Ocarina

A Compiler for the AADL
for Ocarina 1.1w, 3 June 2007

Jérôme Hugues, Thomas Vergnaud, Bechir Zalila

Table of Contents

About This Guide 1
- What This Guide Contains 1
- Conventions 1

1 Introduction 2

2 The Architecture Analysis & Design Language 3

3 Installation 4
- 3.1 Supported Platforms 4
- 3.2 Build requirements 4
- 3.3 Build instructions 4
- 3.4 Additional instructions for cross platforms 4
- 3.5 Building the documentation 4
  - 3.5.1 Build Options 5
  - 3.5.2 Compiler, Tools and Run-Time libraries Options 5

4 Using Ocarina 6
- 4.1 `ocarina_sh` 6
- 4.2 AADL Scenario Files 6
- 4.3 `ocarina` 7
- 4.4 `ocarina-config` 7

5 A Tutorial of AADL & Ocarina 9

6 Ocarina API Reference Manual 10
- 6.1 The Ocarina Core Library 10
  - 6.1.1 Rationale of the core library 10
    - 6.1.1.1 Internal representation 10
  - 6.1.2 Code organization 10
    - 6.1.2.1 AADL nodes 11
    - 6.1.2.2 AADL models 11
    - 6.1.2.3 AADL instances 11
  - 6.1.3 Low level API to manipulate tree nodes 11
    - 6.1.3.1 Ocarina.Entities 11
    - 6.1.3.2 Ocarina.Entities.Components 12
    - 6.1.3.3 Ocarina.Entities.Components.Connections 13
    - 6.1.3.4 Ocarina.Entities.Components.Flows 13
    - 6.1.3.5 Ocarina.Entities.Components.Subcomponents 13
    - 6.1.3.6 Ocarina.Entities.Components.Subprogram_Calls 13
    - 6.1.3.7 Ocarina.Entities.Messages 13
    - 6.1.3.8 Ocarina.Entities.Namespaces 14
    - 6.1.3.9 Ocarina.Entities.Properties 14
  - 6.1.4 API to build and manipulate AADL models 15
    - 6.1.4.1 Ocarina.Builder 16
    - 6.1.4.2 Ocarina.Builder.Annexes 16

6.1.4.3 Ocarina.Builder.Components . . . 17
6.1.4.4 Ocarina.Builder.Components.Connections . . . 19
6.1.4.5 Ocarina.Builder.Components.Features . . . 20
6.1.4.6 Ocarina.Builder.Components.Flows . . . 21
6.1.4.7 Ocarina.Builder.Components.Modes . . . 22
6.1.4.8 Ocarina.Builder.Components.Subcomponents . . . 23
6.1.4.9 Ocarina.Builder.Components.Subprogram_Calls . . . 23
6.1.4.10 Ocarina.Builder.Namespaces . . . 24
6.1.4.11 Ocarina.Builder.Properties . . . 25
6.1.4.12 Ocarina.Analyzer.Finder . . . 26
6.1.4.13 Ocarina.Analyzer.Queries . . . 28
6.1.5 API to build and manipulate AADL instances . . . 31
6.1.6 Core parsing and printing facilities . . . 31
6.1.6.1 Parser . . . 31
6.1.6.2 Printer . . . 32
6.1.6.3 Using the parser and the printer . . . 32
6.2 Input/Output Modules . . . 34
6.3 Gaia, an Application Generator . . . 34

7 Ada Mapping Rules . . . 35

7.1 Components mapping rules . . . 35
7.1.1 Data components mapping . . . 35
7.1.1.1 Base type mapping . . . 35
7.1.1.2 Composed type mapping . . . 35
7.1.1.3 Protected type mapping . . . 36
7.1.1.4 Accessor usage . . . 37
7.1.1.5 Middleware mapping . . . 40
7.1.1.6 AADL Properties support . . . 40
7.1.2 Subprogram components mapping . . . 41
7.1.2.1 Mapping of empty subprograms . . . 41
7.1.2.2 Mapping of opaque subprograms . . . 41
7.1.2.3 Mapping of pure call sequence subprograms . . . 43
7.1.2.4 Mapping of hybrid subprograms . . . 44
7.1.2.5 Data access . . . 46
7.1.2.6 AADL Properties support . . . 48
7.1.3 Thread components mapping . . . 48
7.1.3.1 Servant mapping . . . 48
7.1.3.2 Shared variables access . . . 49
7.1.3.3 AADL Properties support . . . 50
7.1.4 Process components mapping . . . 51
7.1.4.1 Shared variables declaration and initialization . . . 51
7.1.4.2 AADL Properties support . . . 53
7.2 Setup of the application . . . 53
7.3 Node positioning . . . 57
7.4 Description of the Middleware API . . . 58
7.4.1 API to manipulate PolyORB . . . 58
7.4.1.1 ARAO.Obj_Adapters . . . 58
7.4.1.2 ARAO.RT_Obj_Adapters . . . 58
7.4.1.3 ARAO.Periodic_Threads . . . 58
7.4.1.4 ARAO.Requests . . . 58
7.4.1.5 ARAO.Utils . . . 59
7.4.2 PolyORB Setup files . . . 59
7.4.2.1 ARAO.Setup.Ocarina_OA . . . 59
7.4.2.2 ARAO.Setup.OA.Multithreaded . . . 59

7.4.2.3 ARAO.Setup.OA.Multithreaded.Prio . . . 59

8 Petri Net Mapping Rules . . . 61
8.1 Mapping Patterns . . . 61
8.1.1 Component Features . . . 61
8.1.2 Subprograms . . . 61
8.1.3 Other Components . . . 61
8.1.4 Connections . . . 62
8.1.5 Subprogram Connections . . . 62
8.2 Examples . . . 63

9 AADL modes for Emacs and vim . . . 65
9.1 Emacs . . . 65
9.2 vim . . . 65

10 Dia editor & AADL . . . 66

Appendix A Standard AADL property files . . . 67
A.1 AADL Project . . . 67
A.2 AADL Properties . . . 69

Appendix B Ocarina AADL property files . . . 78
B.1 ARAO . . . 78

Appendix C Conformance to standards . . . 80

Index . . . 81

About This Guide

This guide describes the use of Ocarina, a compiler for the AADL.

It presents the features of the compiler, related APIs and tools; and details how to use them to build and exploit AADL models.

What This Guide Contains

This guide contains the following chapters:

- Chapter 1 [Introduction], page 2 provides a brief description of Ocarina.
- Chapter 2 [AADL], page 3 provides a quick overview of the AADL language.
- Chapter 3 [Installation], page 4 details how to configure and install Ocarina on your system.
- Chapter 4 [Using Ocarina], page 6 details the utilization of Ocarina
- Chapter 6 [Ocarina API Reference Manual], page 10
- Chapter 7 [Ada Mapping Rules], page 35 details the mapping rules used by Ocarina to generate Ada code from AADL models
- Chapter 8 [Petri Net Mapping Rules], page 61 details the mapping rules used by Ocarina to generate Petri nets from AADL models
- Chapter 9 [AADL modes for Emacs and vim], page 65 presents the Emacs and vim modes for AADL packaged with Ocarina
- Chapter 10 [Dia editor & AADL], page 66 presents the Dia editor and its AADL plugin
- Appendix A [Standard AADL property files], page 67 provides the standard AADL property sets.
- Appendix C [Conformance to standards], page 80 discusses the conformance of Ocarina to the AADL standard.

Conventions

Following are examples of the typographical and graphic conventions used in this guide:

- `Functions`, `utility program names`, `standard names`, and `classes`.
- '`Option flags`'
- '`File Names`', '`button names`', and '`field names`'.
- *Variables*.
- *Emphasis*.
- [optional information or parameters]
- Examples are described by text

```
and then shown this way.
```

Commands that are entered by the user are preceded in this manual by the characters "`$ `" (dollar sign followed by space). If your system uses this sequence as a prompt, then the commands will appear exactly as you see them in the manual. If your system uses some other prompt, then the command will appear with the `$` replaced by whatever prompt character you are using.

Full file names are shown with the "`/`" character as the directory separator; e.g., '`parent-dir/subdir/myfile.adb`'. If you are using GNAT on a Windows platform, please note that the "\" character should be used instead.

1 Introduction

Ocarina is an application that can be used to build applications from AADL descriptions. Because of its modular architecture, Ocarina can also be used to add AADL functions to existing applications.

2 The Architecture Analysis & Design Language

AADL stands for Architecture Analysis & Design Language. It can be expressed using graphical and textual syntaxes; an XML representations is also defined to ease the interoperability between tools.

AADL aims at allowing for the description of Distributed Real-Time Embedded (DRE) systems by assembling blocks separately developed. Thus it focuses on the definition of clear block interfaces, and separates the implementations from those interfaces. AADL allows for the description of both software and hardware parts of a system. Here is a brief presentation of the language; more information can be found on the AADL web site.

An AADL description is made of *components*. The AADL standard defines software components (data, threads, thread groups, subprograms, processes), execution platform components (memory, buses, processors, devices) and hybrid components (systems).

Components model well identified elements of the actual system. **Subprograms** model procedures such as those in C or Ada. **Threads** model the active part of an application (such as POSIX threads). **Processes** are memory spaces that contain the **threads**. **Thread groups** are used to create an hierarchy among threads. **Processors** model micro-processors and a minimal operating system (mainly a scheduler). **Memories** model hard disks, RAMs, etc. **Buses** model all kinds of networks, wires, etc. **Devices** model sensors, etc. Unlike other components, **systems** do not represent anything concrete; they actually create building blocks to help structure the description.

Component declarations have to be instantiated into subcomponents of other components in order to model an architecture. At the top-level, a system contains all the component instances. Most components can have subcomponents, so that an AADL description is hierarchical.

Each component has an interface (called **component type**) that provides **features** (e.g. communication ports). Components communicate one with another by **connecting** their features. To a given component type correspond zero or several implementations. Each of them describe the internals of the components: subcomponents, connections between those subcomponents, etc. An implementation of a thread or a subprogram can specify **call sequences** to other subprograms. This helps describe the whole execution flows in the architecture.

AADL defines a set of standard **properties** that can be attached to most elements (components, connections, features, etc.). Standard properties are used to specify things such as the clock frequency of a processor, the execution time of a thread, the bandwidth of a bus, etc. In addition, it is possible to add user-defined properties, to express specific description constraints.

By default, all elements of an AADL description are declared in a global name space. To avoid possible name conflicts in the case of a large description, it is possible to gather components within **packages**. Thus, packages help structure the description, while systems help structure the architecture. A package can have a public part and a private part; only the elements of the package can have a visibility on the private part. Packages can contain components declarations. So, they can be used to structure the description from a logical point of view. Unlike systems, they do not impact the architecture.

3 Installation

3.1 Supported Platforms

Ocarina has been compiled and successfully tested on the following platforms:

- Linux
- MacOS X
- Solaris
- Windows

Note: Ocarina should compile and run on every target for which GNAT is available.

