GPR2 Library Reference

Release 27.0w

AdaCore

Jun 10, 2026

CONTENTS:

Version 27.0w
Date: Jun 10, 2026

CHAPTER

ONE

OVERVIEW

The GPR2 library provides a complete parser and project model for GPR (GNAT Project) files. It is the foundation used by all AdaCore build tools (`gprbuild`, `gprclean`, `gprinstall`, `gprconfig`, ...) and is available to third-party tools that need to reason about Ada project structure.

This reference covers the core API of the GPR2 library: project tree loading, view and attribute access, source and unit enumeration, diagnostics reporting, and the incremental build infrastructure. For a task-oriented introduction with working code examples, start with the *GPR2 Library Quick Start*.

Note

This document assumes familiarity with the GPR project file language. If you are new to GPR, read the *GPR User Guide* for a task-oriented introduction or the *GPR Reference Manual* for a complete language specification before proceeding.

1.1 Core capabilities

- **Loading** a project tree, resolving all imports, aggregations, extensions, the Ada runtime project, and the configuration project (`.cgpr`).
- **Reading** project attributes, variables, types, and packages from any view in the tree, with full evaluation of default values, aliases, and configuration overrides.
- **Enumerating** source files and Ada compilation units, including their relationships (spec/body/separate, other-part links, dependencies).
- **Building incrementally** using a DAG of typed actions whose signatures are persisted across runs.

1.2 Design principles

Reference counting. All library objects (`Tree`, `View`, `Attribute`, ...) use reference counting internally. There is no manual memory management; objects are safe to copy, store in containers, and pass by value.

No exceptions for expected conditions. Operations that can fail in normal use (missing attribute, source not found, ...) return an undefined object or a Boolean, rather than raising an exception. Callers guard calls with `Has_XXX` or `Is_Defined` checks. Preconditions on getters enforce this at runtime when the library is built with `GPR2_BUILD=debug` or `GPR2_BUILD=release_checks`. At the highest level, `GPR2.Project.Tree.Load` and `GPR2.Build.Tree_Db.Execute` return a simple success/failure status so callers do not need exception handlers for normal error paths.

Typed paths. All filesystem paths in the API use `GPR2.Path_Name.Object` rather than raw strings.

Reporter-based diagnostics. Diagnostic messages are automatically forwarded to a `GPR2.Reporter` object during tree loading and by the incremental builder. Tools configure a reporter to control how messages reach the end user - for example routing them to a console, a GUI panel, or a structured logger. The underlying `GPR2.Log.Object` is accessible for post-processing purposes, but direct use is rarely needed.

1.3 Package naming

Ada child packages follow the GNAT file-naming convention and mirror the directory hierarchy. The package `GPR2.Project.Tree` lives in `gpr2-project-tree.ads`. Public API packages are under `src/lib/`; build infrastructure is under `src/build/`.

1.4 Key packages at a glance

`GPR2.Options`
: Parses common GPR tool switches and loads a tree

`GPR2.Project.Tree`
: Root object; owns all views and the build database

`GPR2.Project.View`
: One project in the tree; source of attributes, sources, units

`GPR2.Project.Attribute`
: A single evaluated attribute value

`GPR2.Project.Variable`
: A project-level or package-level variable

`GPR2.Build.Source`
: A source file as seen by the build system, with unit and dependency info

`GPR2.Build.Compilation_Unit`
: An Ada compilation unit (spec, body, separates) across the build tree

`GPR2.Path_Name`
: Value type for filesystem paths

`GPR2.Reporter`
: Configurable sink that routes diagnostics to the end user

`GPR2.Build.Tree_Db`
: Persistent build database and action DAG

`GPR2.Build.Actions`
: Abstract base type for a single build step

CHAPTER

TWO

LOADING A PROJECT

Loading a project tree is the first step in using the GPR2 library. It parses the root `.gpr` file, resolves all imports, extensions, and aggregations, and optionally locates or generates the configuration project (`.cgpr`) that describes the available compilers for each language.

The result is a `GPR2.Project.Tree.Object` that gives access to every project view in the tree.

2.1 Using GPR2.Options

`GPR2.Options.Object` is the standard way to configure a load. It mirrors the switches accepted by all standard GPR tools (`-P`, `-X`, `--target`, `--RTS`, `--subdirs`, etc.) and passes them to `Tree.Load` as a single object.

```
with GPR2.Options;
with GPR2.Project.Tree;

Tree    : GPR2.Project.Tree.Object;
Options : GPR2.Options.Object;

Options.Add_Switch (GPR2.Options.P, "myproject.gpr");
Options.Add_Switch (GPR2.Options.X, "BUILD=release");
Options.Add_Switch (GPR2.Options.Target, "aarch64-elf");
Options.Add_Switch (GPR2.Options.RTS, "ravenscar-sfp", Index => "Ada");

if not Tree.Load (Options) then
   --  Errors have already been reported via the Reporter
   return;
end if;
```

The `Index` parameter of `Add_Switch` holds the language qualifier for switches such as `--RTS:<lang>`.

`Tree.Load` returns `False` if loading failed. All diagnostic messages are forwarded to the reporter (a console reporter by default) as they are emitted, so there is no need to iterate the log on failure.

