, uplevel variables (variables declared in some enclosing subprogram), and dereferences of parameters of access types passed in IN mode (e.g., updating *P.all* where *P* is an IN parameter). Infer switches `--ignore-global-side-effects`, `--ignore-uplevel-side-effects`, and `--ignore-param-derefs-side-effects` can be used to ignore global variable, uplevel variables, and dereferences of IN parameters respectively. The Infer switch `--side-effects-ignore` can be used to specify global or uplevel variables that will be ignored by the analysis, and can be repeated (e.g. `--side-effects-ignore P1.G1 --side-effects-ignore P2.G2`).
	Here is an example:
	`package body P is`
	`   G : Integer;`
	`   function Side_Effects (I : Integer) return Integer`
	`   is`
	`   begin`
	`      G := G + I;`
	`      return G;`
	`   end Side_Effects;`
	Infer will flag:
	`p.adb:5:4: medium warning: function with side effects (Infer): Function reprod.side_effects has side effects on global variable `G``

continues on next page

Table 3 – continued from previous page

Message	Description

function with side effects in conditional	A warning is emitted when a function having side effects is used in the right hand side of a short-circuit conditional.
	This checker has to be explicitly enabled using the Infer-specific switch `--side-effects-in-conditionals`.
	Infer switches `--ignore-global-side-effects`, `--ignore-uplevel-side-effects`, `--ignore-param-derefs-side-effects`, and `--side-effects-ignore` have the same effect as for the checker finding functions with side effects.
	Here is an example:
	`package body P is`
	
	`   G : Integer;`
	
	`   function Side_Effects (I : Integer) return Integer`
	`   is`
	`   begin`
	`      G := G + I;`
	`      return G;`
	`   end Side_Effects;`
	
	`   procedure P`
	`   is`
	`   begin`
	`      if G = 2 or else Side_Effects (2) = 2 then`
	`         G := 1;`
	`      end if;`
	`   end P;`
	Infer will flag:
	`p.adb:15:24: medium warning: function with side effects in conditional (Infer):`
	`Function reprod.side_effects with side effects on global variable `G` in conditional`

continues on next page

Table 3 – continued from previous page

Message	Description

write through global and parameter	Emitted on calls where a global variable is used as parameter to a subprogram that modifies said global both through its global reference and through the parameter reference. For example:
	`procedure P is`
	`  X : Integer := 0;`
	`  procedure I (A : in out Integer) is`
	`  begin`
	`  X := 1;`
	`  A := 2;`
	`  end;`
	`begin`
	`  I (X);`
	`end P;`
	Infer will emit:
	`p.adb:9:3: medium: write through global and parameter (Infe r): p.i␣ ↪may write to`
	`X through a global reference, but X may be overwritten by an out␣ ↪parameter copy`

os command injection	The product constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.
	A message of this kind is emitted when a flow from a predefined *source* reaches a predefined *sink* without proper sanitization.
	`procedure P is`
	`  function Get return String`
	`  with Taint_Source => (My_Source'Result => Console);`
	`  procedure Spawn (Cmd : String)`
	`  with Taint_Sink => (Cmd => OS_Command_Injection);`
	`begin`
	`  Spawn (Get);`
	`end P;`
	Infer will emit:
	`p.adb:8:3: medium: os command injection [CWE 78] (Infer): untrusted␣ ↪data Get with`
	`kind 'console' flows to a sink`
	For more details, see *Taint Analysis* below.

cross site scripting	The product does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users. A message of this kind is emitted when a flow from a predefined *source* reaches a predefined *sink* without proper sanitization. For more details, see *Taint Analysis* below.

continues on next page

Table 3 – continued from previous page

Message	Description
sql injection	The product constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data. A message of this kind is emitted when a flow from a predefined *source* reaches a predefined *sink* without proper sanitization. For more details, see *Taint Analysis* below.
path traversal	The product uses external input to construct a pathname that should be within a restricted directory, but it does not properly neutralize '.../...//' (doubled triple dot slash) sequences that can resolve to a location that is outside of that directory. A message of this kind is emitted when a flow from a predefined *source* reaches a predefined *sink* without proper sanitization. For more details, see *Taint Analysis* below.
code injection	The product constructs all or part of a code segment using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the syntax or behavior of the intended code segment. A message of this kind is emitted when a flow from a predefined *source* reaches a predefined *sink* without proper sanitization. For more details, see *Taint Analysis* below.
command injection	The product constructs all or part of a command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended command when it is sent to a downstream component. A message of this kind is emitted when a flow from a predefined *source* reaches a predefined *sink* without proper sanitization. For more details, see *Taint Analysis* below.
information leak	The product exposes sensitive information to an actor that is not explicitly authorized to have access to that information. A message of this kind is emitted when a flow from a predefined *source* reaches a predefined *sink* without proper sanitization. For more details, see *Taint Analysis* below.
improper input validation	The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly. A message of this kind is emitted when a flow from a predefined *source* reaches a predefined *sink* without proper sanitization. For more details, see *Taint Analysis* below.
server side request forgery	The web server receives a URL or similar request from an upstream component and retrieves the contents of this URL, but it does not sufficiently ensure that the request is being sent to the expected destination. A message of this kind is emitted when a flow from a predefined *source* reaches a predefined *sink* without proper sanitization. For more details, see *Taint Analysis* below.
cleartext storage	The product stores sensitive information in cleartext within a resource that might be accessible to another control sphere. A message of this kind is emitted when a flow from a predefined *source* reaches a predefined *sink* without proper sanitization. For more details, see *Taint Analysis* below.

11.5 Taint Analysis

Taint analysis tracks the flow of untrusted data (originating from *sources*) through a program. It identifies cases where this data reaches sensitive operations (*sinks*) without passing through appropriate validation or cleaning mechanisms (*sanitizers*). Such cases may result in serious security vulnerabilities, including SQL injection, cross-site scripting (XSS), or command injection.

Taint analysis is performed by the Infer engine and is enabled by default. It can be controlled with the following switches:

- `--taint` (default): enable taint analysis.
- `--no-taint`: disable taint analysis entirely.

Note that taint analysis requires the Infer engine to be enabled (see `--infer` / `--no-infer`).

11.5.1 Configuration

Taint sources, sinks, and sanitizers are declared using the aspects Taint_Source, Taint_Sink, and Taint_Sanitizer, respectively.

Here is an example of a taint source declaration:

```
function My_Source return String
   with Taint_Source => (My_Source'Result => Filesystem);
```

Each taint source is associated with a user-defined *Kind* that categorizes the type of untrusted data. This kind identifies the *origin* of the data (see *source kinds*).

Taint sinks represent operations where the use of untrusted data may lead to a vulnerability if left unsanitized. Each sink must specify a *Kind* that indicates the *weakness kind* it represents:

```
procedure Spawn (Args : Argument_List)
   with Taint_Sink => (Args => OS_Command_Injection);
```

Taint sanitizers indicate where tainted data is validated or cleaned. Unlike sources, sanitizers are tied to the **weakness kind** that is, a sanitizer is declared to mitigate a *specific kind of weakness*, regardless of where the taint originated:

```
function Sanitize_Args (Args : Argument_List) return Argument_List
   with Taint_Sanitizer => (Sanitize_Args'Result => OS_Command_Injection);
```

The expected argument of a taint aspect (`Taint_Source`, `Taint_Sink`, or `Taint_Sanitizer`) is an aggregate, where each designator is refered to the *taint target*. It specifies the entity **on which the taint property is applied** - meaning:

- For Taint_Source: the entity that will **store the untrusted data** after the subprogram is called
- For Taint_Sink: the entity that **must not contain tainted data** when passed to the subprogram
- For Taint_Sanitizer: the entity **being sanitized** as a result of calling the subprogram

The taint target must be one of the following:

- A parameter of the specified subprogram
- A visible uplevel variable (declared in an enclosing scope)
- A visible global variable
- The `'Result` attribute of the current subprogram (for functions)
- A pointer dereference, to indicate the taint applies to the pointee rather than the pointer itself

When the taint applies to several entities together, use a list of taint targets:

```
procedure My_Source (Arg1 : out Integer; Arg2 : out String)
   with Taint_Source => ((Arg1, Arg2) => Filesystem);
```

When each target has its own taint kind, list them separately:

```
procedure My_Source (Arg1 : out Integer; Arg2 : out String)
   with Taint_Source => (Arg1 => Filesystem, Arg2 => Network);
```

When several kinds apply to the same set of taint targets, group the kinds in another aggregate:

```
procedure My_Source (Arg1 : out Integer; Arg2 : out String)
   with Taint_Source => ((Arg1, Arg2) => (Filesystem, Network));
```

If aspects are not available, the same annotations can be expressed using pragmas. Place the pragma immediately after the subprogram declaration, just as you would for a Precondition pragma:

```
procedure My_Source (Arg1 : out Integer; Arg2 : out String; Arg3 : out String);
pragma Taint_Source (((Arg1, Arg2) => (Filesystem, Network), Arg3 => Console));
```

The double parentheses are intentional and mandatory: this particular pragma only accepts a single argument which is itself an aggregate.

11.5.2 Kinds of Sources

The expressions in the taint-specification aggregate for `Taint_Source` is an indentifier that categorizes the type of untrusted data.

The following table lists supported source kinds:

Table 4: Sources

Kind	Description
Cookie	Data from HTTP cookies
Command_Line	Data from the command line
Console	Data from the console
Database	Data from a database
Environment	Data from environment variables
Filesystem	Data from a file
HTTP	Data from incoming HTTP requests
Network	Data from network connections

11.5.3 Trusting Sources

Trusting and distrusting sources can be achieved with the `Trusted_Taint_Sources` and `Distrusted_Taint_Sources` attributes of the Analyzer package of your GPR file. These two attributes are a list of *source kinds*. No taint sources are trusted by default, so for example:

```
package Analyzer is
   for Trusted_Taint_Sources use ("network");
end Analyzer;
```

This piece of code will prevent GNATSAS from emitting messages caused by passing data from tainted network sources to vulnerable sinks, but all other messages caused by other taint sources will be preserved. Disabling messages about all taint sources except for specific ones can be achieved by trusting the special `"all"` taint source and then listing the distrusted sources in the `Distrusted_Taint_Sources` attribute:

```
package Analyzer is
   for Trusted_Taint_Sources use ("all");
   for Distrusted_Taint_Sources use ("network", "filesystem");
end Analyzer;
```

11.5.4 Kinds of Weaknesses

The expressions in the taint-specification aggregate for `Taint_Sink` and `Taint_Sanitizer` is an indentifier that specifies the type of weakness.

The following table lists supported weakness kinds:

Table 5: Weaknesses

Kind	Description	CWE weakness
Cross_	The product does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.	CWE-79
Sql_inj	The product constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data.	CWE-89
Path_tr	The product uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the product does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.	CWE-22
Os_cor	The product constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.	CWE-78
Code_i	The product constructs all or part of a code segment using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the syntax or behavior of the intended code segment.	CWE-94
Command_i	The product constructs all or part of a command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended command when it is sent to a downstream component.	CWE-77
Information_le	The product exposes sensitive information to an actor that is not explicitly authorized to have access to that information.	CWE-200
Improper_	The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly.	CWE-20
Server_	The web server receives a URL or similar request from an upstream component and retrieves the contents of this URL, but it does not sufficiently ensure that the request is being sent to the expected destination.	CWE-918
Clear-t-ext_sto	Information stored in an environment variable can be accessible by other processes with the execution context, including child processes that dependencies are executed in, or serverless functions in cloud environments. An environment variable's contents can also be inserted into messages, headers, log files, or other outputs. Often these other dependencies have no need to use the environment variable in question. A weakness that discloses environment variables could expose this information.	CWE-526

11.5.5 Semantics

The analysis tracks flows of tainted data from sources (of any kind) to sinks (of a particular weakness kind). A taint error is reported when:

1. Tainted data (from a source of any kind) reaches a sink of kind `K`, and
2. The data was **not sanitized for kind** `K` along the way.

This means that:

- The source kind and sink kind can differ.
- Only sanitizers **matching the sink's weakness kind** can prevent a taint error.
- Sanitizers are not generic: they mitigate specific weakness kinds.

This design supports precise modeling of weaknesses. For example, the same user input might require different sanitization depending on whether it's used in a shell command or an SQL query.

If a call is made to a subprogram with no taint summary that is, it is not analyzed or annotated the analysis conservatively assumes that any tainted input may taint all `out` or `in out` parameters, as well as the function result. This does not apply to global variables.

11.6 GNAT Warnings Messages

By default, GNAT SAS reports a selection of warnings from the GNAT compiler, as listed in the table below. The list of all available GNAT warnings and their description can be found in the GNAT user's guide in the Warning Message Control section. For more details on this capability see *Configuring GNAT Warnings*.

Message	GNAT switch	Description
normal warnings mode	`-gnatwn`	Assorted set of messages with no common theme.
bad fixed value	`-gnatwb`	Warnings for static fixed-point expressions whose value is not an exact multiple of Small.
unused assignment	`-gnatwm`	Warnings for variables that are assigned but whose value is never read.
missing parenthesis	`-gnatwq`	Warnings for expressions whose lack of parenthesis may be ambiguous from a reader's point of view.
redundant construct	`-gnatwr`	Warnings for redundant constructs.
unassigned variable	`-gnatwv`	Warnings on access to variables which may not be properly initialized.
low bound assumption	`-gnatww`	Warnings on accessing unconstrained string parameters with a literal or S'Length.
export/import mismatch	`-gnatwx`	Warnings on potential issues arising from interfacing with other programming languages.
assertion failure	`-gnatw.a`	Warnings on assertions the compiler can tell will always fail.
biased representation	`-gnatw.b`	Warnings on clauses that may force the compiler to use a biased representation for an integer type (e.g. using a single bit to represent the range 10..11).
overlapping actuals	`-gnatw.i`	Warnings on statically detectable overlapping actuals in a subprogram call.
suspicious modulus value	`-gnatw.m`	Warnings on modulus values matching the size of the type they're attached to.
suspicious parameter order	`-gnatw.p`	Warnings on subprogram calls where all parameters are variables whose name match the formals of the subprogram but are not in the same order.
object renaming function	`-gnatw.r`	Warnings on object renamings that rename function calls.

11.7 Inspector Messages

GNAT SAS reports messages issued by Inspector (unless Inspector is disabled). See *Configuring Inspector* for more details.

11.7.1 Understanding messages on *generics*

In some cases, Inspector messages are reported on an instance of a *generic* unit, but the message references entities by the names used in the generic template rather than in the instance. In such cases, it is important to understand that Inspector, like SPARK, only analyzes fully-instantiated code and the message was actually emitted on a given instantiation of the generic. When reading such messages, users should then look at the instantiation at the reported location, and compare with the generic code, to understand how the message applies.

Note that when messages are emitted on multiple instances, GNAT SAS will de-duplicate them and only report one message to avoid extra noise, as in most cases the issue comes from the generic's implementation.

11.7.2 Run-Time Checks

These are language defined run-time checks that are searched by Inspector:

Message	Description

array index check	Index value could be outside the array bounds (CWE 120, 124, 125-127, 129-131). This is also known as *buffer overflow*. For example in the following code:
	`procedure Buffer_Overflow is`
	`    type Int_Array is array (0 .. 2) of Integer;`
	`    X, Y : Int_Array;`
	`begin`
	`    for I in X'Range loop`
	`    X (I) := I + 1;`
	`    end loop;`
	
	`    for I in X'Range loop`
	`    Y (X (I)) := I;  -- Bad when I = 2, since X (I) = 3`
	`    end loop;`
	`end Buffer_Overflow;`
	the buffer overflow will be flagged with:
	`buffer_overflow.adb:10:7: high: array index check (Inspector): check fails here;`
	`requires (X (I)) in 0..2`

continues on next page

Table 7 – continued from previous page

Message	Description

divide by zero	The second operand of a divide, mod or rem operation could be zero (CWE 369). For example in the following code:
	`procedure Div is`
	`    type Int is range 0 .. 2**32-1;`
	`    A : Int := Int'Last;`
	`    X : Integer;`
	`begin`
	`    for I in Int range 0 .. 2 loop`
	`       X := Integer (A / I);  --  division by zero when I = 0`
	`    end loop;`
	`end Div;`
	Inspector will detect the division by zero with:
	`div.adb:7:23: high: divide by zero (Inspector): check fails here; requires I /= 0`

access check	Attempting to dereference a reference that could be null (CWE 476). For example in the following code:
	`procedure Null_Deref is`
	`    type Int_Access is access Integer;`
	`    X : Int_Access;`
	`begin`
	`    if X = null then`
	`       X.all := 1;  -- null dereference`
	`    end if;`
	`end Null_Deref;`
	the dereference at line 6 will generate:
	`null_deref.adb:6:7: high: access check (Inspector): check fails here`

continues on next page

Table 7 – continued from previous page

Message	Description

range check	A calculation may generate a value outside the bounds of an Ada type or subtype and generate an invalid value (CWE 682). For example in the following code:
	1 `procedure Out_Of_Range is`
	2 `    subtype Constrained_Integer is Integer range 1 .. 2;`
	3 `    A : Integer;`
	4
	5 `    procedure Proc_1 (I : in Constrained_Integer) is`
	6 `    begin`
	7 `       A := I + 1;`
	8 `    end Proc_1;`
	9
	10 `begin`
	11 `    A := 0;`
	12 `    Proc_1 (I => A);  --  A is out-of-range of parameter I`
	13 `end Out_Of_Range;`
	the call at line 12 will generate:
	`out_of_range.adb:12:17: high: range check (Inspector): check fails here;`
	`requires A in 1..2`
	because A is not in the range of `Constrained_Integer`.

continues on next page

Table 7 – continued from previous page

Message	Description
overflow check	A calculation may overflow the bounds of a numeric type and wrap around. The likelihood this will affect operation of the program depends on how narrow is the range of the numeric value (CWE 190-191). For example in the following code:

```
1  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  procedure Overflow is
5     Attempt_Count : Integer := Integer'Last;
6     --  Gets reset to zero before attempting password read
7     PW            : Natural;
8  begin
9     --  Oops forgot to reset Attempt_Count
10    loop
11       Put ("Enter password to delete system disk");
12       Get (PW);
13       if PW = 42 then
14          Put_Line ("system disk deleted");
15          exit;
16       else
17          Attempt_Count := Attempt_Count + 1;
18
19          if Attempt_Count > 3 then
20             Put_Line ("max password count reached");
21             raise Program_Error;
22          end if;
23       end if;
24    end loop;
25 end Overflow;
```

the expression `Attempt_Count + 1` will overflow since Attempt_Count is initialized with the largest integer value (`Integer'Last`):

```
overflow.adb:17:41: high: overflow check (Inspector): check fails␣
↪here;
requires Attempt_Count /= Integer_32'Last
```

continues on next page

Table 7 – continued from previous page

Message	Description

aliasing check	A parameter that can be passed by reference is aliased with another parameter or a global object and a subprogram call might violate the associated precondition by writing to one of the aliased objects and reading the other aliased object, possibly resulting in undesired behavior. Aliasing checks are generally expressed as a requirement that a parameter not be the same as some other parameter, or not match the address of some global object and will be flagged as a precondition check in the caller. For example in the following code:
	`procedure Alias is`
	`   type Int_Array is array (1 .. 10) of Integer;`
	`   A, B : Int_Array := (others => 1);`
	`   procedure In_Out`
	`     (A : Int_Array;`
	`      B : Int_Array;`
	`      C : out Int_Array) is`
	`   begin`
	`      --  Read A multiple times, and write C multiple times:`
	`      --  if A and C alias and are passed by reference, we are in trouble!`
	`      C (1) := A (1) + B (1);`
	`      C (1) := A (1) + B (1);`
	`   end In_Out;`
	`begin`
	`   --  We pass A as both an 'in' and 'out' parameter: danger!`
	`   In_Out (A, B, A);`
	`end Alias;`
	the call to `In_Out` will be flagged with:
	`alias.adb:18:4: high: precondition <aliasing check> (Inspector):`
	`precondition failure on call to alias.in_out; requires C /= A`

continues on next page

Table 7 – continued from previous page

Message	Description

tag check	A tag check (incorrect tag value on a tagged object) may fail (CWE 136, 137). For example in the following code:
	`procedure Tag is`
	`   type T1 is tagged null record;`
	
	`   package Pkg is`
	`      type T2 is new T1 with null record;`
	`      procedure Op (X : T2) is null;`
	`   end Pkg;`
	`   use Pkg;`
	
	`   type T3 is new T2 with null record;`
	
	`   procedure Call (X1 : T1'Class) is`
	`   begin`
	`      Op (T2'Class (X1));`
	`   end Call;`
	
	`   X1 : T1;`
	`   X2 : T2;`
	`   X3 : T3;`
	`begin`
	`   Call (X1); -- not OK, Call requires T2'Class`
	`   Call (X2); -- OK`
	`   Call (X3); -- OK`
	`end Tag;`
	the first call will be flagged with:
	`tag.adb:21:4: high: precondition <tag check> (Inspector):`
	`precondition failure on call to tag.call; requires X1'Tag in {tag.`
	`↪pkg.t2, tag.t3}`

continues on next page

Table 7 – continued from previous page

Message	Description

discriminant check	A field for the wrong variant/discriminant is accessed (CWE 136, 137). For example in the following code:
	`with Ada.Text_IO; use Ada.Text_IO;`
	
	`procedure P is`
	`  type Device_State is (Off, On);`
	
	`  type Device (State : Device_State := On) is record`
	`  case State is`
	`  when Off =>`
	`  null;`
	`  when On =>`
	`  Power_Level : Integer;`
	`  end case;`
	`  end record;`
	
	`  function Get_Device return Device is`
	`  begin`
	`  return (State => Off);`
	`  end Get_Device;`
	
	`  My_Device : Device;`
	`begin`
	`  My_Device := Get_Device;`
	`  Put_Line (My_Device.Power_Level'Image);`
	`end P;`
	reading the `Power_Level` attribute will generate the following message:
	`p.adb:23:23: high: discriminant check [CWE 137] (Inspector): check fails here;`
	`  requires not (device'Discriminant_Check#2 (My_Device.State))`

negative operand of exponentiation	The operand of integer exponentiation is negative. For example in the following code:
	`function Neg_Operand return Integer is`
	`  function Compute return Integer`
	`  is`
	`  begin`
	`  return -2;`
	`  end Compute;`
	`begin`
	`  return 2 ** Compute;`
	`end Neg_Operand;`
	the exponentiation on line 8 will generate the following message:
	`neg_operand.adb:8:16: high: negative operand of exponentiation (Inspector):`
	`check fails here; requires (compute'Result) >= 0`

continues on next page

Table 7 – continued from previous page

Message	Description
precondition	A subprogram call that might violate the subprogram's preconditions (CWE 628). Checks (as listed above) associated with this precondition are listed as part of the message. In the expression associated with the message, the precondition being checked is expressed in terms of the variables of the called subprogram, rather than the calling one. In GNAT Studio, you can use the `Goto body of xxx` capability to view the called subprogram and its associated preconditions. A run-time check as listed above (e.g. range check) may add a precondition on a subprogram, which is then checked at all call sites and will generate a precondition message in case a failure of this precondition may occur. See *Inspector Annotations* and *Understanding the Differences between Preconditions and Run-time Errors* for more information on preconditions. (Same CWE values as the associated checks). See `aliasing check` and `tag check` examples above.

11.7.3 User Checks

Inspector will analyze checks inserted manually by the programmer either as an explicit Assert or via Ada 2012 aspects.

Message	Description

assertion	A user assertion (using e.g. pragma Assert) could fail. For example in the following code:
	`procedure Assert is`
	`  function And_Or (A, B : Boolean) return Boolean is`
	`  begin`
	`  return False;`
	`  end And_Or;`
	`begin`
	`  pragma Assert (And_Or (True, True));`
	`end Assert;`
	Inspector will detect that the pragma Assert will never be True:
	`assert.adb:9:19: high: assertion (Inspector): check fails here;`
	`requires (and_or'Result) /= false`

continues on next page

Table 8 – continued from previous page

Message	Description

conditional check	An exception could be raised depending on the outcome of a conditional test in user code. For example in the following code:
	`with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;`
	`with Ada.Text_IO; use Ada.Text_IO;`
	
	`procedure Overflow is`
	`  Attempt_Count : Integer := Integer'Last;`
	`  --  Gets reset to zero before attempting password read`
	`  PW            : Natural;`
	`begin`
	`  --  Oops forgot to reset Attempt_Count`
	`  loop`
	`  Put ("Enter password to delete system disk");`
	`  Get (PW);`
	`  if PW = 42 then`
	`  Put_Line ("system disk deleted");`
	`  exit;`
	`  else`
	`  Attempt_Count := Attempt_Count + 1;`
	
	`  if Attempt_Count > 3 then`
	`  Put_Line ("max password count reached");`
	`  raise Program_Error;`
	`  end if;`
	`  end if;`
	`  end loop;`
	`end Overflow;`
	Inspector will detect that `Attempt_Count` will always be greater than 3:
	`overflow.adb:21:13: high: conditional raise (Inspector):`
	`exception is raised in a conditional branch here;`
	`requires Attempt_Count <= 3`

raise exception	An exception is being raised on a reachable path. This is similar to `conditional check`, but the exception is raised systematically instead of conditionally. For example:
	`procedure Raise_Exc is`
	`  X : Integer := (raise Program_Error);`
	`begin`
	`  null;`
	`end Raise_Exc;`
	will generate:
	`raise_exc.adb:2:20: low: raise exception (Inspector): unconditional raise`

continues on next page

Table 8 – continued from previous page

Message	Description

user precondition	A call might violate a subprogram's specified precondition (CWE 628). This specification may be written as a pragma Precondition, or as a Pre aspect in Ada 2012 syntax. For example the following code:
	`procedure Pre is`
	`  function "**" (Left, Right : Float) return Float`
	`  with Import, Pre => Left /= 0.0;`
	
	`  A : Float := 1.0;`
	`begin`
	`  A := (A - 1.0) ** 2.0;`
	`end Pre;`
	will generate:
	`pre.adb:7:19: high: precondition <user precondition> (Inspector):precondition`
	`failure on call to pre."**";`
	`requires Left in ([IEEE_32_Float'First..0.0) | (0.0..IEEE_32_Float'Last])`

continues on next page

Table 8 – continued from previous page

Message	Description

postcondition	The subprogram's body may violate its specified postcondition (CWE 682). This specification may be written as a pragma Postcondition, or as a Post aspect in Ada 2012 syntax. For example in the following code:
	`procedure Post is`
	`   type States is (Normal_Condition, Under_Stress, Bad_Vibration);`
	`   State : States;`
	`   function Stress_Is_Minimal return Boolean is (State = Normal_Condition);`
	`   function Stress_Is_Maximal return Boolean is (State = Bad_Vibration);`
	`   procedure Decrement with`
	`     Pre  => not Stress_Is_Minimal,`
	`     Post => not Stress_Is_Maximal;`
	`   procedure Decrement is`
	`   begin`
	`      State := States'Val (States'Pos (State) + 1);`
	`   end Decrement;`
	`begin`
	`   ...`
	`end Post;`
	Inspector detects that the postcondition of `Decrement` will always be violated, given the precondition:
	`post.adb:11:14: high: postcondition (Inspector): postcondition failure on call to`
	`post.decrement; requires not (stress_is_maximal'Result)`

In addition, Inspector treats the run-time check associated with an Assume pragma differently in this respect than other checks (notably, than the check associated with an Assert pragma). Pragma Assume is defined for the specific purpose of communicating assumptions to static-analysis tools (e.g., GNAT SAS or SPARK) that the tools can assume to be correct without any further analysis - the burden of ensuring the correctness of the assumption is on the user, not the tool, and misuse of Assume pragmas can invalidate Inspector's analysis.

Assert and Assume pragmas take the same arguments and have the same effect at run time (except that they are enabled/disabled by different assertion policies), but GNAT SAS does not generate a message if the check of an Assume pragma cannot be shown to always pass.

See *Improve Your Code Specification* for more details on pragma Assert and Assume.

11.7.4 Uninitialized and Invalid Variables

Inspector and Infer analyses attempt to detect as many cases of accessing uninitialized (or invalid, for example due to reading an external input) variables as possible, including accessing global uninitialized variables. This includes detecting cases of uninitialized components. In the case of out parameters of composite types, if there are components of an out parameter that are not initialized within the called subprogram, then detection of that will happen for references to the out parameter's uninitialized components on the calling side. Note that no messages will be issued within the called subprogram if the formal out parameter or any of its components are not assigned (as it's not an error in Ada to fail to assign to a composite out parameter's components).