3.2 Build requirements

An Ada compiler:

- GNAT Pro 5.04a or later
- GNAT GPL 2006 or later

Optional:

- XML/Ada (http://libre.adacore.com/xmlada/) if you want to build the XMI printer and dia plugin.
- PolyORB (http://libre.adacore.com/polyorb/) if you want to build distributed applications using Ocarina.

Note: per construction, the macro `configure` used to find your GNAT compiler looks first to the executable `gnatgcc`, then `adagcc` and finally to `gcc` to find out which Ada compiler to use. You should be very careful with your path and executables if you have multiple GNAT versions installed. See below explanations on the ADA environment variable if you need to override the default guess.

3.3 Build instructions

To compile and install Ocarina, execute:

```
% ./configure [some options]
% make              (or gmake if your make is not GNU make)
% make install      (ditto)
```

This will install files in standard locations. If you want to choose another prefix than '`/usr/local`', give configure a '`--prefix=whereveryouwant`' argument.

Note: at this time, you MUST use GNU make to compile this software.

3.4 Additional instructions for cross platforms

Only one Ocarina installation is currently possible with a given `--prefix`. If both a native and a cross installation are needed on the same machine, distinct prefixes must be used.

3.5 Building the documentation

Ocarina documentation is built automatically with the Ocarina libraries and tools. It is installed in the '`${prefix}/share/doc/ocarina`'

3.5.1 Build Options

Available options for the 'configure' script include:

- '`--enable-debug`': enable debugging information generation and supplementary runtime checks. Note that this option has a significant space and time cost, and is not recommended for production use.

3.5.2 Compiler, Tools and Run-Time libraries Options

The following environment variables can be used to override configure's guess at what compilers to use:

- `CC`: the C compiler
- `ADA`: the Ada 95 compiler (e.g. gcc, gnatgcc or adagcc)

For example, if you have two versions of GNAT installed and available in your `PATH`, and configure picks the wrong one, you can indicate what compiler should be used with the following syntax:

```
% ADA=/path/to/good/compiler/gcc ./configure [options]
```

Ocarina will be compiled with GNAT build host's configuration, including run-time library. You may override this setting using `ADA_INCLUDE_PATH` and `ADA_OBJECTS_PATH` environment variables. See GNAT User's Guide for more details.

NOTE: Developers building Ocarina from the version control repository who need to rebuild the configure and Makefile.in files should use the script '`support/reconfig`' for this purpose. This should be done after each update from the repository. In addition to the requirements above, they will need autoconf 2.57 or newer, automake 1.6.3 or newer.

4 Using Ocarina

4.1 ocarina_sh

This command is a simplified front-end for the functions provided by Ocarina.

```
Usage: ocarina_sh [-v] [-p] [-c] [-n] [-h] [-s <scenario_part_1> <scenario_part_2>...]
 scenario_part_? is the AADL file LIST describing the application scenario
 -b: build the generated application code
 -p: only parse the application model
 -n: only produce Petri Net
 -c: only perform schedulability analysis
 -h: only perform checks on the application model
 -v: output Ocarina version
```

This script taks an AADL scenario files as an input and, depending on the givent suitches, invokes the '`ocarina`' command with the proper command line options.

If the user gave the "-b" command line switch, the generated code will be compiled.

4.2 AADL Scenario Files

AADL scenario files are a very simple way to configure an AADL application. AADL scenario may consist of more than one AADL file but they all should be located in the same directory. Example:

The following file containing the common part of 2 AADL scenarios:

```
system RMA
properties
  Ocarina_Config::AADL_Files => ("rma.aadl");
  --  "rma.aadl" contains common AADL components (processes,
  --  threads, data types)

  Ocarina_Config::Needed_Property_Sets => (ARAO, Cheddar_Properties);
  --  The non standard predefined property sets needed by the
  --  application.
end RMA;
```

The following files contains a system implmentation of the previous one by adding specific parts for an application that will leads to a C code generation:

```
system implementation RMA.Impl_C
properties
  Ocarina_Config::AADL_Files +=> ("software_c.aadl");
  --  Note that this is an additive property
  --  association.

  Ocarina_Config::Generator => PolyORB_HI_C;
  --  The code generator
end RMA.Impl_C;
```

The following files sontains a system implementation of the first one by adding specific parts for an application that will leads to a Ada code generation:

```
system implementation RMA.Impl_Ada
properties
  Source_Text +=> ("software_ada.aadl");
  --  Note that this is an additive property
  --  association.

  Ocarina_Config::Generator => PolyORB_HI_Ada;
  --  The code generator
```

```
end RMA.Impl_Ada;
```

Node that for the 2 last files, we used the "additive" for of AADL properties to "add" AADL files.

If the user invokes '`ocarina_sh`' on both '`scenario_common.aadl`' and '`scenario_ada.aadl`', then '`ocarina`' will be invoked to generate C code for the PolyORB-HI middleware.

If the user invokes '`ocarina_sh`' on both '`scenario_common.aadl`' and '`scenario_ada.aadl`', then '`ocarina`' will be invoked to generate Ada code for the PolyORB-HI middleware.

4.3 ocarina

This commnad is a extended front-end for the functions provided by Ocarina.

Note: this script is the actual driver for the Ocarina compiler, its use is for advanced users. `ocarina_sh` *relies on this driver to produce all-in-on model analysis and code generation*

```
Usage: ocarina [opts] files

  files are a non null sequence of AADL or DIA files

  Options:
  -v  Output Ocarina version, then exit
  -s  Output Ocarina search directory, then exit
  -f  Parse Ocarina predefined NON STANDARD property sets
  -a  Alternative legality rules. Extensions to AADL 1.0
  -c  To generate Petri Net from an intermediary tree
  -n  To generate Petri Net from the AADL instance model
  -g  To generate code from the AADL instance tree
      Registered generators:
       PolyORB-QoS-Ada
       PolyORB-HI-Ada
       PolyORB-HI-C
  -u  Dump the gaia tree
  -e  To only expand the AADL model
  -d  Specify output directory
  -o  Specify output file
  -p  Specify the printer to use
      Registered printers:
       aadl
       aadl_min
       aadl_tree_p
       aadl_tree_e
       dia
  -I  Specify the inclusion paths
  -q  Quiet mode, no debugging messages (default mode)
  -V  Verbose mode, display debugging messages
  -h  Hardware checking, perform hardware checking
```

4.4 ocarina-config

`ocarina-config` returns path and library information on Ocarina's installation.

```
Usage: ocarina-config [OPTIONS]
Options:
        No option:
            Output all the flags (compiler and linker) required
            to compile your program.
        [--prefix[=DIR]]
            Output the directory in which Ocarina architecture-independent
           files are installed, or set this directory to DIR.
        [--exec-prefix[=DIR]]
```

```
    Output the directory in which Ocarina architecture-dependent
   files are installed, or set this directory to DIR.
[--version|-v]
    Output the version of Ocarina.
[--config]
    Output Ocarina's configuration parameters.
[--runtime[=<Runtime_Name>]]
    Checks the validity and the presence of the given runtime and
    then, outputs its path. Only one runtime can be requested at
    a time. If no runtime name is given, outputs the root directory
    of all runtimes.
[--libs]
    Output the linker flags to use for Ocarina.
[--properties]
    Output the location of the standard property file.
[--resources]
    Output the location of resource files
    (typically the standard properties)
[--cflags]
    Output the compiler flags to use for Ocarina.
[--help]
    Output this message
```

This script can be used to compile user program that uses Ocarina's API. See Chapter 6 [Ocarina API Reference Manual], page 10.

5 A Tutorial of AADL & Ocarina

This chapter will appear in a future revision of Ocarina.

6 Ocarina API Reference Manual

This chapter describes Ocarina's API, an API to manipulate AADL models.

6.1 The Ocarina Core Library

6.1.1 Rationale of the core library

The core library holds the tree structures of AADL descriptions. It provides the facilities required to manipulate the AADL descriptions.

6.1.1.1 Internal representation

AADL descriptions are usually sets of declarations. Verifications can be performed on these representations, but the declarations must be instantiated in order to be able to fully compute the architectures. Therefore, the processing of an AADL description is usually performed in two steps:

- the building of an AADL model that corresponds to the AADL declarations;
- then the instantiation of the model to manipulate the actual architecture.

The Ocarina core library provides the necessary functions to build AADL declaration, validate the model and then instantiate it.

Both model and instance AADL descriptions are represented by abstract syntax trees. These trees are made of several nodes, defined in '`src/core/tree/ocarina-nodes.idl`'. As the structure is rather complex, higher lever notions are defined. Thus, Ocarina mainly manipulate *entities*. AADL entities are:

- namespaces:
  - AADL specification,
  - packages,
  - property sets;
- components:
  - component types and implementations,
  - port group types;
- properties:
  - property names, types and constants,
  - property associations;
- subclauses:
  - features (ports, port group specifications, subprograms as features, subcomponent accesses),
  - subcomponents,
  - subprogram call sequences and subprogram calls,
  - connections,
  - modes,
  - flows;
- annexes (libraries and subclauses).

6.1.2 Code organization

The Ocarina core is made of three main parts: the manipulation of the tree nodes, the manipulation of AADL models and the manipulation of AADL architecture instances.

6.1.2.1 AADL nodes

The manipulation of the tree nodes consists of the structures of the tree, and functions to manipulate them at a low level. The corresponding files are located in '`src/core/tree`'.

The functions of lowest level are located in the package `Ocarina.Nodes`. They allow for the direct manipulation of tree, without checking any semantics. Some sort of assembly language to manipulate nodes.

Some higher-level access functions are provided in packages such as `Ocarina.Entities`. Those functions provide higher level access to entity information such as getting their name, etc. without dealing with the actual structure of the nodes. They can be considered as low level functions. However, these functions should always be preferred to lowest level ones as they manipulates entities. These functions will be described in the next section;

6.1.2.2 AADL models

Functions are provided to manipulate trees of AADL models. They allow to build, check and interrogate model trees. The files are located in '`src/core/model`'.

High level functions are provided to manipulate AADL models. They are meant to hide the actual structure of the Ocarina tree and only manipulate AADL notions (components, connections, etc.) Several packages are located in '`src/core/model`' and provide facilities to build, verify and interrogate AADL models.

6.1.2.3 AADL instances

Other functions are provided to manipulate trees of AADL instances. Like for AADL models, it is possible to build, check and interrogate trees. However, instance trees cannot be directly built: they are computed from model trees, thus instantiating AADL models. The files are located in '`src/core/instance`'.

6.1.3 Low level API to manipulate tree nodes

Low level functions that are provided in `Ocarina.Nodes` and the package `Ocarina.Entities` and its child packages.

Functions of `Ocarina.Nodes` allow the direct manipulation of the node tree. Therefore it is difficult to use them. We do not describe this API. The packages `Ocarina.Entities` provide an API to manipulate entities, thus easing the manipulation of the tree elements.