Common switches:

P
: Project file path (`-P`)

X
: Scenario variable assignment (`-X name=value`)

Target
: Cross-compilation target (`--target=`)

RTS
Runtime selection; use `Index` for the language (`--RTS[:lang]=`)

Autoconf
Config file to generate if absent (`--autoconf=`)

Config
Explicit config file to use (`--config=`)

Subdirs
Suffix appended to obj/lib/exec directories (`--subdirs=`)

Src_Subdirs
Extra source subdirectory prepended to each project (`--src-subdirs=`)

Relocate_Build_Tree
Root directory for out-of-tree builds (`--relocate-build-tree`)

Root_Dir
Base path used to compute relative relocations (`--root-dir=`)

AP
Extra project search path directory (`-aP`)

Implicit_With
Project added as an implicit dependency of all projects

Db
Additional knowledge base directory (`--db`)

Db_Minus
Skip the default knowledge base (`--db-`)

2.2 Integrating with GNATCOLL.Opt_Parse

For tools that use `GNATCOLL.Opt_Parse` for command-line parsing, the `GPR2.Options.Opt_Parse` package provides a generic package that registers all standard GPR switches into an existing `Argument_Parser` and returns a ready-to-use `Options.Object`:

```
with GNATCOLL.Opt_Parse;  use GNATCOLL.Opt_Parse;
with GPR2.Options.Opt_Parse;

Parser : Argument_Parser :=
  Create_Argument_Parser (Help => "My GPR tool");

package GPR_Args is new GPR2.Options.Opt_Parse.Args (Parser => Parser);

--  ... declare additional tool-specific arguments ...

if not Parser.Parse then
   return;
end if;

declare
   Options : constant GPR2.Options.Object :=
               GPR_Args.Parsed_GPR2_Options;
begin
   if not Tree.Load (Options) then
```

(continues on next page)

(continued from previous page)

```
      return;
   end if;
end;
```

2.3 Load parameters

`Tree.Load` accepts several optional parameters beyond `Options`:

`Reporter`
: Reporter used for all diagnostics during and after load. Defaults to a console reporter. Pass a custom reporter here or call `Tree.Set_Reporter` beforehand.

`With_Runtime`
: Whether runtime sources are included when populating source information.

`Artifacts_Info_Level`
: Source information to compute at load time. `No_Source` (default) skips source enumeration; `Sources_Only` resolves source lists; `Sources_Units` also parses unit information.

`Config`
: An explicit `Configuration.Object`. When provided, `--config` and `--autoconf` options in `Options` are ignored.

`Environment`
: Environment variable set to use. Defaults to the process environment.

`Absent_Dir_Error`
: Whether a missing obj/lib/exec directory is a warning or an error.

`Create_Missing_Dirs`
: Whether to create missing obj/lib/exec directories automatically.

`Allow_Implicit_Project`
: When no project is specified and exactly one `.gpr` file exists in the current directory, load it implicitly.

`Check_Shared_Libs_Import`
: Report an error if a shared library project imports a static library.

CHAPTER

THREE

VIEWS

A **view** (`GPR2.Project.View.Object`) represents one project in the loaded tree. It is the entry point for reading attributes, enumerating sources, and navigating the project graph.

3.1 Project kinds

Every view has a `Kind` (`GPR2.Project_Kind`):

`K_Standard`
: Regular project that owns source files and produces an executable or object files.

`K_Library`
: Produces a static or shared library.

`K_Abstract`
: No sources, no artifacts; used purely to share attribute definitions.

`K_Aggregate`
: Groups other projects for a single build invocation. Must be the root of the tree; cannot be used as an import inside another project.

`K_Aggregate_Library`
: A library built from the object files of its aggregated projects. Unlike `K_Aggregate` it can appear anywhere in the project graph.

`K_Configuration`
: Describes available compilers and tool settings. Loaded automatically by `Tree.Load`; not written by application code.

The `Qualifier` accessor returns the kind as declared in the `.gpr` file; `Kind` resolves `K_Standard` vs `K_Library` when the qualifier was omitted.

3.2 Accessing views

After a successful `Tree.Load`, views are accessed via the tree.

By namespace root

The most general entry point is `Tree.Namespace_Root_Projects`, which returns a `View.Set.Object`:

- For a plain (non-aggregate) tree it contains the single root project.
- For a `K_Aggregate` tree it contains all directly aggregated sub-project roots.

```
for Root of Tree.Namespace_Root_Projects loop
   --  process each namespace root
end loop;
```

Root project

`Tree.Root_Project` returns the top-level project that was passed to `-P`. For a non-aggregate tree this is also the only namespace root. For aggregate trees the root is the aggregate project itself; the aggregated sub-projects are reached via `Tree.Namespace_Root_Projects` or `Root.Aggregated`. Use `Tree.Root_Project` when you specifically need the project that was explicitly loaded.

3.3 Iterating the tree

`Tree.Iterate` (or the Ada `for ... of Tree` loop via the `Default_Iterator` aspect) visits views in the tree:

```
for View of Tree loop          -- uses Default_Iterator
   Ada.Text_IO.Put_Line (String (View.Name));
end loop;
```

The iterator is controlled by three parameters:

Kind : Iterator_Control
: Which relationship edges to follow. The predefined constants are:

  - `Default_Iterator` - follows imported and extended projects, skips runtime and configuration projects.
  - `Full_Iterator` - follows all edges including runtime and configuration.

  The individual flags are `I_Imported`, `I_Extended`, `I_Aggregated`, `I_Recursive`, and `I_Runtime`.

Filter : Filter_Control
: Which project kinds to yield. `Default_Filter` yields all kinds. `Library_Filter` yields only `K_Library` and `K_Aggregate_Library` projects.

Status : Status_Control
: Whether to include or exclude externally-built projects. `Default_Status` includes them.