Message	Description

validity check	The code may be reading an uninitialized or invalid value (e.g. corrupted data) (CWE 457). For example in the following code:
	`procedure Uninit is`
	`  A : Integer;`
	`  B : Integer;`
	`begin`
	`  A := B;  --  we are reading B which is uninitialized!`
	`end Uninit;`
	GNAT SAS will flag the read of B:
	`uninit.adb:5:9: high: validity check (Inspector): B is uninitialized here`
	When an OUT parameter is read without being initialized, it is flagged in the same way:
	`procedure Uninit_Out (B : out Integer) is`
	`  A : Integer;`
	`begin`
	`  A := B;  --  we are reading B which is uninitialized!`
	`end Uninit_Out;`
	`uninit.adb:11:9: high: validity check (Inspector): B is uninitialized here`
	Infer also flags the read before initialization of the field of a record passed as an OUT parameter. While this can be valid code since the OUT parameter may be passed by reference, it is prone to errors.
	`  type Rec is record`
	`  F1 : Integer;`
	`  end record;`
	`  procedure Uninit_Out_Rec (R : out Rec) is`
	`  A : Integer;`
	`  begin`
	`  A := R.F1;  --  R.F1 can be initialzed, but shouldn't be accessed`
	`  end Uninit_Out_Rec;`
	Infer will flag the read of R.F1 with a validity check message:
	`uninit.adb:20:12: high: validity check (Infer): `_.F1` is read without initialization`
	Note that for the same example, Inspector flags the problem with a message of a kind `suspicious input`.
	Finally, both Infer and Inspector flag with a validity check an OUT parameter of scalar type that isn't initialized at the end of a subprogram:
	`  procedure Out_Not_Set (O : out Integer) is`
	`  begin`
	`  null;`
	`  end Out_Not_Set;`
	GNAT SAS flags such subprogram with:
	`uninit.adb:22:4: high: validity check (Inspector): out parameter O is uninitialized here`

11.7.5 Warning Messages

In addition to run-time checks, Inspector will also attempt to detect many cases of logic errors, which often point to suspicious/possibly incorrect code. Unlike check-related messages, these messages do not always point to a real error if the message is correct, and are based on heuristics.

Message	Description

dead code	Also called "unreachable code." Indicates logical errors as the programmer assumed the unreachable code could be executed (CWE 561). For example in the following code:
	`procedure Dead_Code (X : out Integer) is`
	`   I : Integer := 10;`
	`begin`
	`   if I < 4 then`
	`      X := 0;`
	`   elsif I >= 10 then`
	`      X := 0;`
	`   else`
	`      X := 0;`
	`   end if;`
	`end Dead_Code;`
	GNAT SAS will flag:
	`dead_code.adb:5:9: medium warning: dead code (Inspector): dead code because I = 10`
	`dead_code.adb:9:9: medium warning: dead code (Inspector): dead code because I = 10`

test always false	Indicates redundant conditionals, which could flag logical errors where the test always evaluates to false (CWE 570). For example in the code above for `dead code`, GNAT SAS will also flag: `dead_code.adb:4:9: low warning: test always false (Inspector): test always false because I = 10`
test always true	Indicates redundant conditionals, which could flag logical errors where the test always evaluates to true (CWE 571). For example in the code above for `dead code`, GNAT SAS will also flag: `dead_code.adb:6:4: medium warning: test always true (Inspector): test always true because I = 10`

continues on next page

Table 9 – continued from previous page

Message	Description

test predetermined	Indicates redundant conditionals, which could flag logical errors (CWE 570, 571). This is similar to `test always true` and `test always false` and is only emitted when there is no real polarity associated with the test such as in a case statement:
	`procedure Predetermined is`
	`   I : Integer := 0;`
	`begin`
	`   case I is`
	`      when 0 =>`
	`         ...`
	`      when 1 =>`
	`         ...`
	`      when others =>`
	`         ...`
	`   end case;`
	`end Predetermined;`
	GNAT SAS will flag:
	`predetermined.adb:4:4: low warning: test predetermined (Inspector):`
	`test predetermined because I = 0`

condition predetermined	Indicates redundant condition inside a conditional, like the left or right operand of a boolean operator which is always true or false (CWE 570, 571). For example in the following code:
	`procedure Condition is`
	`   type L is (A, B, C);`
	`   procedure Or_Else (V : L) is`
	`   begin`
	`      if V /= A or else V /= B then`
	`         return;`
	`      else`
	`         raise Program_Error;`
	`      end if;`
	`   end Or_Else;`
	GNAT SAS will flag:
	`condition.adb:6:27: medium warning: condition predetermined (Inspector):`
	`condition predetermined because (V /= B) is always true`

continues on next page

Table 9 – continued from previous page

Message	Description

loop does not complete normally	Indicates loops that either run forever or fail to terminate normally (CWE 835). For example in the following code:
	`procedure Loops is`
	`   Buf : String := "The" & ASCII.NUL;`
	`   BP  : Natural;`
	`begin`
	`   Buf (4) := 'a';   -- Eliminates null terminator`
	`   BP := Buf'First;`
	
	`   while True loop`
	`      BP := BP + 1;`
	`      exit when Buf (BP - 1) = ASCII.NUL;  -- Condition never reached`
	`   end loop;`
	`end Loops;`
	GNAT SAS will flag:
	`loops.adb:8:10: medium warning: loop does not complete normally (Inspector)`

unused assignment	Indicates redundant assignment. This may be an indication of unintentional loss of result or unexpected flow of control (CWE 563). Note that Inspector recognizes special variable patterns as temporary variables that will be ignored by this check: *ignore*, *unused*, *discard*, *dummy*, *tmp*, *temp*. This can be tuned via the MessagePatterns.xml file if needed. An object marked as unreferenced via an Unreferenced pragma is similarly ignored (see the Implementation Defined Pragmas section of the Gnat Pro Reference Manual for information about this pragma). For example in the following code:
	`procedure Unused_Assignment is`
	`   C : Character := Get_Character;`
	`begin`
	`   null;  -- C is not used in this subprogram`
	`end Unused_Assignment;`
	GNAT SAS will flag:
	`unused_assignment.adb:2:4: medium warning: unused assignment (Inspector):`
	`unused assignment into C`

continues on next page

Table 9 – continued from previous page

Message	Description

unused assignment to global	Indicates that a subprogram call modifies a global variable, which is then overwritten following the call without any uses between the assignments. Note that the redundant assignment may occur inside another subprogram call invoked by the current subprogram (CWE 563). For example in the following code:
	`package body Unused_Global is`
	`   G : Integer;`
	`   procedure Proc0 is`
	`   begin`
	`      G := 123;`
	`   end Proc0;`
	`   procedure Proc1 is`
	`   begin`
	`      Proc0;`
	`   end Proc1;`
	`   procedure Proc2 is`
	`   begin`
	`      Proc1;`
	`      G := 456;  -- override effect of calling Proc1`
	`   end Proc2;`
	`end Unused_Global;`
	GNAT SAS will flag:
	`unused_global.adb:16:7: low warning: unused assignment to global (Inspector):`
	`unused assignment to global G in unused_global.proc1`

continues on next page

Table 9 – continued from previous page

Message	Description

unused out parameter	Indicates that an actual parameter of a call is ignored (either never used or overwritten) (CWE 563). For example in the following code:
	`procedure Unused_Out is`
	`   procedure Linear_Search`
	`     (Table : Int_Array;`
	`      Item : Integer;`
	`      Found: out Boolean;`
	`      Index: out Integer);`
	
	`   Table : Int_Array (1..10) := (others => 0);`
	`   Found : Boolean;`
	`   Index : Integer;`
	`begin`
	`   Search.Linear_Search (Table, 0, Found, Index); -- Found and Index are passed as out parameters, but not used afterwards`
	`end Unused_Out;`
	GNAT SAS will flag:
	`unused_out.adb:8:33: medium warning: unused out parameter (Inspector):`
	`unused out parameter Found`
	`unused_out.adb:8:33: medium warning: unused out parameter (Inspector):`
	`unused out parameter Index`

useless reassignment	Indicates when an assignment does not modify the value stored in the variable being assigned (CWE 563). For example in the following code:
	`procedure Self_Assign (A : in out Integer) is`
	`   B : Integer;`
	`begin`
	`   B := A;`
	`   A := B;`
	`end Self_Assign;`
	GNAT SAS will flag:
	`self_assign.adb:5:6: medium warning: useless reassignment (Inspector):`
	`useless reassignment of A`

continues on next page

Table 9 – continued from previous page

Message	Description

suspicious precondition	The precondition has a form that indicates there might be a problem with the algorithm. If the allowable value set of a given input expression is not contiguous, that is, there are certain values of the expression that might cause a run-time problem inside the subprogram in between values that are safe, then this might be an indication that certain cases are not being properly handled by the code. In other situations, this might simply reflect the inherent nature of the algorithm involved. For example in the following code extracted from the GNAT SAS tutorial:
	`package body Stack is`
	
	`   procedure Push (S : in out Stack_Type; V : Value) is`
	`   begin`
	`      if S.Last = S.Tab'Last then`
	`         raise Overflow;`
	`      end if;`
	
	`      S.Last := S.Last - 1;  --  Should be S.Last + 1`
	`      S.Tab (S.Last) := V;`
	`   end Push;`
	GNAT SAS will flag:
	`stack.adb:3:4: medium warning: suspicious precondition (Inspector):`
	`precondition for S.Last does not have a contiguous range of values`

suspicious input	Inputs mention a value reachable through an out-parameter of the subprogram before this parameter is assigned. Although the value may sometimes be initialized as the Ada standard allows, it generally uncovers a bug where the subprogram reads an uninitialized value or a value that the programmer did not mean to pass to the subprogram as an input value. For example in the following code:
	`procedure In_Out is`
	`   type T is record`
	`      I : Integer;`
	`   end record;`
	
	`   procedure Take_In_Out (R : in out T) is`
	`   begin`
	`      R.I := R.I + 1;`
	`   end Take_In_Out;`
	
	`   procedure Take_Out (R : out T; B : Boolean) is`
	`   begin`
	`      Take_In_Out (R);  -- R is 'out' but used as 'in out'`
	`   end Take_Out;`
	GNAT SAS will flag:
	`in_out.adb:13:7: medium warning: suspicious input (Inspector):`
	`input R.I depends on input value of out-parameter`

continues on next page

Table 9 – continued from previous page

Message	Description

unread parameter	A parameter of an elementary type of mode in out is assigned on all paths through the subprogram before any reads, and so could be declared with mode out. For example in the following code:
	`procedure Unread (X : in out Integer) is`
	`begin`
	`   X := 0;  -- X is assigned but never read`
	`end Unread;`
	GNAT SAS will flag:
	`unread.adb:1:1: medium warning: unread parameter (Inspector):`
	`parameter X could have mode out`

unassigned parameter	Inspector emits a warning when a parameter of a scalar type of mode in out is never assigned, and so could be declared with mode in. For example in the following code:
	`procedure Unassigned (X : in out Integer; Y : out Integer) is`
	`begin`
	`   Y := X;  -- X is read but never assigned`
	`end Unassigned;`
	GNAT SAS will flag:
	`unassigned.adb:1:1: medium warning: unassigned parameter (Inspector):`
	`parameter X could have mode in`
	In addition, Infer emits the same warning when any component of a parameter of a composite type of mode in out or out is never assigned. For example in the following code:
	`type Rec is record`
	`   F1 : Integer;`
	`   F2 : Integer;`
	`end record;`
	
	`procedure Unassigned_Composite (R1, R2 : out Rec) is`
	`begin`
	`   R2.F1 := 10;`
	`end Unassigned_Composite;`
	GNAT SAS will flag:
	`unassigned_infer.adb:1:4: medium warning: unassigned parameter (Infer): out`
	`parameter R1 not written in a subprogram`
	The warning is not emitted for primitives of tagged types and for null records.

continues on next page

Table 9 – continued from previous page

Message	Description

suspicious constant operation	An operation computes a constant value from non-constant operands. This is characteristic of a typographical mistake, where a variable is used instead of another one, or a missing part in the operation, like the lack of conversion to a floating-point or fixed-point type before division. For example in the following code:
	`procedure Constant_Op is`
	`   type T is new Natural range 0 .. 14;`
	`   function Incorrect (X : T) return T is`
	`   begin`
	`      return X / (T'Last + 1);`
	`   end;`
	`begin`
	`   null;`
	`end Constant_Op;`
	GNAT SAS will flag:
	`constant_op.adb:6:16: medium warning: suspicious constant operation (Inspector):`
	`operation X/15 always evaluates to 0`

subp never returns	The subprogram will never return, presumably because of an infinite loop. There will typically be an additional message in the subprogram body (e.g. test always false) explaining why the subprogram never returns. For example in the following code:
	`procedure Infinite_Loop is`
	`   X : Integer := 33;`
	`begin`
	`   loop`
	`      X := X + 1;`
	`   end loop;`
	`end Infinite_Loop;`
	GNAT SAS will flag:
	`infinite_loop.adb:1:1: medium warning: subp never returns (Inspector):`
	`infinite_loop never returns`
	`infinite_loop.adb:5:9: medium warning: loop does not complete normally (Inspector)`

continues on next page

Table 9 – continued from previous page

Message	Description

subp always fails	Indicates that a run-time problem is likely to occur on every execution of the subprogram. There will typically be an additional message in the subprogram body explaining why the subprogram always fails. For example in the following code:
	`procedure P is`
	`  X : Integer := (raise Program_Error);`
	`begin`
	`  null;`
	`end P;`
	GNAT SAS will flag:
	`p.adb:1:1: high warning: subp always fails (Inspector): p fails for all possible inputs`
	`p.adb:2:20: low: raise exception (Inspector): unconditional raise`

11.7.6 Information Messages

These are extra information messages generated by Inspector during analysis to complement existing warnings or check messages.

Message	Description
call too complex	Indicates that Inspector skipped analyzing the subprogram call to avoid exhausting resources needed for analyzing the remainder of the system. Inspector will report any presumptions it makes about the results/effects of the otherwise unanalyzed call. See also section *Call Too Complex* for more information on the possible causes of this message.
subp not available	Indicates that Inspector cannot analyze the call because the called subprogram is not available. There are two possible reasons for this: the body of the subprogram is not part of the project sources, or the called subprogram subprogram is analyzed in a different partition. Inspector will report any presumptions it makes about the results/effects of the otherwise unanalyzed call.
subp not analyzed	Indicates that Inspector encountered a problem while analyzing this subprogram and skipped its analysis as well as its nested subprograms. These subprograms need to be reviewed manually instead.
module not analyzed	Indicates that Inspector could not analyze any of the subprograms in this module. These subprograms need to be reviewed manually instead.
module analyzed	Indicates that Inspector completed analysis of this module
dead code continues	Indicates a block that is dead because its predecessor is dead.

11.7.7 Race Condition Messages

When GNAT SAS is run in *deep analysis mode*, Inspector's analysis produces three sorts of race condition messages. Note that when GNAT SAS indicates that no lock is held, it actually means that no lock visible to multiple tasks is held. There may in fact be a local lock held.

Race condition messages are meant to be exhaustive, assuming knowledge about tasks and protected objects in the application are visible to GNAT SAS.

Message	Description
unprotected access	A reentrant task (e.g. task type) reads or writes a potentially shared object without holding a lock. The message is associated with places where the object is accessed in the absence of any lock, or with non-overlapping lock configuration (CWE 362, 366, 667, 820).
unprotected shared access	A task accesses a potentially shared object without holding a lock and this object is also referenced by some other task. The message is associated with places where the object is referenced in the absence of any lock, or with non-overlapping lock configuration (CWE 362, 366, 667, 820).
mismatched protected access	A task references a potentially shared object while holding a lock, and this object is also referenced by another task without holding the same lock. Messages are associated with the second task's references (CWE 362, 366, 667, 821).

For example given the following code:

```
package Race is

   procedure Increment;
   pragma Annotate (GNATSAS, Multiple_Thread_Entry_Point, "Race.Increment");
   --  This is a task type: there will be multiple threads calling this subprogram

   procedure Decrement;
   pragma Annotate (GNATSAS, Multiple_Thread_Entry_Point, "Race.Decrement");
   --  Ditto

end Race;
```

```
package body Race is

   Counter : Natural := 0;

   procedure Acquire;
   pragma Import (C, Acquire);

   procedure Release;
   pragma Import (C, Release);

   pragma Annotate (GNATSAS, Mutex, "Race.Acquire", "Race.Release");

   procedure Increment is
   begin
      Acquire;

      if Counter = Natural'Last then
         Counter := Natural'First;
      else
         Counter := Counter + 1;
      end if;

      Release;
   end Increment;

```

(continues on next page)

(continued from previous page)

```
26    procedure Decrement is
27    begin
28       if Counter = Natural'First then  --  reading Counter without any lock
29          Counter := Natural'Last;      --  writing without any lock
30       else
31          Counter := Counter - 1;       --  reading and writing without any lock
32       end if;
33    end Decrement;
34
35 end Race;
```

GNAT SAS will generate the following messages:

```
race.adb:28:10: medium warning: mismatched protected access (Inspector): mismatched␣
↪protected access of shared object Counter via race.increment
race.adb:28:10: medium warning: unprotected access (Inspector): unprotected access of␣
↪Counter via race.decrement
race.adb:29:18: medium warning: mismatched protected access (Inspector): mismatched␣
↪protected access of shared object Counter via race.increment
race.adb:29:18: medium warning: unprotected access (Inspector): unprotected access of␣
↪Counter via race.decrement
race.adb:31:18: medium warning: mismatched protected access (Inspector): mismatched␣
↪protected access of shared object Counter via race.increment
race.adb:31:21: medium warning: mismatched protected access (Inspector): mismatched␣
↪protected access of shared object Counter via race.increment
race.adb:31:18: medium warning: unprotected access (Inspector): unprotected access of ␣
↪Counter via race.decrement
race.adb:31:21: medium warning: unprotected access (Inspector):  unprotected access of␣
↪Counter via race.decrement
```

If GNAT SAS indicates a race condition at a particular line in your program, it is best to refer to the **Race Conditions** report by clicking on the *Race Conditions* button in the title bar. The **Race Conditions** report is organized by the affected object, and gives an overall perspective on how the various task entry points are involved in creating the potential race condition on the given object. Note that Inspector does not detect *deadlocks* between tasks (sometimes called *deadly embraces*). Rather it identifies data objects that are potentially referenced without adequate synchronization.

The following section gives more explanations on how to identify possible race conditions.

Identify Possible Race Conditions

Note: This section only applies when using GNAT SAS in *deep analysis mode*.

When GNAT SAS is run in *deep analysis mode*, Inspector detects common forms of *race conditions*. More precisely, Inspector focuses on the *data race* subset of race conditions: Race conditions that may exist if there are two or more concurrent tasks that attempt to access the same object and at least one of them is doing an update. For example, if a Reader task makes a copy of a List Object at the same time a Writer task is modifying the List Object, the copy of the List Object may be corrupt. Languages such as Ada use *synchronization* or **locking** as a means to guard against race conditions. Inspector identifies race conditions where synchronization is not used or is used incorrectly. Since Inspector focuses on data races, concurrent references to atomic objects are presumed to be free of race conditions. (Note that the current release of GNAT SAS does not identify potential deadlocks, also known as deadly embraces, where two tasks are stuck waiting on locks held by the other.)

A lock is held during any protected subprogram or protected entry call. Any variable that can be accessed by more than one referencing task simultaneously must be locked at every reference to guard against race conditions. Furthermore, the referencing lock should match the lock used by other tasks to prevent updates during access. If locking is absent, or if one reference uses a different lock than some other reference, GNAT SAS identifies a possible race condition. Note that an identified race condition is not *guaranteed* to create problems on every execution, but it *might* cause a problem, depending on specific run time circumstances.

Note that if a lock is associated with an object that is newly created each time a subprogram is called, it does not actually provide any synchronization between distinct calls to that subprogram. A lock is only effective if it is associated with an object visible to multiple tasks. Inspector ignores locks on objects that are not visible to multiple tasks since they have no synchronizing effect. This means GNAT SAS may indicate there are no locks held at the point of a reference to a potentially shared object even though there are in fact some *local* locks held. A future release will identify any potential problems associated with local locks more explicitly.

Inspector must understand the tasking structure of the program being analyzed to detect race conditions. There are two types of entry points that are important to race condition analysis: *Reentrant* entry points and *Daemon* entry points. A *Reentrant* entry point represents code that can be invoked by multiple tasks concurrently (e.g. task types). A Daemon entry point (also called a *singleton*) is presumed to be invoked only by a single task at a time (e.g. task bodies).

Standard Ada tasking constructs (such as tasks and protected objects) are identified automatically by GNAT SAS as needed. In addition, you can manually identify reentrant entry points (aka task types) with the *-reentrant "package.subp"* switch on the GNAT SAS command line. Use the *-daemon "package.subp"* to identify daemon entry points (aka tasks). See *GNAT SAS CLI Reference* for the syntax for these switches.

You may also identify tasks and task types for GNAT SAS by using the GNAT-defined *pragma Annotate*. This pragma has no effect on the code generated for the execution of a program: it only affects Inspector's race condition analysis. In this example:

```
package Pkg is
   procedure Single;
   pragma Annotate (GNATSAS, Single_Thread_Entry_Point, "Pkg.Single");
   procedure Multiple;
   pragma Annotate (GNATSAS, Multiple_Thread_Entry_Point, "Pkg.Multiple");
end Pkg;
```

Inspector will assume that Pkg.Single is a single thread entry point (or "daemon", or "task") procedure and that Pkg.Multiple is a multiple thread entry point (or "reentrant", or "task type") procedure. An Annotate pragma used in this way must have exactly three operands: the identifier GNATSAS, one of the identifiers Single_Thread_Entry_Point or Multiple_Thread_Entry_Point, and a string literal whose value is the fully qualified name of the procedure being identified.

To allow one pragma to apply to multiple subprograms, the final string literal may also have the same "wildcard" syntax supported by the *-reentrant* and *-daemon* command line switches. In this example:

```
package Foo_Procs is
   procedure Foo_123;
   procedure Foo_456;
   pragma Annotate (GNATSAS, Single_Thread_Entry_Point, "Foo_Procs.Foo*");
   procedure Foo_789;
end Foo_Procs;
```

the pragma would apply to the two procedures which precede it. If the same pragma were used as a configuration pragma in an associated configuration pragma file (described below), the pragma would apply to all three procedures.

Except when used as a configuration pragma (described below), the pragma must occur in the same immediately enclosing declarative_part or package_specification as the procedure declaration, not before the procedure's declaration, and not after its completion. For a general description of *pragma Annotate*, see the GNAT Reference Manual.

Annotate pragmas may be used as configuration pragmas. In the preceding example, the same pragmas could have been present in an associated configuration pragma file (e.g., a `gnatsas.adc` file).

You might need to specify your own entry points explicitly to include all relevant external entry points, including call backs from external subsystems and interrupt entry points.

For partitions that include one or more task entry points, an indication of zero detected race conditions ensures there is no path within that partition from one of these entry points to any of the three kinds of unsynchronized access to shared data objects identified by Inspector.

Inspector performs race condition analysis only with the deep analysis mode. This helps to ensure that potential race conditions are identified early. Locating race conditions with run-time testing can be difficult since they normally cause problems only intermittently or under heavy load. Note that you may add the `--no-race-conditions` switch to your project file's "inspector" switches to suppress race condition analysis.

Some programs make use of user-defined mutual exclusion mechanisms instead of using language-defined protected actions. If a pair of procedures (often with names like Lock and Unlock, Acquire and Release, or P and V) are used to implement mutual exclusion, *pragma Annotate* may be used to communicate this information to GNAT SAS. This pragma has no effect on the code generated for the execution of a program; the pragma only affects Inspector's race condition analysis. Given this example:

```
package Locking is
    procedure Lock;
    procedure Unlock;
    pragma Annotate (GNATSAS, Mutex, "Locking.Lock", "Locking.Unlock");
end Locking;
```

Inspector will assume that a call to Lock acquires a lock and a call to Unlock releases it. If the following procedure is then called, for example, from the body of a task type:

```
procedure Increment_Global_Counters is
begin
    Counter_1 := Counter_1 + 1;
    Locking.Lock;
    Counter_2 := Counter_2 + 1;
    Locking.Unlock;
end Increment_Global_Counters;
```

Inspector's race condition analysis will flag only the use of Counter_1 as being potentially unsafe.

Inspector trusts the accuracy of the pragma; no attempt is made to verify that the two procedures really do somehow implement mutual exclusion. An Annotate pragma used in this way must have exactly four operands: the identifier GNATSAS, the identifier Mutex, a string literal whose value is the fully qualified name of (only) the lock-acquiring procedure, and a corresponding string literal for the lock-releasing procedure. Except when used as a configuration pragma (described below), the pragma must occur in the same immediately enclosing declarative_part or package_specification as the two procedure declarations, not before either procedure's declaration, and not after either procedure's completion.

The **Race Condition** report in *GNATstudio* provides details about the shared objects in your program that might be subject to unsafe access. In GNAT Studio, there is one global report accessible from the main GNAT SAS report. Clicking on each shared object displays information about entry points associated with possible race conditions, including the information whether it is a read or a write access. As mentioned above, GNAT SAS only concerns itself with locks that are visible to multiple tasks, so an empty column *Locks* means no *task-visible* locks are held. There may be locks associated with locally created objects, but these provide no effective synchronization between distinct tasks.

11.7.8 Understanding the Differences between Preconditions and Run-time Errors

One possibly surprising behavior of Inspector is that potential run-time errors are not always flagged on the line that actually raises the exception. Using the following example:

```
Arr : array (Integer range 1 .. 10) of Integer := (others => 0);

function Get1 (X : Integer) return Integer is
begin
   return Arr (X + 10);
end Get1;

...

R := Get1 (-10);
```

There is a potential run-time error on the access to the array:

```
return Arr (X + 10);
```

However, Inspector does not flag the potential problem here. Instead, it "protects" the calls to Get1 with a deduced precondition:

```
X in -9..0
```

This precondition has to be respected by callers. This is clearly not the case in the call:

```
R := Get1 (-10);
```

Which is why Inspector flags a precondition violation here. This precondition violation is due to a potential run-time error, which is indicated in angle brackets as part of the message generated by GNAT SAS:

```
bug.adb:13:12: high: precondition <array index check> (Inspector):  precondition failure␣
↪on bug.get1; requires X in -9..0
```

Let's look at a slightly (admittedly artificially) more complex example than the above:

```
Arr : array (Integer range 1 .. 10) of Integer := (others => 0);

function Get2 (X : Integer) return Integer is
   Index : Integer := 10;
begin
   for J in 1 .. X loop
      Index := Index + 1;
   end loop;

   return Arr (Index);
end Get1;

...

R := Get1 (-10);
```

Although the nature of the checks is very similar (array checks), the additional complexity of the code added by the loop leads GNAT SAS to report the potential errors on the array access, as opposed to the precondition. Therefore, in

this case the message is reported directly on the line that may raise the exception, and not on the caller.

In summary, when looking for potential run-time errors it's important to keep in mind that Inspector may flag problems either on the actual line where the error may occur or in callers that may lead to the error. This is only the case for potential run-time errors: warnings on the other hand (e.g. dead code, unused assignments) are always flagged directly on the incriminated line.

Note that you can change this behavior (at the expense of additional false alarms) by adding the `--no-preconditions` switch to your project file's list of inspector switches. This will disable the generation and propagation of preconditions and will generate messages instead at the point of the potential failure.

11.8 Inspector Annotations

In addition to messages, GNAT SAS' Inspector analysis also generates annotations on your source code. Annotations do not need to be reviewed like messages, and do not in general represent errors in the code. Instead, they express properties and assumptions of the code, and can be used to help reviewing the code manually and spot inconsistencies, and also to help understand messages generated by Inspector.

11.8.1 Annotations

The Inspector engine generates the following kinds of annotations on each Ada subprogram:

pre	*Preconditions* specify requirements that the subprogram imposes on its inputs. For example, a subprogram might require a certain parameter to be non-null for proper operation of the subprogram. These preconditions are checked at every call site. A message is given for any precondition that a caller might violate. Precondition messages include in parenthesis a list of the checks involved in the requirements.
presumption	*Presumptions* display what Inspector presumes about the results of an external subprogram whose code is unavailable, or are in a separate partition. There are separate presumptions for each call site, with a string in the form @<*line-number-of-the-call*> appended to the name of the subprogram. Presumptions are not generally used to determine preconditions of the calling routine, but they might influence postconditions of the calling routine. See below for further discussion of presumptions.
post	*Postconditions* characterize the behavior of the subprogram in terms of its outputs and the presumptions made.
unanalyzed	*Unanalyzed Calls* display the external calls to subprograms that the Inspector has not analyzed, and so participate in the determination of presumptions. Note that these annotations include all directly unanalyzed calls as well as the unanalyzed calls in the call graph subtree that have an influence on the current subprograms.
global inputs	*Global Inputs* comprise a list of all global variables referenced by each subprogram. Note that this only includes enclosing objects and not e.g. specific components. In the case of pointers, only the pointer is listed. Dereference to pointers may be implied by the pointer listed.
global outputs	*Global Outputs* comprise a list of all global variables (objects and components) modified by each subprogram.
new obj	*New Objects* comprise a list of heap-allocated objects, created by a subprogram, that are not reclaimed during the execution of the subprogram itself; these are new objects that are accessible after return from the subprogram.