6.1.3.1 Ocarina.Entities

This package defines the following subprograms:

Get_Entity_Category: Return the entity referenced by an entity reference, or No_Node if nothing is pointed

```
function Get_Entity_Category (Node : Types.Node_Id) return Entity_Category;
```

Get_Name_Of_Entity: Return the entity referenced by an entity reference, or No_Node if nothing is pointed

```
function Get_Name_Of_Entity
     (Entity : Types.Node_Id;
      Get_Display_Name : Boolean := True)
     return Types.Name_Id;
```

Get_Name_Of_Entity: Return the entity referenced by an entity reference, or No_Node if nothing is pointed

```
function Get_Name_Of_Entity
     (Entity : Types.Node_Id;
      Get_Display_Name : Boolean := True)
```

```
return String;
```

Get_Name_Of_Entity_Reference: Return the entity referenced by an entity reference, or No_Node if nothing is pointed

```
function Get_Name_Of_Entity_Reference
     (Entity_Ref : Types.Node_Id;
      Get_Display_Name : Boolean := True)
     return Types.Name_Id;
```

Get_Name_Of_Entity_Reference: Return the entity referenced by an entity reference, or No_Node if nothing is pointed

```
function Get_Name_Of_Entity_Reference
     (Entity_Ref : Types.Node_Id;
      Get_Display_Name : Boolean := True)
     return String;
```

Get_Referenced_Entity: Return the entity referenced by an entity reference, or No_Node if nothing is pointed

```
function Get_Referenced_Entity
     (Entity_Ref : Types.Node_Id)
     return Types.Node_Id;
```

Set_Referenced_Entity: Set the entity that is to be referenced by the entity reference

```
procedure Set_Referenced_Entity (Entity_Ref, Entity : Types.Node_Id);
```

Add_Path_Element_To_Entity_Reference: Add Item to the end of the path that constitutes the reference to the entity.

```
procedure Add_Path_Element_To_Entity_Reference
     (Entity_Ref, Item : Types.Node_Id);
```

Entity_Reference_Path_Has_Several_Elements: return True if the path has more than one element.

```
function Entity_Reference_Path_Has_Several_Elements
     (Entity_Ref : Types.Node_Id)
     return Boolean;
```

Duplicate_Identifier:

```
function Duplicate_Identifier
     (Identifier : Types.Node_Id)
     return Types.Node_Id;
```

6.1.3.2 Ocarina.Entities.Components

This package provides all the required functions to build and manipulate AADL components. The operations performed through this API return True if everything is correct or False if the operation is illegal.

This package defines the following subprograms:

Get_Category_Of_Component: return the category of the component type, implementation or instance.

```
function Get_Category_Of_Component
     (Component : Types.Node_Id)
     return Component_Category;
```

6.1.3.3 Ocarina.Entities.Components.Connections

This package provides functions to build connection within component implementations.

This package defines the following subprograms:

Get_Category_Of_Connection:

```
function Get_Category_Of_Connection
      (Connection : Types.Node_Id)
      return Connection_Type;
```

6.1.3.4 Ocarina.Entities.Components.Flows

This package provides functions to build flows within component implementations.

This package defines the following subprograms:

Get_Category_Of_Flow:

```
function Get_Category_Of_Flow (Flow : Types.Node_Id) return Flow_Category;
```

6.1.3.5 Ocarina.Entities.Components.Subcomponents

This package provides functions to build subcomponents within component implementations.

This package defines the following subprograms:

Get_Category_Of_Subcomponent: Return the category of the subcomponent or subcomponent instance.

```
function Get_Category_Of_Subcomponent
      (Subcomponent : Types.Node_Id)
      return Component_Category;
```

Get_Corresponding_Component: return the corresponding component declaration, if there is any, or No_Node.

```
function Get_Corresponding_Component
      (Subcomponent : Types.Node_Id)
      return Types.Node_Id;
```

6.1.3.6 Ocarina.Entities.Components.Subprogram_Calls

This package provides functions to build subcomponents within component implementations.

This package defines the following subprograms:

Get_Corresponding_Subprogram: return the corresponding subprogram declaration, if there is any, or No_Node.

```
function Get_Corresponding_Subprogram
      (Call : Types.Node_Id)
      return Types.Node_Id;
```

6.1.3.7 Ocarina.Entities.Messages

This package defines the following subprograms:

Display_Node_Kind_Error:

```
function Display_Node_Kind_Error      (Node : Types.Node_Id)
      return Boolean;
```

DNKE:

```
function DNKE (Node : Types.Node_Id) return Boolean
      renames Display_Node_Kind_Error
```

6.1.3.8 Ocarina.Entities.Namespaces

This package provides API to ease the manipulation of allowed elements into the unnamed namespace. They return No_Node if there was an error, else they return the element that has been passed as parameter.

This package defines the following subprograms:

Package_Has_Public_Declarations_Or_Properties: Returns True if the package has public elements, else False. Pack must reference a package specification.

```
function Package_Has_Public_Declarations_Or_Properties
      (Pack : Types.Node_Id)
      return Boolean;
```

Package_Has_Private_Declarations_Or_Properties: Returns True if the package has private elements, else False. Pack must reference a package specification.

```
function Package_Has_Private_Declarations_Or_Properties
      (Pack : Types.Node_Id)
      return Boolean;
```

6.1.3.9 Ocarina.Entities.Properties

This package provides functions to create or read property names, types, constants and associations.

This package defines the following subprograms:

Value_Of_Property_Association_Is_Undefined:

```
function Value_Of_Property_Association_Is_Undefined
      (Property : Types.Node_Id)
      return Boolean;
```

Type_Of_Property_Is_A_List:

```
function Type_Of_Property_Is_A_List
      (Property : Types.Node_Id)
      return Boolean;
```

Get_Type_Of_Property:

```
function Get_Type_Of_Property
      (Property : Types.Node_Id;
       Use_Evaluated_Values : Boolean := True)
      return Property_Type;
```

Get_Type_Of_Property_Value:

```
function Get_Type_Of_Property_Value
      (Property_Value : Types.Node_Id;
       Use_Evaluated_Values : Boolean := True)
      return Property_Type;
```

Get_Integer_Of_Property_Value:

```
function Get_Integer_Of_Property_Value
      (Property_Value : Types.Node_Id)
      return Types.Unsigned_Long_Long;
```

Get_Float_Of_Property_Value:

```
function Get_Float_Of_Property_Value
      (Property_Value : Types.Node_Id)
      return Long_Long_Float;
```

Get_String_Of_Property_Value:

function Get_String_Of_Property_Value
(Property_Value : Types.Node_Id)
return Types.Name_Id;

Get_String_Of_Property_Value:

function Get_String_Of_Property_Value
(Property_Value : Types.Node_Id)
return String;

Get_Enumeration_Of_Property_Value:

function Get_Enumeration_Of_Property_Value
(Property_Value : Types.Node_Id)
return Types.Name_Id;

Get_Enumeration_Of_Property_Value:

function Get_Enumeration_Of_Property_Value
(Property_Value : Types.Node_Id)
return String;

Get_Boolean_Of_Property_Value:

function Get_Boolean_Of_Property_Value
(Property_Value : Types.Node_Id)
return Boolean;

Get_Classifier_Of_Property_Value:

function Get_Classifier_Of_Property_Value
(Property_Value : Types.Node_Id)
return Types.Node_Id;

Get_Reference_Of_Property_Value:

function Get_Reference_Of_Property_Value
(Property_Value : Types.Node_Id)
return Types.Node_Id;

Get_Value_Of_Property_Association:

function Get_Value_Of_Property_Association
(Property : Types.Node_Id)
return Ocarina.AADL_Values.Value_Type;

Find_Property_Association_From_Name:

function Find_Property_Association_From_Name
(Property_List : Types.List_Id;
Property_Name : Types.Name_Id)
return Types.Node_Id;

Find_Property_Association_From_Name:

function Find_Property_Association_From_Name
(Property_List : Types.List_Id;
Property_Name : String)
return Types.Node_Id;

6.1.4 API to build and manipulate AADL models

This section describes the high level function's to manipulate AADL models:

1. Builder API~: These functions (declared in the `Ocarina.Builder` hierarchy) are provided to create the main kinds of nodes: components, namespaces, subclauses, annexes, properties, etc. They are named `Add_New_.....` They create a new node and insert it as a child of a

parent node, except for the root node. This helps ensure that the description structure is valid, since the API only allows valid constructions. All these functions take at least three parameters:

- the location of the node; for example the position in a file that corresponds to the element that is parsed; the location is of type Location as defined in the package `Locations`;
- the name that should be set for the node, since all those nodes can have names; the name is an identifier node; for some entities it can be set to `No_Node` (connections, etc.);
- the namespace or the parent node, which shall contain the newly created node; this node must be different from `No_Node`, and has to be of a correct kind, according to the AADL standard BNF.

Other parameters may be required, to specify information such as the category of the component (bus, process, etc.), whether the subclause is a refinement, etc. Some parameters have default values, meaning that they can be omitted if the value is not known when the node is created; the values can be set later, using lower-level access functions. Functions simply named `Add_...` are used internally to add child nodes to their parents. No verification is performed; they just ensure the child lists are created before inserting the node in them.

2. Verification API: Functions are provided to check the validity of AADL models. They ensure that all referenced AADL entities are declared, that their type is correct, etc. They also check that AADL properties are consistent.
3. Interrogation API: These functions (declared in the `Ocarina.Analyzer` hierarchy) are provided to interrogate AADL models in order to find entities, get properties, etc.

6.1.4.1 Ocarina.Builder

This package defines no subprogram

6.1.4.2 Ocarina.Builder.Annexes

This package provides functions to build annex nodes into the AADL tree.