Example - iterate only over library projects:

```
for View of Tree.Iterate
  (Kind   => GPR2.Project.Default_Iterator,
   Filter => GPR2.Project.Library_Filter)
loop
   Ada.Text_IO.Put_Line (String (View.Name));
end loop;
```

3.4 Navigating from a view

From any view you can reach related views:

View.Imports
: The set of projects directly imported by `with` clauses.

View.Closure
: The transitive closure of imported projects. Optional flags control whether extended and aggregated projects are included.

View.Extended
The set of projects extended by this view (non-empty only when the view uses `extends`).

View.Extended_Root
The root project of the extension chain.

View.Extending
The view that extends this one (if any).

View.Aggregated
The set of projects aggregated by this view. Only valid when `View.Kind in Aggregate_Kind`.

3.5 View properties

View.Name
Project name as declared in the `.gpr` file (`Name_Type`).

View.Path_Name
Absolute path to the `.gpr` file (`GPR2.Path_Name.Object`).

View.Dir_Name
Directory containing the `.gpr` file.

View.Kind
Resolved project kind.

View.Object_Directory
Resolved `Object_Dir` attribute path. Valid for standard, library, and aggregate-library projects.

View.Source_Directories
Set of resolved source directory paths.

View.Library_Directory
Resolved `Library_Dir` attribute. Valid for library projects.

View.Executable_Directory
Resolved `Exec_Dir` attribute. Valid for standard and library projects.

3.6 Aggregate projects and namespace roots

For a non-aggregate tree there is exactly one namespace root, which is the root project itself. For an aggregate tree every directly aggregated sub-project is a namespace root.

Ada requires all compilation-unit names to be unique within a namespace. GNAT derives source filenames from unit names by default, so this uniqueness requirement translates to unique filenames within a namespace. When iterating sources or resolving units, always start from a namespace root rather than from an arbitrary view to work within one consistent namespace.

`View.Is_Namespace_Root` indicates whether a view is a namespace root. Only namespace-root views may be passed to `View.Units` and `View.Unit`.

3.7 Commonly used view API

3.7.1 Attributes

Attributes are the primary way to read project configuration. Each attribute is identified by its name (a constant from `GPR2.Project.Registry.Attribute`) and an optional index.

```
--  Check whether an attribute is present before reading it
if View.Has_Attribute (PRA.Compiler.Default_Switches) then
   Attr : constant GPR2.Project.Attribute.Object :=
            View.Attribute (PRA.Compiler.Default_Switches);
   Put_Line (Attr.Value.Text);
end if;
```

Many attributes are indexed. Use `View.Attribute` with an `Attribute_Index` to look up a specific entry:

```
--  Look up Compiler.Switches for a specific source file.
--  The index is the concrete filename; the project may define it with a
--  pattern such as Compiler.Switches ("autogen-*") - the library matches
--  the concrete name against all defined patterns.
Attr := View.Attribute
  (PRA.Compiler.Switches,
   GPR2.Project.Attribute_Index.Create ("autogen-blah.adb"));

--  Or look up the switches for the Ada language:
Attr := View.Attribute
  (PRA.Compiler.Switches,
   GPR2.Project.Attribute_Index.Create (Ada_Language));
```

To enumerate all entries of an indexed attribute:

```
for Attr of View.Attributes (PRA.Compiler.Switches) loop
   Put_Line (Attr.Index.Text & " => " & Attr.Value.Text);
end loop;
```

3.7.2 Sources

`View.Sources` returns all sources owned by the view (a `Build.Source` set). `View.Visible_Sources` resolves visibility at the namespace level: when two views in the same namespace own a source with the same filename, `Visible_Sources` returns only the one that takes precedence. For namespace-root traversals prefer `View.Visible_Sources`.

```
for Src of View.Visible_Sources loop
   Put_Line (String (Src.Path_Name.Simple_Name));
end loop;
```

3.7.3 Units

`View.Units` returns all Ada compilation units visible within the namespace rooted at this view. It is only valid on a namespace root. Each `GPR2.Build.Compilation_Unit.Object` groups the spec, body, and any separates for one logical unit.

```
--  View must satisfy View.Is_Namespace_Root
for CU of View.Units loop
   Put_Line (String (CU.Name));
end loop;
```

CHAPTER

FOUR

ATTRIBUTES

Attributes are the primary mechanism for storing project configuration in a `.gpr` file. Each attribute has a name, an optional index, and a value that is either a single string or a list of strings.

The attribute API is spread across four packages:

- `GPR2.Project.Registry.Pack` - registers packages (built-in and custom).
- `GPR2.Project.Registry.Attribute` - registers attribute definitions and provides constants for every standard attribute.
- `GPR2.Project.Attribute_Index` - index values used to look up indexed attributes.
- `GPR2.Project.Attribute` - the attribute object returned by view queries.

4.1 Attribute registry

Every attribute that GPR2 recognizes must be registered before the project tree is loaded. Registration records the attribute's metadata: whether it takes an index, what kind of value it holds, and which project kinds it is allowed in.

GPR2 pre-registers all standard GPR attributes (`Compiler'Switches`, `Object_Dir`, `Main`, etc.) at library elaboration time. Tools that define their own GPR packages must register those packages and their attributes before calling `Tree.Load`.

4.1.1 Registering a custom package

Use `GPR2.Project.Registry.Pack.Add` to introduce a new package:

```
with GPR2.Project.Registry.Pack;

--  Register a custom package allowed in all project kinds
if not GPR2.Project.Registry.Pack.Exists (+"MyTool") then
   GPR2.Project.Registry.Pack.Add
     (Name     => +"MyTool",
      Projects => GPR2.Project.Registry.Pack.Everywhere);
end if;
```

`Projects` restricts which project kinds may contain the package. `Everywhere` allows it in all project kinds; `No_Aggregates` excludes aggregate projects.