Note that the `global inputs` and `global outputs` annotations are by default limited in length and may be truncated as a result. In order to generate these annotations fully, you can add the `--dbg-off inout_limit` switch to the list

of Inspector switches in your project file..

See *Use Annotations generated by Inspector for Code Reviews* for more details on how annotations can be used to aid in source code review.

11.8.2 Annotation Syntax

In the context of pre/post conditions and messages, a syntax close to Ada is used to express these conditions, including some Inspector specific notations:

X'Initialized	Means that *X* is expected to be initialized, but no further requirement or knowledge is imposed on its value. Inspector issues a precondition *X'Initialized* whenever the value of *X* is read in the subprogram and it is not already mentioned in another precondition. As part of a postcondition, this means that X is known to be initialized when the subprogram exits, with no more precise information.
X'Accessibility_Level	Every entity has an associated number called its accessibility level. Inspector tracks this number and makes reference to it using the attribute *X'Accessibility_Level*. This number relates to the accessibility rules of Ada. Every access type also has an underlying accessibility level. When needed, there is a check between the accessibility level of the object and the accessibility level of the access type. When the object has a greater accessibility level than the access type, a PROGRAM_ERROR is raised at run time. Inspector tracks these levels and reports on possible exceptions using this attribute notation.
(soft)	Means that a precondition is associated with code that might not be executed on certain paths, so violating the precondition might be safe under those circumstances. However, violating the precondition on every call would not be wise, as there are values for other inputs that could cause a violation of the precondition to lead to a run-time failure.
possibly_updated(X)	Used to convey in a postcondition that *X* is a possible output, but there is no specific set of values that it is known to have after the subprogram completes.
possibly_init'ed(X)	Used to convey in a postcondition that *X* is initialized on some but not all paths through the subprogram.
base...fld	Refers to the *fld component* of any object accessible by following one or more levels of indirection starting at *base*, not including *base.fld* itself. This notation is used to handle iterative or recursive algorithms that walk recursive data structures. For example, in an algorithm that walks a linked list, a possible precondition might be *(X...next)'Initialized* meaning that the *next* component of any object reachable from *X* must be initialized (not including *X.next* itself: there will be a separate *X.next'Initialized* precondition if appropriate.
X in (a..b | c | d)	This is the Ada 2012 set notation, and means that *X* is either in the range *a..b*, or equal to c, or equal to d.
X in (1.1 .. 5.2]	Floating point ranges are expressed using this mathematical notation, where *(* and *)* mean that the value is excluded (not part of the range), and *[* and *]* mean that the value is included in the range. In other words, this is equivalent to *X > 1.1 and X <= 5.2*. In some cases the names IEEE_32_Float and IEEE_64_Float are used as the prefix of an attribute. For example, the name "IEEE_32_Float'Last" is used to represent the largest (finite) value representable in the IEEE 754 single precision (i.e., 32-bit) representation. The names "IEEE_32_Float'First", "IEEE_64_Float'Last" and "IEEE_64_Float'First" are used analogously. A name such as "IEEE_32_Float'Succ(0.0)" refers to the successor (or predecessor, if Succ is replaced with Pred) of the argument value in the given IEEE representation; for example, the name "IEEE_64_Float'Pred(1.0) refers to the predecessor of 1.0 in the 64-bit IEEE representation. The Succ/Pred notation may be used in cases where this choice shortens the resulting message text.

continues on next page

Table 12 – continued from previous page

Global_Var@Pkg'Ela	Refers to the value of a global variable at the time the specified package is elaborated. This might occur if there is a global constant in Pkg that captures the value of a global variable declared in some other packge, during elaboration of Pkg.
X'Old	Refers to the value *X* had on entry to the subprogram, as defined in Ada 2012.
Func'Result	Refers to the return value of the given function *Func*, as defined in Ada 2012.
new My_Type(in Subp)#N	Refers to the *My_Type* object created by the Nth allocator within *Subp*. The notation *new My_Type(in Subp)#N.<num objects>* refers to the number of times this allocator might be invoked at run-time for each call of the subprogram. If an asterisk follows the index (e.g. *new My_Type(Subp)#3**)) then the allocator arises from a nested call on *Subp*, which may represent a recursive call, or simply a nested call on a same-named subprogram.
Pkg'Body	Refers to the package body of unit *Pkg* (as opposed to the spec or to the whole unit). This is used in particular to refer to variables declared in package bodies, e.g. *Pkg1'Body.Global_Var1* points to the variable *Global_Var1* declared in the body of package *Pkg1*.
//	This operator is used to represent the division operator used for converting (with rounding) a fixed point value to an integer value, with the semantics described in Ada RM 4.6(33). Here is an example precondition: *requires (if Arg >= 1_415_577_600 then 0 else (Arg//16_384)/3_600) in 0..23*. Note that, in the precondition above, *1_415_577_600* is not a value of the type of *Arg*, but of the underlying integer type GNAT uses to represent its (fixed-point) value. Likewise, *Arg//16_384* is actually an integer division, but with special rounding properties. Reading such annotations may be confusing at first, and special care should be exercised.

11.8.3 Use Annotations generated by Inspector for Code Reviews

Note: This section only applies in *deep mode*.

Whether a formal team process or an ad-hoc, one-person activity, manually reviewing source code is a good way to identify problems early in the development cycle. Unfortunately, it can also be quite time consuming, especially when the reviewer is not the author of the code. Inspector reduces the effort required for understanding source code by characterizing the input requirements and the net effect of each component of the code base.

Specifically, Inspector determines **preconditions** and **postconditions** for every Ada subprogram it analyzes. It also makes **presumptions** about external subprograms it calls whose source code is not available, or which are so complex that they threaten to exhaust the available machine resources. GNAT SAS displays **preconditions**, **presumptions**, and **postconditions** within the GNAT Studio source editor as an Ada comment block immediately preceding the first executable line of the subprogram.

The **preconditions** displayed by GNAT SAS are implicit requirements that are imposed on the inputs to a subprogram, as determined by analyzing the algorithms used within the subprogram. Violating **preconditions** might cause the subprogram to fail or to give meaningless results. During code review, the reviewer can verify that the **preconditions** determined by Inspector for the code as written are appropriate and meet the underlying requirements for the subprogram.

Early in a development cycle, system documentation might be missing or incomplete. Since Inspector generates **preconditions** for each module without requiring the entire enclosing system to be available, it can be used before system integration to understand subprograms as they are developed. In a mature, maintained codebase the system documentation might no longer agree with current code's behavior. In either case, Inspector's generated **preconditions** can be used to verify both the written and unwritten assumptions made by the codewriters.

Presumptions represent assumptions made by Inspector about the results of a call on a subprogram whose code is unavailable for analysis. A separate presumption is made for each call site to the unanalyzed subprogram, with a string

in the form @<*line-number-of-the-call*> appended to the name of the subprogram. **Presumptions** do not generally affect the **preconditions** of the calling routine, but they might influence **postconditions** of the calling routine.

Postconditions are characteristics of the output which a subprogram could produce, presuming its **preconditions** are satisfied and the **presumptions** made about unanalyzed calls are appropriate. Even in the absence of other documentation, postconditions can help a reviewer understand the purpose and effect of code. Likewise, **postconditions** can be helpful to software developers who use a subprogram. Comparing **postconditions** to either **preconditions** or the context of calling routines can provide valuable insight into the workings of the code which might not be obvious from solely a manual review.

CHAPTER

TWELVE

GNAT SAS CLI REFERENCE

12.1 gnatsas

12.1.1 NAME

gnatsas - GNAT Static Analysis Suite

12.1.2 SYNOPSIS

gnatsas [*COMMAND*] ...

12.1.3 DESCRIPTION

Analyze a project with `gnatsas analyze -P project.gpr` and export the results with `gnatsas report -P project.gpr`. All `gnatsas` commands (listed below in the **COMMANDS** section) accept a `--help` switch to print more usage information.

Find a full reference, guide and tutorial in the GNAT SAS tab on: https://support.adacore.com/csm?id=documentation

12.1.4 COMMANDS

analyze [*OPTION*]... [*ARG*]...
Analyze a project.

baseline [*OPTION*]...
Modify baselines.

help [--man-format=*FMT*] [*OPTION*]... [*TOPIC*]
Display GNATSAS help.

metrics [*OPTION*]...
Launch GNATmetric with the specified command-line arguments. Refer to the GNATmetric documentation for usage.

report [*COMMAND*] ...
Display results or generate a report in one of the supported formats once an analysis has been performed. If no format is specified, text format is applied.

review [--from-csv=*VAL*] [--import=*VAL*] [*OPTION*]...
Review file interaction.

12.1.5 COMMON OPTIONS

These options are common to all commands.

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.1.6 GPR Switches

-aP
Add directory to the project search path

--autoconf
Specify/create the main config project file name

--config
Specify the configuration project file name

--db
Parse dir as an additional knowledge base

--db-
Do not load the standard knowledge base

--implicit-with
Add the given projects as a dependency on all loaded projects

-eL
Follows symlinks for project files

--no-project
Do not use a project file

-P
The project file

--print-gpr-registry
Print the GPR registry

--root-dir
Root directory of obj/lib/exec to relocate

--relocate-build-tree
Root obj/lib/exec dirs are current-directory or dir

--RTS
Use --RTS=<runtime> to specify the Ada runtime

--RTS
Use --RTS:<lang>=<runtime> to specify the runtime for language <lang>

--src-subdirs
prepend <obj>/dir to the list of source dirs for each project

--subdirs
Use dir as suffix to obj/lib/exec directories

--target
Specify a target for cross platforms

-X
Specify an external reference for Project Files

12.1.7 MORE HELP

Use **gnatsas** *COMMAND* --help for help on a single command.

12.1.8 EXIT STATUS

gnatsas exits with the following status:

0
on success.

1
on internal tool error (e.g., analysis engine failure).

2
on user error (e.g., wrong option, options combination, incorrect option argument).

3
on setup error (e.g., missing read/write right for a file).

4
on parsing error.

100
on interruption (e.g., out of memory, user interruption).

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.1.9 BUGS

Open bug reports by visiting **https://support.adacore.com/csm**.

12.2 gnatsas-analyze

12.2.1 NAME

gnatsas-analyze - Analyze a project.

12.2.2 SYNOPSIS

gnatsas analyze [*OPTION*]... [*ARG*]...

12.2.3 DESCRIPTION

Analyze a project.

The default behavior (when none of -U, --no-subprojects, --file, --files-from is specified) is to analyze the closure of the main units specified in the GPRFILE project file for the full project tree, except externally built ones.

12.2.4 OPTIONS

--bits=*VAL*
: How many bits the analysis should assume.

-f, --force
: Force analysis of all files (disable incrementality).

--file=*VAL*
: Analyze a single file, ignoring other files from the project.

--files-from=*VAL*
: Specify the path of a file containing a list of files to analyze.

--gnat
: (Default) enable launching GNAT front end and collecting its warnings.

--gnatcheck
: (Default) enable GNATcheck analysis.

--incrementality-method=*VAL*
: Whether to use checksums or timestamps to determine files changes when performing SCIL generation or computing GNAT warnings.

--infer
: (Default) enable Infer analysis.

--inspector
: (Default) enable Inspector analysis.

-j *VAL*, --jobs=*VAL*
: Set maximum number of jobs. Set to 0 to let each analysis engine choose an optimal amount of cores.

--keep-going
If set, do not stop on failure of an analysis engine, but only if all analysis engines fail.

--mode=*VAL*
Analysis mode. Supported modes are 'fast' (default if not specified), 'deep'.

--no-gnat
Disable launching GNAT front end and collecting its warnings.

--no-gnatcheck
Disable GNATcheck analysis.

--no-infer
Disable Infer analysis.

--no-inspector
Disable Inspector analysis.

--no-subprojects
Analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.

--no-taint
Disable taint analyses.

--no-unused-annotate-warning
Do not emit warnings about unused "Annotate" pragmas.

--progress-bar=*VAL*
Print a progress bar.

--run-name=*NAME*
Set a name for this analysis run. When set, the run name is displayed in the GNAT SAS report window of GNAT Studio. Also impacts the name of the file containing the analysis results, which becomes *NAME*.sam instead of <project>.<timeline>.sam.

--skip-spark-mode
If set, skip messages that are in 'SPARK mode On' code.

--taint
(Default) enable taint analyses.

--timeline=*VAL*
Timeline to use for the analysis.

-U [*FILE*] (default=)
When *FILE* is specified, analyze files contained in the closure of the *FILE* unit. If set and no *FILE* is specified, analyze all files in the full project tree, except externally built ones.

12.2.5 COMMON OPTIONS

These options are common to all commands.

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.2.6 GPR Switches

-aP
Add directory to the project search path

--autoconf
Specify/create the main config project file name

--config
Specify the configuration project file name

--db
Parse dir as an additional knowledge base

--db-
Do not load the standard knowledge base

--implicit-with
Add the given projects as a dependency on all loaded projects

-eL
Follows symlinks for project files

--no-project
Do not use a project file

-P
The project file

--print-gpr-registry
Print the GPR registry

--root-dir
Root directory of obj/lib/exec to relocate

--relocate-build-tree
Root obj/lib/exec dirs are current-directory or dir

--RTS
Use --RTS=<runtime> to specify the Ada runtime

--RTS
Use --RTS:<lang>=<runtime> to specify the runtime for language <lang>

--src-subdirs
prepend <obj>/dir to the list of source dirs for each project

--subdirs
Use dir as suffix to obj/lib/exec directories

--target
Specify a target for cross platforms

-X
Specify an external reference for Project Files

12.2.7 MORE HELP

Use **gnatsas** *COMMAND* --help for help on a single command.

12.2.8 EXIT STATUS

analyze exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.2.9 BUGS

Open bug reports by visiting **https://support.adacore.com/csm**.

12.2.10 SEE ALSO

gnatsas(1)

12.3 gnatsas-baseline

12.3.1 NAME

gnatsas-baseline - Modify baselines.

12.3.2 SYNOPSIS

gnatsas baseline [*OPTION*]...

12.3.3 DESCRIPTION

Modify baselines.

12.3.4 OPTIONS

--bump-baseline
Bump baseline.

--set-baseline=*VAL*
Specify that sam_file is the new default baseline.

--set-current=*VAL*
Set sam_file as the results of the last run. Let the user import results as if they were produced locally.

--timeline=*VAL*
Timeline to use for the baseline.

(Deprecated) --file=*VAL*
[use --timeline instead] Analyze a single file, ignoring other files from the project.

(Deprecated) --files-from=*VAL*
[use --timeline instead] Specify the path of a file containing a list of files to analyze.

(Deprecated) --mode=*VAL*
[use --timeline instead] Analysis mode. Supported modes are 'fast' (default if not specified), 'deep'.

(Deprecated) --no-subprojects
[use --timeline instead] Analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.

(Deprecated) -U [*FILE*] (default=)
[use --timeline instead] When *FILE* is specified, analyze files contained in the closure of the *FILE* unit. If set and no *FILE* is specified, analyze all files in the full project tree, except externally built ones.

12.3.5 COMMON OPTIONS

These options are common to all commands.

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.3.6 GPR Switches

-aP
Add directory to the project search path

--autoconf
Specify/create the main config project file name

--config
Specify the configuration project file name

--db
Parse dir as an additional knowledge base

--db-
Do not load the standard knowledge base

--implicit-with
Add the given projects as a dependency on all loaded projects

-eL
Follows symlinks for project files

--no-project
Do not use a project file

-P
The project file

--print-gpr-registry
Print the GPR registry

--root-dir
Root directory of obj/lib/exec to relocate

--relocate-build-tree
Root obj/lib/exec dirs are current-directory or dir

--RTS
Use --RTS=<runtime> to specify the Ada runtime

--RTS
Use --RTS:<lang>=<runtime> to specify the runtime for language <lang>

--src-subdirs
prepend <obj>/dir to the list of source dirs for each project

--subdirs
Use dir as suffix to obj/lib/exec directories

--target
Specify a target for cross platforms

-X
Specify an external reference for Project Files

12.3.7 MORE HELP

Use **gnatsas** *COMMAND* --help for help on a single command.

12.3.8 EXIT STATUS

baseline exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.3.9 BUGS

Open bug reports by visiting **https://support.adacore.com/csm**.

12.3.10 SEE ALSO

gnatsas(1)

12.4 gnatsas-help

12.4.1 NAME

gnatsas-help - Display GNATSAS help.

12.4.2 SYNOPSIS

gnatsas help [**--man-format**=*FMT*] [*OPTION*]... [*TOPIC*]

12.4.3 DESCRIPTION

Prints help about gnatsas commands and other subjects...

12.4.4 ARGUMENTS

TOPIC
The topic to get help on. **topics** lists the topics.

12.4.5 OPTIONS

--man-format=*FMT* (absent=pager)
Show output in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

12.4.6 COMMON OPTIONS

These options are common to all commands.

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.4.7 GPR Switches

-aP
Add directory to the project search path

--autoconf
Specify/create the main config project file name

--config
Specify the configuration project file name

--db
Parse dir as an additional knowledge base

--db-
Do not load the standard knowledge base

--implicit-with
Add the given projects as a dependency on all loaded projects

-eL
Follows symlinks for project files

--no-project
Do not use a project file

-P
The project file

--print-gpr-registry
Print the GPR registry

--root-dir
Root directory of obj/lib/exec to relocate

--relocate-build-tree
Root obj/lib/exec dirs are current-directory or dir

--RTS
Use --RTS=<runtime> to specify the Ada runtime

--RTS
Use --RTS:<lang>=<runtime> to specify the runtime for language <lang>

--src-subdirs
prepend <obj>/dir to the list of source dirs for each project

--subdirs
Use dir as suffix to obj/lib/exec directories

--target
Specify a target for cross platforms

-X
Specify an external reference for Project Files

12.4.8 MORE HELP

Use **gnatsas** *COMMAND* --help for help on a single command.

12.4.9 EXIT STATUS

help exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.4.10 BUGS

Open bug reports by visiting **https://support.adacore.com/csm**.

12.4.11 SEE ALSO

gnatsas(1)

12.5 gnatsas-report

12.5.1 NAME

gnatsas-report - Display results or generate a report in one of the supported formats once an analysis has been performed. If no format is specified, text format is applied.

12.5.2 SYNOPSIS

gnatsas report [*COMMAND*] ...

12.5.3 DESCRIPTION

Display results of an analysis.

12.5.4 COMMANDS

code-climate [*OPTION*]... [*FILE.sam*]
Display results in Code Climate format.

csv [*OPTION*]... [*FILE.sam*]
Display results in CSV format.

exit-code [*OPTION*]... [*FILE.sam*]
Count messages and exit with the result as exit code. Count all the messages that are not info or annotation by default. You can use --show to specify which messages to count.

gnat-studio [*OPTION*]... [*FILE.sam*]
Generate report for GNAT Studio.

gnathub [*OPTION*]... [*FILE.sam*]
Generate report for GNAThub.

html [*OPTION*]... [*FILE.sam*]
Generate an HTML report.

sarif [*OPTION*]... [*FILE.sam*]
Generate report in SARIF format.

security [*OPTION*]... [*FILE.sam*]
Generate a security report.

text [*OPTION*]... [*FILE.sam*]
Display results in text format.

12.5.5 ARGUMENTS

FILE.sam
[Optional] Specify a SAM file from which to generate a report. This can be used 1) in addition to a GPR project file specified with -P or 2) instead of such specification. In case 1), this can be used to generate a report from an arbitrary SAM file, whereas the one from the latest analysis (or corresponding to the specified timeline) is used otherwise. In case 2), the report format must be explicitly provided (e.g. 'text') and only the following formats are supported: [text, csv, exit-code, security, gnathub]. This can be useful in workflows where the project dependencies are not available in the environment. Note that options (such as filters) specified in the project file will not apply in such case and should be passed to the command-line if required.

12.5.6 OPTIONS

--compare-with=*VAL*
Specify a SAM file for a one-off comparison with the selected analysis run (the last run of the specified timeline, or the current run if no timeline was specified).

--no-show-backtraces
(Default) disable displaying backtraces.

--no-show-header
(Default) disable displaying a header with information about the run before the messages.

--no-show-reviews
(Default) disable printing review information, if any, alongside the messages. This prints the current review status, date of review, name of reviewer and associated comment.

-o *FILE*, --out=*FILE*
File in which the report should be written. '-' means standard output.

--show=*VAL*
Selectively hide or show messages according to the tool that emitted them, what they're about, the file or project they come from or whether they're new or not. See the **SHOW OPTION FORMATTING** section of the "gnatsas report --help" documentation page for detailed format of this option. Specifying this switch multiple times performs a union of the results of each filter.

--show-backtraces
Enable displaying backtraces.

--show-header
Enable displaying a header with information about the run before the messages.

--show-reviews
Enable printing review information, if any, alongside the messages. This prints the current review status, date of review, name of reviewer and associated comment.

--timeline=*VAL*
Timeline to use for the report.

(Deprecated) --file=*VAL*
[use --timeline instead] Analyze a single file, ignoring other files from the project.

(Deprecated) --files-from=*VAL*
[use --timeline instead] Specify the path of a file containing a list of files to analyze.

(Deprecated) --mode=*VAL*
[use --timeline instead] Analysis mode. Supported modes are 'fast' (default if not specified), 'deep'.

(Deprecated) --no-subprojects
[use --timeline instead] Analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.

(Deprecated) -U [*FILE*] (default=)
[use --timeline instead] When *FILE* is specified, analyze files contained in the closure of the *FILE* unit. If set and no *FILE* is specified, analyze all files in the full project tree, except externally built ones.

12.5.7 COMMON OPTIONS

These options are common to all commands.

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.5.8 SHOW OPTION FORMATTING

This section desribes the format of the value to pass to the report command's --show option in order to filter messages. Its format is:

--show [**default**|**all**|*category_constraint*]{,*category_constraint*}*

If **all** is specified, the default constraints are ignored and only the specified ones are applied. Otherwise, if nothing (or **default**) is specified, specified constraints are added to the default ones.

category_constraint's format is: *category*[=[**all**|**default**]|[**=**|**+**|**-**]*constraint*]{[**+**|**-**]*constraint*}*

Using **=** resets the constraints of the corresponding category while **+** (resp. **-**) adds them to (resp. removes them from) the previous ones, if any.

The *category* is one of **age**, **kind**, **rank**, **tool**, **cwe**, **review_status**, **review_kind**, **prj** or **file**.

The *constraint* depends on *category*:

age
constraints are: **unchanged**, **added** and **removed**

kind
constraints are: **warning**, **check**, **info**, **race_condition**, **annotation** and all message kinds. Spaces in message kinds must be replaced with underscores (e.g. "access check" must be specified as "access_check")

rank
constraints are **info**, **low**, **medium** and **high**

tool
constraints are **inspector**, **infer**, **gnatcheck** and **gnat**

cwe
constraints are any CWE or **none**

review_status
constraints are all review statuses plus **none**

review_kind
constraints are **not_a_bug**, **pending**, **bug**, **uncategorized** and **none**

prj
constraints are **runtime** and any project basename, or relative paths

file
constraints are source files basename, or relative paths

If a *constraint* contains one of the characters ',', '-', '+' or '=', this character should be quoted or escaped.

For **cwe**, **review_status** and **review_kind**, **none** matches those messages that do not have the corresponding information attached (i.e., no cwe and no review, respectively).

12.5.9 SHOW OPTION DEFAULT VALUE

The default filter for the text, code-climate and exit-code formats is equivalent to:
--show age=added+unchanged,kind=all-annotation-info,review_kind=all-not_a_bug

The default filter for csv and sarif formats is equivalent to:
--show kind=all-annotation-info

Other formats do not have any filtering. Refer to the User Guide for more information about default filtering.

12.5.10 SHOW OPTION EXAMPLES

Adding info messages to default output:
--show kind+info

Adding annotations to default output:
--show kind+annotation

Hiding messages with low rank from default output:
--show rank-low

Showing all messages:
--show all

Showing only warnings and checks:
--show all,kind=warning+check

Showing only added messages:
--show all,age=added

Showing all reviewed messages:
--show all,review_status-none

Showing only added high messages with no user review:
--show all,age=added,rank=high,review_status=none

Showing only messages from file a-b.adb:
--show 'file="a-b.adb"'

Showing all Infer messages and only high Inspector messages:
--show tool=inspector,rank=high --show tool=infer

12.5.11 GPR Switches

-aP
Add directory to the project search path

--autoconf
Specify/create the main config project file name

--config
Specify the configuration project file name

--db
Parse dir as an additional knowledge base

--db-
Do not load the standard knowledge base

--implicit-with
Add the given projects as a dependency on all loaded projects

-eL
Follows symlinks for project files

--no-project
Do not use a project file

-P
The project file

--print-gpr-registry
Print the GPR registry

--root-dir
Root directory of obj/lib/exec to relocate

--relocate-build-tree
Root obj/lib/exec dirs are current-directory or dir

--RTS
Use --RTS=<runtime> to specify the Ada runtime

--RTS
Use --RTS:<lang>=<runtime> to specify the runtime for language <lang>

--src-subdirs
prepend <obj>/dir to the list of source dirs for each project

--subdirs
Use dir as suffix to obj/lib/exec directories

--target
Specify a target for cross platforms

-X
Specify an external reference for Project Files

12.5.12 MORE HELP

Use **gnatsas** *COMMAND* --help for help on a single command.

12.5.13 EXIT STATUS

report exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.5.14 BUGS

Open bug reports by visiting **https://support.adacore.com/csm**.

12.5.15 SEE ALSO

gnatsas(1)

12.6 gnatsas-review

12.6.1 NAME

gnatsas-review - Review file interaction.

12.6.2 SYNOPSIS

gnatsas review [**--from-csv**=*VAL*] [**--import**=*VAL*] [*OPTION*]...

12.6.3 DESCRIPTION

Review file interaction.

12.6.4 OPTIONS

--from-csv=*VAL*
Import reviews from specified csv file

--import=*VAL*
Import reviews from specified review file

12.6.5 COMMON OPTIONS

These options are common to all commands.

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.6.6 GPR Switches

-aP
Add directory to the project search path

--autoconf
Specify/create the main config project file name

--config
Specify the configuration project file name

--db
Parse dir as an additional knowledge base

--db-
Do not load the standard knowledge base

--implicit-with
Add the given projects as a dependency on all loaded projects

-eL
Follows symlinks for project files

--no-project
Do not use a project file

-P
The project file

--print-gpr-registry
Print the GPR registry

--root-dir
Root directory of obj/lib/exec to relocate

--relocate-build-tree
Root obj/lib/exec dirs are current-directory or dir

--RTS
Use --RTS=<runtime> to specify the Ada runtime

--RTS
Use --RTS:<lang>=<runtime> to specify the runtime for language <lang>

--src-subdirs
prepend <obj>/dir to the list of source dirs for each project

--subdirs
Use dir as suffix to obj/lib/exec directories

--target
Specify a target for cross platforms

-X
Specify an external reference for Project Files

12.6.7 MORE HELP

Use **gnatsas** *COMMAND* --help for help on a single command.

12.6.8 EXIT STATUS

review exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.6.9 BUGS

Open bug reports by visiting **https://support.adacore.com/csm**.

12.6.10 SEE ALSO

gnatsas(1)

12.7 gnatsas-report-code-climate

12.7.1 NAME

gnatsas-report-code-climate - Display results in Code Climate format.

12.7.2 SYNOPSIS

gnatsas report code-climate [*OPTION*]... [*FILE.sam*]

12.7.3 ARGUMENTS

FILE.sam
[Optional] Specify a SAM file from which to generate a report. This can be used 1) in addition to a GPR project file specified with -P or 2) instead of such specification. In case 1), this can be used to generate a report from an arbitrary SAM file, whereas the one from the latest analysis (or corresponding to the specified timeline) is used otherwise. In case 2), the report format must be explicitly provided (e.g. 'text') and only the following formats are supported: [text, csv, exit-code, security, gnathub]. This can be useful in workflows where the project dependencies are not available in the environment. Note that options (such as filters) specified in the project file will not apply in such case and should be passed to the command-line if required.

12.7.4 OPTIONS

--compare-with=*VAL*
Specify a SAM file for a one-off comparison with the selected analysis run (the last run of the specified timeline, or the current run if no timeline was specified).