This package defines the following subprograms:

Set_Annex_Content: Set the text of the annex. Annex is the Node_Id of the annex library or subclause, returned by Add_New_Annex_Subclause or Add_New_Annex_Library. Text is the Name_Id referencing the text of the annex. Return True is everything went right, else False.

```
function Set_Annex_Content
     (Annex : Types.Node_Id;
      Text : Types.Name_Id)
     return Boolean;
```

Add_New_Annex_Subclause: Create a new annex subclause. An annex subclause can be inserted into a component declaration (type or implementation) or a port group declaration. Loc is the location of the annex in the parsed text. Annex_Name is the name of the annex subclause. Namespace is the component or the port group where the annex must be inserted. This functions returns the Node_Id of the newly created annex subclause node, or No_Node if something went wrong.

```
function Add_New_Annex_Subclause      (Loc : Locations.Location;
      Annex_Name : Types.Node_Id;
      Namespace : Types.Node_Id)
     return Types.Node_Id;
```

Add_New_Annex_Library: Create a new annex library. An annex library can be inserted into a package or the top level AADL specification (i.e. the unnamed namespace). Loc is the location of the annex in the parsed text. Annex_Name is the name of the annex library. Namespace is the package specification or the top level AADL specification where the annex must be inserted. This functions returns the Node_Id of the newly created annex library node, or No_Node if something went wrong.

```
function Add_New_Annex_Library
     (Loc : Locations.Location;
      Annex_Name : Types.Node_Id;
      Namespace : Types.Node_Id;
      Is_Private : Boolean := False)
     return Types.Node_Id;
```

6.1.4.3 Ocarina.Builder.Components

This package defines the following subprograms:

Add_Annex: Add an annex subclause into a component (type or implementation). Component is a Node_Id referencing the component. Annex is a Node_Id referencing the annex subclause. Returns True if the annex was correctly added into the component, else False.

```
function Add_Annex
     (Component : Node_Id;
      Annex     : Node_Id)
     return Boolean;
```

Add_Connection: Add a connection into a component implementation. Component is a Node_Id referencing the component implementation. Connection is a Node_Id referencing the connection. Returns True if the connection was correctly added into the component, else False.

```
function Add_Connection
     (Component  : Node_Id;
      Connection : Node_Id)
     return Boolean;
```

Add_Feature: Add a feature into a component type. Component is a Node_Id referencing the component type. Feature is a Node_Id referencing the feature. Returns True if the feature was correctly added into the component, else False.

```
function Add_Feature
     (Component : Node_Id;
      Feature   : Node_Id)
     return Boolean;
```

Add_Refined_Type: Add a refined type into a component implementation. Refined types correspond to refinements of the component type features.

```
function Add_Refined_Type
     (Component : Node_Id;
      Refined_Type : Node_Id)
     return Boolean;
```

Add_Subcomponent: Add a subcomponent into a component implementation. Component is a Node_Id referencing the component implementation. Subcomponent is a Node_Id referencing the subcomponent. Returns True if the subcomponent was correctly added into the component, else False.

```
function Add_Subcomponent
     (Component    : Node_Id;
      Subcomponent : Node_Id)
```

```
return Boolean;
```

Add_Subprogram_Call_Sequence: Add a subprogram call sequence into a component implementation. Component is a Node_Id referencing the component implementation. Call_Sequence is a Node_Id referencing the subcomponent. Returns True if the sequence was correctly added into the component, else False.

```
function Add_Subprogram_Call_Sequence
     (Component     : Node_Id;
      Call_Sequence : Node_Id)
     return Boolean;
```

Add_Flow_Spec: Add a flow specification into a component type. Component is a Node_Id referencing the component type. Flow_Spec is a Node_Id referencing the flow. Returns True if the flow was correctly added into the component, else False.

```
function Add_Flow_Spec
     (Component : Node_Id;
      Flow_Spec : Node_Id)
     return Boolean;
```

Add_Flow_Implementation: Add a flow implementation into a component implementation. Component is a Node_Id referencing the component implementation. Flow_Impl is a Node_Id referencing the flow. Returns True if the flow was correctly added into the component, else False.

```
function Add_Flow_Implementation
     (Component : Node_Id;
      Flow_Impl : Node_Id)
     return Boolean;
```

Add_End_To_End_Flow_Spec: Add an end to end flow specification into a component implementation. Component is a Node_Id referencing the component implementation. Flow_Impl is a Node_Id referencing the flow. Returns True if the flow was correctly added into the component, else False.

```
function Add_End_To_End_Flow_Spec
     (Component       : Node_Id;
      End_To_End_Flow : Node_Id)
     return Boolean;
```

Add_Mode: Add a mode (declaration or transition) into a component implementation. Component is a Node_Id referencing the component implementation. Mode is a Node_Id referencing the mode declaration or mode transition. Returns True if the mode was correctly added into the component, else False.

```
function Add_Mode
     (Component : Node_Id;
      Mode      : Node_Id)
     return Boolean;
```

Add_Property_Association: Add a property association into a component (type or implementation). Component is a Node_Id referencing the component type or implementation. Property_Association is a Node_Id referencing the property association. Returns True if the property was correctly added into the component, else False. Component creation

```
function Add_Property_Association
     (Component            : Node_Id;
      Property_Association : Node_Id)
     return Boolean;
```

Add_New_Component_Type: Create a new component type node. A component type can be inserted into a package or the top level AADL specification (aka the unnamed namespace). Loc is the location of the component in the parsed text. Identifier is a Node_Id referencing the name of the component. Namespace is either a package specification or the top level AADL specification. Component_Type is the component category (processor, memory, process, etc.). Is_Private indicates if the component is declaraed in the private or the public part of the package; it is only relevant if Namespace references a package specification. Returns the Node_Id of the newly created component type node, or No_Node if something went wrong.

```
function Add_New_Component_Type
     (Loc            : Location;
      Identifier     : Node_Id;
      Namespace      : Node_Id;
      Component_Type : Ocarina.Entities.Components.Component_Category;
      Is_Private     : Boolean := False)
     return Node_Id;
```

Add_New_Component_Implementation: Create a new component implementation node. A component implementation can be inserted into a package or the top level AADL specification (aka the unnamed namespace). Loc is the location of the component in the parsed text. Identifier is a Node_Id referencing the name of the component. Namespace is either a package specification or the top level AADL specification. Component_Type is the component category (processor, memory, process, etc.). Is_Private indicates if the component is declaraed in the private or the public part of the package; it is only relevant if Namespace references a package specification. Returns the Node_Id of the newly created component implementation node, or No_Node if something went wrong.

```
function Add_New_Component_Implementation
     (Loc            : Location;
      Identifier     : Node_Id;
      Namespace      : Node_Id;
      Component_Type : Ocarina.Entities.Components.Component_Category;
      Is_Private     : Boolean := False)
     return Node_Id;
```

Add_New_Port_Group: Create a new port group type. It can be inserted into a package or the top level AADL specification.

```
function Add_New_Port_Group
     (Loc        : Location;
      Name       : Node_Id;
      Namespace  : Node_Id;
      Is_Private : Boolean := False)
     return Node_Id;
```

6.1.4.4 Ocarina.Builder.Components.Connections

This package defines the following subprograms:

Add_Property_Association: Add a property association to the connection declaration. Connection must reference a connection declaration. Property_Association references the property association. Return True if everything went right, else False.

```
function Add_Property_Association
     (Connection           : Node_Id;
      Property_Association : Node_Id)
     return Boolean;
```

Add_New_Connection: Create and add a new connection into a component implementation. Loc is the location of the connection in the parsed text. Name references an identifier which contains the name of the connection, if any. Comp_Impl references the component implementation. Category is the type of the connection. Is_Refinement indicates wether the connection is a refinement or not. Source and Destination are the left and right memebers of the connection. In_Modes contains the list of the modes associated to the connection. Name can be No_Node, if the connection is not nammed. Return the Node_Id of the newly created connection if everything went right, else No_Node.

```
function Add_New_Connection
     (Loc            : Location;
      Name           : Node_Id;
      Comp_Impl      : Node_Id;
      Category       : Ocarina.Entities.Components.Connections.Connection_Type;
      Is_Refinement  : Boolean := False;
      Source         : Node_Id := No_Node;
      Destination    : Node_Id := No_Node;
      In_Modes       : Node_Id := No_Node)
     return Node_Id;
```

6.1.4.5 Ocarina.Builder.Components.Features

The core API for the feature subclause of the component types and the port group types.

This package defines the following subprograms:

Add_Property_Association:

```
function Add_Property_Association
     (Feature               : Node_Id;
      Property_Association  : Node_Id)
     return Boolean;
```

Add_New_Port_Spec:

```
function Add_New_Port_Spec
     (Loc                : Location;
      Name               : Node_Id;
      Container          : Node_Id;
      Is_In              : Boolean;
      Is_Out             : Boolean;
      Is_Data            : Boolean;
      Is_Event           : Boolean;
      Is_Refinement      : Boolean := False;
      Associated_Entity  : Node_Id := No_Node)
     return Node_Id;
```

Add_New_Port_Group_Spec:

```
function Add_New_Port_Group_Spec
     (Loc            : Location;
      Name           : Node_Id;
      Container      : Node_Id;
      Is_Refinement  : Boolean := False)
     return Node_Id;
```

Add_New_Server_Subprogram:

```
function Add_New_Server_Subprogram
     (Loc            : Location;
```

```
 Name          : Node_Id;
 Container     : Node_Id;
 Is_Refinement : Boolean := False)
return Node_Id;
```

Add_New_Data_Subprogram_Spec:

```
function Add_New_Data_Subprogram_Spec
(Loc           : Location;
 Name          : Node_Id;
 Container     : Node_Id;
 Is_Refinement : Boolean := False)
return Node_Id;
```

Add_New_Subcomponent_Access:

```
function Add_New_Subcomponent_Access
(Loc           : Location;
 Name          : Node_Id;
 Container     : Node_Id;
 Is_Refinement : Boolean := False;
 Category      : Ocarina.Entities.Components.Component_Category;
 Is_Provided   : Boolean)
return Node_Id;
```

Add_New_Parameter:

```
function Add_New_Parameter
(Loc           : Location;
 Name          : Node_Id;
 Container     : Node_Id;
 Is_In         : Boolean := True;
 Is_Out        : Boolean := True;
 Is_Refinement : Boolean := False)
return Node_Id;
```

6.1.4.6 Ocarina.Builder.Components.Flows

This package defines the following subprograms:

Add_Property_Association: Add a property association to the flow declaration. Flow must reference a flow implementation or a flow specification. Property_Association references the property association. Return True if everything went right, else False.

```
function Add_Property_Association
(Flow                 : Node_Id;
 Property_Association : Node_Id)
return Boolean;
```

Add_New_Flow_Spec: Create a new flow specification inside a component type

```
function Add_New_Flow_Spec
(Loc           : Location;
 Name          : Node_Id;
 Comp_Type     : Node_Id;
 Category      : Ocarina.Entities.Components.Flows.Flow_Category;
 Source_Flow   : Node_Id;
 Sink_Flow     : Node_Id;
 Is_Refinement : Boolean := False)
return Node_Id;
```

Add_New_Flow_Implementation: Create a new flow implementation inside a component implementation

```
function Add_New_Flow_Implementation
     (Loc           : Location;
      Container     : Node_Id;
      Name          : Node_Id;
      Category      : Ocarina.Entities.Components.Flows.Flow_Category;
      In_Modes      : Node_Id;
      Is_Refinement : Boolean)
     return Node_Id;
```

Add_New_End_To_End_Flow_Spec: Create a new end to end flow specification inside a component implementation

```
function Add_New_End_To_End_Flow_Spec
     (Loc           : Location;
      Container     : Node_Id;
      Name          : Node_Id;
      In_Modes      : Node_Id;
      Is_Refinement : Boolean)
     return Node_Id;
```

6.1.4.7 Ocarina.Builder.Components.Modes

This package provides functions to handle modes in the component implementations.