Once registered, packages with this name will be recognized in `.gpr` files:

```
package MyTool is
   for Switches use ("-O2");
end MyTool;
```

4.1.2 Registering custom attributes

Attributes are registered with `GPR2.Project.Registry.Attribute.Add`. The package the attribute belongs to must already be registered.

```
with GPR2.Project.Registry.Attribute;
with GPR2.Project.Registry.Pack;

My_Package  : constant Package_Id     := +"MyTool";
My_Switches : constant Q_Attribute_Id := (My_Package, +"Switches");

if not GPR2.Project.Registry.Attribute.Exists (My_Switches) then
   GPR2.Project.Registry.Attribute.Add
     (Name                 => My_Switches,
      Index_Type           => GPR2.Project.Registry.Attribute.Language_Index,
      Value                => GPR2.Project.Registry.Attribute.List,
      Value_Case_Sensitive => True,
      Is_Allowed_In        => GPR2.Project.Registry.Attribute.Everywhere);
end if;
```

Key parameters of `Add`:

`Index_Type`
: Kind of index the attribute accepts. `No_Index` for unindexed attributes; `Language_Index` for language names; `FileGlob_Or_Language_Index` for source globs or languages; `File_Index` / `FileGlob_Index` for source filenames or patterns.

`Value`
: `Single` for a scalar value, `List` for a list of strings.

`Value_Case_Sensitive`
: Whether the value strings are case-sensitive.

`Is_Allowed_In`
: Array of `Project_Kind` booleans; use `Everywhere` to allow the attribute in all project kinds.

`Index_Optional`
: If `True`, the index may be omitted.

`Empty_Value`
: How empty values are treated: `Allow`, `Ignore` (warning), or `Error`.

`Inherit_From_Extended`
: Whether and how the attribute value is inherited from an extended project: `Inherited`, `Concatenated`, or `Not_Inherited`.

`Config_Concatenable`
: If `True`, the value is concatenated with any value found in the configuration project rather than overriding it.

4.2 Standard attribute constants

`GPR2.Project.Registry.Attribute` exposes a constant of type `Q_Attribute_Id` for every standard GPR attribute. Top-level attributes use a simple name; package attributes are nested under a child package:

`GPR2.Project.Registry.Attribute.Object_Dir`
: `Object_Dir`

`GPR2.Project.Registry.Attribute.Exec_Dir`
: `Exec_Dir`

GPR2.Project.Registry.Attribute.Source_Files
Source_Files

GPR2.Project.Registry.Attribute.Source_List_File
Source_List_File

GPR2.Project.Registry.Attribute.Main
Main

GPR2.Project.Registry.Attribute.Library_Name
Library_Name

GPR2.Project.Registry.Attribute.Compiler.Switches
Compiler'Switches

GPR2.Project.Registry.Attribute.Compiler.Default_Switches
Compiler'Default_Switches

GPR2.Project.Registry.Attribute.Compiler.Driver
Compiler'Driver

GPR2.Project.Registry.Attribute.Binder.Switches
Binder'Switches

GPR2.Project.Registry.Attribute.Linker.Switches
Linker'Switches

GPR2.Project.Registry.Attribute.Naming.Body_Suffix
Naming'Body_Suffix

GPR2.Project.Registry.Attribute.Builder.Default_Switches
Builder'Default_Switches

4.3 Reading attributes from a view

Use `View.Has_Attribute` before accessing an attribute that may be absent:

```
with GPR2.Project.Registry.Attribute;

if View.Has_Attribute (GPR2.Project.Registry.Attribute.Object_Dir) then
   Attr : constant GPR2.Project.Attribute.Object :=
            View.Attribute (GPR2.Project.Registry.Attribute.Object_Dir);
   Put_Line (Attr.Value.Text);
end if;
```

`View.Attribute` returns `Attribute.Undefined` when the attribute is absent and has no default, so the `Has_Attribute` guard is necessary for optional attributes.

4.4 Single vs. list attributes

An attribute's `Kind` is either `Single` or `List`:

```
Attr : constant GPR2.Project.Attribute.Object :=
         View.Attribute (GPR2.Project.Registry.Attribute.Main);

case Attr.Kind is
   when Single =>
```

(continues on next page)

(continued from previous page)

```
      Put_Line (Attr.Value.Text);
   when List =>
      for V of Attr.Values loop
         Put_Line (V.Text);
      end loop;
end case;
```

In practice the kind of each attribute is fixed by its registration, so you rarely need to check `Kind` at runtime - just call `Value` for single attributes and `Values` for list attributes.

4.5 Indexed attributes

Many attributes are indexed (e.g. `Compiler'Switches` is indexed by language or source filename). Pass an `Attribute_Index` to select one entry:

```
with GPR2.Project.Attribute_Index;
with GPR2.Project.Registry.Attribute;

--  Look up Compiler'Switches for the Ada language
Attr := View.Attribute
  (GPR2.Project.Registry.Attribute.Compiler.Switches,
   GPR2.Project.Attribute_Index.Create (Ada_Language));

--  Look up Compiler'Switches for a specific source file.
--  The index passed is a concrete filename. The project may define the
--  attribute with a glob-pattern index such as Compiler'Switches ("autogen-*");
--  the library resolves the lookup by pattern matching against defined
--  indexes, or by falling back to the language of the source file.
Attr := View.Attribute
  (GPR2.Project.Registry.Attribute.Compiler.Switches,
   GPR2.Project.Attribute_Index.Create_Source ("autogen-blah.adb"));
```

Creating an `Attribute_Index`:

`Attribute_Index.Create (Value_Type)`
: String index (case-insensitive by default)

`Attribute_Index.Create_Source (Simple_Name)`
: Source filename (respects filesystem case sensitivity)

`Attribute_Index.Create (Language_Id)`
: Language identifier

`Attribute_Index.I_Others`
: The special `others` index

Enumeration vs. lookup

`View.Attributes` returns all entries as they are written in the project file - glob patterns appear as-is (e.g. the index value is `"autogen-*"` rather than any resolved filename). Use it to inspect or iterate raw definitions.