--long-desc
Include check name and engine name at the beginning of the description. This is not required by the format but provides more information in CodeClimate viewers with limited display capabilities.

-o *FILE*, --out=*FILE*
File in which the report should be written. '-' means standard output.

--root=*DIR*
Make the paths in the codeClimate files relative to *DIR* instead of the project directory. This is typically used to specify the root of the Git repository in order to generate a report for a GitLab integration.

--show=*VAL*
Selectively hide or show messages according to the tool that emitted them, what they're about, the file or project they come from or whether they're new or not. See the **SHOW OPTION FORMATTING** section of the "gnatsas report --help" documentation page for detailed format of this option. Specifying this switch multiple times performs a union of the results of each filter.

--timeline=*VAL*
Timeline to use for the report.

(Deprecated) --file=*VAL*
[use --timeline instead] Analyze a single file, ignoring other files from the project.

(Deprecated) --files-from=*VAL*
[use --timeline instead] Specify the path of a file containing a list of files to analyze.

(Deprecated) --mode=*VAL*
[use --timeline instead] Analysis mode. Supported modes are 'fast' (default if not specified), 'deep'.

(Deprecated) --no-subprojects
[use --timeline instead] Analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.

(Deprecated) -U [*FILE*] (default=)
[use --timeline instead] When *FILE* is specified, analyze files contained in the closure of the *FILE* unit. If set and no *FILE* is specified, analyze all files in the full project tree, except externally built ones.

12.7.5 COMMON OPTIONS

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.7.6 EXIT STATUS

code-climate exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.7.7 SEE ALSO

gnatsas(1)

12.8 gnatsas-report-csv

12.8.1 NAME

gnatsas-report-csv - Display results in CSV format.

12.8.2 SYNOPSIS

gnatsas report csv [*OPTION*]... [*FILE.sam*]

12.8.3 ARGUMENTS

FILE.sam
: [Optional] Specify a SAM file from which to generate a report. This can be used 1) in addition to a GPR project file specified with -P or 2) instead of such specification. In case 1), this can be used to generate a report from an arbitrary SAM file, whereas the one from the latest analysis (or corresponding to the specified timeline) is used otherwise. In case 2), the report format must be explicitly provided (e.g. 'text') and only the following formats are supported: [text, csv, exit-code, security, gnathub]. This can be useful in workflows where the project dependencies are not available in the environment. Note that options (such as filters) specified in the project file will not apply in such case and should be passed to the command-line if required.

12.8.4 OPTIONS

--compare-with=*VAL*
: Specify a SAM file for a one-off comparison with the selected analysis run (the last run of the specified timeline, or the current run if no timeline was specified).

-o *FILE*, --out=*FILE*
: File in which the report should be written. '-' means standard output.

--show=*VAL*
: Selectively hide or show messages according to the tool that emitted them, what they're about, the file or project they come from or whether they're new or not. See the **SHOW OPTION FORMATTING** section of the "gnatsas report --help" documentation page for detailed format of this option. Specifying this switch multiple times performs a union of the results of each filter.

--timeline=*VAL*
: Timeline to use for the report.

(Deprecated) --file=*VAL*
: [use --timeline instead] Analyze a single file, ignoring other files from the project.

(Deprecated) --files-from=*VAL*
: [use --timeline instead] Specify the path of a file containing a list of files to analyze.

(Deprecated) --mode=*VAL*
: [use --timeline instead] Analysis mode. Supported modes are 'fast' (default if not specified), 'deep'.

(Deprecated) --no-subprojects
: [use --timeline instead] Analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.

(Deprecated) -U [*FILE*] (default=)
[use --timeline instead] When *FILE* is specified, analyze files contained in the closure of the *FILE* unit. If set and no *FILE* is specified, analyze all files in the full project tree, except externally built ones.

12.8.5 COMMON OPTIONS

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.8.6 EXIT STATUS

csv exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.8.7 SEE ALSO

gnatsas(1)

12.9 gnatsas-report-exit-code

12.9.1 NAME

gnatsas-report-exit-code - Count messages and exit with the result as exit code. Count all the messages that are not info or annotation by default. You can use --show to specify which messages to count.

12.9.2 SYNOPSIS

gnatsas report exit-code [*OPTION*]... [*FILE.sam*]

12.9.3 ARGUMENTS

FILE.sam
: [Optional] Specify a SAM file from which to generate a report. This can be used 1) in addition to a GPR project file specified with -P or 2) instead of such specification. In case 1), this can be used to generate a report from an arbitrary SAM file, whereas the one from the latest analysis (or corresponding to the specified timeline) is used otherwise. In case 2), the report format must be explicitly provided (e.g. 'text') and only the following formats are supported: [text, csv, exit-code, security, gnathub]. This can be useful in workflows where the project dependencies are not available in the environment. Note that options (such as filters) specified in the project file will not apply in such case and should be passed to the command-line if required.

12.9.4 OPTIONS

--compare-with=*VAL*
: Specify a SAM file for a one-off comparison with the selected analysis run (the last run of the specified timeline, or the current run if no timeline was specified).

--show=*VAL*
: Selectively hide or show messages according to the tool that emitted them, what they're about, the file or project they come from or whether they're new or not. See the **SHOW OPTION FORMATTING** section of the "gnatsas report --help" documentation page for detailed format of this option. Specifying this switch multiple times performs a union of the results of each filter.

--timeline=*VAL*
: Timeline to use for the report.

(Deprecated) --file=*VAL*
: [use --timeline instead] Analyze a single file, ignoring other files from the project.

(Deprecated) --files-from=*VAL*
: [use --timeline instead] Specify the path of a file containing a list of files to analyze.

(Deprecated) --mode=*VAL*
: [use --timeline instead] Analysis mode. Supported modes are 'fast' (default if not specified), 'deep'.

(Deprecated) --no-subprojects
[use --timeline instead] Analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.

(Deprecated) -U [*FILE*] (default=)
[use --timeline instead] When *FILE* is specified, analyze files contained in the closure of the *FILE* unit. If set and no *FILE* is specified, analyze all files in the full project tree, except externally built ones.

12.9.5 COMMON OPTIONS

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.9.6 EXIT STATUS

exit-code exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.9.7 SEE ALSO

gnatsas(1)

12.10 gnatsas-report-html

12.10.1 NAME

gnatsas-report-html - Generate an HTML report.

12.10.2 SYNOPSIS

gnatsas report html [*OPTION*]... [*FILE.sam*]

12.10.3 ARGUMENTS

FILE.sam
: [Optional] Specify a SAM file from which to generate a report. This can be used 1) in addition to a GPR project file specified with -P or 2) instead of such specification. In case 1), this can be used to generate a report from an arbitrary SAM file, whereas the one from the latest analysis (or corresponding to the specified timeline) is used otherwise. In case 2), the report format must be explicitly provided (e.g. 'text') and only the following formats are supported: [text, csv, exit-code, security, gnathub]. This can be useful in workflows where the project dependencies are not available in the environment. Note that options (such as filters) specified in the project file will not apply in such case and should be passed to the command-line if required.

12.10.4 OPTIONS

--compare-with=*VAL*
: Specify a SAM file for a one-off comparison with the selected analysis run (the last run of the specified timeline, or the current run if no timeline was specified).

-o *FILE*, --out=*FILE*
: File in which the report should be written. '-' means standard output.

--show=*VAL*
: Selectively hide or show messages according to the tool that emitted them, what they're about, the file or project they come from or whether they're new or not. See the **SHOW OPTION FORMATTING** section of the "gnatsas report --help" documentation page for detailed format of this option. Specifying this switch multiple times performs a union of the results of each filter.

--timeline=*VAL*
: Timeline to use for the report.

(Deprecated) --file=*VAL*
: [use --timeline instead] Analyze a single file, ignoring other files from the project.

(Deprecated) --files-from=*VAL*
: [use --timeline instead] Specify the path of a file containing a list of files to analyze.

(Deprecated) --mode=*VAL*
: [use --timeline instead] Analysis mode. Supported modes are 'fast' (default if not specified), 'deep'.

(Deprecated) --no-subprojects
[use --timeline instead] Analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.

(Deprecated) -U [*FILE*] (default=)
[use --timeline instead] When *FILE* is specified, analyze files contained in the closure of the *FILE* unit. If set and no *FILE* is specified, analyze all files in the full project tree, except externally built ones.

12.10.5 COMMON OPTIONS

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.10.6 EXIT STATUS

html exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.10.7 SEE ALSO

gnatsas(1)

12.11 gnatsas-report-sarif

12.11.1 NAME

gnatsas-report-sarif - Generate report in SARIF format.

12.11.2 SYNOPSIS

gnatsas report sarif [*OPTION*]... [*FILE.sam*]

12.11.3 ARGUMENTS

FILE.sam
: [Optional] Specify a SAM file from which to generate a report. This can be used 1) in addition to a GPR project file specified with -P or 2) instead of such specification. In case 1), this can be used to generate a report from an arbitrary SAM file, whereas the one from the latest analysis (or corresponding to the specified timeline) is used otherwise. In case 2), the report format must be explicitly provided (e.g. 'text') and only the following formats are supported: [text, csv, exit-code, security, gnathub]. This can be useful in workflows where the project dependencies are not available in the environment. Note that options (such as filters) specified in the project file will not apply in such case and should be passed to the command-line if required.

12.11.4 OPTIONS

--compare-with=*VAL*
: Specify a SAM file for a one-off comparison with the selected analysis run (the last run of the specified timeline, or the current run if no timeline was specified).

--inline-cwes
: Inline information about CWEs related to the rule of a given result into the result's message (disabled by default). This is useful when the SARIF consumer lacks the feature of displaying relations between rules.

-o *FILE*, --out=*FILE*
: File in which the report should be written. '-' means standard output.

--print-all-rules
: Include all rules (including CWEs) checked by GNATSAS in the output SARIF log. If not specified, only the rules (and CWEs) corresponding to actual results are included. Use this option when stability of the SARIF log is a strong requirement, e.g. for GitHub code scanning.

--root=*DIR*
: Make the URIs in the SARIF files relative to *DIR* instead of the project directory. The purpose of this switch is to provide portability by letting users control the relative path (uri) to source files in the generated SARIF results.

--show=*VAL*
: Selectively hide or show messages according to the tool that emitted them, what they're about, the file or project they come from or whether they're new or not. See the **SHOW OPTION FORMATTING** section of the "gnatsas report --help" documentation page for detailed format of this option. Specifying this switch multiple times performs a union of the results of each filter.

--timeline=*VAL*
Timeline to use for the report.

--use-abs-paths
Use absolute paths in URIs instead of defining uriBaseIds and using relative paths to those. This is useful when the SARIF consumer does not support resolution of relative paths to uriBaseIds.

(Deprecated) --file=*VAL*
[use --timeline instead] Analyze a single file, ignoring other files from the project.

(Deprecated) --files-from=*VAL*
[use --timeline instead] Specify the path of a file containing a list of files to analyze.

(Deprecated) --mode=*VAL*
[use --timeline instead] Analysis mode. Supported modes are 'fast' (default if not specified), 'deep'.

(Deprecated) --no-subprojects
[use --timeline instead] Analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.

(Deprecated) -U [*FILE*] (default=)
[use --timeline instead] When *FILE* is specified, analyze files contained in the closure of the *FILE* unit. If set and no *FILE* is specified, analyze all files in the full project tree, except externally built ones.

12.11.5 COMMON OPTIONS

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.11.6 EXIT STATUS

sarif exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.11.7 SEE ALSO

gnatsas(1)

12.12 gnatsas-report-security

12.12.1 NAME

gnatsas-report-security - Generate a security report.

12.12.2 SYNOPSIS

gnatsas report security [*OPTION*]... [*FILE.sam*]

12.12.3 ARGUMENTS

FILE.sam
[Optional] Specify a SAM file from which to generate a report. This can be used 1) in addition to a GPR project file specified with -P or 2) instead of such specification. In case 1), this can be used to generate a report from an arbitrary SAM file, whereas the one from the latest analysis (or corresponding to the specified timeline) is used otherwise. In case 2), the report format must be explicitly provided (e.g. 'text') and only the following formats are supported: [text, csv, exit-code, security, gnathub]. This can be useful in workflows where the project dependencies are not available in the environment. Note that options (such as filters) specified in the project file will not apply in such case and should be passed to the command-line if required.

12.12.4 OPTIONS

--compare-with=*VAL*
Specify a SAM file for a one-off comparison with the selected analysis run (the last run of the specified timeline, or the current run if no timeline was specified).

-o *FILE*, --out=*FILE*
File in which the report should be written. '-' means standard output.

--show=*VAL*
Selectively hide or show messages according to the tool that emitted them, what they're about, the file or project they come from or whether they're new or not. See the **SHOW OPTION FORMATTING** section of the "gnatsas report --help" documentation page for detailed format of this option. Specifying this switch multiple times performs a union of the results of each filter.

--show-mitigated-cwes
Show full security report, including CWEs mitigated by Ada.

--timeline=*VAL*
Timeline to use for the report.

(Deprecated) --file=*VAL*
[use --timeline instead] Analyze a single file, ignoring other files from the project.

(Deprecated) --files-from=*VAL*
[use --timeline instead] Specify the path of a file containing a list of files to analyze.

(Deprecated) --mode=*VAL*
[use --timeline instead] Analysis mode. Supported modes are 'fast' (default if not specified), 'deep'.

(Deprecated) --no-subprojects
[use --timeline instead] Analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.

(Deprecated) -U [*FILE*] (default=)
[use --timeline instead] When *FILE* is specified, analyze files contained in the closure of the *FILE* unit. If set and no *FILE* is specified, analyze all files in the full project tree, except externally built ones.

12.12.5 COMMON OPTIONS

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.12.6 EXIT STATUS

security exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.12.7 SEE ALSO

gnatsas(1)

12.13 gnatsas-report-text

12.13.1 NAME

gnatsas-report-text - Display results in text format.

12.13.2 SYNOPSIS

gnatsas report text [*OPTION*]... [*FILE.sam*]

12.13.3 ARGUMENTS

FILE.sam
[Optional] Specify a SAM file from which to generate a report. This can be used 1) in addition to a GPR project file specified with -P or 2) instead of such specification. In case 1), this can be used to generate a report from an arbitrary SAM file, whereas the one from the latest analysis (or corresponding to the specified timeline) is used otherwise. In case 2), the report format must be explicitly provided (e.g. 'text') and only the following formats are supported: [text, csv, exit-code, security, gnathub]. This can be useful in workflows where the project dependencies are not available in the environment. Note that options (such as filters) specified in the project file will not apply in such case and should be passed to the command-line if required.

12.13.4 OPTIONS

--compare-with=*VAL*
Specify a SAM file for a one-off comparison with the selected analysis run (the last run of the specified timeline, or the current run if no timeline was specified).

--no-show-backtraces
(Default) disable displaying backtraces.

--no-show-header
(Default) disable displaying a header with information about the run before the messages.

--no-show-reviews
(Default) disable printing review information, if any, alongside the messages. This prints the current review status, date of review, name of reviewer and associated comment.

-o *FILE*, --out=*FILE*
File in which the report should be written. '-' means standard output.

--show=*VAL*
Selectively hide or show messages according to the tool that emitted them, what they're about, the file or project they come from or whether they're new or not. See the **SHOW OPTION FORMATTING** section of the "gnatsas report --help" documentation page for detailed format of this option. Specifying this switch multiple times performs a union of the results of each filter.

--show-backtraces
Enable displaying backtraces.

--show-header
Enable displaying a header with information about the run before the messages.

--show-reviews
Enable printing review information, if any, alongside the messages. This prints the current review status, date of review, name of reviewer and associated comment.

--timeline=*VAL*
Timeline to use for the report.

(Deprecated) --file=*VAL*
[use --timeline instead] Analyze a single file, ignoring other files from the project.

(Deprecated) --files-from=*VAL*
[use --timeline instead] Specify the path of a file containing a list of files to analyze.

(Deprecated) --mode=*VAL*
[use --timeline instead] Analysis mode. Supported modes are 'fast' (default if not specified), 'deep'.

(Deprecated) --no-subprojects
[use --timeline instead] Analyze all files in the root project only, ignoring all subprojects and also ignoring main units defined in the root project.

(Deprecated) -U [*FILE*] (default=)
[use --timeline instead] When *FILE* is specified, analyze files contained in the closure of the *FILE* unit. If set and no *FILE* is specified, analyze all files in the full project tree, except externally built ones.

12.13.5 COMMON OPTIONS

--force-lock
Force gnatsas lock file.

--help[=*FMT*] (default=auto)
Show this help in format *FMT*. The value *FMT* must be one of **auto**, **pager**, **groff** or **plain**. With **auto**, the format is **pager** or **plain** whenever the **TERM** env var is **dumb** or undefined.

--list-timelines
List available timelines for project with detailed information and exit with return code 0 without performing other actions.

--log-to=*VAL*
Specify log file.

-q, --quiet
Suppress info output.

--save-temps, --keep-temp-files
Do not remove temporary files. Only useful for debugging.

--unset-log-flag=*VAL*
Deactivate a flagged output.

-v, --verbose
Give verbose output.

--version
Show version information.

12.13.6 EXIT STATUS

text exits with the following status:

0
on success.

123
on indiscriminate errors reported on standard error.

124
on command line parsing errors.

125
on unexpected internal errors (bugs).

12.13.7 SEE ALSO

gnatsas(1)

CHAPTER

THIRTEEN

FAQ

13.1 Migrating Away from the Historical CodePeer Database

Starting with GNAT SAS 24.0w (20230404), GNAT SAS's historical database is replaced by a set of files, the most important ones being the SAM and SAR files, that store GNAT SAS's messages and reviews, respectively (see *Message Files*).

After updating and when running GNAT SAS, a user with a historical database will be prompted to import its database before doing any new analysis. This is done with the *sam-from-db* tool. To remove the migration message, simply move (or remove) the database `codepeer/<prj>.db/Sqlite.db`. Similarly, if the project file contains a *CodePeer* package, the user is prompted to migrate it to the new *Analyzer package* (see *Ensuring compatibility with GNAT SAS*) before doing any new analysis.

This move away from the database implies the following changes:

- GNAT SAS does not maintain a full history of analysis anymore.
  - However, it is now easy to save the results of a particular run, by simply saving the associated SAM file (see *GNAT SAS Files Reference*).
  - We encourage users to version the SAM files along with their sources, this way the analysis results can be kept in sync with a given revision of the corresponding sources.
- The IDE server is discontinued.

  Users should do reviews locally (from scratch or from a copied SAR, see *Reviewing Messages*), and update their changes. Concurrent reviewing can be handled with the *merge-reviews* tool, see *Importing and Sharing GNAT SAS User Reviews*.
- The *run id* notion is now replaced by SAM files themselves. The `--current` switch that took a run id as an argument is replaced by the capability to display an arbitrary SAM file with `gnatsas report <sam-file>`. Similarly, `--set-baseline-id` is replaced by `gnatsas baseline --set-baseline <sam-file>`.

Switching to SAM/SAR files introduces a lot of flexibility with regard to result and review sharing, run comparison, backups, as well as *advanced workflows setup*. All that while keeping the gist of the historical database: baselines, run comparison, and reviews.

In addition, this complete re-implementation of the GNAT SAS pipeline results in sizable speed up in particular when storing results or generating reports of any format.

13.1.1 GNAT SAS Switches Changes

GNAT SAS 24.0 implements a new command line interface. It is possible to convert your old `codepeer` commands to the new `gnatsas` commands by just running your old `codepeer` command and following the instructions that are printed.

You can learn more about GNAT SAS' command line switches by passing the `--help` switch to `gnatsas` and its various subcommands, e.g. `gnatsas analyze --help` will print the `analyze`'s command help, while `gnatsas report --help` will print the `report`'s command help.

See also:

See *GNAT SAS CLI Reference* for more information about available command-line switches.

Additionally, you should make sure that the `Switches` listed in the *CodePeer* package of your project files are adapted to the new `Switches` syntax of the new *Analyzer package*, check *Ensuring compatibility with GNAT SAS* for more information.

13.1.2 sam-from-db

The *sam-from-db* tool is a tool to extract data from a CodePeer historical database and generate SAM and SAR files from it.

The default invocation is:

```
$ sam-from-db -P <prj>.gpr
```

You should add any `-X` or `--subdirs` switch that you use with CodePeer to this command line.

This extracts, for each analysis level in the database, a pair of SAM files for the baseline run and the last run, as well as a SAR file containing all reviews associated to the exported messages. Once extracted, the tool puts the files in the *output directory* of your project, and writes the *prj.runs_info.json* file that contains the levels information.

See also:

See also *GNAT SAS Files Reference*.

Warning: GNAT SAS does not have analysis levels, instead it has two analysis modes, namely *fast analysis mode* and *deep analysis mode* (see *Analysis modes*). Data from each legacy CodePeer analysis level is exported in a timeline with the level number as name (see *Timelines*). As a result, the imported results are not directly available through the GNAT SAS analysis modes. In order to get the imported results with an analysis mode, you can either:

- use the `--set-baseline` and `--set-current` switches with, e.g.:

  ```
  gnatsas baseline -P<prj> --set-baseline <output_dir>/<prj>.1.baseline.sam --set-
  ↪current <output_dir>/<prj>.1.sam
  ```

  (see *Comparing GNAT SAS Runs* and *Importing GNAT SAS Results*),

- or directly use the imported timeline, e.g., use:

  ```
  gnatsas report -P<prj> --timeline 1
  ```

  or

  ```
  gnatsas analyze -P<prj> --mode fast --timeline 1
  ```

  Adding `--timeline 1` to a gnatsas command puts this command in the context of the imported CodePeer level 1 analysis.

If *sam-from-db* is not able to locate your database, or if you want to specify a custom path, use the `--db-path` switch to manually set it.

Warning: *sam-from-db* does not extract all the information stored in the database. By default, it only extracts the baseline and last run of each analysis level.

You can use this tool to do selective exportation with either:

- `--level <level>` to only export one level,
- `--run-id <id>` to only export one run-id (requires `--level`),
- `--reviews-only` to only export reviews.

If one of those switches is set, *sam-from-db* generates the specified files in the current directory, or `--out-dir` if set, and exits.

Tool Limitation with GNATcheck's Messages

Due to a change in the way GNATcheck messages are stored, GNAT SAS is not able to properly identify a GNATcheck message from a database import with one from a new run. As a result, when you use a SAM file generated from the database as a baseline, all GNATcheck messages from the database are marked as *Removed*, and all the ones from the current run are marked as *Added*. The problem disappears once the baseline is updated.

13.2 What is the Difference between GNAT SAS and SPARK?

Static analyzers fall into two broad categories: bug finders and verifiers. Bug finders detect violations of properties. Verifiers guarantee the absence of violations of properties. Because they target opposite goals, bug finders and verifiers usually have different architectures, are based on different technologies, and require different methodologies.

Typically, bug finders require little upfront work, but may generate many false alarms which need to be manually triaged and addressed, while verifiers require some upfront work, but generate fewer false alarms.

GNAT SAS belongs to the bug finder category, while SPARK belongs to the verifier category.

In other words, if you want to assess the quality of your code and detect potential run-time or security errors for manual review in Ada code then GNAT SAS is the right tool.

If you want instead to guarantee absence of run-time or security errors automatically and demonstrate useful properties of your application then SPARK is the right tool.

13.3 What is the difference between GNAT SAS Messages and Compiler Warnings?

A question asked by some users is: *Why not just use the compiler? The Ada compiler generates warnings, so what added value does GNAT SAS bring?*.

It is true that Ada compilers (and in particular GNAT) generate warnings, and these warnings are useful, but they are definitely not comparable to what GNAT SAS' Inspector and Infer engines can achieve for several fundamental reasons:

- Most messages generated by Inspector and Infer are not considered at all by Ada compilers beyond really obvious cases (e.g. all potential cases of run-time errors are not flagged by Ada compilers, detection of race conditions, etc...)

- Compilers in general will do no or little data and control flow tracing to emit warnings. GNAT SAS, and in particular Inspector, performs full data and control flow tracing as explained in *How Does Inspector Work?*.
- Compiler warnings operate only on a single subprogram (or in the best cases, single unit), while Inspector and Infer do interunit analysis and propagate information from one unit to the other.
- Inspector and Infer will keep much more precise track of values of e.g. individual fields or array elements and generate more precise messages as a result.

GNAT SAS also comes with many additional capabilities not available in compilers such as baselines, the ability to generate contracts for subprograms, etc...

Finally, when using GNAT SAS you can decide to take advantage of GNAT Warnings via the `--gnat` switch, see *Configuring GNAT Warnings* for more details.

13.4 Common Issues

13.4.1 Compilation Errors

Note: This section only applies to GNAT-based engines (i.e. Inspector and GNAT Warnings).

GNAT-based engines may issue compilation errors during the GNAT SAS analysis, but that does not always prevent completing the analysis since both Inspector and GNAT warnings are able to partially analyze projects with these errors. Inspector needs a file to be compilable to analyze it, but analyzes the rest of the files treating the calls to subprograms defined in the files with compilation errors as *Calls to Unknown Subprograms*. GNAT Warnings can in some cases partially analyze even files with compilation errors.

A compilation error can be caused either by the fact that the code is not valid Ada code and cannot be compiled by GNAT compiler or by the fact that that project is not properly set up.

To resolve compilation errors due to improper setup, see the instructions in the section *Ensuring compatibility with GNAT SAS*.

In the case of invalid Ada code, you can resolve the compilation error either by fixing the code or by excluding the file from the compilation using the *Excluded_Source_Files* attribute in the top-level of the project:

```
project Work  is
  ...
  for Excluded_Source_Files use ("pack.adb");
  ...
end Work;
```

Note: Using this attribute at the project level is different from using it in the *Analyzer package* as described in the section *Partial Analysis*. When the attribute is used in the *Analyzer package*, the file is still compiled.

Warning: There are limitations to this exclusion mechanism. In particular, as mentioned in *Inspector-specific handling*:

- It is problematic to do this for specification files since excluding them may cause compilation errors in all the files depending on them.
- Excluding a file at the project level has an impact on Inspector's incremententality.

CHAPTER

FOURTEEN

APPENDIX

14.1 Inspector Reference

14.1.1 How Does Inspector Work?

At its heart, Inspector's technology is most like a whole-program "optimizer": it uses extended versions of algorithms familiar to those writing compiler optimizers. First, a conventional compiler front end is used to convert the program into a treelike intermediate language, called "SCIL" (Statically Checkable Intermediate Language). Then, the Inspector "engine" takes over and begins with a global aliasing analysis followed by translation of each subprogram into a static single assignment (SSA) representation and then a global value numbering over the subprogram. At this point, most compiler optimizers would begin the process of generating code. Inspector instead begins the process of determining whether the program might violate any run-time checks or user assertions and whether the algorithms have any "race conditions" or security "holes."

Inspector analyzes each subprogram separately, by taking into account only the information computed for callees of the subprogram. In particular, the analysis of a subprogram does not depend on its callers. To perform this analysis, Inspector, using the global value numbering, determines the possible value set for every variable and expression appearing in the program at every point where it appears. It uses this information to determine where checks or assertions might fail and also uses it to determine the "preconditions" and "postconditions" of every subprogram. The preconditions represent what must be true about the inputs to the subprogram for the subprogram's algorithm to operate without overflowing or failing a run-time check. The postconditions represent what is true about the outputs after the subprogram completes, presuming that the inputs satisfied the preconditions.

Once the pre- and postconditions for a subprogram have been determined, they can be propagated to every point where the subprogram is called, iterating as necessary in the presence of recursion, until the whole program has been analyzed. Once this process stabilizes, a final pass is performed to report any potential run-time or assertion failures, race conditions, or security holes that exist anywhere within the program. During that pass, Inspector may generate error messages whenever a caller could violate the precondition generated for a callee to guard against an error in the callee. As a result, an error may not be reported by Inspector at the point where there is a fault in the program, but rather at the point where this fault results in an inconsistency. Reviewing the preconditions at this point allows identifying whether the caller or the callee is the culprit by matching the preconditions generated by Inspector and the expectations of the programmer. Reviewing the preconditions generated by Inspector is another way of detecting errors before they occur in the program.

The Inspector technology is fundamentally a "bottom up" approach, which provides excellent scalability. The complexity of the analysis is not dependent on the number of paths in the call graph but rather merely on the number of subprograms in the graph, with a factor based on the amount of recursion. The technology is also very precise, in that the pre- and postconditions for every subprogram include information about all inputs and outputs, including all global variables and objects reachable through chains of pointers. This precision helps to reduce the false-positive rate while at the same time avoiding false negatives.

The book *Static Analysis of Software: The Abstract Interpretation*, published by Wiley (2012), contains a chapter on the mathematics underlying the inner working of Inspector.