This package defines the following subprograms:

Add_Property_Association: Add a property association to the mode declaration or mode transition. Mode must either reference a mode declaration or a mode transition. Property_Association references the property association. Return True if everything went right, else False.

```
function Add_Property_Association
     (Mode : Node_Id;
      Property_Association : Node_Id)
     return Boolean;
```

Add_New_Mode: Add a new mode declaration into a component implementation. Loc is the location of the mode declaration in the parsed text. Identifier references an identifier containing the name of the mode. Component references the component implementation. Is_Implicit is used by other parts of the builder API, for "in modes" clauses. You should always keep it False. Return a Node_Id referencing the newly created mode if everything went right, else False.

```
function Add_New_Mode
     (Loc : Location;
      Identifier : Node_Id;
      Component : Node_Id)
     return Node_Id;
```

Add_New_Mode_Transition: Add a new empty mode transition into a component implementation. Source, Destination, etc. of the mode transition must be added manually after the node has been created. Loc is the location of the mode transition in the parsed text. Identifier references an identifier containing the name of the mode. Component references the component implementation. Return a Node_Id referencing the newly created mode if everything went right, else False.

```
function Add_New_Mode_Transition
     (Loc : Location;
```

```
 Component : Node_Id)
return Node_Id;
```

6.1.4.8 Ocarina.Builder.Components.Subcomponents

This package defines the following subprograms:

Add_Property_Association: Add a property association to the subcomponent declaration. Subcomponent must reference a Subcomponent declaration. Property_Association references the property association. Return True if everything went right, else False.

```
function Add_Property_Association
     (Subcomponent          : Node_Id;
      Property_Association : Node_Id)
     return Boolean;
```

Add_New_Subcomponent: Create and add a new subcomponent into a component implementation. Loc is the location of the subcomponent in the parsed text. Name references an identifier which contains the name of the subcomponent. Comp_Impl references the component implementation. Category is the type of the subcomponent. Is_Refinement indicates wether the connection is a refinement or not. In_Modes contains the list of the modes associated to the connection. Return the Node_Id of the newly created subcomponent if everything went right, else No_Node.

```
function Add_New_Subcomponent
     (Loc            : Location;
      Name           : Node_Id;
      Comp_Impl      : Node_Id;
      Category       : Ocarina.Entities.Components.Component_Category;
      Is_Refinement  : Boolean := False;
      In_Modes       : Node_Id := No_Node)
     return Node_Id;
```

6.1.4.9 Ocarina.Builder.Components.Subprogram_Calls

This package defines the following subprograms:

Add_Property_Association: Add a property association to the subprogram call. Subprogram_Call must reference a subprogram call (not a call sequence). Property_Association references the property association. Return True if everything went right, else False.

```
function Add_Property_Association
     (Subprogram_Call      : Node_Id;
      Property_Association : Node_Id)
     return Boolean;
```

Add_Subprogram_Call: Add a subprogram call to the subprogram call sequence. Subprogram_Call must reference a subprogram call (not a call sequence). Call_Sequence references the subprogram call sequence. Return True if everything went right, else False.

```
function Add_Subprogram_Call      (Call_Sequence    : Node_Id;
      Subprogram_Call : Node_Id)
     return Boolean;
```

Add_New_Subprogram_Call: Create and add a new subprogram call into a subprogram call sequence. Loc is the location of the call sequence in the parsed text. Name references an identifier which contains the name of the subprogram call. Call_Sequence references the subprogram call sequence that contains the subprogram call. The function return the Node_Id of the newly created subprogram call if everything went right, else No_Node.

```
function Add_New_Subprogram_Call      (Loc            : Location;
```

```
 Name          : Node_Id;
 Call_Sequence : Node_Id)
return Node_Id;
```

Add_New_Subprogram_Call_Sequence: Create and add a new subprogram call sequence into a component implementation. Loc is the location of the call sequence in the parsed text. Name references an identifier which contains the name of the call sequence, if any. Comp_Impl references the component implementation. In_Modes contains the list of the modes associated to the connection. Name can be No_Node, if the sequence is not nammed. Subprogram calls Return the Node_Id of the newly created call sequence if everything went right, else No_Node.

```
function Add_New_Subprogram_Call_Sequence
     (Loc       : Location;
      Name      : Node_Id;
      Comp_Impl : Node_Id;
      In_Modes  : Node_Id := No_Node)
     return Node_Id;
```

6.1.4.10 Ocarina.Builder.Namespaces

This package defines the following subprograms:

Add_Declaration: Insert any component, property_set, package or port_group into the AADL specification. Namespace must reference the node created with Initialize_Unnamed_Namespace or a package specification. Return True if the element was correctly inserted, else False

```
function Add_Declaration
     (Namespace : Types.Node_Id;
      Element : Types.Node_Id)
     return Boolean;
```

Initialize_Unnamed_Namespace: Create the AADL specification node, which corresponds to the top level of the AADL description. This function must be invoked first, as all the other elements of the description will be added to this one. Loc is the location of the AADL specification in the parsed text. Return a reference to the newly created node if everything went right, else False.

```
function Initialize_Unnamed_Namespace
     (Loc : Locations.Location)
     return Types.Node_Id;
```

Add_New_Package: Checks if a package of that name already exists. If so, return this one, else create a new one and return it. Loc is the location of the package specification in the parsed text. Pack_Name is a Node_Id referencing an identifier which contains the name of the package. Namespace must reference the top level AADL specification node.

```
function Add_New_Package
     (Loc : Locations.Location;
      Pack_Name : Types.Node_Id;
      Namespace : Types.Node_Id)
     return Types.Node_Id;
```

Add_Property_Association: Add a property association to the list of the package properties, without checking for homonyms or whatever. This function should be only used by other functions of the core API. Namespace must reference a package specification. Return True if the property was added, else False.

```
function Add_Property_Association
     (Pack : Types.Node_Id;
      Property_Association : Types.Node_Id)
     return Boolean;
```

6.1.4.11 Ocarina.Builder.Properties

This package defines the following subprograms:

Add_New_Property_Set: Either Single_Value /= No_Node and Mulitple_Values = No_Node, then we have a single valued constant; or Single_Value = No_Node, then we have a muli valued constant

```
function Add_New_Property_Set
     (Loc       : Location;
      Name      : Node_Id;
      Namespace : Node_Id)
     return Node_Id;
```

Add_New_Property_Constant_Declaration: Either Single_Value /= No_Node and Mulitple_Values = No_Node, then we have a single valued constant; or Single_Value = No_Node, then we have a muli valued constant

```
function Add_New_Property_Constant_Declaration
     (Loc             : Location;
      Name            : Node_Id;
      Property_Set    : Node_Id;
      Constant_Type   : Node_Id;
      Unit_Identifier : Node_Id;
      Single_Value    : Node_Id;
      Multiple_Values : List_Id)
     return Node_Id;
```

Add_New_Property_Type_Declaration: Either Applies_To_All is set to True and Applies_To is empty, or Applies_To_All is False and Applies_To is not empty

```
function Add_New_Property_Type_Declaration
     (Loc             : Location;
      Name            : Node_Id;
      Property_Set    : Node_Id;
      Type_Designator : Node_Id)
     return Node_Id;
```

Add_New_Property_Name_Declaration: Either Applies_To_All is set to True and Applies_To is empty, or Applies_To_All is False and Applies_To is not empty

```
function Add_New_Property_Name_Declaration
     (Loc                    : Location;
      Name                   : Node_Id;
      Property_Set           : Node_Id;
      Is_Inherit             : Boolean;
      Is_Access              : Boolean;
      Single_Default_Value   : Node_Id;
      Multiple_Default_Value : List_Id;
      Property_Name_Type     : Node_Id;
      Property_Type_Is_A_List : Boolean;
      Applies_To_All         : Boolean;
      Applies_To             : List_Id)
     return Node_Id;
```

Add_New_Property_Association: If Check_For_Conflicts is set to True, then the function checks whether there is a property association of that name already. If override is set to True and there is a conflict, then it is overridden by the new association. Else the new association is ignored. If Check_For_Conflicts is set to False, then the value of Override is ignored.

```
function Add_New_Property_Association
     (Loc                 : Location;
      Name                : Node_Id;
      Property_Name       : Node_Id;
      Container           : Node_Id;
      In_Binding          : Node_Id;
      In_Modes            : Node_Id;
      Property_Value      : Node_Id;
      Is_Constant         : Boolean;
      Is_Access           : Boolean;
      Is_Additive         : Boolean;
      Applies_To          : List_Id;
      Check_For_Conflicts : Boolean := False;
      Override            : Boolean := False)
     return Node_Id;
```

6.1.4.12 Ocarina.Analyzer.Finder

This package provides functions to search nodes in the abstract tree. The functions return No_Node if nothing was found.

This package defines the following subprograms:

Select_Nodes: Build a list (chained using the accessor Next_Entity) from Decl_List and appends it to Last_Node. This list will contain the nodes whose kinds correspond to Kinds.

```
procedure Select_Nodes
     (Decl_List  :        List_Id;
      Kinds      :        Node_Kind_Array;
      First_Node : in out Node_Id;
      Last_Node  : in out Node_Id);
```

Find_Property_Entity: Find a property entity (type, name or constant). If Property_Set_Identifier is No_Node and the current scope is the one of a property set, try to find the property in it. Finally, look for the implicit property sets (AADL_Project and AADL_Properties).

```
function Find_Property_Entity
     (Root                    : Node_Id;
      Property_Set_Identifier : Node_Id;
      Property_Identifier     : Node_Id;
      Options                 : Analyzer_Options)
     return Node_Id;
```

Find_Component_Classifier: Same as above, but find a component classifier

```
function Find_Component_Classifier
     (Root                  : Node_Id;
      Package_Identifier    : Node_Id;
      Component_Identifier  : Node_Id;
      Options               : Analyzer_Options)
     return Node_Id;
```

Find_Port_Group_Classifier: Same as above, but find a port group

```
function Find_Port_Group_Classifier
     (Root                  : Node_Id;
      Package_Identifier    : Node_Id;
      Port_Group_Identifier : Node_Id;
      Options               : Analyzer_Options)
```

```
      return Node_Id;
```

Find_Feature: Find a feature in a component type or implementation

```
   function Find_Feature
      (Component          : Node_Id;
       Feature_Identifier : Node_Id)
      return Node_Id;
```

Find_Mode: Same as above, but find a mode

```
   function Find_Mode
      (Component       : Node_Id;
       Mode_Identifier : Node_Id)
      return Node_Id;
```

Find_Subcomponent: Find a subcomponent in a component implementation. If In_Modes is specified, return the subcomponent that are set in the given modes.

```
   function Find_Subcomponent
      (Component               : Node_Id;
       Subcomponent_Identifier : Node_Id;
       In_Modes                : Node_Id := No_Node)
      return Node_Id;
```

Find_Subprogram_Call: Same as above but find a subprogram call

```
   function Find_Subprogram_Call      (Component       : Node_Id;
       Call_Identifier : Node_Id;
       In_Modes        : Node_Id := No_Node)
      return Node_Id;
```