`View.Attribute` with a concrete index performs resolution: the library tries to match the given value against each defined index pattern (`FileGlob` matching), and for source-file indexes also falls back to the language of that source file. This is the right call when you want the effective switches that apply to a particular source.

```
--  Enumerate raw definitions (patterns as written in the project)
for Attr of View.Attributes
  (GPR2.Project.Registry.Attribute.Compiler.Switches)
loop
   Put_Line (Attr.Index.Value & " => " & Attr.Value.Text);
   --  Index.Value may be "autogen-*", "Ada", "others", etc.
end loop;

--  Resolve the effective switches for one specific source file
Attr := View.Attribute
  (GPR2.Project.Registry.Attribute.Compiler.Switches,
   GPR2.Project.Attribute_Index.Create_Source ("autogen-blah.adb"));
```

4.6 Attribute metadata

Beyond the value, an `Attribute.Object` carries useful metadata:

`Attr.Kind`
: `Single` or `List`.

`Attr.Value`
: The single value (`Source_Reference.Value.Object`). Call `.Text` to obtain the `Value_Type` string.

`Attr.Values`
: The list of values. Each element has a `.Text` accessor.

`Attr.Index`
: The `Attribute_Index` for this entry; call `.Value` or `.Is_Others` on the result.

`Attr.Has_Index`
: `True` if this attribute entry has an index.

`Attr.Is_Default`
: `True` if the value was synthesised from a GPR default rule rather than written explicitly in the project file.

`Attr.Is_From_Config`
: `True` if the value originates from the configuration project (`.cgpr`) rather than from the user project.

CHAPTER

FIVE

SOURCES AND UNITS

Source and unit information is not populated during `Tree.Load` by default. It must be explicitly requested, either at load time via the `Artifacts_Info_Level` parameter or afterwards by calling `Tree.Update_Sources`.

5.1 Populating source information

The `Source_Info_Option` type controls how much work is done when enumerating sources:

`Sources_Only`
: Source files are enumerated and unit names are inferred from filenames according to the naming scheme. Unit information may be inaccurate for Ada (ambiguity between body and separate, krunched filenames).

`Sources_Units`
: Unit information is resolved accurately by parsing ALI files (when available) or source content. This is the default for `Update_Sources`.

`Sources_Units_Artifacts`
: As above, and also loads dependency data from ALI files.

At load time - pass `Artifacts_Info_Level` to `Tree.Load`:

```
Options.Add_Switch (GPR2.Options.P, "myproject.gpr");
if not Tree.Load (Options,
                  Artifacts_Info_Level => GPR2.Sources_Units)
then
   return;
end if;
```

After load - call `Tree.Update_Sources` explicitly:

```
if not Tree.Update_Sources (Option => GPR2.Sources_Units) then
   --  errors were reported to the reporter
   return;
end if;
```

`Update_Sources` can be called multiple times: to upgrade the level of information (e.g. from `Sources_Only` to `Sources_Units`), or to accommodate filesystem changes - it performs a delta update of the source base, adding newly appearing files and removing files that no longer exist, without re-processing the entire tree.

5.2 The Source object

`GPR2.Build.Source.Object` (extending `GPR2.Build.Source_Base.Object`) represents one source file as seen by the build system.

Key accessors:

`Src.Path_Name`
: Absolute path to the source file (`GPR2.Path_Name.Object`).

`Src.Language`
: Language identifier (e.g. `Ada_Language`).

`Src.Kind`
: `S_Spec` (spec or header file), `S_Body` (body or implementation source), or `S_Separate` (Ada separates only).

`Src.Has_Units`
: `True` for Ada sources. Ada is the only language for which GPR2 tracks formal compilation units; for all other languages this is `False`.

`Src.Owning_View`
: The view that owns this source.

`Src.Is_Inherited`
: `True` if the source is inherited from an extended project rather than owned directly.

`Src.Is_Visible`
: `True` if this source takes precedence within its namespace (i.e. it is the one returned by `View.Visible_Sources`).

For Ada sources, a single file may contain more than one compilation unit (`at` clauses in the naming package). Use `Src.Units` to iterate all unit entries, or `Src.Unit` to retrieve the one at a given index.

5.3 Iterating sources

`View.Sources` returns all sources owned by a view. `View.Visible_Sources` additionally resolves visibility: when two views in the same namespace own a source with the same filename, only the one that takes precedence is returned.

```
--  All sources owned by a view
for Src of View.Sources loop
   Put_Line (String (Src.Path_Name.Simple_Name)
             & "  [" & GPR2.Image (Src.Language) & "]");
end loop;

--  Visibility-resolved sources at a namespace root
for Src of Root_View.Visible_Sources loop
   Put_Line (String (Src.Path_Name.Simple_Name));
end loop;
```

Prefer `Visible_Sources` when working at the namespace level to avoid processing shadowed sources.

5.4 Compilation units

`GPR2.Build.Compilation_Unit.Object` groups all parts of one Ada compilation unit - spec, body, and separates - across the build tree. Compilation units are Ada-specific; for other languages, source files are individually enumerated via `View.Sources`.