14.1.2 Partitioning of Analysis

When analysis a large set of sources and depending on how much memory is available, Inspector may decide to split the analysis into multiple *partitions*, consisting of a subset of the sources.

The partitioning is done based on the following criteria:

1. When using Inspector's `-global` switches, the analysis is always done globally, with a single partition.
2. Otherwise, based on the size of the sources the analysis may be split into multiple partitions. This is done using a call-graph based analysis which attempts to minimize the number of cases where a caller and a callee are assigned to different partition elements. The default size of partitions is set per level to use a certain maximum amount of memory:

Level	Memory limit per partition
1	1GB
2	3GB
3	4GB
4	Unlimited

   This can be further tuned via Inspector's `--partition-memory-limit` switch, e.g. `--partition-memory-limit 8GB`.

Each partition is saved in a file called `part-xxx-of-project.library` where `xxx` is the partition number (e.g. `001`, `002`, etc...).

For subsequent analysis, the previous partition files - if found - are reused from one run to another, except that files no longer present in a new run are removed, and new files are appended at the end of the last partition, or in a new partition if the last partition was already full.

Note that it is more likely that a global analysis might run out of memory during the processing of some source file. In that case, the source file is skipped and analysis proceeds as though that file was not present in the library, unless memory is really stressed too far, in which case the entire analysis may fail.

There is also a switch `--project-partitions` which is provided. With this switch, in deep analysis mode, at least one partition is created for each project file (.gpr) in your project hierarchy, assuming you are using more than one project file. In other cases (fast analysis mode or a single project file), this switch is ignored. Specification of `--project-partitions` is not usually recommended unless partitioning compatibility with older versions of Inspector is desired, or if you want to manually influence Inspector's partition selection algorithm. When this switch is used, the partition files are regenerated after each run (as opposed to being reused run after run).

Finally, the optional switch `--simple-partitions` disables partitioning algorithm based on a call-graph analysis and assigns source files to partitions based on an alphabetical order.

14.1.3 Inspector Limitations and Heuristics

The most accurate static error detection is possible when the entire program representation is available to Inspector. There are two reasons why the entire program might not be available: a portion of the program might simply not be available: perhaps it hasn't been written yet or perhaps it's from a restricted library; the other reason is that Inspector might have to partition the program representation into manageable sized *partitions*.

Inspector performs static error detection by holding in main memory a representation of the entire subsystem being analyzed. It is possible that the subsystem being analyzed is so large that its representation cannot be contained in the memory of the host machine on which Inspector is running. Inspector automatically divides the subsystem being analyzed into partitions that are more likely to be analyzable within the memory constraints of the host machine. See *Partitioning of Analysis* for more details on how Inspector decides to partition an analysis.

Evaluation Order

Inspector assumes that arguments of calls are evaluated in left-to-right order. In practice, compilers are allowed to choose any evaluation order for arguments of calls. GNAT in particular may evaluate call arguments in different orders depending on the target platform. If the order of evaluation matters for the side effects of the call, then Inspector analysis may not be sound with respect to the behavior observed when executing on the target.

Inspector Presumptions

Sometimes the entire program is not available to Inspector, either because the program source is simply not available, or because the program was so large it needed to be partitioned. In either of these cases, important information about a subprogram invocation might not be available to Inspector at the point of subprogram invocation. Inspector will not be able to calculate the **preconditions** of the subprogram being called and it will not be able to precisely characterize the return value nor the effects of calling that subprogram. Invocations of subprograms that haven't been processed by Inspector are referred to as **unanalyzed subprogram calls**.

Inspector makes **presumptions** about the return values and other side effects of unanalyzed subprogram calls. The **presumptions** of each unanalyzed subprogram call are displayed in the subprogram information block of the calling subprogram. The source line number of the unanalyzed subprogram call is shown after an @ sign, and information about its presumed return value or other side effect is displayed.

The actual unanalyzed call might be directly on the line specified in the **presumption** (by the @*nnn*) or it might be a call that occurs inside the subprogram called on the given line. You can tell if it's an indirect call or a direct call by the name given in the **presumption**. If it doesn't correspond to the name of the subprogram directly called on the given line, then it must be something called indirectly.

Sometimes the **presumptions** made by Inspector are overly "optimistic", such as a given call will never return a null, when in fact it can return a null. For example, suppose that under some circumstances, an unanalyzed subprogram call can return a null, but that the calling subprogram never checks for this possibility, and instead dereferences the return value without checking. This could cause an exception when the program is run, and implies that the program being analyzed contains a latent defect. If Inspector makes the **presumption** that the called subprogram does not return null, then it would not identify this latent defect. For this reason, the programmer should check the **presumptions** about unanalyzed subprogram calls and verify that they are appropriate.

Alternatively, the **presumptions** made by Inspector might be "insufficient". For example Inspector might presume that some unanalyzed subprogram call *could* return null, when in fact it never does. Later, in some use of the returned value, far away from the unanalyzed call, Inspector might complain that a possibly-null value is being dereferenced. The original **presumption** was insufficient in this case, because it didn't match the actual behavior of the unanalyzed call, and thereby led to the complaint. The programmer can eliminate such a complaint by adding code immediately after the point of the call to assert that the returned value is not null. This will cause Inspector to tighten up its presumption about the unanalyzed call to satisfy the assertion, and the complaints at other points in the code will go away. For example:

```
Result := Call_To_Unknown_Function (...);
pragma Assert (Result /= null);
```

If you do not like Inspector's default handling of presumptions, you can disable this behavior via the `--no-presumptions` switch.

Handling of Generic Units

Inspector does not analyze generic units per se, since it doesn't have sufficient context in order to perform a precise enough analysis of the generic code (since it is dependent on its generic parameters which can change significantly the code generated).

Instead, Inspector analyzes each instantiation of generic units. This means in particular that if you do not instantiate a generic unit, it won't be analyzed. In addition if there are multiple instantiations of a single generic unit, Inspector will analyze each instantiation separately. Note that this is supported by Inspector, but in general not by the other analysis engines. For example, this is not yet supported by Infer as described in the section *Configuring Infer*.

Any messages generated during the analysis of one instantiation might very likely be similar or identical to those generated for other instantiations. The net effect is that there may be multiple similar messages associated with a single line in the source listing for a generic unit.

In addition, when using e.g. the *--no-subprojects* switch (analyze only sources from the root project), since instantiations are analyzed in the context of the client code (where the instantiation is located), you will get an analysis and potentially messages related to source files located outside the root project.

Loop Unrolling

In some cases, Inspector may decide that it is worthwhile to unroll a loop when generating the SCIL representation of the loop. For example, given Ada source code

```
for I in 1 .. 3 loop
   A (I) := I * 100;
end loop;
```

Inspector might (or might not) choose to represent this in the generated SCIL as something more like

```
A (1) := 100;
A (2) := 200;
A (3) := 300;
```

In most cases, this sort of thing is an internal implementation decision which the user need not be aware of. However, Inspector might then go on to emit messages which pertain to one specific iteration of the loop (or, as is discussed later, to iterations other than the first iteration). In such cases, Inspector will include additional text in each message in order to unambiguously identify the subject of the message. This additional text may take either of two forms. In the case of a for loop (or of an implicitly looping construct such as a quantified expression) with static bounds for which Inspector decides full unrolling is desirable, text such as "(iteration 3 of 5)" would be included. In the case of a loop for which only the first iteration (or perhaps only some part of the first iteration, such as the initial test condition evaluation for a while loop) is unrolled, text such as "(initial iteration)" or "(subsequent iterations)" would be included. In either case, such additional text refers to the nearest enclosing loop (or implicitly looping construct).

For example, given the subprogram

```
function Foo return Integer is
   type Vec is array (1 .. 3) of Integer;
   Nums : Vec := (1, 2, 3);
   Dens : Vec := (-1, 0, 1);
   Result : Integer := 0;
begin
   for I in Vec'Range loop
      Result := Result + (Nums (I) / Dens (I));
   end loop;
```

(continues on next page)

(continued from previous page)

```
   return Result;
end Foo;
```

Inspector generates the message:

```
foo.adb:8:38: high: divide by zero (Inspector): check fails here: requires Dens(I) /= 0␣
↪(iteration 2 of 3)
```

In order to decide whether or not to unroll a loop, Inspector uses some heuristics and in particular limits the number of generated statements after unrolling. This limit is 128 statements by default, and can be changed by setting the environment variable *CODEPEER_UNROLL_LIMIT* to a lower or greater value. Setting this environment variable to 0 will in effect disable loop unrolling. Note that this setting is taken into account when generating SCIL files, so you need to force generation of the SCIL file after enabling it, by e.g. using the GNAT SAS menu *GNAT SAS>Advanced>Remove SCIL*

Calls to Unknown Subprograms

If Inspector encounters calls to subprograms whose body is not available (e.g. imported from another language, or a runtime call not known to Inspector), it will report such calls as part of the info messages.

Unless the `--no-presumptions` switch is used, Inspector will generate presumptions (see *Inspector Presumptions*) for each call to unknown subprograms that might need to be reviewed for accuracy. When using `--no-presumptions`, no such presumptions are generated, at the expense of potentially more false positives.

Skipped Subprograms or Files

The static error detection analysis performed by Inspector is space- and time-intensive, because Inspector tracks the value of all program expressions across all possible execution paths. The present implementation of Inspector holds all this information in main memory on the host machine.

Certain Ada programs are too large or too complex to analyze within the memory constraints of the host machine. When Inspector detects this circumstance, it will skip analyzing the subprograms or files which are too complex. No annotations or error messages will be generated for these subprograms or files, and calls to subprograms which were skipped will be analyzed as though the subprogram body were not available. Detailed information about which files were skipped is available in the info messages and may also be found in the low level `Inspection.log` file when the switch `--dbg-on limitations` is given.

This kind of limitation is inherent to static analysis, and as for calls to unknown subprograms, calls to these skipped subprograms need to be reviewed manually instead.

As an example, certain syntax parsers can be too complex to analyze within the memory constraints of the host machine. The parsing routine will typically contain a loop and deeply nested *switch* or *if-then-else* statements with complex conditional expressions. These can introduce a very large number of paths and distinct computations into the program graph, which may bump into the host machine memory limit.

If you want Inspector to analyze these subprograms, you can try one of the following solutions:

- increase the number of steps: by default the number of steps for each subprogram is limited to 30 in fast analysis mode, and 1000 in deep analysis mode. You can increase this value by using the `--steps` switch.

  Note that an exception is made for elaboration procedures, whose analysis is very important in practice. (An analysis failure for an elaboration procedure means that the analysis for the entire module is abandoned.) Consequently, such subprograms are given thrice as many steps as the other ones.

- increase the amount of memory available on your machine: in case of a memory error reported by Inspector, runing on a machine with more main memory should help analyze complex subprograms.

- increase the maximum number of values computed: in case Inspector reports a *too many value numbers* in `Inspection.log` (when the switch `--dbg-on limitations` is given), then you might want to use the debug switch `--dbg-vn-limit` with a value above 100000, e.g: `--dbg-vn-limit 150000`. If this is not sufficient, you will need to simplify your subprogram, see below.
- simplify your subprogram: when Inspector reports an inability to analyze a subprogram, it often means that the subprogram is too complex to be analyzed, and should be split into multiple, simpler subprograms which Inspector can analyze separately.

Note that increasing one of the above limits may well run into another limit due to the inherent complexity of these subprograms (e.g. going from a "too many value numbers" limit to a steps limit, to an out of memory limit.

In general our recommendation is to either analyze such complex subprograms manually and let Inspector skip them, or alternatively refactor your code as mentioned above by e.g. splitting it into multiple simpler subprograms.

Since Inspector's analysis is modular, the rest of the code will still be analyzed and produce useful and relevant results.

Call Too Complex

When Inspector analyzes a call to a subprogram, it can hit two different kinds of limits. If one of these limits is hit, Inspector will analyze the call as if there were no body available.

If you want Inspector to analyze these calls, you can try to increase the limit. If the switch `--dbg-on limitations` is given, `Inspection.log` file contains messages about which limit is hit. By looking for *call too complex* in a line, you can see two different kinds of limits:

- in the case of hitting the *too many possible call targets* limit, it means that there are too many targets for a dispatching call or call through an access-to-subprogram value. You can increase the maximum number of targets with the debug switch `--dbg-call-target-limit <limit>`.
- in the case of hitting the *too many obj-ids to import* limit, you can increase the limit using debug switch `--dbg-obj-ids-import-limit <limit>`.

Dispatching Calls and Access to Subprograms

When Inspector encounters a dispatching call or a call via an access to subprogram variable, Inspector attempts to identify all possible such target calls by considering all the compatible primitive operations (or subprograms whose address was taken and whose profile is compatible in the case of an access to subprogram object). By default, the effect of these calls is taken into account in the caller, but the preconditions for these multiple targets (which may correspond to subprograms that cannot be called at this point) are ignored, to avoid an undue number of false positives. The net effect is that Inspector may not report certain possible precondition failures for multi-target calls.

When `--messages max` is used, then Inspector checks that the preconditions of all targets are satisfied at the point of call.

Null Array Bounds

In most cases, Ada requires that the high and low bounds of an index constraint for an array must belong to the corresponding index subtype of the array type. For example, because Standard.String is declared as:

```
type String is array (Positive range <>) of Character;
```

both bounds of an object or value of type String will usually be positive. There is an exception to this rule in the case of an array which has no elements because the low bound is greater than the high bound. In this case, values outside of the index subtype are permitted. It is not at all unusual to see high bounds outside of the index subtype, as in:

```
X : String(1 .. 0);
```

and Inspector handles this case. It is much more unusual to see low bounds outside of the index subtype, as in:

```
Unlikely  : String (-10 .. -20);
```

It happens that the possibility of an out-of-index-subtype lower bound significantly impairs Inspector's analysis in some cases that do commonly come up in practice. In order to pragmatically deal with this situation, Inspector will, in some cases, treat an out-of-index-subtype lower bound as an error. An array declaration such as the "Unlikely" example above will elaborate without raising any exception, as specified by the Ada language definition. Inspector, however, may treat it as an error. More importantly, in cases where the bounds of array are not known (e.g., a subprogram with a parameter of an unconstrained array subtype such as String), Inspector may assume for purposes of its analysis that the low bound belongs to the index subtype. Given the following example:

```
procedure P (X : String) is
   First_Is_Positive : Boolean := X'First > 0;
   Last_Is_Positive  : Boolean := X'Last > 0;
```

Inspector will assume that the local variable First_Is_Positive is initialized to True, whereas it knows that Last_Is_Positive might be False if X'Length = 0.

Handling of 'Valid

Inspector interprets `X'Valid` as `X` being in the range of its subtype if the declared range is static and if the subtype does not have any subtype predicates. `X'Valid` is interpreted as an unknown call in other cases.

Given the following example:

```
procedure P1 (X : Positive) is
begin
   if X'Valid then
      pragma Assert (X > 10);
   end if;
end P1;
```

Inspector interprets `X'Valid` as `X in 0 .. 2**31-1`. Since for an IN [OUT] parameter of a given subtype, Inspector assumes that the parameter is in its statically declared range, the code above will generate a `test always true` message and a precondition `X > 10`. Since such messages concerning `'Valid` are in general not useful, Inspector suppresses them unless `--messages max` is used.

Given the following example:

```
subtype Even is Integer
   with Dynamic_Predicate => Even mod 2 = 0;

procedure P2 (X : Even) is
begin
   if X'Valid then
       pragma Assert (X > 10);
   end if;
end P2;
```

Inspector interprets `X'Valid` as an unknown call. It thus does not know anything about its result and cannot generate a precondition for the assertion. This means that Inspector will emit a message about the potentially failing assertion.

Inspector emits a high-ranking message if the prefix of `'Valid` is known to be uninitialized. No message is emitted in the case where the prefix of `'Valid` only might be uninitialized, unless `--messages max` is used.

14.1.4 Advanced Inspector Command Line Switches

Advanced Inspector Switches

As described in *Analyzer package attributes*, these switches should be specified with the *for Switches ("inspector")* attribute of the project file.

- `--additional-patterns <file>`

  When present, this file will be merged with `MessagePatterns.xml`

- `--daemon "<package.subp>"`

  Specify that the named subprogram is the entry point of a daemon task (corresponding to an Ada task).

  Subprograms are identified by prefixing the name of the subprogram with the name of the package in which the subprogram is defined. Wildcard matching of the string providing the module and subprogram name (via the asterisk character) is available at the beginning or end of the specification, for example: "*package.subp*".

  See *Race Condition Messages* for more information.

- `--dbg-vn-limit <num_vns>`

  Specify maximum number of "value numbers" allowed during Inspector analysis of a single method/subprogram. See section *Skipped Subprograms or Files* for more information.

- `--dbg-call-target-limit <limit>`

  Specify maximum number of targets for a dispatching call or a call through an access-to-subprogram value. For Inspector analysis only. See section *Call Too Complex* for more information.

- `--dbg-obj-ids-import-limit <limit>`

  Specify maximum number of "obj-ids" to import from a callee to the caller. For Inspector analysis only. See section *Call Too Complex* for more information.

- `--message-patterns <file>`

  Specify an alternate `MessagePatterns.xml` file.

- `--method-memory-size <megs>`

  Specify the maximum memory size in MB to be used by Inspector for analyzing a single method/subprogram. By default, Inspector allocates a maximum of 1/3 of the amount of physical memory on the system to analyze an individual subprogram, to limit memory explosions in some corner cases. For very complex subprograms, this limit may not be sufficient and may be bumped via this switch. See section *Skipped Subprograms or Files* for more information.

- `--method-timeout <seconds>`

  Specify the maximum time spent, in seconds, by Inspector to analyze a single method/subprogram, to avoid spending too much time analyzing very complex subprograms. The default value is set to 600 (10 minutes) in fast mode and 1800 (30 minutes) in deep mode. Note that you should in general use the `--steps` switch instead.

- `--steps <num>`

  Specify the maximum number of steps for allowed before stopping Inspector's analysis of a method/subprogram, to avoid spending too much time analyzing very complex subprograms. The number of steps is calculated deterministically, which means that it will be the same for each run with the same analysis, regardless of the platform's speed. See section *Skipped Subprograms or Files* for more information.

- `--partition-memory-limit <GB>`

  Specify the maximum memory that should be used for each partition of Inspector analysis, in gigabytes. The optional suffix `GB` may be used explicitly, for example: `--partition-memory-limit 8GB` or simply: `--partition-memory-limit 8`

- `--no-preconditions`

  Do not propagate checks as implicit preconditions during Inspector analysis. This generally means that code within a subprogram that would be identified as a check will not be promoted as a precondition of the subprogram, typically resulting in messages flagged on lines corresponding to the checks rather than flagged on calls. As an exception to implicit precondition suppression, checks that inputs to subprograms (parameters and referenced globals) are initialized will still be propagated as preconditions, because flagging all uses of inputs would result in too many false-positive messages. In addition, any user-defined preconditions will remain as preconditions and be taken into account on calls. See also --no-propagation below.

- `--no-presumptions`

  Instruct Inspector to not generate any presumptions for unanalyzed subprogram calls. Use of this switch will generally result in additional messages being reported in the vicinity of unanalyzed calls, since assumptions about the result of such calls are no longer made.

- `--no-propagation "<package.subp>"`

  Instruct Inspector to not propagate checks as implicit preconditions for subprogram "package.subp." This is the same as --no-preconditions, but limited to the named subprogram, which might be a top-level operation for which it is not appropriate to presume the inputs will conform to any non-explicit precondition. The "--no-propagation <package.subp>" switch may appear multiple times, to suppress precondition propagation for more than one subprogram. An asterisk ('*') character may appear at the start or end of the name to indicate a wildcard match.

- `--no-race-conditions`

  Suppress analysis and reporting of race conditions in deep analysis mode. This may result in faster analysis.

- `--project-partitions`

  Create partitions based on the project hierarchy. See *Partitioning of Analysis* for more details.

- `--simple-partitions`

  Create partitions based on an alphabetical order of package names. See *Partitioning of Analysis* for more details.

- `--reentrant "<package.subp>"`

  Specify that the named subprogram is a reentrant entry point (corresponding to an Ada task type).

  Subprograms are identified by prefixing the name of the subprogram with the name of the package in which the subprogram is defined. Wildcard matching of the string providing the module and subprogram name (via the asterisk character) is available at the beginning or end of the specification, for example: "*package.subp*".

  See *Race Condition Messages* for more information.

14.1.5 Format of `MessagePatterns.xml` File

`MessagePatterns.xml` is a file to control the output of Inspector. Internally, Inspector generates a number of messages, each with a message text and ranking set to **Low** (see the section about ranking below). With `MessagePatterns.xml`, one can change the ranking of a message from Inspector or even suppress the message. You can either modify the file that is part of the GNAT SAS installation, or better specify a project-specific file or (preferred when possible) specify an additional project-specific file which comes in addition to the predefined file, see *Analyzer package attributes*.

Note that a simpler way to filter out messages based on message categories is via the `--show` command line switch.

Structure of the File

The file `MessagePatterns.xml` is simply a list of *rules*. The structure is the following:

```
<?xml version="1.0"?>
<Message_Probability_Rules>
   <!--  the list of rules here -->
</Message_Probability_Rules>
```

Each rule consists of two parts: a pattern and a target ranking. The pattern describes the type of message that this rule matches. The target ranking is the new ranking of the message as it will appear in the GNAT SAS output. In the XML format, a rule looks like this:

```
<Message_Rule Matching_Probability="RANKING">
  <Message_Pattern>
     <!-- definition of the pattern -->
  </Message_Pattern>
</Message_Rule>
```

The target ranking is marked *RANKING* in this rule, and we will describe below how a pattern can be defined.

The following ranking levels can be used as target ranking, in increasing order:

Info	Info messages about e.g. subprogram not being analyzed.
Low	Low ranking messages.
Medium	Medium ranking messages.
High	High ranking messages.
Suppressed	Used to suppress a message from the reports.

Matching of rules

For each message, Inspector will try to match each pattern of each rule. If no pattern matches, the ranking of the message remains **Low**. If a single pattern matches, the target ranking of the corresponding rule becomes the new ranking of the message. If several rules match, the highest ranking wins. Note in particular that **Suppressed** being the highest ranking, if a matching rule has target ranking **Suppressed**, the message is suppressed regardless of other matching rules.

Defining patterns

A pattern is a list of specifiers. There are three types of specifiers, Boolean specifiers, String specifiers, and computed specifiers. They are written as follows:

```
<Bool_Specifier Name_Of_Boolean_Attribute="NAME"></Bool_Specifier>

<String_Specifier
   Name_Of_String_Attribute="NAME"
   String_To_Match="REGEXP">
</String_Specifier>

<Computed_Specifier Name_of_Computed_Attribute="NAME"></Computed_Specifier>
```

For each type of specifier the *NAME* can be chosen among a predefined set. In addition, each specifier can be negated using the attribute `Complement="YES"`.

Boolean specifiers

For Boolean specifiers, the following attributes are allowed:

Attribute Name	Description
Is_Big_Int	The message concerns an integer value.
Is_Big_Rat	The message concerns a real value.
Is_Side_Effect	The message is related to a side effect generated after calling a subprogram.
Is_Pointer	The message concerns a pointer.
Bad_Set_Includes_Null_Literal	The message concerns a pointer value which should not be null.
Expected_Set_Is_Subset_Plus_Minus_1000	The message concerns a value that is in -1000 .. 1000.
Expected_Set_Is_Singleton_Set	The message concerns a value that can have only one concrete value.
Valid_Bad_Set_Is_Singleton_Set	The message concerns a value which can take any concrete values except one.
Valid_Bad_Set_Overlaps_Plus_Minus_1000	The message concerns a value that should not be in -1000 .. 1000.
Precondition_Set_Contains_Holes	The message is about a precondition set which is not a contiguous range.
Precondition_Set_Has_Singleton_Hole_At_Zero	The message is about a precondition set which includes the values around zero, but not zero itself.
Check_Is_Soft	The message is *soft* (see *Inspector Annotations*)
Check_Is_Loop_Bound_Check	This message concerns a loop bound check
Bad_Set_Only_Invalid	The message is about a possibly uninitialized memory location.
Message_Depends_On_Unknown_Call	Message depends on unknown call.
Exception_Handler_Present	The enclosing basic block contains an exception handler. This is relevant in the context of validity checks when Inspector is checking whether scalar out parameters have been initialized.
Uninteresting_RHS	Uninteresting RHS in the context of dead store message.
Message_About_Implicit_Call	Message is about implicit (internal) call introduced by the compiler.

continues on next page

Table 1 – continued from previous page

Attribute Name	Description
Message_Is_Uncertain	The message reflects some uncertainty. Alias for: Message_Depends_On_Unknown_Call or Exception_Handler_Present or Uninteresting_RHS or Message_About_Implicit_Call
Message_Depends_On_Object_Merging	Message depends on approximating more objects with one abstract object. This is the case when, e.g., array others or wildcard dereferences are involved.
Message_Involves_Loop_Imprecision	The message depends on a variable modified in a loop for which Inspector was not able to compute a precise range.
Message_Likely_False_Positive	Approximations were involved when computing information relevant for the message meaning that the message is more likely false positive. Is implied by: Message_Is_Uncertain, Message_Depends_On_Object_Merging, and Message_Involves_Loop_Imprecision

String specifiers

For String specifiers, the chosen attribute name will influence the meaning of the regular expression that is given with the second attribute. The following table gives an overview over the accepted attribute names.

Attribute Name	Description
Text	Regexp applies to text of the message. See below for details.
Subp	Regexp applies to subprogram name
File	Regexp applies to file name
Directory	Regexp applies to directory name
Likelihood	Regexp applies to likelihood of the failure, allowed values are: CHECK_WILL_FAIL, CHECK_MIGHT_FAIL, CHECK_CANNOT_FAIL
Check_Kind	Regexp applies to the kind of check that is described by the message. See below for a list of possible checks. Note that you can use a regular expression to match several kinds of checks at once. The regexp is not case sensitive for the attribute Check_Kind.
Original_Check_Kind	This is only valid for Precondition_Annotations and Precondition_Checks. Regexp applies to the kind of checks that caused the corresponding precondition.

The regexp above are those used in file names by the Unix shell or DOS prompt:

- * matches any string of 0 or more characters
- ? matches any character
- [char char ...] matches any character listed
- [char - char] matches any character in given range
- {elmt, elmt, ...} is an alternation (any of elmt)

A typical Inspector message like

```
div.adb:7:23: high: divide by zero (Inspector): check fails here; requires I >= 1
```

starts with the location of the message, followed by its rank, then its kind, and ends with some text with more details. In the case of a regexp with the attribute `Text`, the regexp applies to the rightmost part of the message which we call text in the following. Depending on the kind of message the regexp might not apply to the whole text. If the text start

with `requires`, like above, the regexp only applies to the text after the `requires`, here `I >= 1`. In some cases the kind is not singled out with a colon, as in

```
main.adb:16:31: medium warning: dead code (Inspector): dead code because (is_equal
↪'Result) = true
test.adb:8:7: medium warning: unused assignment (Inspector): unused assignment into A
```

In this case the regexp is only applied to the part starting after the kind of the message, here `because (is_equal'Result) = true` and `into A`. To get the exact text against which your regexp is applied you can use the debug switch `--dbg-on patterns`, see below for more information.