Find_Connection: Same as above but find a connection

```
   function Find_Connection
      (Component             : Node_Id;
       Connection_Identifier : Node_Id;
       In_Modes              : Node_Id := No_Node)
      return Node_Id;
```

Find_Flow_Spec: Find a flow in a component type or implementation

```
   function Find_Flow_Spec
      (Component       : Node_Id;
       Flow_Identifier : Node_Id)
      return Node_Id;
```

Find_Subclause: Same as above but find a subclause

```
   function Find_Subclause      (Component  : Node_Id;
       Identifier : Node_Id)
      return Node_Id;
```

Find_All_Declarations: Returns the first node of a list of declarations corresponding to the Kinds requested. Following nodes are accessed through the Next_Entity accessor. If no Kinds are requested, then return all the declarations found. If the Namespace is not given, search the declaration in the whole AADL specification declarations and its namespaces. Otherwise, search the declaration in the given namespace.

```
   function Find_All_Declarations
      (Root      : Node_Id;
       Kinds     : Node_Kind_Array;
       Namespace : Node_Id := No_Node)
      return Entity_List;
```

Find_All_Component_Types: Return the first component type found in the Namespace. If Namespace is No_Node, then return the first component type declaration in the whole AADL specification. Following declarations are accessed using the Next_Entity accessor.

```
function Find_All_Component_Types
     (Root      : Node_Id;
      Namespace : Node_Id := No_Node)
     return Entity_List;
```

Find_All_Top_Level_Systems: Return all systems implementations whose component type do not have any feature. Those systems correspond to the roots of the instantiated architecture. Note that there should be only one such system; else this would mean several independent architectures are described.

```
function Find_All_Top_Level_Systems (Root : Node_Id) return Entity_List;
```

Find_All_Subclauses: General function that returns the first node of a list of subclauses corresponding to the Kinds requested. Following nodes are accessed through the Next_Entity accessor.

```
function Find_All_Subclauses
     (AADL_Declaration : Node_Id;
      Kinds            : Node_Kind_Array)
     return Entity_List;
```

Find_All_Features: Applicable to component types and implementations, and port group types.

```
function Find_All_Features
     (AADL_Declaration : Node_Id)
     return Entity_List;
```

Find_All_Subclause_Declarations_Except_Properties: Applicable to component types and implementations, and port group types.

```
function Find_All_Subclause_Declarations_Except_Properties
     (AADL_Declaration : Node_Id)
     return Entity_List;
```

Find_All_Property_Associations: Applicable to component types and implementations, and port group types.

```
function Find_All_Property_Associations
     (AADL_Declaration : Node_Id)
     return Entity_List;
```

Find_Property_Association: Find the property association named Property_Association_Name. Return No_Node if nothing was found.

```
function Find_Property_Association
     (AADL_Declaration          : Node_Id;
      Property_Association_Name : Name_Id)
     return Node_Id;
```

6.1.4.13 Ocarina.Analyzer.Queries

This package contains routines that are used to get several information from the AADL tree.

This package defines the following subprograms:

Is_An_Extension: Returns True if Component is an extension of Ancestor, whether by the keyword 'extends' or because Ancestor is a corresponding component type.

```
function Is_An_Extension
     (Component : Node_Id;
```

```
 Ancestor  : Node_Id)
return Boolean;
```

Needed_By: Return True iff N is needed by Entity (for example Entity has a subcompnent of type N). It also return True if N is needed indirectly by Entity (through another intermediate need). In order for this function to work fine, the AADL tree must have been expanded. However, since it acts only on the AADL syntax tree, this function is put in this package. NOTE: If N is a property *declaration* node, the result will be True reguardless the actual need of Entity to N.

```
function Needed_By (N : Node_Id; Entity : Node_Id) return Boolean;
```

Property_Can_Apply_To_Entity: Return True if the property association Property can be applied to Entity. Otherwise, return False. Beware that this function performs exact verification; a property cannot apply to a package.

```
function Property_Can_Apply_To_Entity
     (Property : Node_Id;
      Entity   : Node_Id)
     return Boolean;
```

Is_Defined_Property: Return True if the property named 'Name' is defined for the AADL entity 'Entity'.

```
function Is_Defined_Property
     (Entity : Node_Id;
      Name   : String)
     return Boolean;
```

Is_Defined_String_Property: Return True if the aadlstring property named 'Name' is defined for the AADL entity 'Entity'.

```
function Is_Defined_String_Property
     (Entity : Node_Id;
      Name   : String)
     return Boolean;
```

Is_Defined_Integer_Property: Return True if the aadlinteger property named 'Name' is defined for the AADL entity 'Entity'.

```
function Is_Defined_Integer_Property
     (Entity : Node_Id;
      Name   : String)
     return Boolean;
```

Is_Defined_Boolean_Property: Return True if the aadlboolean property named 'Name' is defined for the AADL entity 'Entity'.

```
function Is_Defined_Boolean_Property
     (Entity : Node_Id;
      Name   : String)
     return Boolean;
```

Is_Defined_Float_Property: Return True if the aadlreal property named 'Name' is defined for the AADL entity 'Entity'.

```
function Is_Defined_Float_Property
     (Entity : Node_Id;
      Name   : String)
     return Boolean;
```

Is_Defined_Reference_Property: Return True if the component reference property named 'Name' is defined for the AADL entity 'Entity'.

```
function Is_Defined_Reference_Property
     (Entity : Node_Id;
      Name   : String)
     return Boolean;
```

Is_Defined_List_Property: Return True if the 'list of XXX' property named 'Name' is defined for the AADL entity 'Entity'.

```
function Is_Defined_List_Property
     (Entity : Node_Id;
      Name   : String)
     return Boolean;
```

Is_Defined_Enumeration_Property: Return True if the enumeration property named 'Name' is defined for the AADL entity 'Entity'.

```
function Is_Defined_Enumeration_Property
     (Entity : Node_Id;
      Name   : String)
     return Boolean;
```

Get_Value_Of_Property_Association: Return the value of the property association named 'Name' if it is defined defined for 'Entity'. Otherwise, return No_Node.

```
function Get_Value_Of_Property_Association
     (Entity : Node_Id;
      Name   : String)
     return Node_Id;
```

Get_String_Property: Return the value of the aadlstring property association named 'Name' if it is defined defined for 'Entity'. Otherwise, return "".

```
function Get_String_Property
     (Entity : Node_Id;
      Name   : String)
     return String;
```

Get_String_Property: Return the value of the aadlstring property association named 'Name' if it is defined defined for 'Entity'. Otherwise, return No_Name.

```
function Get_String_Property
     (Entity : Node_Id;
      Name   : String)
     return Name_Id;
```

Get_Integer_Property: Return the value of the aadlinteger property association named 'Name' if it is defined defined for 'Entity'. Otherwise, return 0.

```
function Get_Integer_Property
     (Entity : Node_Id;
      Name   : String)
     return Unsigned_Long_Long;
```

Get_Float_Property: Return the value of the aadlreal property association named 'Name' if it is defined defined for 'Entity'. Otherwise, return 0.0.

```
function Get_Float_Property
     (Entity : Node_Id;
      Name   : String)
     return Long_Long_Float;
```

Get_Boolean_Property: Return the value of the aadlboolean property association named 'Name' if it is defined defined for 'Entity'. Otherwise, return False.

```
function Get_Boolean_Property
     (Entity : Node_Id;
      Name : String)
     return Boolean;
```

Get_Reference_Property: Return the value of the component reference property association named 'Name' if it is defined for 'Entity'. Otherwise, return No_Node.

```
function Get_Reference_Property
     (Entity : Node_Id;
      Name   : String)
     return Node_Id;
```

Get_List_Property: Return the value of the 'list of XXX' property association named 'Name' if it is defined for 'Entity'. The returned list is a Node_Container list. Otherwise, return No_List.

```
function Get_List_Property
     (Entity : Node_Id;
      Name   : String)
     return List_Id;
```

Get_Enumeration_Property: Return the value of the enumeration property association named 'Name' if it is defined defined for 'Entity'. Otherwise, return "".

```
function Get_Enumeration_Property
     (Entity : Node_Id;
      Name   : String)
     return String;
```

Get_Enumeration_Property: Return the value of the enumeration property association named 'Name' if it is defined defined for 'Entity'. Otherwise, return No_Name.

```
function Get_Enumeration_Property
     (Entity : Node_Id;
      Name : String)
     return Name_Id;
```

Compute_Property_Value: Compute the value of a property value and return a Node_Id containing this value. This value does not contain any reference (value ()).

```
function Compute_Property_Value (Property_Value : Node_Id) return Node_Id;
```

6.1.5 API to build and manipulate AADL instances

This section presents the functions that allow the manipulation of AADL instances.

The routines that create the instance tree from the model tree are located in the `Ocarina.Expander` package. The main function that should be used is the `Ocarina.Expander.Expand_Model` function. It Expands the tree of the model and returns the expanded architecture. The first parameter given to this function is the root of the model tree. The second parameter designs the root system implementation, used when several system implementations are electable as root system.

6.1.6 Core parsing and printing facilities

6.1.6.1 Parser

The main parsing function is the `Ocarina.Parser.Parse` function. This function selects automatically the right parser depending on the file suffix: If the file suffix is ".aadl", then the parsing function used is `Ocarina.AADL.Parser.Process` and if the file suffix is ".dia", then the parsing function used is `Ocarina.Dia.Parser.Process`. Therefore, the input files given to the Parse function must have valid suffixes.

The return value of the `Ocarina.Parser.Parse` function is the node corresponding to the root of the tree. If something went wrong during the parsing, the return value is `No_Node`. The first parameter given to the `Ocarina.Parser.Parse` function is the name of the file to parse. The second parameter corresponds to the tree root, this way, it's possible to parse many files by calling the function several times and by giving to it the same root as second parameter. After each call, the returned value is the old tree to which are added the AADL entities of the last parsed file. Hence Ocarina supports multiple file AADL descriptions.

6.1.6.2 Printer

Unlike the parsing facility, the printing facility cannot be selected automatically. The user must precise which printer he wants to use.

The data structure `Ocarina.Printer.Output_Options` contains a field named `Printer_Name` which allows to select a registered printer. There are other fields in this structure, they allow to configure some printing options (output file, output directory...).