CU.Name
: Ada unit name (e.g. `"My_Package.Child"`).

CU.Has_Part (Kind)
: Whether the unit has a spec (`S_Spec`), body (`S_Body`), or any separate.

CU.Spec
: `Unit_Location` for the spec (view + source path + index).

CU.Main_Body
: `Unit_Location` for the body.

CU.Separates
: Map of separate subunit names to their `Unit_Location`.

CU.Owning_View
: The view that provides the main part (body if present, else spec).

CU.Root_View
: The namespace root this unit belongs to.

A `Unit_Location` record carries three fields: `View`, `Source` (a `Path_Name.Object`), and `Index` (the `at` position within the file, or `No_Index` for single-unit files).

5.5 Iterating units

`View.Units` returns all compilation units whose main part is visible within the namespace rooted at the given view. It requires `Sources_Units` or higher and the view must be a namespace root.

```
--  View must satisfy View.Is_Namespace_Root
for CU of Root_View.Units loop
   Put_Line (String (CU.Name));
   if CU.Has_Part (GPR2.S_Spec) then
      Put_Line ("  spec: "
                & String (CU.Spec.Source.Simple_Name));
   end if;
   if CU.Has_Part (GPR2.S_Body) then
      Put_Line ("  body: "
                & String (CU.Main_Body.Source.Simple_Name));
   end if;
end loop;
```

To look up a specific unit by name:

```
CU : constant GPR2.Build.Compilation_Unit.Object :=
       Root_View.Unit (+"My_Package.Child");

if CU.Is_Defined then
   --  found
end if;
```

5.6 Ada closure

`Tree.For_Each_Ada_Closure` visits every compilation unit in the Ada transitive dependency closure of the tree's main programs (or library interface units). It requires `Sources_Units` or higher.

```
Tree.For_Each_Ada_Closure
  (Action => procedure (CU : GPR2.Build.Compilation_Unit.Object) is
   begin
      Put_Line (String (CU.Name));
   end);
```

Optional parameters:

Mains
: Restricts the closure entry points to a specific set of source filenames or unit names. Defaults to the `Main` attribute of the root project.

All_Sources
: When `True`, visits all sources regardless of the closure.

Root_Project_Only
: When `True`, only sources belonging to the root project are included.

Externally_Built
: When `True`, also visits units from externally-built projects.

CHAPTER

SIX

REPORTER

The `GPR2.Reporter` package defines the abstract diagnostic sink used throughout the library. Two distinct message streams flow through it:

- **Diagnostic messages** (`Warning`, `Error`, `Hint`, `Lint`) are produced during tree loading and source population. They are associated with a location in a project file and describe problems or observations about the project structure. Their visibility is controlled by `Verbosity_Level`.
- **End_User messages** are live informational feedback not tied to any project file location. They are emitted during loading operations (e.g. "Using project file p.gpr") and by the incremental builder as actions run (e.g. "Compiling foo.adb"). Their visibility is controlled independently via `User_Verbosity_Level` and the message's own `User_Level_Value` (`Optional`, `Regular`, `Important`).

Both streams are forwarded automatically during `Tree.Load`, `Tree.Update_Sources`, and by the build infrastructure.

6.1 Built-in reporters

Two concrete reporters are provided out of the box.

`GPR2.Reporter.Console`
Writes messages to standard output (`End_User` messages) or standard error (warnings and errors). This is the default reporter used by `Tree.Load` when no explicit reporter is provided.

```
with GPR2.Reporter.Console;

Reporter : GPR2.Reporter.Console.Object :=
             GPR2.Reporter.Console.Create
               (Verbosity      => GPR2.Reporter.Regular,
                User_Verbosity => GPR2.Reporter.Unset);
```

Optional `Create` parameters:

`Verbosity`
Controls which diagnostic severity levels are shown (default `Regular`).

`User_Verbosity`
Independent control over `End_User` build-progress messages (default `Unset`, meaning it follows `Verbosity`).

`Use_Full_Pathname`
When `True`, file paths in diagnostic messages are absolute rather than basenames.

`Level_Report_Format`
`None`, `Short`, or `Long` (default); controls the level prefix in formatted diagnostic output.

GPR2.Reporter.Log
: Stores messages in an internal `GPR2.Log.Object` instead of printing them. Useful when the caller wants to post-process or display diagnostics in a custom way (e.g. a GUI tool that renders messages in a panel).

```
with GPR2.Reporter.Log;

Reporter : GPR2.Reporter.Log.Object :=
             GPR2.Reporter.Log.Create
               (Verbosity => GPR2.Reporter.Regular);
```

After loading, retrieve the collected messages:

```
for Msg of Reporter.Log loop
   My_UI.Show (Msg.Format);
end loop;
```

6.2 Installing a reporter

Pass the reporter to `Tree.Load`:

```
if not Tree.Load (Options, Reporter => Reporter) then
   return;
end if;
```

Or set it explicitly before loading:

```
Tree.Set_Reporter (Reporter);
```

Only one reporter can be active at a time. Setting a new reporter replaces the previous one.

6.3 Verbosity levels

`Verbosity_Level` controls which diagnostic messages are shown:

Quiet
: None.

No_Warnings
: Errors only.

Regular
: Errors and warnings (default).

Verbose
: Errors, warnings, hints, and lint messages.

Very_Verbose
: Same as `Verbose`; available for subclass use.

`User_Verbosity_Level` provides independent control over `End_User` build-progress messages:

Unset
: `End_User` visibility follows `Verbosity_Level` (default).

Quiet
: Suppress all `End_User` messages.