The list of Inspector check kind ids in `MessagePatterns.xml` corresponds mostly to GNAT SAS messages as described in *GNAT SAS Messages Reference*:

Check Kind	Description
NON_ANALYZED_CALL_WARNING	call not analyzed (due to e.g. tool limitation or code too complex)
SUSPICIOUS_PRECONDITION_WARNING	suspicious precondition
SUSPICIOUS_INPUT_WARNING	suspicious input
SUSPICIOUS_CONSTANT_OPERATION_WARNING	suspicious constant operation
UNREAD_IN_OUT_PARAMETER_WARNING	suspicious in out: could have out mode
UNASSIGNED_IN_OUT_PARAMETER_WARNING	suspicious in out: could have in mode
UNKNOWN_CALL_WARNING	unknown call (due to e.g. partitioning)
DEAD_STORE_WARNING	unused assignment
DEAD_OUTPARAM_STORE_WARNING	unused out parameter
POTENTIALLY_DEAD_STORE_WARNING	unused assignment to global
SAME_VALUE_DEAD_STORE_WARNING	useless reassignment
DEAD_BLOCK_WARNING	dead code
INFINITE_LOOP_WARNING	loop does not complete normally
PLAIN_DEAD_EDGE_WARNING	test predetermined
TRUE_DEAD_EDGE_WARNING	test always true
FALSE_DEAD_EDGE_WARNING	test always false
TRUE_CONDITION_DEAD_EDGE_WARNING	condition predetermined to true
FALSE_CONDITION_DEAD_EDGE_WARNING	condition predetermined to false
DEAD_BLOCK_CONTINUATION_WARNING	continuation of dead code
ANALYZED_MODULE_WARNING	module included in analysis
NON_ANALYZED_MODULE_WARNING	module not included, due to Inspector limitations
NON_ANALYZED_PROCEDURE_WARNING	subprogram not included, due to Inspector limitations
PROCEDURE_DOES_NOT_RETURN_ERROR	subprogram never returns
CHECK_FAILS_ON_EVERY_CALL_ERROR	subprogram always fails
UNLOCKED_REENTRANT_UPDATE_ERROR	unprotected access
UNLOCKED_SHARED_DAEMON_UPDATE_ERROR	unprotected shared access
MISMATCHED_LOCKED_UPDATE_ERROR	mismatched protected access
PRECONDITION_CHECK	precondition
INVALID_OR_NULL_CHECK	access check
DIVIDE_BY_ZERO_CHECK	divide by zero
ARRAY_INDEXING_CHECK	array index check
NUMERIC_OVERFLOW_CHECK	overflow check
USER_ASSIGN_STM_CHECK	overflow check
PRE_ASSIGN_STM_CHECK	overflow check (before a call)
POST_ASSIGN_STM_CHECK	overflow check (after a call)
NUMERIC_RANGE_CHECK	range check
TAG_CHECK	tag check
TYPE_VARIANT_CHECK	discriminant check

continues on next page

Table 3 – continued from previous page

Check Kind	Description
ALIASING_CHECK	parameter aliasing
RAISE_CHECK	raise exception
CONDITIONAL_RAISE_CHECK	conditional check
ASSERTION_CHECK	assertion
POSTCONDITION_CHECK	postcondition
INVALID_CHECK	validity check

Note that Inspector issues a warning when a check kind id (or regular expression) does not match any check.

In order to help you come up with correct rules you can use the Inspector debug switch `--dbg-on patterns`. This switch will make Inspector print all the messages in the following format:

```
Finding Rule for DEAD_STORE_WARNING: test.adb: test.p: into A
```

Where the first part is the check kind in the format to be used in a rule, the second part is the file, the third part the procedure containing the message, and the last part the text against which your regexp is matched. The above message is usually printed as:

```
test.adb:8:7: medium warning: unused assignment (Inspector): unused assignment into A
```

After each message, the rules that match the corresponding message (if any) are printed, precising the resulting ranking, e.g. :

```
Finding Rule for DEAD_STORE_WARNING: test.adb: test.p: into A
Rule # 57 matches.
  (first match)
winning probability: MEDIUM
```

Here you see that rule 57 is the first and only rule that matches the message, and that the resulting ranking is MEDIUM.

Computed specifiers

For computed specifiers, the allowed set of attribute names is the following:

Attribute Name	Description
Is_Check	The message concerns a check. (see *GNAT SAS Messages Reference*)
Is_Assign_Stm_Check	The message concerns a check for an assignment.
Is_Assign_Stm_Or_Precond_Check	The message concerns a check for an assignment or a precondition.
Source_Is_Ada	The source for that message is Ada (always true currently).
Message_Is_New	The message is new wrt. the baseline run.
Message_Is_Dropped	The message is in the baseline run, but not this run.
Message_Is_Unchanged	The message is unchanged wrt. the baseline run.
Max_Messages	Switch --messages was given with argument *max*.
Normal_Messages	Switch --messages was given with argument *normal*.
Min_Messages	Switch --messages was given with argument *min*.

Example

The following simple example shows a rule which will suppress all messages from the *validity check* category. Note that a simpler way to achieve the same effect would be to use `--be-messages=-validity_check`.

```
<Message_Probability_Rules>
  <Message_Rule Matching_Probability="SUPPRESSED">
    <Message_Pattern>
      <String_Specifier
        Name_Of_String_Attribute="Check_Kind"
        String_To_Match="INVALID_CHECK">
      </String_Specifier>
    </Message_Pattern>
  </Message_Rule>
</Message_Probability_Rules>
```

The following more complex example shows a rule which will flag as *HIGH* all *check* messages which are *not* soft, will always fail, and which are *not* user raise statements:

```
<Message_Probability_Rules>
 <Message_Rule Matching_Probability="HIGH">
   <Message_Pattern>
     <Computed_Specifier Name_Of_Computed_Attribute="Is_Check">
     </Computed_Specifier>

     <Bool_Specifier
       Complemented="YES"
       Name_Of_Boolean_Attribute="CHECK_IS_SOFT">
     </Bool_Specifier>

     <String_Specifier
       Name_Of_String_Attribute="Likelihood"
       String_To_Match="CHECK_WILL_FAIL">
     </String_Specifier>

     <String_Specifier
       Complemented="YES"
       Name_Of_String_Attribute="Check_Kind"
       String_To_Match="RAISE_CHECK">
     </String_Specifier>

   </Message_Pattern>
 </Message_Rule>
</Message_Probability_Rules>
```

14.2 Inspector By Example

GNAT SAS' Inspector engine is very different from most static analyzers, which execute symbolically the program until a safe approximation is reached. Instead, Inspector works bottom-up from callees to callers, generating a contract for each subprogram along the way. Contracts are the basis for generating error messages related to possible run-time errors and logic errors, and they can be inspected during code reviews to detect further errors. Thus, it is essential to understand how Inspector generates contracts, and how it uses contracts to detect errors. This section aims at providing a deeper insight into how GNAT SAS' Inspector engine works, through a step-by-step exploration of simple code examples. The code for these examples is available in the distribution of GNAT SAS, under `<gnatsas install>/share/examples/gnat_sas/inspector_by_example`, and in GNAT Studio through menu *Help* → *GNATSAS* → *Examples* → *Inspector by Example*.

14.2.1 Basic Examples

This section presents the results of running GNAT SAS' Inspector on simple subprograms composed of assignments, branchings and calls.

Scalar Assignment

Assign

Inspector is able to follow very precisely the assignments to scalar variables throughout the program. Take a very simple program `Assign` that assigns the value `Y+1` to the variable `X`:

```
1 procedure Assign (X : out Integer; Y : in Integer) is
2 begin
3    X := Y + 1;
4 end Assign;
```

On this subprogram, Inspector generates the following contract:

```
1 assign.adb:1: (pre)- assign:(overflow check) Y /= 2_147_483_647
2 assign.adb:1: (post)- assign:X /= -2_147_483_648
3 assign.adb:1: (post)- assign:X = Y + 1
```

The precondition on line 1 requires that the value of `Y` passed in argument to `Assign` be less than the maximal integer value. This is good advice indeed, because otherwise the addition on line 3 of `Assign` overflows.

The postcondition on line 3 indicates that the value of `X` after the call is equal to `Y+1`. The postcondition on line 2 indicates that this value is different from (hence greater than) the minimal integer value.

On this first example, Inspector is able to generate the most precise contract for subprogram `Assign`. On more complex subprograms, Inspector will generate a comprehensive and safe approximation.

Bad_Assign

Consider a version of `Assign` where `X` is read inside the subprogram before it is assigned a value:

```
1 procedure Bad_Assign (X : out Integer; Y : in Integer) is
2 begin
3    X := X + 1;
4 end Bad_Assign;
```

Inspector detects that `X` is not initialized when it is read on line 3, and thus it generates an error:

```
bad_assign.adb:3:9: high: validity check [CWE 457] (Inspector): X is uninitialized here
```

Assign_To_Pos

When constrained types are used, for example the type `Positive` of positive integers in Ada, then Inspector uses this information in the contracts it generates. Take a variation of `Assign` where parameter `X` is a positive integer:

```
1 procedure Assign_To_Pos (X : out Positive; Y : in Integer) is
2 begin
3    X := Y + 1;
4 end Assign_To_Pos;
```

On this subprogram, Inspector generates the following contract:

```
1 assign_to_pos.adb:1: (pre)- assign_to_pos:(range check) Y in 0..2_147_483_646
2 assign_to_pos.adb:1: (post)- assign_to_pos:X = Y + 1
3 assign_to_pos.adb:1: (post)- assign_to_pos:X'Initialized
```

Note how the range of `Y` in the precondition on line 1 has been restricted to positive integers less than the maximal integer value. Note also that the postcondition on line 3 only indicates now that `X` is initialized on exit, which is the most precise information here given that the type of `X` already excludes all negative values. The attribute `'Initialized` is not an actual Ada attribute. It is used to denote that a location has been initialized. This is different from the Ada attribute `'Valid` which only denotes that a location has a valid bit representation. For example, a variable of a machine integer type is always valid, even if it is not initialized.

Bad_Assign_To_Pos

Consider a version of `Assign_To_Pos` where the assignment to `X` always results in a constraint error, whatever the precondition:

```
1 procedure Bad_Assign_To_Pos (X : out Positive; Y : in Positive) is
2 begin
3    X := -Y + 1;
4 end Bad_Assign_To_Pos;
```

Inspector detects that no suitable precondition can be generated for this subprogram, and it generates errors:

```
1 bad_assign_to_pos.adb:1:1: high warning: subp always fails (Inspector): bad_assign_to_
  ↪pos fails for all possible inputs
2 bad_assign_to_pos.adb:3:12: high: range check [CWE 682] (Inspector): check fails here
```

Self_Assign

As shown in the examples above, the precondition refers to values of variables before a call, while the postcondition refers to values of variables after a call. In general, though, postconditions may relate values on entry to a subprogram and values on exit to the same subprogram. Consider the code of `Self_Assign` which increments the value of its parameter `X`:

```
1 procedure Self_Assign (X : in out Integer) is
2 begin
3    X := X + 1;
4 end Self_Assign;
```

On this subprogram, Inspector generates the following contract:

```
1 self_assign.adb:1: (pre)- self_assign:(overflow check) X /= 2_147_483_647
2 self_assign.adb:1: (post)- self_assign:X /= -2_147_483_648
3 self_assign.adb:1: (post)- self_assign:X = X'Old + 1
```

The precondition on line 1 indeed refers to the value of `X` before the call, and the postcondition on line 2 refers to the value of `X` after the call. The interesting case is the postcondition on line 3, which relates the value of `X` before the call, denoted `X'Old`, and the value of `X` after the call.

Bad_Self_Assign

Consider a version of `Self_Assign` where `X` is assigned its current value previously copied:

```
1 procedure Bad_Self_Assign (X : in out Integer) is
2    Y : Integer := X - 1;
3 begin
4    X := Y + 1;
5 end Bad_Self_Assign;
```

When `--mode=deep` is given, Inspector detects that the assignment is useless, and it generates a warning:

```
1 bad_self_assign.adb:4:6: medium warning: useless reassignment [CWE 563] (Inspector):␣
  ↪useless reassignment of X
```

Aggregate Assignment

Assign_Rec

Inspector is able to follow assignments to variables of composite type as precisely as assignments to scalar variables. Take for example a record type `Rec` aggregating an integer and a `Pair`, which is itself aggregating two integers:

```
1 package Assign_Rec is
2    type Pair is record
3       A, B : Integer;
4    end record;
5    type Rec is record
6       C : Integer;
7       D : Pair;
8    end record;
```

(continues on next page)

(continued from previous page)

```
9     procedure Assign (X : out Rec; Y : in Integer);
10 end Assign_Rec;
```

The subprogram `Assign_Rec.Assign` updates some components of its record parameter `X`:

```
1  package body Assign_Rec is
2     procedure Assign (X : out Rec; Y : in Integer) is
3     begin
4        X.C := Y + 1;
5        X.D.B := Y - 1;
6     end Assign;
7  end Assign_Rec;
```

On this subprogram, Inspector generates the following contract:

```
1  assign_rec.adb:2: (pre)- assign_rec.assign:(overflow check) Y in -2_147_483_647..2_147_
   ↪483_646
2  assign_rec.adb:2: (post)- assign_rec.assign:X.C = Y + 1
3  assign_rec.adb:2: (post)- assign_rec.assign:X.C >= -2_147_483_646
4  assign_rec.adb:2: (post)- assign_rec.assign:X.D.B <= 2_147_483_645
5  assign_rec.adb:2: (post)- assign_rec.assign:X.D.B = Y - 1
```

The postconditions on lines 2-5 correspond exactly to the most precise postconditions on this subprogram, which one would also get with assignments to scalar variables.

Bad_Assign_Rec

Consider a version of `Assign_Rec` where `Y` is multiplied and divided by one million, instead of being incremented and decremented by one:

```
1  with Assign_Rec; use Assign_Rec;
2
3  procedure Bad_Assign_Rec (X : out Rec; Y : in Integer) is
4  begin
5     X.C := Y * 1_000_000;
6     X.D.B := Y / 1_000_000;
7  end Bad_Assign_Rec;
```

Inspector detects that the expression `Y / 1_000_000` on line 5 always evaluates to zero, and it issues a warning:

```
1  bad_assign_rec.adb:5:15: medium warning: suspicious constant operation (Inspector):␣
   ↪operation Y/1_000_000 always evaluates to 0
```

Indeed, since `Y` is multiplied by one million, its absolute value should be less than 2147, as indicated by the contract generated by Inspector:

```
1  bad_assign_rec.adb:2: (pre)- bad_assign_rec:(overflow check) Y in -2_147..2_147
2  bad_assign_rec.adb:2: (post)- bad_assign_rec:X.C = Y*1_000_000
3  bad_assign_rec.adb:2: (post)- bad_assign_rec:X.C in -2_147_000_000..2_147_000_000
4  bad_assign_rec.adb:2: (post)- bad_assign_rec:X.D.B = 0
```

Then, dividing `Y` by one million always gives the value zero.

Assign_Arr

Similarly, take an array type `Arr` aggregating ten integers:

```
1  package Assign_Arr is
2     type Arr is array (1 .. 10) of Integer;
3     procedure Assign (X : out Arr; Y : in Integer);
4  end Assign_Arr;
```

The subprogram `Assign_Arr.Assign` updates some elements of its array parameter `X`:

```
1  package body Assign_Arr is
2     procedure Assign (X : out Arr; Y : in Integer) is
3     begin
4        X (1) := Y + 1;
5        X (4) := Y - 1;
6     end Assign;
7  end Assign_Arr;
```

On this subprogram, Inspector generates the following contract:

```
1  assign_arr.adb:2: (pre)- assign_arr.assign:(overflow check) Y in -2_147_483_647..2_147_
   ↪483_646
2  assign_arr.adb:2: (post)- assign_arr.assign:X(1 | 4)'Initialized
3  assign_arr.adb:2: (post)- assign_arr.assign:X(1) = Y + 1
4  assign_arr.adb:2: (post)- assign_arr.assign:X(4) = Y - 1
```

The postconditions on lines 2-4 correspond exactly to the most precise postconditions already shown for `Assign_Rec.Assign`.

Bad_Assign_Arr

Consider a version of `Assign_Arr` where `X` is assigned at indexes `Y` and `Y + 10`:

```
1  with Assign_Arr; use Assign_Arr;
2
3  procedure Bad_Assign_Arr (X : out Arr; Y : in Integer) is
4  begin
5     X (Y) := 1;
6     X (Y + 10) := -1;
7  end Bad_Assign_Arr;
```

Inspector detects that the expression `Y + 10` on line 5 is outside of the bounds of `X`, which ranges from 1 to 10, and it issues an error:

```
1  bad_assign_arr.adb:2:1: high warning: subp always fails (Inspector): bad_assign_arr␣
   ↪fails for all possible inputs
2  bad_assign_arr.adb:4:4: medium: array index check [CWE 120] (Inspector): requires Y in 1.
   ↪.10
3  bad_assign_arr.adb:5:4: high: array index check [CWE 120] (Inspector): check fails here;␣
   ↪requires Y in -9..0
```

Indeed, since index `Y` is used to access `X`, its value should be between 1 and 10. Then, `Y + 10` is greater than the upper bound of `X`.

Assign_Arr_Unk

Finally, take an array type `Arr` aggregating an unknown number of integers:

```
1 package Assign_Arr_Unk is
2    type Arr is array (Integer range <>) of Integer;
3    procedure Assign (X : out Arr; Y : in Integer);
4 end Assign_Arr_Unk;
```

The subprogram `Assign_Arr_Unk.Assign` updates some elements of its array parameter `X`:

```
1 package body Assign_Arr_Unk is
2    procedure Assign (X : out Arr; Y : in Integer) is
3    begin
4       X (1) := Y + 1;
5       X (4) := Y - 1;
6    end Assign;
7 end Assign_Arr_Unk;
```

On this subprogram, Inspector generates the following contract:

```
1 assign_arr_unk.adb:2: (pre)- assign_arr_unk.assign:(array index check) X'First <= 1
2 assign_arr_unk.adb:2: (pre)- assign_arr_unk.assign:(array index check) X'Last >= 4
3 assign_arr_unk.adb:2: (pre)- assign_arr_unk.assign:(overflow check) Y in -2_147_483_647..
  ↪2_147_483_646
4 assign_arr_unk.adb:2: (post)- assign_arr_unk.assign:X(1 | 4)'Initialized
5 assign_arr_unk.adb:2: (post)- assign_arr_unk.assign:X(1) = Y + 1
6 assign_arr_unk.adb:2: (post)- assign_arr_unk.assign:X(4) = Y - 1
```

The postconditions on lines 4-6 correspond exactly to the most precise postconditions already shown.

Branching

Cond_Assign

A common limitation of static analyzers is that they cannot follow precisely disjunctions that occur due to branches in the program. Inspector uses advanced techniques to overcome this difficulty and actually follow these branches. Take a simple subprogram `Cond_Assign` that assigns a different value to parameter `X` depending on the value of another parameter `B`:

```
1 procedure Cond_Assign (X : out Integer; Y : in Integer; B : in Boolean) is
2 begin
3    if B then
4       X := Y + 1;
5    else
6       X := Y - 1;
7    end if;
8 end Cond_Assign;
```

On this subprogram, Inspector generates the following contract:

```
1 cond_assign.adb:1: (pre)- cond_assign:(overflow check) B or Y /= -2_147_483_648
2 cond_assign.adb:1: (pre)- cond_assign:(overflow check) not B or Y /= 2_147_483_647
```

(continues on next page)

(continued from previous page)

```
cond_assign.adb:1: (post)- cond_assign:X = One-of{Y + 1, Y - 1}
cond_assign.adb:1: (post)- cond_assign:X'Initialized
```

The preconditions on line 1 and 2 precisely represent the conditions that input parameters should respect to avoid an integer negative overflow (on line 6 of `Cond_Assign`) or positive overflow (on line 4 of `Cond_Assign`). Note how these preconditions are expressed as disjunctions. This is because the negative or positive overflow is only possible for a particular value of `B` (which differs between the negative overflow and the positive overflow).

The postcondition on line 3 indicates that the value of `X` on exit either comes from the assignment on line 4, or the one on line 6. This is not the most precise postcondition, as one could state that, on exit:

```
(B = True and X = Y + 1) or (B = False and X = Y - 1)
```

As seen with the `Call_Assign` example below, Inspector actually generates internally this more precise postcondition, from which it outputs on line 3 a more user-friendly postcondition. The internal (more precise) postcondition is used for analyzing callers of `Cond_Assign`.

Bad_Cond_Assign

Consider a version of `Cond_Assign` with a modified test:

```
1 procedure Bad_Cond_Assign (X : out Integer; Y : in Integer; B : in Boolean) is
2 begin
3    if B or not B then
4       X := Y + 1;
5    else
6       X := Y - 1;
7    end if;
8 end Bad_Cond_Assign;
```

Inspector detects that the test on line 3 is always true and it issues a warning:

```
bad_cond_assign.adb:3:9: medium warning: test always true [CWE 571] (Inspector): test␣
↪always true because B or not B
bad_cond_assign.adb:6:9: medium warning: dead code [CWE 561] (Inspector): dead code␣
↪because B or not B
```

Multi_Cond_Assign

The precise analysis of branches seen on `Cond_Assign` is performed by Inspector on any number of if statements, as shown on `Multi_Cond_Assign`:

```
1 procedure Multi_Cond_Assign
2   (X : out Integer; Y : in Integer; B1, B2 : in Boolean) is
3 begin
4    if B1 then
5       X := Y + 1;
6    else
7       if B2 then
8          X := Y - 1;
9       else
```

(continues on next page)

(continued from previous page)

```
10          X := Y * 2;
11       end if;
12    end if;
13 end Multi_Cond_Assign;
```

On this subprogram, Inspector generates the following contract:

```
1 multi_cond_assign.adb:1: (pre)- multi_cond_assign:(overflow check) B1 or B2 or Y*2 in -2_
  ↪147_483_648..2_147_483_647
2 multi_cond_assign.adb:1: (pre)- multi_cond_assign:(overflow check) B1 or not B2 or Y /= -
  ↪2_147_483_648
3 multi_cond_assign.adb:1: (pre)- multi_cond_assign:(overflow check) not B1 or Y /= 2_147_
  ↪483_647
4 multi_cond_assign.adb:1: (post)- multi_cond_assign:X = One-of{Y + 1, Y - 1, Y*2}
5 multi_cond_assign.adb:1: (post)- multi_cond_assign:X'Initialized
```

The postcondition on line 4 is similar to the one obtained for `Cond_Assign`, except here it deals with two if statements. Our comment about Inspector generating internally a contract more precise than the one displayed for `Cond_Assign` is also applicable here.

Bad_Multi_Cond_Assign

Consider a version of `Multi_Cond_Assign` with modified tests:

```
1  procedure Bad_Multi_Cond_Assign
2    (X : out Integer; Y : in Integer; B1, B2 : in Boolean) is
3  begin
4     if B1 or Y < 10 then
5        X := Y + 1;
6     else
7        if B2 and not (Y > 0) then
8           X := Y - 1;
9        else
10          X := Y * 2;
11       end if;
12    end if;
13 end Bad_Multi_Cond_Assign;
```

Inspector detects that the test on line 6 is always false and it issues a warning:

```
1 bad_multi_cond_assign.adb:6:13: low warning: test always false [CWE 570] (Inspector):␣
  ↪test always false because not B2 or Y >= 1
2 bad_multi_cond_assign.adb:7:12: medium warning: dead code [CWE 561] (Inspector): dead␣
  ↪code because not B2 or Y >= 1
```

Case_Assign

The analysis of case statements is similar to the analysis of if statements. Consider the subprogram `Case_Assign.Assign` which takes an operation `Op` in parameter, among three possibilities:

```
1  package Case_Assign is
2     type Oper is (Incr, Decr, Double);
3     procedure Assign (X : out Integer; Y : in Integer; Op : in Oper);
4  end Case_Assign;
```

It does the obvious assignment in each case:

```
1  package body Case_Assign is
2     procedure Assign (X : out Integer; Y : in Integer; Op : in Oper) is
3     begin
4        case Op is
5           when Incr   =>
6              X := Y + 1;
7
8           when Decr   =>
9              X := Y - 1;
10
11          when Double =>
12             X := Y * 2;
13       end case;
14    end Assign;
15 end Case_Assign;
```

On this subprogram, Inspector generates the following contract:

```
1  case_assign.adb:2: (pre)- case_assign.assign:(overflow check) Op /= Decr or Y /= -2_147_
   ↪483_648
2  case_assign.adb:2: (pre)- case_assign.assign:(overflow check) Op /= Double or Y*2 in -2_
   ↪147_483_648..2_147_483_647
3  case_assign.adb:2: (pre)- case_assign.assign:(overflow check) Op /= Incr or Y /= 2_147_
   ↪483_647
4  case_assign.adb:2: (post)- case_assign.assign:X = One-of{Y + 1, Y - 1, Y*2}
5  case_assign.adb:2: (post)- case_assign.assign:X'Initialized
```

The postcondition on line 4 is the same as what is obtained for if statements in `Multi_Cond_Assign`. The same comment about the internally more precise contract still applies.

Call Statement

Call_Assign

The modular treatment of calls is a cornerstone of Inspector analysis. Before a caller of subprogram S is analyzed, the body of S is analyzed and Inspector generates a contract for S. Then, this contract is used to completely abstract the calls to S during the analysis of its callers. In fast mode, Inspector uses precise contracts for all the calls in the same library unit and treats other as unknown. In deep mode, Inspector uses precise contracts for all the calls in the same partition - see *Partitioning of Analysis* for more details. Direct and mutually recursive calls require that a coarser contract is used for bootstrapping. This coarser contract is then iteratively refined. Consider the subprogram `Call_Assign.Call`:

```
package Call_Assign is
   type Choice is (Simple, Conditional, Self);
   procedure Call (X : in out Integer; C : in Choice);
end Call_Assign;
```

It calls one of `Assign`, `Cond_Assign` or `Self_Assign` seen previously, depending on the value of its parameter `C`:

```
with Assign, Cond_Assign, Self_Assign;

package body Call_Assign is
   procedure Call (X : in out Integer; C : in Choice) is
   begin
      case C is
         when Simple      =>
            Assign (X, 1);

         when Conditional =>
            Cond_Assign (X, 2, True);

         when Self        =>
            Self_Assign (X);
      end case;
   end Call;
end Call_Assign;
```

On this subprogram, when run with `--mode=deep`, Inspector generates the following contract:

```
call_assign.adb:3: (pre)- call_assign.call:(overflow check) C /= Self or X /= 2_147_483_
↪647
call_assign.adb:3: (post)- call_assign.call:X /= -2_147_483_648
call_assign.adb:3: (post)- call_assign.call:X = One-of{2, 3, X'Old + 1}
```

The precondition on line 1 only constrains the input value of `X` when `C` is equal to `Self`. Indeed, the generated preconditions for *Assign* and *Cond_Assign* are always satisfied by the values passed in parameters in `Call_Assign`, while the generated precondition for *Self_Assign* imposes that `X` is less than the maximal integer value.

The postcondition on line 3 enumerates the possible values for `X` on exit. The value `2` corresponds to the call to `Assign`. The value `3` corresponds to the call to `Cond_Assign`. Note here that Inspector used the more precise internal contract for `Cond_Assign` instead of the one displayed, which would have lead to an additional possible value of `1` for `X`. The value `X'Old + 1` corresponds to the call to `Self_Assign`.

The postcondition on line 2 is a common constraint on `X` on exit, whatever the procedure called.

Bad_Call_Assign

Consider a version of `Call_Assign` in which subprograms `Assign`, `Cond_Assign` and `Self_Assign` are all called in contexts that violate their generated preconditions:

```
with Call_Assign; use Call_Assign;
with Assign, Cond_Assign, Self_Assign;

procedure Bad_Call_Assign (X : in out Integer; C : in Choice) is
begin
   case C is
```

(continues on next page)

(continued from previous page)

```
      when Simple      =>
         Assign (X, Integer'Last);

      when Conditional =>
         Cond_Assign (X, Integer'Last, True);

      when Self        =>
         X := Integer'Last;
         Self_Assign (X);
   end case;
end Bad_Call_Assign;
```

Inspector detects these violations and it issues corresponding precondition errors:

```
bad_call_assign.adb:3:1: medium warning: unread parameter (Inspector): parameter X could␣
↪have mode out
```

Reverse_Call_Assign

You might be wondering whether Inspector computed a more precise internal postcondition for the postcondition of *Call_Assign* displayed on line 3. This is indeed the case, as shown in the following example. Consider three subprograms for each value of the `Choice` enumeration:

```
package Reverse_Call_Assign is
   procedure Call_Simple (X : in out Integer);
   procedure Call_Conditional (X : in out Integer);
   procedure Call_Self (X : in out Integer);
end Reverse_Call_Assign;
```

Each subprogram calls `Call_Assign.Call` with the corresponding value for parameter `C`:

```
with Call_Assign; use Call_Assign;

package body Reverse_Call_Assign is
   procedure Call_Simple (X : in out Integer) is
   begin
      Call (X, Simple);
   end Call_Simple;
   procedure Call_Conditional (X : in out Integer) is
   begin
      Call (X, Conditional);
   end Call_Conditional;
   procedure Call_Self (X : in out Integer) is
   begin
      Call (X, Self);
   end Call_Self;
end Reverse_Call_Assign;
```

On these subprograms, when run with `--mode=deep`, Inspector generates the following contracts:

```
reverse_call_assign.adb:3: (post)- reverse_call_assign.call_simple:X = 2
reverse_call_assign.adb:7: (post)- reverse_call_assign.call_conditional:X = 3
```

(continues on next page)

(continued from previous page)

```
reverse_call_assign.adb:11: (pre)- reverse_call_assign.call_self:(overflow check) X /= 2_
↪147_483_647
reverse_call_assign.adb:11: (post)- reverse_call_assign.call_self:X /= -2_147_483_648
reverse_call_assign.adb:11: (post)- reverse_call_assign.call_self:X = X'Old + 1
```

The postcondition on line 1 is indeed the most precise contract on `Call_Simple`, as it is the same as the contract generated for *Assign* with the value of `1` for `Y`.