6.1.6.3 Using the parser and the printer

The user should not directly use the parsing and printing subprogram supplied by each module. To parse a file, the function that should be used is `Ocarina.Parser.Parse`. To print a file, the user must specify the printer options in a variable of type `Ocarina.Printer.Output_Options`, then call the `Ocarina.Printer.Print` function with the Root of the AADL tree and the options variable as parameters. Here is a sample code that shows how to initialize Ocarina, parse and print a set of AADL files. The example describes a classical way to use the Ocarina libraries to build a basic program that loops indefinitely and parses at each iteration all the AADL files given to its command line. If the parsing has been successful, the program prints the AADL sources corresponding to the built AADL syntax tree. The example illustrates also the *Reset* capabilities of Ocarina allowing to reinitialize all the Ocarina engines at the end of an iteration in order to use them in the incoming iteration.

```
------------------------------------------------------------------------------
--                                                                          --
--                           OCARINA COMPONENTS                             --
--                                                                          --
--                  P A R S E _ A N D _ P R I N T _ A A D L                  --
--                                                                          --
--                                B o d y                                   --
--                                                                          --
--                  Copyright (C) 2007, GET-Telecom Paris.                  --
--                                                                          --
-- Ocarina  is free software;  you  can  redistribute  it and/or  modify    --
-- it under terms of the GNU General Public License as published by the     --
-- Free Software Foundation; either version 2, or (at your option) any      --
-- later version. Ocarina is distributed  in  the  hope  that it will be    --
-- useful, but WITHOUT ANY WARRANTY;  without even the implied warranty of  --
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General --
-- Public License for more details. You should have received  a copy of the --
-- GNU General Public License distributed with Ocarina; see file COPYING.   --
-- If not, write to the Free Software Foundation, 51 Franklin Street, Fifth --
-- Floor, Boston, MA 02111-1301, USA.                                       --
--                                                                          --
-- As a special exception,  if other files  instantiate  generics from this --
-- unit, or you link  this unit with other files  to produce an executable, --
-- this  unit  does not  by itself cause  the resulting  executable to be   --
-- covered  by the  GNU  General  Public  License. This exception does not  --
-- however invalidate  any other reasons why the executable file might be   --
-- covered by the GNU Public License.                                       --
--                                                                          --
--                 Ocarina is maintained by the Ocarina team                --
```

```
--                      (ocarina-users@@listes.enst.fr)                      --
--                                                                           --
-------------------------------------------------------------------------------

with Ada.Command_Line;
with GNAT.OS_Lib;

with Types;
with Namet;

with Ocarina.Configuration;
with Ocarina.Parser;
with Ocarina.Printer;
with Ocarina.Analyzer;

procedure Parse_And_Print_AADL is

   use Types;
   use Namet;

   Root              : Node_Id;
   Printer_Options   : Ocarina.Printer.Output_Options :=
     Ocarina.Printer.Default_Output_Options;
   Analysis_Options  : constant Ocarina.Analyzer.Analyzer_Options :=
     Ocarina.Analyzer.Default_Analyzer_Options;
   Success           : Boolean;

begin
   loop
      --  Initialization step

      Ocarina.Initialize;
      Ocarina.Configuration.Init_Modules;

      --  Parse the aadl source file, the right parser is selected
      --  automatically depending on the file suffix. It is important
      --  that the root node is set to No_Node at the very beginning
      --  or parsing.

      Root := No_Node;

      for J in 1 .. Ada.Command_Line.Argument_Count loop
         --  Parse the file corresponding to the Jth argument opf the
         --  commant line and enrich the global AADL tree.

         Root := Ocarina.Parser.Parse (Ada.Command_Line.Argument (J), Root);
      end loop;

      --  If something went wrong, Root = No_Node

      if Root /= No_Node then
         --  Analyze the tree

         Success := Ocarina.Analyzer.Analyze_Tree (Root, Analysis_Options);

         if Success then
            --  Select the printer

            Set_Str_To_Name_Buffer ("aadl");
            Printer_Options.Printer_Name := Name_Find;

            --  Print to an AADL File

            Success := Ocarina.Printer.Print
              (Root    => Root,
```

```
                Options => Printer_Options);
            else
               GNAT.OS_Lib.OS_Exit (1);
            end if;
         else
            GNAT.OS_Lib.OS_Exit (1);
         end if;

         --  Pause for 1 second

         delay 1.0;

         --  Reset step

         Ocarina.Configuration.Reset_Modules;
         Ocarina.Reset;
      end loop;

   end Parse_And_Print_AADL;
```

6.2 Input/Output Modules

At the time this documentation is being written, Ocarina can handle two kinds of files: AADL source files and Dia source files. For each kind, a parser and a printer are provided.

- **AADL** The `Print_Node` procedure provided by this module allows to print a node and its subnodes. The result is an AADL source file. The printer name associated to this module is `aadl`.
- **Dia** The `Print_Node` procedure provided by this module generate a Dia file which contains the graphical description of the tree. The printer name associated to this module is `dia`
- **Dumper** This module is used for debugging purpose. Its printing facility allows to dump the AADL tree or the AADL instance tree. The printer names associated to this module are `aadl_tree_p` for the parsed tree and `aadl_tree_e` for the expanded tree (instance tree).

6.3 Gaia, an Application Generator

Gaia is an application generator. It processes AADL descriptions and produces the distribution infrastructure (source code and Makefiles) to deploy and run applications. The architectural part of the distributed application is described by the AADL code. The functional part is described by Ada source code. This source code has to be specified within the AADL components so that Gaia can integrate it when deploying the distributed application.

A distributed application consists of nodes that communicate between each other. A node corresponds generally to a process. For each node in the distributed application, a code is generated.

7 Ada Mapping Rules

We choose to use a middleware in order to ensure the communication between the nodes of the distributed application is the schizophrenic middleware PolyORB. It's obvious that a large part of the code (thread creation for example) is the same for distributed application. This code is written once and used as the middleware API. The use of PolyORB implies that communications between the application node are performed by requests and rely on an ORB (Object Request Broker). A full description of the Middleware API is given in Section 7.4 [Description of the Middleware API], page 58.

The present chapter defines the mapping Gaia use to generate Ada code and PolyORB primitives.

7.1 Components mapping rules

The unnamed namespace of an AADL description is mapped to a conventional Ada package called `Namespaces`. The several namespaces (which are children of the unnamed namespace) are mapped to subpackages of the `Namespaces` package. For each node of the distributed application, a `Namespaces` package "family" is generated; it contains all the data and subprograms mappings that are used by this node.

7.1.1 Data components mapping

7.1.1.1 Base type mapping

A subcomponent-free `data` component should contain a `ARAO::data_type` property in order to generate the corresponding Ada type. ARAO predefined types are : `integer`, `float`, `null`, `string` and `boolean`. Normal data components are mapped to the related Ada type, as seen in the following AADL example :

```
data message
properties
  ARAO::data_type => integer;
end message;
```

is mapped to the following Ada95 code:

```
type message is new Integer;
```

7.1.1.2 Composed type mapping

AADL `data` component implementations may contain others `data` subcomponents. In this case, the `data` component is mapped to an Ada record type.

As an example, the following AADL component implementation:

```
data integer_type
properties
  ARAO::data_type => integer;
end integer_type;

data structure
end structure;

data implementation structure.impl
subcomponents
  d1 : data integer_type;
  d2 : data integer_type;
end structure.impl;
```

is mapped to the following Ada code :

```
type Integer_Type is new Integer;

type Structure_Impl is
   record
  D1 : Integer_Type;
  D2 : Integer_Type;
   end record;
```

7.1.1.3 Protected type mapping

AADL protected `data` components must be declared in the same way composed types are, i.e. by encapsulating them within another AADL type declaration.

For each protected `data` components, a new type is declared which contains all data components (i.e. fields of the related composed type) plus a mutex object within a Ada record. As for composed types, all fields must be either a previously user-declared type or a ARAO base type. Accessors and building features for the type will be declared too, as for the data-owned procedures ("methods") designated in the `features` part of the `data` component.

The generated code enforces access protection, and declare type's object-oriented procedures ("methods", as defined by user), using the middleware mutexes. A "method" of a type must always has as `features` a `requires data access` on this data.

For example, the following AADL declaration :

```
data internal_message
properties
  ARAO::data_type => integer;
end internal_message;

data message
subcomponents
  Field : data internal_message;
features
  method : subprogram update;
properties
  ARAO::Access_Control_Protocol => Protected_Access;
end message;
```

would generate a code like this :

```
package Partition is

   type Message is private;

   procedure Build
     (This : out Message);

   procedure Get_Data
     (This  : in Message;
      Value : out Internal_Message);

   procedure Set_Data
     (This  : in out Message;
      Value : in Internal_Message);

private

   type Message is
```

```
      record
         Data  : Partition.Internal_Message;
         Mutex : PolyORB.Tasking.Mutexes.Mutex_Access;
      end record;

end Partition;
```

and the `update` method which will be call by generated code is like this :

```
procedure Update
  (This  : in out Message;
   Value : in Partition.Internal_Message) is
begin
   PolyORB.Tasking.Mutexes.Enter (This.Mutex);
   Repository.Update (This, Value);
   PolyORB.Tasking.Mutexes.Leave (This.Mutex);
end Update;
```

Where the procedures `Enter` and `Leave` are middleware mutexes' `take` and `release` procedures.

We define *protected data type internal type* as the components (usually only one) , excluding the mutex, which are embedded in a protected type as a `subcomponent`. The *protected data type internal type* could be either a protected type or a "normal" (non-protected) type. Eventually, all protected types can be decomposed in a set of basic types.

7.1.1.4 Accessor usage

Data accessors can be used by the user exactly as data-owned procedure are. In the current version, they are the only ones actually called by Gaia runtime, contrary to the generated interfaces which are not called at all.

Protected type accessors include `Set_X` and `Get_X` Ada procedures, where `X` is the name of the Field which contains the real (internal) data type. Those procedures are access-protected, using the protected object's middleware mutex to ensure mutual exclusion. The `Build` procedure will ensure mutex initialization.