Important_Only
Show only `Important`-level `End_User` messages.

Regular
Show `Regular` and `Important` `End_User` messages.

Verbose
Show all `End_User` messages including `Optional`-level ones.

6.4 Reporting messages manually

Any code that has access to a reporter can push messages directly:

```
--  From a GPR2.Log collected elsewhere
Reporter.Report (Some_Log);

--  A single diagnostic Message.Object
Reporter.Report (GPR2.Message.Create
  (Level   => GPR2.Message.Warning,
   Message => "project has no sources"));

--  Plain string convenience overload (creates an End_User message)
Reporter.Report ("Build complete", Level => GPR2.Message.Important);
```

6.5 The Message type

`GPR2.Message.Object` is the fundamental unit for both streams. Key fields:

`Msg.Level`
`Warning`, `Error`, `End_User`, `Hint`, or `Lint`.

`Msg.Message`
The raw text of the message.

`Msg.Sloc`
Source location (project file, line, column); undefined for `End_User` messages.

`Msg.User_Level`
For `End_User` messages: `Optional`, `Regular`, or `Important`. Controls under which `User_Verbosity` setting the message is shown.

`Msg.Format`
Formatted string in compiler style: `[<file>:<line>:<col>: ]<level>: <text>`. `End_User` messages without a source location are returned as plain text with no prefix.

`GPR2.Message.Treat_Warnings_As_Error` is a process-wide flag: when enabled, any subsequently created `Warning`-level message is promoted to `Error` at construction time.

6.6 Writing a custom reporter

Extend `GPR2.Reporter.Object` and override two primitives:

```
with GPR2.Reporter;
with GPR2.Message;
```

(continues on next page)

(continued from previous page)

```
type My_Reporter is new GPR2.Reporter.Object with null record;

overriding function Verbosity
  (Self : My_Reporter) return GPR2.Reporter.Verbosity_Level
is (GPR2.Reporter.Regular);

overriding procedure Internal_Report
  (Self    : in out My_Reporter;
   Message : GPR2.Message.Object)
is
begin
   --  Route the message wherever the tool needs it
   My_Log.Append (Message.Format);
end Internal_Report;
```

`Internal_Report` is called only for messages that pass the verbosity filter; the filtering is done by the class-wide `Report` wrapper before `Internal_Report` is invoked. Override `User_Verbosity` as well to provide independent control over the `End_User` stream.

CHAPTER

SEVEN

CUSTOM INCREMENTAL BUILDER

The GPR2 build infrastructure provides a complete framework for implementing incremental builders on top of the project model. It is used by `gprbuild` but is fully available to third-party tools. The key packages are:

- `GPR2.Build.Tree_Db` - persistent build database and action DAG.
- `GPR2.Build.Actions` - abstract base type for a single build step.
- `GPR2.Build.Artifacts` - abstract base type for the inputs and outputs that connect actions to each other.
- `GPR2.Build.Actions_Population` - populates the action graph from a project tree using the standard GPR2 build actions.
- `GPR2.Build.Actions_Scheduler` - parallel action execution engine.

7.1 Overview

A build proceeds in four stages:

1. **Load** the project tree (`Tree.Load`).
2. **Populate sources** (`Tree.Update_Sources`).
3. **Populate the action graph** - either via `Actions_Population.Populate_Actions` for standard GPR builds, or by inserting custom actions directly into `Tree.Artifacts_Database`.
4. **Execute** the graph (`Tree.Artifacts_Database.Execute`).

7.2 The build database

`Tree.Artifacts_Database` returns an access to the `GPR2.Build.Tree_Db.Object` for the tree. The database is created automatically when the tree is loaded.

```
Db : constant GPR2.Build.Tree_Db.Object_Access :=
       Tree.Artifacts_Database;
```

The database holds a directed acyclic graph of actions connected by artifacts. Each action's output artifacts become input artifacts of downstream actions, establishing the dependency order for execution. Signature checksums persist on disk so that unchanged actions are skipped on the next run.

7.3 Populating the standard action graph

For standard GPR builds (compile, bind, link), use `GPR2.Build.Actions_Population.Populate_Actions`:

```
with GPR2.Build.Actions_Population;
with GPR2.Build.Options;

Build_Opts : GPR2.Build.Options.Build_Options;
--  Build_Opts.Mains may be set to restrict to specific mains;
--  leave empty to build all mains from the root project.

if not GPR2.Build.Actions_Population.Populate_Actions
  (Tree    => Tree,
   Options => Build_Opts)
then
   return;
end if;

if not Db.Propagate_Actions then
   return;
end if;
```

`Populate_Actions` inserts compile, bind, and link actions for every source in the tree. `Propagate_Actions` then calls `On_Tree_Propagation` on each action to resolve cross-action dependencies (e.g. the Ada binder closure).

7.4 Executing the graph

Pass a scheduler and options to `Db.Execute`:

```
with GPR2.Build.Actions_Scheduler;

Scheduler  : GPR2.Build.Actions_Scheduler.Object;
Sched_Opts : GPR2.Build.Actions_Scheduler.Options;

Sched_Opts.Jobs         := 0;    --  0 = auto-detect CPU count
Sched_Opts.Stop_On_Fail := True;

case Db.Execute (Scheduler, Sched_Opts) is
   when GPR2.Build.Actions_Scheduler.Success => null;
   when GPR2.Build.Actions_Scheduler.Errors  =>
      --  some actions reported errors
      return;
   when GPR2.Build.Actions_Scheduler.Failed  =>
      --  some actions failed to launch
      return;
end case;
```

Key `Options` fields:

`Jobs`
: Parallel job count; `0` auto-detects the number of CPUs.