The postcondition on line 2 is indeed the most precise contract on `Call_Conditional`, as it is the same as the contract generated for *Cond_Assign* with the value of `2` for `Y` and `True` for `B`.

The precondition on line 3 and the postconditions on lines 4 and 5 are indeed the most precise contract on `Call_Self`, as they are the same as the contract generated for *Self_Assign*.

Inspector could only generate these most precise contracts because it did not lose the relation between the input value of parameter `C` and the output value of parameter `X` in `Call_Assign.Call`. The internal most precise postcondition computed by Inspector for `Call_Assign.Call` is really:

```
(C = Simple and X = 2) or (C = Conditional and X = 3) or (C = Self and X = X'Old + 1)
```

Top_Down

Inspector completely works bottom-up on the call-graph, propagating information gathered from callees in their generated contracts to their callers. This strategy ensures that partial applications can be handled easily, and that the analysis is much faster than the classical top-down approach. Although Inspector does not use knowledge of the calling context when analyzing a subprogram, in many cases it generates contracts precise enough to take this context into account.

Consider the local function `Ident` which is always called in a context where the content of array `A` is all ones:

```
with Assign_Arr; use Assign_Arr;

procedure Top_Down (Y : out Integer) is
   A : Arr;
   function Ident (X : Integer) return Integer is
   begin
      if A (X) > 0 then
         return X;
      else
         return 0;
      end if;
   end Ident;
begin
   A := (others => 1);
   Y := A (Ident (3));
end Top_Down;
```

Inspector does not use this information in the analysis of `Ident`, but instead generates a conditional postcondition taking the value of X into account (even though this is not visible in the printable version, internally Inspector knows that `One-of{X, 0}` really means `X` when `A(X) > 0` and 0 otherwise):

```
top_down.adb:2: (post)- top_down:Y = 1
top_down.adb:4: (pre)- top_down.ident:(validity check) A(X)'Initialized
top_down.adb:4: (pre)- top_down.ident:(array index check) X in 1..10
```

(continues on next page)

(continued from previous page)

```
top_down.adb:4: (post)- top_down.ident:top_down.ident'Result = One-of{X, 0}
top_down.adb:4: (post)- top_down.ident:top_down.ident'Result in 0..10
```

Thus, Inspector can then check that the call to `Ident(3)` returns a positive value since at this point it knows that `X(3) > 0`.

14.2.2 Loop Examples

This section presents the results of running GNAT SAS' Inspector on subprograms containing loops.

Induction Variables

Induction

The first difficulty in analyzing loops is that it is not possible in general to completely unroll the loop, either because the number of iterations is very large, or because it is not known statically. Thus, the static analysis must analyze together all iterations beyond a certain point. This is a source of approximations, in particular for induction variables, that is, those variables assigned in the loop. However, Inspector analyzes precisely those induction variables that are incremented or decremented at each run through the loop.

For example, take a simple subprogram incrementing a variable `X` 10 times:

```
procedure Induction (X : out Integer) is
begin
   X := 0;
   for J in 1 .. 10 loop
      X := X + 1;
   end loop;
end Induction;
```

On this subprogram, Inspector generates the following contract:

```
induction.adb:1: (post)- induction:X = 10
```

Inspector manages here to generate the most precise postcondition for the loop.

Bad_Induction

Consider a version of `Induction` in which `X` is not initialized before the loop:

```
procedure Bad_Induction (X : out Integer) is
begin
   for J in 1 .. 10 loop
      X := X + 1;
   end loop;
end Bad_Induction;
```

Inspector detects that `X` might be uninitialized on line 4 and it issues an error:

```
bad_induction.adb:4:12: high: validity check [CWE 457] (Inspector): check fails here;␣
↪requires X'Initialized (iteration 1 of 10)
```

Multi_Induction

In many cases, Inspector can also follow precisely the relations between induction variables in a loop, as in subprogram `Multi_Induction`:

```
1 procedure Multi_Induction (X1, X2 : out Integer; Y : in Integer) is
2 begin
3    X1 := 0;
4    X2 := 0;
5    while X1 < Y loop
6       X1 := X1 + 2;
7       X2 := X2 + 6;
8    end loop;
9 end Multi_Induction;
```

On this subprogram, Inspector generates the following contract:

```
1 multi_induction.adb:1: (pre)- multi_induction:Y <= 715_827_882
2 multi_induction.adb:1: (post)- multi_induction:X1 - Y in 0..2_147_483_648
3 multi_induction.adb:1: (post)- multi_induction:X1 in (0 | 2 | 4 | 6 | ... | 44..715_827_
  ↪882)
4 multi_induction.adb:1: (post)- multi_induction:X2 - Y*3 in -2_147_483_646..6_442_450_944
5 multi_induction.adb:1: (post)- multi_induction:X2 = X1*3
6 multi_induction.adb:1: (post)- multi_induction:X2 in (0 | 6 | 12 | 18 | ... | 132..2_147_
  ↪483_646)
```

Note how the postcondition on line 5 correctly indicates the relation between `X1` and `X2` after the loop.

Bad_Multi_Induction

Consider a version of `Multi_Induction` in which `X1` is not assigned in the loop:

```
1 procedure Bad_Multi_Induction (X1, X2 : out Integer; Y : in Positive) is
2 begin
3    X1 := 0;
4    X2 := 0;
5    while X1 < Y loop
6       X2 := X2 + 6;
7    end loop;
8    return;
9 end Bad_Multi_Induction;
```

Inspector detects that the loop never terminates, because the exit test on line 5 is always true:

```
1 bad_multi_induction.adb:1:1: medium warning: subp never returns (Inspector): bad_multi_
  ↪induction never returns
2 bad_multi_induction.adb:5:13: medium warning: test always true [CWE 571] (Inspector):␣
  ↪test always true because X1 < Y
3 bad_multi_induction.adb:5:13: medium warning: loop does not complete normally [CWE 835]␣
  ↪(Inspector)
4 bad_multi_induction.adb:8:4: medium warning: dead code [CWE 561] (Inspector): dead code␣
  ↪because X1 < Y
```

Array Scanning

The most frequent use of loops is to iterate over a collection, for example an array. Use of arrays introduces a new difficulty, namely that the static analysis must handle a large or unbounded number of memory locations. This requires performing some abstractions, to compute a safe approximation.

Search

Take a simple function `Search` scanning an array for a value. We are reusing here the type `Assign_Arr.Arr` from example *Assign_Arr* of arrays of ten integers.

```
1  with Assign_Arr; use Assign_Arr;
2
3  function Search (X : in Arr; Y : in Integer) return Integer is
4  begin
5     for J in X'Range loop
6        if X (J) = Y then
7           return J;
8        end if;
9     end loop;
10    return 0;
11 end Search;
```

On this subprogram, Inspector generates the following contract:

```
1  search.adb:2: (pre)- search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or X(4) =␣
   ↪Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8) = Y or X(9) = Y or X(10)'Initialized
2  search.adb:2: (pre)- search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or X(4) =␣
   ↪Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8) = Y or X(9)'Initialized
3  search.adb:2: (pre)- search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or X(4) =␣
   ↪Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8)'Initialized
4  search.adb:2: (pre)- search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or X(4) =␣
   ↪Y or X(5) = Y or X(6) = Y or X(7)'Initialized
5  search.adb:2: (pre)- search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or X(4) =␣
   ↪Y or X(5) = Y or X(6)'Initialized
6  search.adb:2: (pre)- search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or X(4) =␣
   ↪Y or X(5)'Initialized
7  search.adb:2: (pre)- search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or X(4)
   ↪'Initialized
8  search.adb:2: (pre)- search:(validity check) X(1) = Y or X(2) = Y or X(3)'Initialized
9  search.adb:2: (pre)- search:(validity check) X(1) = Y or X(2)'Initialized
10 search.adb:2: (pre)- search:(validity check) X(1)'Initialized
11 search.adb:2: (post)- search:search'Result in 0..10
```

The preconditions on lines 1-10 exactly capture in which case the function can be called, regarding the initialization of the array `X`. Indeed, it is safe to call `Assign_arr` with a partially initialized array, provided the value `Y` is found before reaching the uninitialized part.

The postcondition on line 11 is a safe approximation on `Search`. The most precise postcondition would have to state that either the array `X` does not contain `Y`, in which case `Search` returns `0`, or the array `X` contains `Y`, in which case `Search` returns the smallest corresponding index.

Bad_Search

Consider a version of `Search` in which the final return statement is missing:

```
1  with Assign_Arr; use Assign_Arr;
2
3  function Bad_Search (X : in Arr; Y : in Integer) return Integer is
4  begin
5     for J in X'Range loop
6        if X (J) = Y then
7           return J;
8        end if;
9     end loop;
10 end Bad_Search;
```

On this subprogram, Inspector generates the following contract:

```
1  bad_search.adb:2: (pre)- bad_search:(conditional raise) X(1) = Y or X(2) = Y or X(3) = Y␣
   ↪or X(4) = Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8) = Y or X(9) = Y or X(10) = Y
2  bad_search.adb:2: (pre)- bad_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or␣
   ↪X(4) = Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8) = Y or X(9) = Y or X(10)
   ↪'Initialized
3  bad_search.adb:2: (pre)- bad_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or␣
   ↪X(4) = Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8) = Y or X(9)'Initialized
4  bad_search.adb:2: (pre)- bad_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or␣
   ↪X(4) = Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8)'Initialized
5  bad_search.adb:2: (pre)- bad_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or␣
   ↪X(4) = Y or X(5) = Y or X(6) = Y or X(7)'Initialized
6  bad_search.adb:2: (pre)- bad_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or␣
   ↪X(4) = Y or X(5) = Y or X(6)'Initialized
7  bad_search.adb:2: (pre)- bad_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or␣
   ↪X(4) = Y or X(5)'Initialized
8  bad_search.adb:2: (pre)- bad_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y or␣
   ↪X(4)'Initialized
9  bad_search.adb:2: (pre)- bad_search:(validity check) X(1) = Y or X(2) = Y or X(3)
   ↪'Initialized
10 bad_search.adb:2: (pre)- bad_search:(validity check) X(1) = Y or X(2)'Initialized
11 bad_search.adb:2: (pre)- bad_search:(validity check) X(1)'Initialized
12 bad_search.adb:2: (post)- bad_search:bad_search'Result in 1..10
```

Inspector will figure out that in order for the subprogram to exit successfully, the loop should never terminate without a return, otherwise the subprogram would raise `Program_Error` (as mandated by the Ada standard). This is expressed by the precondition at line 1.

Call_Search

Inspector also computes the set of inputs and outputs of `Search`, whether these are reachable from parameters or global variables. These inputs and outputs form a part of the generated contract not shown here. Now consider a caller of `Search`, that calls it twice on the same inputs:

```
1  with Assign_Arr; use Assign_Arr;
2  with Search;
3
4  procedure Call_Search (X : in Arr; Y : in Integer; U, V : out Integer) is
5  begin
6     U := Search (X, Y);
7     V := Search (X, Y);
8  end Call_Search;
```

On this subprogram, when run with `--mode=deep`, Inspector generates the following contract:

```
1  call_search.adb:3: (pre)- call_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y␣
   ↪or X(4) = Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8) = Y or X(9) = Y or X(10)
   ↪'Initialized
2  call_search.adb:3: (pre)- call_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y␣
   ↪or X(4) = Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8) = Y or X(9)'Initialized
3  call_search.adb:3: (pre)- call_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y␣
   ↪or X(4) = Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8)'Initialized
4  call_search.adb:3: (pre)- call_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y␣
   ↪or X(4) = Y or X(5) = Y or X(6) = Y or X(7)'Initialized
5  call_search.adb:3: (pre)- call_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y␣
   ↪or X(4) = Y or X(5) = Y or X(6)'Initialized
6  call_search.adb:3: (pre)- call_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y␣
   ↪or X(4) = Y or X(5)'Initialized
7  call_search.adb:3: (pre)- call_search:(validity check) X(1) = Y or X(2) = Y or X(3) = Y␣
   ↪or X(4)'Initialized
8  call_search.adb:3: (pre)- call_search:(validity check) X(1) = Y or X(2) = Y or X(3)
   ↪'Initialized
9  call_search.adb:3: (pre)- call_search:(validity check) X(1) = Y or X(2)'Initialized
10 call_search.adb:3: (pre)- call_search:(validity check) X(1)'Initialized
11 call_search.adb:3: (post)- call_search:U in 0..10
12 call_search.adb:3: (post)- call_search:V = U
```

Inspector uses the knowledge that the same inputs produce the same outputs to compute the postcondition on line 12.

Bad_Call_Search

Consider a caller of `Search` which does not initialize the last element of parameter `X` before the call:

```
1  with Assign_Arr; use Assign_Arr;
2  with Search;
3
4  function Bad_Call_Search return Integer is
5     X : Arr;
6  begin
7     for J in 1 .. 9 loop
8        X (J) := J;
```

(continues on next page)

(continued from previous page)

```
   end loop;
   return Search (X, 10);
end Bad_Call_Search;
```

When run with `--mode=deep`, Inspector generates an error, as it understands that `X(10)` is not initialized, and that `10` is not among the possible values at indices `1..9`.

```
bad_call_search.adb:3:1: high warning: subp always fails (Inspector): bad_call_search␣
↪fails for all possible inputs
bad_call_search.adb:9:11: high: precondition <validity check> [CWE 457] (Inspector):␣
↪precondition failure on call to search; requires X(1) = Y or X(2) = Y or X(3) = Y or␣
↪X(4) = Y or X(5) = Y or X(6) = Y or X(7) = Y or X(8) = Y or X(9) = Y or X(10) to be␣
↪initialized
```

Search_Unk

Take now a slightly different function `Search_Unk`. It also scans an array for a value, but we are reusing here the type `Assign_Arr_Unk.Arr` from example *Assign_Arr_Unk* of unconstrained arrays, which may contain an unbounded number of integers.

```
with Assign_Arr_Unk; use Assign_Arr_Unk;

function Search_Unk (X : in Arr; Y : in Integer) return Integer is
begin
   for J in X'Range loop
      if X (J) = Y then
         return J;
      end if;
   end loop;
   return 0;
end Search_Unk;
```

On this subprogram, Inspector generates the following contract:

```
search_unk.adb:2: (pre)- search_unk:X'Length = 0 or X(J)'Initialized
search_unk.adb:2: (pre)- search_unk:(validity check) X'Length = 0 or X(X'First)
↪'Initialized
search_unk.adb:2: (post)- search_unk:search_unk'Result'Initialized
```

Inspector generates a precondition that array `X` is completely initialized upon entry to `Search_Unk`. However, this is too strong for the function to execute safely: it would be sufficient to have `X(X'First)` initialized, and equal to *Y*.

Search_While

The same reasoning as above applies to other forms of loops. Suppose we rewrite `Search` using a while loop instead of a for loop:

```
with Assign_Arr_Unk; use Assign_Arr_Unk;

function Search_While (X : in Arr; Y : in Integer) return Integer is
   J : Integer := X'First;
begin
   while J <= X'Last loop
      if X (J) = Y then
         return J;
      end if;
      J := J + 1;
   end loop;
   return 0;
end Search_While;
```

On this subprogram, Inspector generates the following contract:

```
search_while.adb:2: (pre)- search_while:(overflow check) X'Length = 0 or X(J)'Initialized
search_while.adb:2: (pre)- search_while:(validity check) X'Length = 0 or X(X'First)
↪'Initialized
search_while.adb:2: (pre)- search_while:(overflow check) X(X'First) = Y or X'Length = 0␣
↪or X'First /= 2_147_483_647
search_while.adb:2: (post)- search_while:search_while'Result'Initialized
```

Here we have one additional precondition that worries about the possibility of overflow on adding one to J, which is not an issue for "for" loops, but otherwise, we have the same contract as the one for *Search_Unk*.

Search_Loop

Now, we rewrite `Search` using an indefinite loop instead of a for loop:

```
with Assign_Arr_Unk; use Assign_Arr_Unk;

function Search_Loop (X : in Arr; Y : in Integer) return Integer is
   J : Integer := X'First;
begin
   loop
      if J > X'Last then
         return 0;
      end if;
      if X (J) = Y then
         return J;
      end if;
      J := J + 1;
   end loop;
end Search_Loop;
```

On this subprogram, Inspector generates the following contract:

```
search_loop.adb:2: (pre)- search_loop:(overflow check) X'Length = 0 or X(J)'Initialized
search_loop.adb:2: (pre)- search_loop:(validity check) X'Length = 0 or X(X'First)
↪'Initialized
search_loop.adb:2: (pre)- search_loop:(overflow check) X(X'First) = Y or X'Length = 0 or␣
↪X'First /= 2_147_483_647
search_loop.adb:2: (post)- search_loop:search_loop'Result'Initialized
```

Again, this is the same contract as the one for *Search_While*.

Array Increments

Assign_All_Arr

Let's consider now subprograms that iterate over arrays to increment their elements. The procedure `Assign_All_Arr` increments each element of its array parameter `X`:

```
1  with Assign_Arr; use Assign_Arr;
2
3  procedure Assign_All_Arr (X : in out Arr) is
4  begin
5     for J in X'Range loop
6        X (J) := X (J) + 1;
7     end loop;
8  end Assign_All_Arr;
```

On this subprogram, Inspector generates the following contract:

```
assign_all_arr.adb:2: (pre)- assign_all_arr:(overflow check) X(1) /= 2_147_483_647
assign_all_arr.adb:2: (pre)- assign_all_arr:(overflow check) X(10) /= 2_147_483_647
assign_all_arr.adb:2: (pre)- assign_all_arr:(overflow check) X(2) /= 2_147_483_647
assign_all_arr.adb:2: (pre)- assign_all_arr:(overflow check) X(3) /= 2_147_483_647
assign_all_arr.adb:2: (pre)- assign_all_arr:(overflow check) X(4) /= 2_147_483_647
assign_all_arr.adb:2: (pre)- assign_all_arr:(overflow check) X(5) /= 2_147_483_647
assign_all_arr.adb:2: (pre)- assign_all_arr:(overflow check) X(6) /= 2_147_483_647
assign_all_arr.adb:2: (pre)- assign_all_arr:(overflow check) X(7) /= 2_147_483_647
assign_all_arr.adb:2: (pre)- assign_all_arr:(overflow check) X(8) /= 2_147_483_647
assign_all_arr.adb:2: (pre)- assign_all_arr:(overflow check) X(9) /= 2_147_483_647
assign_all_arr.adb:2: (post)- assign_all_arr:X(1) = X(1)'Old + 1
assign_all_arr.adb:2: (post)- assign_all_arr:X(1..10) /= -2_147_483_648
assign_all_arr.adb:2: (post)- assign_all_arr:X(10) = X(10)'Old + 1
assign_all_arr.adb:2: (post)- assign_all_arr:X(2) = X(2)'Old + 1
assign_all_arr.adb:2: (post)- assign_all_arr:X(3) = X(3)'Old + 1
assign_all_arr.adb:2: (post)- assign_all_arr:X(4) = X(4)'Old + 1
assign_all_arr.adb:2: (post)- assign_all_arr:X(5) = X(5)'Old + 1
assign_all_arr.adb:2: (post)- assign_all_arr:X(6) = X(6)'Old + 1
assign_all_arr.adb:2: (post)- assign_all_arr:X(7) = X(7)'Old + 1
assign_all_arr.adb:2: (post)- assign_all_arr:X(8) = X(8)'Old + 1
assign_all_arr.adb:2: (post)- assign_all_arr:X(9) = X(9)'Old + 1
```

The preconditions on lines 1 to 10 ensure that all additions on line 5 of `Assign_All_Arr` will not overflow. These preconditions are really the most precise ones for subprogram `Assign_All_Arr`, guaranteeing that no run-time error can be raised during subprogram execution.

Bad_Assign_All_Arr

Consider a version of `Assign_All_Arr` in which elements are shifted to the left of the array:

```
with Assign_Arr; use Assign_Arr;

procedure Bad_Assign_All_Arr (X : in out Arr) is
begin
   for J in X'Range loop
      X (J) := X (J + 1);
   end loop;
end Bad_Assign_All_Arr;
```

Inspector generates an index error because the last iteration of the loop will access past the upper bound of `X`:

```
bad_assign_all_arr.adb:2:1: high warning: subp always fails (Inspector): bad_assign_all_
↪arr fails for all possible inputs
bad_assign_all_arr.adb:5:16: high: array index check [CWE 120] (Inspector): check fails␣
↪here; requires 10 + 1 <= 10 (iteration 10 of 10)
```

Assign_All_Arr_Incr

Take now a slight variation over example `Assign_All_Arr` that increments each element by a different amount:

```
with Assign_Arr; use Assign_Arr;

procedure Assign_All_Arr_Incr (X : in out Arr) is
begin
   for J in X'Range loop
      X (J) := X (J) + J;
   end loop;
end Assign_All_Arr_Incr;
```

On this subprogram, Inspector generates the following contract:

```
assign_all_arr_incr.adb:2: (pre)- assign_all_arr_incr:(overflow check) X(1) /= 2_147_483_
↪647
assign_all_arr_incr.adb:2: (pre)- assign_all_arr_incr:(overflow check) X(10) <= 2_147_
↪483_637
assign_all_arr_incr.adb:2: (pre)- assign_all_arr_incr:(overflow check) X(2) <= 2_147_483_
↪645
assign_all_arr_incr.adb:2: (pre)- assign_all_arr_incr:(overflow check) X(3) <= 2_147_483_
↪644
assign_all_arr_incr.adb:2: (pre)- assign_all_arr_incr:(overflow check) X(4) <= 2_147_483_
↪643
assign_all_arr_incr.adb:2: (pre)- assign_all_arr_incr:(overflow check) X(5) <= 2_147_483_
↪642
assign_all_arr_incr.adb:2: (pre)- assign_all_arr_incr:(overflow check) X(6) <= 2_147_483_
↪641
assign_all_arr_incr.adb:2: (pre)- assign_all_arr_incr:(overflow check) X(7) <= 2_147_483_
↪640
assign_all_arr_incr.adb:2: (pre)- assign_all_arr_incr:(overflow check) X(8) <= 2_147_483_
↪639
```

(continues on next page)

(continued from previous page)

```
assign_all_arr_incr.adb:2: (pre)- assign_all_arr_incr:(overflow check) X(9) <= 2_147_483_
↪638
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(1) = X(1)'Old + 1
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(1..10) /= -2_147_483_648
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(10) = X(10)'Old + 10
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(2) = X(2)'Old + 2
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(3) = X(3)'Old + 3
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(4) = X(4)'Old + 4
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(5) = X(5)'Old + 5
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(6) = X(6)'Old + 6
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(7) = X(7)'Old + 7
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(8) = X(8)'Old + 8
assign_all_arr_incr.adb:2: (post)- assign_all_arr_incr:X(9) = X(9)'Old + 9
```

Again, the preconditions on lines 1 to 10 ensure that all additions on line 5 of `Assign_All_Arr_Incr` will not overflow. These preconditions are really the most precise ones for subprogram `Assign_All_Arr_Incr`, guaranteeing that no run-time error can be raised during subprogram execution.

Assign_All_Arr_Incr_Unk

Take yet another variation over the previous example that uses arrays of unbounded size:

```
with Assign_Arr_Unk; use Assign_Arr_Unk;

procedure Assign_All_Arr_Incr_Unk (X : in out Arr) is
begin
   for J in X'Range loop
      X (J) := X (J) + J;
   end loop;
end Assign_All_Arr_Incr_Unk;
```

On this subprogram, Inspector generates the following contract:

```
assign_all_arr_incr_unk.adb:2: (pre)- assign_all_arr_incr_unk:(overflow check) X'Length␣
↪= 0 or X(J)'Initialized
assign_all_arr_incr_unk.adb:2: (pre)- assign_all_arr_incr_unk:(overflow check) X'Length␣
↪= 0 or X(X'First)'Initialized
assign_all_arr_incr_unk.adb:2: (pre)- assign_all_arr_incr_unk:(overflow check) X'Length␣
↪= 0 or X(X'Last)'Initialized
assign_all_arr_incr_unk.adb:2: (post)- assign_all_arr_incr_unk:X(..) = One-of{X(J)'Old,␣
↪(for _I in X'Range => _I + X(_I))}
```

Here, Inspector generates no precondition to guarantee that the addition on line 5 of `Assign_All_Arr_Incr_Unk` cannot overflow. However, Inspector does not issue any error on this subprogram. This is because these errors are suppressed by default, because they are likely false alarms.

Bad_Assign_All_Arr_Unk

Now take the subprogram just seen, and use an unbounded array instead:

```
with Assign_Arr_Unk; use Assign_Arr_Unk;

procedure Bad_Assign_All_Arr_Unk (X : in out Arr) is
begin
   for J in X'Range loop
      X (J) := X (J + 1);
   end loop;
end Bad_Assign_All_Arr_Unk;
```

Inspector can still detect that the last iteration accesses the array at index `X'Last + 1`, because it treats array bounds symbolically:

```
bad_assign_all_arr_unk.adb:5:16: high: array index check [CWE 120] (Inspector): check␣
↪fails here; requires X'First <= J + 1 and J + 1 <= X'Last
```

Array Treatments

Let's consider now more general array treatments, in which subprograms iterate over arrays to update their content.

Map

Take the subprogram `Map` which stores in array `Z` an incremented version of array `X`:

```
with Assign_Arr; use Assign_Arr;

procedure Map (X : in Arr; Y : in Integer; Z : out Arr) is
begin
   for J in X'Range loop
      Z (J) := X (J) + Y;
   end loop;
end Map;
```

On this subprogram, Inspector generates the following contract:

```
map.adb:2: (pre)- map:(validity check) X(1)'Initialized
map.adb:2: (pre)- map:(validity check) X(10)'Initialized
map.adb:2: (pre)- map:(validity check) X(2)'Initialized
map.adb:2: (pre)- map:(validity check) X(3)'Initialized
map.adb:2: (pre)- map:(validity check) X(4)'Initialized
map.adb:2: (pre)- map:(validity check) X(5)'Initialized
map.adb:2: (pre)- map:(validity check) X(6)'Initialized
map.adb:2: (pre)- map:(validity check) X(7)'Initialized
map.adb:2: (pre)- map:(validity check) X(8)'Initialized
map.adb:2: (pre)- map:(validity check) X(9)'Initialized
map.adb:2: (pre)- map:(overflow check) Y + X(1) in -2_147_483_648..2_147_483_647
map.adb:2: (pre)- map:(overflow check) Y + X(10) in -2_147_483_648..2_147_483_647
map.adb:2: (pre)- map:(overflow check) Y + X(2) in -2_147_483_648..2_147_483_647
map.adb:2: (pre)- map:(overflow check) Y + X(3) in -2_147_483_648..2_147_483_647
```

(continues on next page)

(continued from previous page)

```
map.adb:2: (pre)- map:(overflow check) Y + X(4) in -2_147_483_648..2_147_483_647
map.adb:2: (pre)- map:(overflow check) Y + X(5) in -2_147_483_648..2_147_483_647
map.adb:2: (pre)- map:(overflow check) Y + X(6) in -2_147_483_648..2_147_483_647
map.adb:2: (pre)- map:(overflow check) Y + X(7) in -2_147_483_648..2_147_483_647
map.adb:2: (pre)- map:(overflow check) Y + X(8) in -2_147_483_648..2_147_483_647
map.adb:2: (pre)- map:(overflow check) Y + X(9) in -2_147_483_648..2_147_483_647
map.adb:2: (post)- map:Z(1) = Y + X(1)
map.adb:2: (post)- map:Z(1..10)'Initialized
map.adb:2: (post)- map:Z(10) = Y + X(10)
map.adb:2: (post)- map:Z(2) = Y + X(2)
map.adb:2: (post)- map:Z(3) = Y + X(3)
map.adb:2: (post)- map:Z(4) = Y + X(4)
map.adb:2: (post)- map:Z(5) = Y + X(5)
map.adb:2: (post)- map:Z(6) = Y + X(6)
map.adb:2: (post)- map:Z(7) = Y + X(7)
map.adb:2: (post)- map:Z(8) = Y + X(8)
map.adb:2: (post)- map:Z(9) = Y + X(9)
```

The overflow check preconditions are sufficient to guarantee that the addition on line 5 of `Map` cannot overflow, so no messages are generated.