`Force`
: Re-execute all actions regardless of signature validity.

`Stop_On_Fail`
: Abort on first failure (default `True`).

Keep_Temp_Files
Preserve temporary files after execution (useful for debugging).

Script_File
If defined, records all executed commands to this file.

Show_Progress
Emit progress counters as actions are dispatched.

No_Warnings_Replay
When set, warnings from skipped (up-to-date) actions are suppressed rather than replayed.

Force_Jobserver
When set, abort if no Make jobserver protocol is available.

7.5 Actions

`GPR2.Build.Actions.Object` is the abstract base type for a build step. Each action owns a view (its context for looking up attributes and directories) and a signature (checksums of all its inputs and outputs).

Built-in concrete actions provided by the library:

GPR2.Build.Actions.Compile
Compiles one source file for any language.

GPR2.Build.Actions.Compile.Ada
Ada-specific compilation (extends Compile).

GPR2.Build.Actions.Ada_Bind
Runs the Ada binder (`gnatbind`) for one main.

GPR2.Build.Actions.Post_Bind
Compiles the binder-generated body.

GPR2.Build.Actions.Link
Links an executable or shared library.

GPR2.Build.Actions.Link.Partial
Partial link step used for standalone libraries.

7.5.1 Action lifecycle hooks

Each action participates in the build graph via the following hooks, called in this order:

On_Tree_Insertion
Called when the action is added to the database. The action declares its output artifacts and may insert follow-up actions (e.g. a bind action inserts the post-bind compile action here).

On_Tree_Propagation
Called after initial population. Used to expand dependencies dynamically (e.g. the binder walks the Ada closure to pull in all required compilation units). Default implementation does nothing.

Compute_Command
Builds the command line just before execution. Also called when the signature is valid (to include the command line in the signature check) with `Signature_Only => True`.

Pre_Command
Called immediately before the process is spawned. Not called when the action is skipped. Default returns `True`.

Post_Command
Called after the process completes, is skipped, or fails. Default returns `True`.

On_Static_Completion
: Replaces `Pre_Command`/`Post_Command` when actions are populated but not executed (e.g. `gprinstall`). Must not modify artifacts. Default returns `True`.

7.6 Artifacts

`GPR2.Build.Artifacts.Object` is the interface that connects actions. An action's outputs become inputs to downstream actions, establishing the DAG edges. Concrete artifact types:

Artifacts.Files.Object
: A filesystem file (source, object, library, ...).

Artifacts.Object_File.Object
: A compiled object file.

Artifacts.Library.Object
: A static or shared library.

Artifacts.Key_Value.Object
: An abstract key/value pair; used for ordering actions that do not produce a file (e.g. the UID artifact that establishes execution order without file dependencies).

Artifacts.Source_Files.Object
: A source file as a build artifact.

Wiring actions to artifacts is done via the database:

```
--  Register an output artifact for an action
if not Db.Add_Output (Action.UID, My_Object_File) then
   --  artifact already owned by another action
end if;

--  Register an input dependency
Db.Add_Input
  (Action   => Downstream_Action.UID,
   Artifact => My_Object_File,
   Explicit => True);
```

7.7 Implementing a custom action

Extend `GPR2.Build.Actions.Object`, implement `Action_Id`, and override the mandatory primitives:

```
with GPR2.Build.Actions;
with GPR2.Build.Tree_Db;
with GPR2.Build.Command_Line;

type My_Action_Id is new GPR2.Build.Actions.Action_Id with record
   View  : GPR2.Project.View.Object;
   Input : GPR2.Path_Name.Object;
end record;

overriding function View
  (Self : My_Action_Id) return GPR2.Project.View.Object
is (Self.View);

```

(continues on next page)

(continued from previous page)

```
overriding function Action_Class
  (Self : My_Action_Id) return Value_Type
is (+"MyAction");

overriding function Language
  (Self : My_Action_Id) return Language_Id
is (No_Language);

overriding function Action_Parameter
  (Self : My_Action_Id) return Value_Type
is (Value_Type (Self.Input.Simple_Name));

type My_Action is new GPR2.Build.Actions.Object with record
   Id    : My_Action_Id;
   Input : GPR2.Path_Name.Object;
end record;

overriding function UID
  (Self : My_Action) return GPR2.Build.Actions.Action_Id'Class
is (Self.Id);

overriding function Working_Directory
  (Self : My_Action) return GPR2.Path_Name.Object
is (Self.View.Object_Directory);

overriding function On_Tree_Insertion
  (Self : My_Action;
   Db   : in out GPR2.Build.Tree_Db.Object) return Boolean
is
begin
   --  Register output artifacts here
   return True;
end On_Tree_Insertion;

overriding procedure Compute_Command
  (Self           : in out My_Action;
   Slot           :        Positive;
   Cmd_Line       : in out GPR2.Build.Command_Line.Object;
   Signature_Only :        Boolean)
is
begin
   Cmd_Line.Add_Argument ("my-tool");
   Cmd_Line.Add_Argument (String (Self.Input.Value));
end Compute_Command;

overriding procedure Compute_Signature
  (Self            : in out My_Action;
   Check_Checksums :        Boolean)
is
begin
   --  Register inputs/outputs in Self.Signature for change detection
   null;
```

(continues on next page)

(continued from previous page)

```
end Compute_Signature;
```

Once implemented, insert the action into the database before calling `Execute`:

```
Action : My_Action := ...;
if not Db.Add_Action (Action) then
   --  action already present
end if;
```

CHAPTER

EIGHT

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

INDEX