Concat

Consider the program `Concat` which takes two arrays of unbounded size in input, and returns their concatenation in array parameter `Z`:

```
with Assign_Arr_Unk; use Assign_Arr_Unk;

procedure Concat (X, Y : in Arr; Z : out Arr) is
   K : Integer := 1;
begin
   for J in X'Range loop
      Z (K) := X (J);
      K := K + 1;
   end loop;
   for J in Y'Range loop
      Z (K) := Y (J);
      K := K + 1;
   end loop;
end Concat;
```

On this subprogram, Inspector generates the following contract:

```
concat.adb:2: (pre)- concat:(overflow check) X'Length <= 2_147_483_646
concat.adb:2: (pre)- concat:(array index check) X'Length = 0 or X'Last - X'First < Z'Last
concat.adb:2: (pre)- concat:X'Length = 0 or X(J)'Initialized
concat.adb:2: (pre)- concat:(array index check) X'Length = 0 or Z'First <= 1
concat.adb:2: (pre)- concat:(array index check) X'Length = 0 or Z'First <= X'Last - X
↪'First + 1
concat.adb:2: (pre)- concat:(array index check) X'Length = 0 or Z'Last >= 1
concat.adb:2: (pre)- concat:(aliasing check) Y'Address /= Z'Address
concat.adb:2: (pre)- concat:(overflow check) Y'Length <= 2_147_483_646
```

(continues on next page)

(continued from previous page)

```
concat.adb:2: (pre)- concat:(overflow check) Y'Length = 0 or Y(J)'Initialized
concat.adb:2: (pre)- concat:(array index check) Z'Last <= X'Last - X'First + 6_442_450_
↪942
concat.adb:2: (post)- concat:Z(..) = One-of{Z(..)'Old, (for _I in Z'Range => X((X'First␣
↪+ _I) - 1)), (for _I in Z'Range => Y(Y'First + (_I - K)))}
```

The precondition on line 4 guarantees that the assignment to `Z(K)` on line 6 of `Concat` is within bounds: either `X` is empty (then the loop is not executed), or it should fit in `Z`.

This time, when run in a deep mode, Inspector issues errors for possible index checks and overflow checks failure:

```
concat.adb:10:7: medium: array index check [CWE 120] (Inspector): requires K <= Z'Last
concat.adb:10:7: medium: array index check [CWE 120] (Inspector): requires Z'First <= K
concat.adb:11:14: low: overflow check [CWE 190] (Inspector): requires K /= Integer_32
↪'Last
```

Concat_Op

Of course, the behavior of `Concat` can be obtained by simply using the Ada concatenation operator, as in subprogram `Concat_Op`:

```
with Assign_Arr_Unk; use Assign_Arr_Unk;

procedure Concat_Op (X, Y : in Arr; Z : out Arr) is
begin
   Z := X & Y;
end Concat_Op;
```

On this subprogram, Inspector generates the following contract:

```
concat_op.adb:2: (pre)- concat_op:X'First /= 2_147_483_647
concat_op.adb:2: (pre)- concat_op:(overflow check) X'Length + Y'Length in 2..4_294_967_
↪296
concat_op.adb:2: (pre)- concat_op:X'Length in 1..4_294_967_295
concat_op.adb:2: (pre)- concat_op:Y'Length in 1..4_294_967_295
concat_op.adb:2: (post)- concat_op:possibly_updated(Z(..))
```

The preconditions above are not sufficient to guarantee that the bounds of `Z` allow such an assignment, hence the error message generated by Inspector in a deep mode:

```
concat_op.adb:4:11: medium: array index check [CWE 120] (Inspector): requires Z'Length =␣
↪((if X'Length + Y'Length = 0 then Y'First else if X'Length /= 0 then X'First else Y
↪'First)..(if X'Length + Y'Length = 0 then Y'Last else if X'Length /= 0 then X'First␣
↪else Y'First + X'Length + Y'Length - 1))'Length
```

Limitations

Rec_Constant

Although recursion and loops are in theory equivalent, Inspector treats them very differently. While the analysis of loops is carefully designed to recognize invariants over induction variables, the analysis of recursive functions may be less precise. Take as a contrived example a simple recursive function returning a constant value:

```
function Rec_Constant (X : in Integer) return Integer is
begin
   if X < 10 then
      return 3;
   else
      return Rec_Constant (5);
   end if;
end Rec_Constant;
```

On this subprogram, Inspector generates the following contract:

```
rec_constant.adb:1: (post)- rec_constant:rec_constant'Result'Initialized
```

The most precise postcondition would state that `Rec_Constant` always return the value 3.

Ident_Arr

Inspector can follow quite precisely the value of individual elements in an array, as well as the possible values for all other elements which are not individually tracked. This is not the same as being able to handle arbitrary quantified expressions over arrays, although in the cases of loop with static bounds, Inspector will automatically unroll these loops, leading to precise messages when the loop is not too large. Take for example a subprogram `Ident_Arr` that initializes an array to the identity function over its indexes:

```
procedure Ident_Arr (X : out Arr) is
begin
   for J in X'Range loop
      X (J) := J;
   end loop;
end Ident_Arr;
```

Using the aspect syntax of Ada 2012, the user can even specify a postcondition for this procedure stating the property above:

```
with Assign_Arr; use Assign_Arr;

procedure Ident_Arr (X : out Arr)
with Post => (for all J in X'Range => X (J) = J);
```

which is proven by Inspector.

Sum_All_Arr

Inspector does not treat specially some quite useful abstractions over array content, like the sum/max/min over array elements, but again, as long as the loop has static bounds and is not too large, Inspector is able to unroll such loops and produce the most precise analysis. Consider a subprogram `Sum_All_Arr` which computes the sum over the elements of an array:

```
with Assign_Arr; use Assign_Arr;

function Sum_All_Arr (X : in Arr) return Integer is
   Sum : Integer := 0;
begin
   for J in X'Range loop
      Sum := Sum + X (J);
   end loop;
   return Sum;
end Sum_All_Arr;
```

Inspector will generate preconditions stating that this sum cannot overflow:

```
sum_all_arr.adb:2: (pre)- sum_all_arr:(overflow check) X(1) + X(2) + X(3) + X(4) + X(5)␣
↪+ X(6) + X(7) + X(8) + X(9) + X(10) in -2_147_483_648..2_147_483_647
sum_all_arr.adb:2: (pre)- sum_all_arr:(overflow check) X(1) + X(2) + X(3) + X(4) + X(5)␣
↪+ X(6) + X(7) + X(8) + X(9) in -2_147_483_648..2_147_483_647
sum_all_arr.adb:2: (pre)- sum_all_arr:(overflow check) X(1) + X(2) + X(3) + X(4) + X(5)␣
↪+ X(6) + X(7) + X(8) in -2_147_483_648..2_147_483_647
sum_all_arr.adb:2: (pre)- sum_all_arr:(overflow check) X(1) + X(2) + X(3) + X(4) + X(5)␣
↪+ X(6) + X(7) in -2_147_483_648..2_147_483_647
sum_all_arr.adb:2: (pre)- sum_all_arr:(overflow check) X(1) + X(2) + X(3) + X(4) + X(5)␣
↪+ X(6) in -2_147_483_648..2_147_483_647
sum_all_arr.adb:2: (pre)- sum_all_arr:(overflow check) X(1) + X(2) + X(3) + X(4) + X(5)␣
↪in -2_147_483_648..2_147_483_647
sum_all_arr.adb:2: (pre)- sum_all_arr:(overflow check) X(1) + X(2) + X(3) + X(4) in -2_
↪147_483_648..2_147_483_647
sum_all_arr.adb:2: (pre)- sum_all_arr:(overflow check) X(1) + X(2) + X(3) in -2_147_483_
↪648..2_147_483_647
sum_all_arr.adb:2: (pre)- sum_all_arr:(overflow check) X(1) + X(2) in -2_147_483_648..2_
↪147_483_647
sum_all_arr.adb:2: (pre)- sum_all_arr:(validity check) X(1)'Initialized
sum_all_arr.adb:2: (pre)- sum_all_arr:(validity check) X(10)'Initialized
sum_all_arr.adb:2: (pre)- sum_all_arr:(validity check) X(2)'Initialized
sum_all_arr.adb:2: (pre)- sum_all_arr:(validity check) X(3)'Initialized
sum_all_arr.adb:2: (pre)- sum_all_arr:(validity check) X(4)'Initialized
sum_all_arr.adb:2: (pre)- sum_all_arr:(validity check) X(5)'Initialized
sum_all_arr.adb:2: (pre)- sum_all_arr:(validity check) X(6)'Initialized
sum_all_arr.adb:2: (pre)- sum_all_arr:(validity check) X(7)'Initialized
sum_all_arr.adb:2: (pre)- sum_all_arr:(validity check) X(8)'Initialized
sum_all_arr.adb:2: (pre)- sum_all_arr:(validity check) X(9)'Initialized
sum_all_arr.adb:2: (post)- sum_all_arr:sum_all_arr'Result = X(1) + X(2) + X(3) + X(4) +␣
↪X(5) + X(6) + X(7) + X(8) + X(9) + X(10)
sum_all_arr.adb:2: (post)- sum_all_arr:sum_all_arr'Result'Initialized
```

14.2.3 Pointer Examples

This section presents the results of running GNAT SAS' Inspector on subprograms containing pointers.

Dereference

Inspector assumes that the strong typing guarantees provided by Ada are not circumvented by the improper use of unchecked conversions and unchecked deallocation. In particular, it assumes that there are no dangling pointers. Thus, it is equivalent to state that pointer `X` can be dereferenced and that `X` is not null.

Deref

Consider an access type `Ptr` on integers, and a function `Deref.Read` which takes a parameter of type `Ptr`:

```
1 package Deref is
2    type Ptr is access all Integer;
3    function Read (X : in Ptr) return Integer;
4 end Deref;
```

It simply returns the value pointed to:

```
1 package body Deref is
2    function Read (X : in Ptr) return Integer is
3    begin
4       return X.all;
5    end Read;
6 end Deref;
```

On this subprogram, Inspector generates the following contract:

```
1 deref.adb:2: (pre)- deref.read:(access check) X /= null
2 deref.adb:2: (pre)- deref.read:(validity check) X.all'Initialized
3 deref.adb:2: (post)- deref.read:deref.read'Result = X.all
4 deref.adb:2: (post)- deref.read:deref.read'Result'Initialized
```

This is indeed the most precise contract for subprogram `Deref.Read`. The preconditions on lines 1 and 2 guarantee that no run-time error can be raised and that no uninitialized value can be read. The postcondition on line 3 completely summarizes the effect of calling `Deref.Read`.

Bad_Deref

Consider a version of `Deref` in which parameter `X` is an `in out` parameter that is always reset to `null`:

```
1 with Deref; use Deref;
2
3 function Bad_Deref (X : in out Ptr) return Integer is
4 begin
5    if X /= null then
6       X := null;
7    end if;
8    return X.all;
9 end Bad_Deref;
```

Inspector detects that `X` is null at the point of dereference, and it generates an error:

```
bad_deref.adb:2:1: high warning: subp always fails (Inspector): bad_deref fails for all␣
↪possible inputs
bad_deref.adb:7:13: high: access check [CWE 476] (Inspector): check fails here; requires␣
↪X /= null
```

Deref_Assign

Consider now subprogram `Deref_Assign` which updates the value pointed to by `X`:

```
1 with Deref; use Deref;
2
3 procedure Deref_Assign (X : in Ptr; Y : in Integer) is
4 begin
5    X.all := Y;
6 end Deref_Assign;
```

On this subprogram, Inspector generates the following contract:

```
1 deref_assign.adb:2: (pre)- deref_assign:(access check) X /= null
2 deref_assign.adb:2: (post)- deref_assign:X.all = Y
3 deref_assign.adb:2: (post)- deref_assign:X.all'Initialized
```

Again, this is the most precise contract for subprogram `Deref_Assign`. The precondition on line 1 guarantees that no run-time error can be raised. The postcondition on line 2 completely summarizes the effect of calling `Deref_Assign`.

Bad_Deref_Assign

Consider a version of `Deref_Assign` in which some defensive code wrongly returns when parameter `X` is not null:

```
1 with Deref; use Deref;
2
3 procedure Bad_Deref_Assign (X : in Ptr; Y : in Integer) is
4 begin
5    if X /= null then
6       return;
7    end if;
8    X.all := Y;
9 end Bad_Deref_Assign;
```

Inspector detects that `X` is null at the point of dereference, and it generates an error:

```
bad_deref_assign.adb:7:6: high: access check [CWE 476] (Inspector): check fails here;␣
↪requires X /= null
```

Self_Deref_Assign

Finally, consider the version of *Self_Assign* seen previously, where `X` is now a pointer:

```
1  with Deref; use Deref;
2
3  procedure Self_Deref_Assign (X : in out Ptr) is
4  begin
5     X.all := X.all + 1;
6  end Self_Deref_Assign;
```

On this subprogram, Inspector generates the following contract:

```
1  self_deref_assign.adb:2: (pre)- self_deref_assign:(access check) X /= null
2  self_deref_assign.adb:2: (pre)- self_deref_assign:(overflow check) X.all /= 2_147_483_647
3  self_deref_assign.adb:2: (post)- self_deref_assign:X.all /= -2_147_483_648
4  self_deref_assign.adb:2: (post)- self_deref_assign:X.all = X.all'Old + 1
```

The preconditions on lines 1 and 2 guarantee that that no run-time error can be raised and that no uninitialized value can be read. Note that the precondition on line 2 implicitly states that `X.all` is initialized, since it expresses a relation involving `X.all`. The lines 2-4 are exactly the same as the contract previously seen for *Self_Assign*, where `X` has been replaced by `X.all`.

Bad_Self_Deref_Assign

Consider a version of `Self_Deref_Assign` in which the increment is missing:

```
1  with Deref; use Deref;
2
3  procedure Bad_Self_Deref_Assign (X : in out Ptr) is
4  begin
5     X.all := X.all;
6  end Bad_Self_Deref_Assign;
```

Inspector detects that the assignment is useless, and in a deep mode, it generates a warning:

```
1  bad_self_deref_assign.adb:4:10: medium warning: useless reassignment [CWE 563]␣
   ↪(Inspector): useless reassignment of X.all
```

Aliasing

Pointer_Assign

What makes pointers so tricky to handle in static analysis is the possibility of aliasing, where two pointers point to the same memory location. Consider the subprogram `Pointer_Assign`:

```
1  with Deref; use Deref;
2
3  procedure Pointer_Assign (X, Y : in Ptr) is
4  begin
5     X.all := Y.all + 1;
6  end Pointer_Assign;
```

On this subprogram, Inspector generates the following contract:

```
pointer_assign.adb:2: (pre)- pointer_assign:(access check) X /= null
pointer_assign.adb:2: (pre)- pointer_assign:(access check) Y /= null
pointer_assign.adb:2: (pre)- pointer_assign:(overflow check) Y.all /= 2_147_483_647
pointer_assign.adb:2: (post)- pointer_assign:X.all /= -2_147_483_648
pointer_assign.adb:2: (post)- pointer_assign:X.all = Y.all + 1
```

Preconditions are the expected ones. The interesting postcondition is the one on line 5, which states the relationship between the input value of `Y.all` and the output value of `X.all`. Indeed, this is true whether `X` and `Y` are equal or not. (Note that on this example, Inspector displays `Y.all` for the input value pointed to by `Y`.)

Bad_Pointer_Assign

Consider a version of `Pointer_Assign` in which the user added an assertion that, after the assignment, `X.all` and `Y.all` should indeed be different:

```
with Deref; use Deref;

procedure Bad_Pointer_Assign (X, Y : in Ptr) is
begin
   X.all := Y.all + 1;
   pragma Assert (X.all = Y.all + 1);
end Bad_Pointer_Assign;
```

This is not true in the case where `X` and `Y` are equal. In a deep mode, Inspector detects this possibility, and it generates an error:

```
bad_pointer_assign.adb:5:19: low: assertion (Inspector): requires Y.all = Y.all
bad_pointer_assign.adb:5:33: low: overflow check [CWE 190] (Inspector): requires Y.all /
↪= Integer_32'Last
```

The message indicates that the value of `Y.all` at subprogram entry, and the value of `Y.all` after the assignment, may differ. This is the case precisely when `X` and `Y` are equal.

Call_Pointer_Assign

Let's see how Inspector handles a caller of `Pointer_Assign` which aliases its parameters `X` and `Y`:

```
with Deref; use Deref;
with Pointer_Assign;

procedure Call_Pointer_Assign (X : in Ptr) is
begin
   Pointer_Assign (X, X);
end Call_Pointer_Assign;
```

On this subprogram, when run in a deep mode, Inspector generates the following contract:

```
call_pointer_assign.adb:3: (pre)- call_pointer_assign:(access check) X /= null
call_pointer_assign.adb:3: (pre)- call_pointer_assign:(overflow check) X.all /= 2_147_
↪483_647
```

(continues on next page)

(continued from previous page)

```
call_pointer_assign.adb:3: (post)- call_pointer_assign:X.all /= -2_147_483_648
call_pointer_assign.adb:3: (post)- call_pointer_assign:X.all = X.all'Old + 1
```

The postcondition of *Pointer_Assign* has lead to a similar postcondition for `Call_Pointer_Assign`, where `Y.all` has correctly been replaced with `X.all'Old` (and not `X.all`).

Double_Pointer_Assign

Consider now the subprogram `Double_Pointer_Assign` which assigns to both `X` and `Y` pointers:

```
with Deref; use Deref;

procedure Double_Pointer_Assign (X, Y : in Ptr) is
begin
   X.all := 1;
   Y.all := 2;
end Double_Pointer_Assign;
```

On this subprogram, Inspector generates the following contract:

```
double_pointer_assign.adb:2: (pre)- double_pointer_assign:(access check) X /= null
double_pointer_assign.adb:2: (pre)- double_pointer_assign:(access check) Y /= null
double_pointer_assign.adb:2: (post)- double_pointer_assign:X.all in 1..2
double_pointer_assign.adb:2: (post)- double_pointer_assign:Y.all = 2
```

Note the asymmetry between postconditions 3 and 4 which deal respectively with the output values of `X.all` and `Y.all`. Since `Y.all` is assigned last, its output value 2 comes from this last assignment. On the contrary, the output value of `X.all` can come either from the explicit assignment to `X.all`, or from the assignment to `Y.all` if `X` and `Y` are equal. Therefore, the postcondition on line 3 correctly considers the two output values for `X.all`.

The postcondition on line 3 is not the most precise postcondition, which would state in which case the value of `X.all` is 1 or 2:

```
(X /= Y and X.all = 1) or (X = Y and X.all = 2)
```

As already seen in section *Basic Examples* in the context of branchings, Inspector generates internally this more precise postcondition, from which it outputs on line 3 a more user-friendly postcondition. The internal more precise postcondition is used for analyzing callers of `Double_Pointer_Assign`.

Call_Double_Pointer_Assign

Take now such a caller `Call_Double_Pointer_Assign`, which calls `Double_Pointer_Assign` on non-aliased parameters:

```
package Call_Double_Pointer_Assign is
   X, Y : aliased Integer;
   procedure Call;
end Call_Double_Pointer_Assign;
```

```
with Deref; use Deref;
with Double_Pointer_Assign;

```

(continues on next page)

(continued from previous page)

```
package body Call_Double_Pointer_Assign is
   procedure Call is
   begin
      Double_Pointer_Assign (X'Access, Y'Access);
   end Call;
end Call_Double_Pointer_Assign;
```

On this subprogram, Inspector generates the following contract:

```
call_double_pointer_assign.adb:4: (post)- call_double_pointer_assign.call:X = 1
call_double_pointer_assign.adb:4: (post)- call_double_pointer_assign.call:Y = 2
```

As mentioned before, Inspector could use the more precise contract for *Double_Pointer_Assign*, which leads here to the precise postcondition that `X` has value `1` after the call.

14.2.4 Unknown Subprograms

This section presents the results of running GNAT SAS' Inspector on partial applications. It is very common to analyze a partial application, where some units are not yet implemented. This may happen either when analyzing a library, or when not having access to parts of an application (third-party development) or when analyzing an application early on, before all the code has been developed. Inspector handles these cases gracefully, by assuming that calling an unknown subprogram has some effects on its calling environment. These assumptions are precisely tuned to avoid false alarms, while keeping the analysis as precise as possible.

Call_Unknown

Take the unknown subprogram `Unknown`:

```
procedure Unknown (X : out Integer; Y : in Integer);
```

It is called by subprogram `Call_Unknown`:

```
with Unknown;

procedure Call_Unknown (X : out Integer) is
begin
   Unknown (X, 1);
end Call_Unknown;
```

On this subprogram, Inspector generates the following contract:

```
call_unknown.adb:2: (post)- call_unknown:X'Initialized
```

Because the parameter `X` of `Unknown` is marked as `out`, Inspector assumes that it is initialized by the call to `Unknown`, hence the corresponding postcondition of `Call_Unknown`.

Call_Unknown_Pos

Now consider a slight variation of `Call_Unknown` in which `X` is a positive integer:

```
1  with Unknown;
2
3  procedure Call_Unknown_Pos (X : out Positive) is
4  begin
5     Unknown (X, 1);
6  end Call_Unknown_Pos;
```

On this subprogram, Inspector generates the following contract:

```
1  call_unknown_pos.adb:2: (presumption)- call_unknown_pos:unknown.X@4 >= 1
2  call_unknown_pos.adb:2: (post)- call_unknown_pos:X'Initialized
```

The postcondition on line 2 is similar to the one seen previously. What is new here is the *presumption* on line 1, which states that Inspector assumes that `Unknown` returns a positive value for `X`, as needed to avoid a possible run-time error in calling `Unknown`. Note the name `X@4` is used instead of `X` to denote the value of `X` on line 4, after the call to `Unknown`. Since this presumption is relative to the call in `Call_Unknown_Pos`, it is part of the contract of `Call_Unknown_Pos`.

Call_Unknown_Rel

Presumptions can be more complex, for example in the subprogram `Call_Unknown_Rel`, an assertion requires that the value of `X` returned by `Unknown` respects a linear relation with `Y`:

```
1  with Unknown;
2
3  procedure Call_Unknown_Rel (X : out Integer; Y : in Integer) is
4  begin
5     Unknown (X, Y);
6     pragma Assert (X < 2 * Y + 1);
7  end Call_Unknown_Rel;
```

On this subprogram, Inspector generates the following contract:

```
1  call_unknown_rel.adb:2: (pre)- call_unknown_rel:(overflow check) Y in -1_073_741_824..1_
   ↪073_741_823
2  call_unknown_rel.adb:2: (presumption)- call_unknown_rel:unknown.X@4 - Y*2 in -4_294_967_
   ↪294..0
3  call_unknown_rel.adb:2: (presumption)- call_unknown_rel:unknown.X@4 /= 2_147_483_647
4  call_unknown_rel.adb:2: (post)- call_unknown_rel:X - Y*2 in -4_294_967_294..0
5  call_unknown_rel.adb:2: (post)- call_unknown_rel:X /= 2_147_483_647
```

The presumption on line 2 indeed propagates the desired assertion on the expected behavior of `Unknown`.

Call_Unknown_Arr

Inspector tries as much as possible to keep precise information through calls to unknown subprograms. Take the unknown subprogram `Unknown_Arr` which takes an array as `in out` parameter:

```
with Assign_Arr; use Assign_Arr;

procedure Unknown_Arr (X : in out Arr);
```

It is called in a context where the value of some array elements is known:

```
with Assign_Arr; use Assign_Arr;
with Unknown_Arr;

procedure Call_Unknown_Arr (X : in out Arr) is
begin
   X (1) := 1;
   X (4) := 2;
   Unknown_Arr (X);
end Call_Unknown_Arr;
```

On this subprogram, Inspector generates the following contract:

```
call_unknown_arr.adb:3: (post)- call_unknown_arr:X(1 | 4) = X(1 | 4)@7
```

Since the array is passed as `in out`, Inspector assumes that the value of `X` may change, but it keeps the information that elements at indexes 1 and 4 are initialized.

Call_Unknown_Ptr

This is even more visible in the way Inspector analyzes calls to unknown subprograms which take pointers in parameter, like `Unknown_Ptr`:

```
with Deref; use Deref;

procedure Unknown_Ptr (X : in out Ptr);
```

It is called twice, first in a context where the value of parameter `X.all` is not known, then in a context where it is known:

```
with Deref; use Deref;
with Unknown_Ptr;

procedure Call_Unknown_Ptr (X, Y : in out Ptr) is
begin
   Unknown_Ptr (X);
   Y.all := 1;
   Unknown_Ptr (Y);
   pragma Assert (X.all < Y.all);
end Call_Unknown_Ptr;
```

On this subprogram, Inspector generates the following contract:

```
call_unknown_ptr.adb:3: (pre)- call_unknown_ptr:(access check) Y /= null
call_unknown_ptr.adb:3: (presumption)- call_unknown_ptr:unknown_ptr.X@5 /= null
call_unknown_ptr.adb:3: (presumption)- call_unknown_ptr:unknown_ptr.X@5.all /= 2_147_483_
↪647
call_unknown_ptr.adb:3: (presumption)- call_unknown_ptr:unknown_ptr.X@5.all < unknown_
↪ptr.X@7.all
call_unknown_ptr.adb:3: (presumption)- call_unknown_ptr:unknown_ptr.X@7 /= null
call_unknown_ptr.adb:3: (presumption)- call_unknown_ptr:unknown_ptr.X@7.all /= -2_147_
↪483_648
call_unknown_ptr.adb:3: (post)- call_unknown_ptr:X /= null
call_unknown_ptr.adb:3: (post)- call_unknown_ptr:Y /= null
call_unknown_ptr.adb:3: (post)- call_unknown_ptr:Y.all = 1
```

The presumption on line 4 shows that Inspector assumes that the first call to `Unknown_Ptr` (on line 5 of `Call_Unknown_Ptr`) returns a value for `X.all` that is less than the value of `Y.all` returned by the second call to `Unknown_Ptr` (on line 7 of `Call_Unknown_Ptr`) as specified by the following assertion.

Note that the postconditions on line 7 and 8 are derived from presumptions.

Above_Call_Unknown

In general, the exact constraint expected from a call to an unknown subprogram may not be given by its direct caller, but possibly by a caller higher in the call chain. Inspector handles these cases by generating presumptions on unknown subprograms in the contract of these indirect callers. Consider a caller of the subprogram *Call_Unknown*, which expects `X` to be different from 10 after the call:

```
with Call_Unknown;

procedure Above_Call_Unknown (X : out Integer) is
begin
   Call_Unknown (X);
   pragma Assert (X /= 10);
end Above_Call_Unknown;
```

On this subprogram, Inspector generates the following contract:

```
above_call_unknown.adb:2: (presumption)- above_call_unknown:unknown.X@4 /= 10
above_call_unknown.adb:2: (post)- above_call_unknown:X /= 10
```

The presumption on line 1 indeed requires that the call to `Unknown` inside `Call_Unknown` returns a value different from 10. Since this presumption is relative to the call in `Call_Unknown` inside a call in `Above_Call_Unknown`, it is part of the contract of `Above_Call_Unknown`.

14.3 Infer's Limitations and Heuristics

Since Infer is a static analysis framework, it can use different analyses to emit different messages, and each analysis can have distinct limitations. The following table summarizes these limitations:

Message kinds	Limitations and heuristics
access check, memory leak, use after free, function with side effects, function with side effects in conditional	The checker typically doesn't follow all paths through the subprogram and can thus miss errors.
unitialized value passed to	The checker is intraprocedural and field-insensitive. It only checks whether any field of a composite type object is written before the object is passed as an IN or IN-OUT parameter to a call. Consequently, it misses issues when the written field is not the one that is later read. The checker doesn't emit messages on tagged types primitives and null records.
write without open, read without open, double open, close without open	The checker is intraprocedural, meaning messages are emitted only for problems in a single subprogram.

CHAPTER

FIFTEEN

GLOSSARY

Analyzer package
: The *Analyzer* package is a project package used to configure GNAT SAS settings. See *Analyzer package attributes* for a complete description of available attributes.

gnatsas directory
: The directory used to store GNAT SAS intermediate files and artifacts. It is named `gnatsas` and generated at: `[<object_dir>/][<subdirs>/]gnatsas`, where `<object_dir>` refers to the *object directory* and `<subdirs>` refers to *subdirs*.

object directory
: The directory specified through the `Object_Dir` attribute of the project file. The project directory is used if not specified.

output directory
: The directory used to store GNAT SAS results for this project. By default, the output directory is generated under the *gnatsas directory* as `gnatsas/<project>.outputs`. Use the `Output_Dir` attribute of the *Analyzer package* to change the location.

review file
: User reviews are stored in a *Static Analysis Reviews (SAR) file*, or simply *review file*, generated in the *output directory* as `<output_dir>/<prj>.sar` by default. Use the `Review_File` attribute of the *Analyzer package* to change the location.

subdirs
: The optional relative path from the *object directory* (if specified, project directory otherwise) to the *gnatsas directory* where results and intermediate files will be output: `[<object_dir>/][<subdirs>/]gnatsas`. Use the `Subdirs` attribute of the *Analyzer package* to specify its value.

CHAPTER

SIXTEEN

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **Document**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **Opaque**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties---for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See https://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

INDEX