An example of safe usage of accessors is :

```
-----------------------
-- Concurrent_Update --
-----------------------

procedure Concurrent_Update (arg : in out Partition.Counter_Type)
is
   Sum  : Partition.Integer_Type;
begin
   --  data initialization
   Partition.Set_Field (Data, 0);

   for I in 0 .. 100 loop
      Partition.Increment (Data);
      Partition.Get_Field (Data, Sum);
      if Integer (Sum) = 100 then
         exit;
      end if;
   end loop;
end Concurrent_Update;
```

relying on the following AADL declarations :

```
data Integer_Type
```

```
properties
  ARAO::data_type => integer;
end Integer_Type;

data Counter_Type
features
  Increment : subprogram Increment;
subcomponents
  field : data Integer_Type;
properties
  Concurency_Control_Protocol => Protected_Access;
end Counter_Type;

subprogram Increment
features
  this : requires data access Counter_Type;
properties
  source_language => Ada95;
  source_name => "Repository";
end Increment;

subprogram Concurrent_Update
features
  arg : requires data access Counter_Type;
properties
  source_language => Ada95;
  source_name => "Repository";
end Concurrent_Update;

thread Task
features
  sh_data : requires data access Counter_Type;
properties
  Dispatch_Protocol => Periodic;
  Period => 1000 Ms;
end Task;

thread implementation Task.impl
calls {
  sp1 : subprogram Concurrent_Update;
};
connections
  Cnx_Th_dat : data access sh_data -> sp1.arg;
end Task.impl;

process implementation global.impl
subcomponents
  th1 : thread Task.impl;
  th2 : thread Task.impl;
  dat : data Counter_Type;
connections
  Cnx_1 : data access dat -> th1.sh_data;
  Cnx_2 : data access dat -> th2.sh_data;
end global.impl;
```

with the following package specification and body generated :

```
package Partition is

   type Integer_Type is new Integer;
   type Counter_Type is private;

   procedure Build
     (This : out Partition.Integer_Type);
```

```
   procedure Get_Field
     (This  : in Message;
      Value : out Partition.Integer_Type);

   procedure Set_Field
     (This  : in out Message;
      Value : in Partition.Integer_Type);

   procedure Increment
     (This : in out Counter_Type);

private
   type Counter_Type is
      record
         Field : Partition.Integer_Type;
         Mutex : PolyORB.Tasking.Mutexes.Mutex_Access;
      end record;

end Partition;



with Repository;

package body Counter_Type_PKG is

   -----------
   -- Build --
   -----------

   procedure Build
     (This : out Message)
   is
   begin
      --  Initialize the middleware's mutex
      PolyORB.Tasking.Mutexes.Create (T.Mutex);
   end Build;

   ---------------
   -- Get_Field --
   ---------------

   procedure Get_Field
     (This  : in Counter_Type;
      Value : out Partition.Integer_Type)
   is
   begin
      PolyORB.Tasking.Mutexes.Enter (T.Mutex);
      Value := This.Field;
      PolyORB.Tasking.Mutexes.Leave (T.Mutex);
   end Get_Field;

   ---------------
   -- Set_Field --
   ---------------

   procedure Set_Field
     (This  : in out Counter_Type;
      Value : in Partition.Integer_Type)
   is
   begin
      PolyORB.Tasking.Mutexes.Enter (T.Mutex);
      This.Field := Value;
      PolyORB.Tasking.Mutexes.Leave (T.Mutex);
   end Set_Field;
```

```
   ---------------
   -- Increment --
   ---------------

   procedure Increment
     (This : in out Counter_Type)
   is
   begin
      PolyORB.Tasking.Mutexes.Enter (This.Mutex);
      Repository.Increment (This.Field);
      PolyORB.Tasking.Mutexes.Leave (This.Mutex);
   end Increment;

end Counter_Type_PKG;
```

Note that the `Set` usage could had been replaced by an `Initialization` method of Counter_Type, and that the `Get` could had been replaced by a `Test_Value` method.

7.1.1.5 Middleware mapping

We have seen that in the translation phase, the AADL data components are mapped to Ada95 types. Since the communication between nodes is performed using the PolyORB tools, all data sent in a request must have the neutral type `PolyORB.Any.Any`. So, conversion functions from and to this neutral type must be generated. For a process named `proc` these conversion functions will be generated in the `proc_Helpers` package. Example:

```
data message
properties
  ARAO::data_type => integer;
end message;
```

is a definition for an integer type, the conversion routines generated in `proc_Helpers` are:

```
with Partition;
with PolyORB.Any;

package proc_helpers is
   --  TypeCode variable used to characterize an Any variable

   TC_message : PolyORB.Any.TypeCode.Object :=
      PolyORB.Any.TypeCode.TC_Alias;

   function From_Any (Item : in PolyORB.Any.Any) return Partition.message;

   function To_Any (Item : in Partition.message) return PolyORB.Any.Any;

end proc_helpers;
```

Note that we use the `Namespaces` package created in the translation phase.

7.1.1.6 AADL Properties support

Available properties for data components can be found in SAE AS5506, in 5.1 page 50 and in Appendix A, pages 197-218.

Concurency_control_protocol	Supported : None, Access_Protected
Not_Collocated	Not Supported
Provided_Access	Not Supported
Required_Access	Not Supported
Source_Code_Size	Not Supported

Source_Language	Not Supported
Source_Name	Not Supported
Source_Text	Not Supported
Type_Source_Name	Not Supported

7.1.2 Subprogram components mapping

AADL subprograms are mapped to Ada procedures. In case of data-owned subprograms, they are managed in the related generated package, as seen in Section 7.1.1 [Data components mapping], page 35. The parameters of the procedure are mapped from the subprogram features with respect to the following rules:

- The parameter name is mapped from the parameter feature name
- The parameter type is mapped from the parameter feature data type as specified in Section 7.1.1 [Data components mapping], page 35
- The parameter orientation is the same as the feature orientation ("in", "out" or "in out").

The body of the mapped procedure depend on the nature of the subprogram component. Subprogram components can be classified in many kind depending on the value of the `Source_Language`, `Source_Name` and `Source_Text` standard AADL properties and the existence or not of call sequences in the subprogram implementation. There are four kinds of subprogram components:

1. The empty subprograms.
2. The opaque subprograms.
3. The pure call sequence subprograms.
4. The hybrid subprograms.

7.1.2.1 Mapping of empty subprograms

Empty subprograms correspond to subprograms for which there is neither `Source_Language` nor `Source_Name` nor `Source_Text` values nor call sequences. Such kind of subprogram components has no particular utility. For example:

```
subprogram sp
features
  e : in parameter message;
  s : out parameter message;
end sp;
```

is en empty subprogram. A possible Ada implementation for this subprogram could be:

```
procedure sp (e : in message; s : out message) is
  NYI : exception;
begin
  raise NYI;
end sp;
```

7.1.2.2 Mapping of opaque subprograms

Opaque subprograms are the simplest "useful" subprogram components (in code generation point of view). For these subprograms, the `Source_Language` property indicates the programming language of the implementation (C or Ada95). The `Source_Name` property indicates the name of the subprogram implementing the subprogram:

- for Ada95 subprograms, the value of the `Source_Name` property is the **fully qualified name** of the subprogram (e.g. `My_Package.My_Spg`). If the package is stored in a file named

according to the GNAT Ada compiler convensions, there is no need to give a `Source_Text` property for Ada95 subprograms. Otherwise the `Source_Text` property is necessary for the compiler to fetch the implementation files.

- for C subprograms, the value of the `Source_Name` property is the **name** of the C subprogram implementing the AADL subprogram. The `Source_Text` is mandatory for this kind of subprogram and it must give one of the following informations:
  - the path to the `.c` source file that contains the implementation of the subprogram.
  - the path to one or more precompiled object files (`.o`) that implment the AADL subprogram.
  - the path to one or more precompiled `C` library (`.a`) that implment the AADL subprogram.

  These information can be used toghether, for example may give the `C` source file that implements the AADL subprogram, an object file that contains entities used by the `C` file and a library that is necessary to the `C` sources or the objects.

In this case, the code generation consist of creating a shell for the implementation code. In the case of Ada subprograms, the generated subprogram renames the implementation subprogram (using the Ada95 renaming facility). Example:

```
subprogram sp
features
  e : in  parameter message;
  s : out parameter message;
end sp;

subprogram implementation sp.impl
properties
  Source_Language => Ada95;
  Source_Name     => "Repository.Sp_Impl";
end sp.impl;
```

The generated code for the `sp.impl` component is:

```
with Repository;
...
procedure sp_impl (e : in message; s : out message)
  renames Repository.Sp_Impl;
```

The code of the `Repository.sp_impl` procedure is provided by the architecture and must be conform with the `sp.impl` signature. The coherence between the two subprograms will be verified by the Ada95 compiler.

The fact that the hand-written code is not inserted in the generated shell allows this code to be written in a programming language other than Ada95. Thus, if the implementation code is C we have this situation:

```
subprogram sp
features
  e : in parameter message;
  s : out parameter message;
end sp;

subprogram implementation sp.impl
properties
  Source_Language => C;
  Source_Name     => "implem";
end sp.impl;
```

The `Source_Name` value is interpreted as the name of the C subprogram implementing the AADL subprogram. The generated code for the `sp.impl` component is:

```
procedure sp_impl (e : in message; s : out message);
pragma Import (C, sp_impl, "implem");
```

This approach will allow us to have a certain flexibility by separating the generated code and the hand-written code. We can modify the AADL description without affecting the hand-written code (the signature should not be modified of course).

7.1.2.3 Mapping of pure call sequence subprograms

In addition to the opaque approach which consist of delegating all the subprogram body writing to the user, AADL allows to model subprogram as a pure call sequence to other subprograms. Example:

```
subprogram spA
features
  s : out parameter message;
end spA;

subprogram spB
features
  s : out parameter message;
end spA;

subprogram spC
features
  e : in  parameter message;
  s : out parameter message;
end spA;

subprogram spA.impl
calls {
  call1 : subprogram spB;
  call2 : subprogram spC;};
connections
  cnx1 : parameter call1.s -> call2.e;
  cnx2 : parameter call2.s -> s;
end spA.impl;
```

In this case, the subprogram connects together a number of other subprograms. In addition to the call sequence, the connections clause completes the description by specifying the connections between parameters. The pure sequence call model allows to generate complete code : the calls in the call sequence corresponds to Ada95 procedure calls and the connections between parameters correspond to eventual intermediary variables. The Ada95 code generated for the subprogram `spA.impl` is:

```
procedure spA_impl (s : out message) is
   cnx1 : message;
begin
   spB (cnx1);
   spC (cnx1, s);
end spA_impl;
```

Note that in case of pure call sequence subprograms, the AADL subprogram must contain only one call sequence. If there are more than one call sequence, it's impossible - in this case - to determine the relation between them.

7.1.2.4 Mapping of hybrid subprograms

The two last kinds of subprogram components describe even an opaque implementation for which all the functional part is written by the user or a pure call sequence for which all the functional part is given by the AADL description. These two cases are relatively simple to implement. However, they don't offer much flexibility. In the general case we want to integrate the maximum of information within the AADL description in order to get an easy assembling of the distributed application components. However, AADL does not provide control structures (conditions, loops). The best way is to combine the opaque model and the pure call sequence model.

To illustrate the problem, let's consider the following example: A subprogram `spA` receives an input integer value . The subprogram behavior depends on the a value:

- If $a < 4$, then a is given to another subprogram `spB`;
- Else, `spA` calls a third subprogram called `spC` which give its return value to `spB`

In all cases, the return value of `spB` is given to a forth subprogram `spD`; the return value of `spD` is returned by `spA`.

The behavior of `spA` is illustrated by this algorithm:

```
if a < 4 then
  b <- spB (a)
else
  c <- spC ()
  b <- spB (c)
end if

d <- spD (b)
return d
```

We assume that the subprograms `spB`, `spC` and `spD` are correctly defined.

We have three call sequences. AADL allows only to describe the architectural aspects of the algorithm (the connections between the different subprograms). The AADL source corresponding to the last example is:

```
data int
properties
  GAIA::Data_Type => Integer;
end int;

subprogram spA
features
  a : in parameter int;
  d : out parameter int;
end spA;

subprogram spB
features
  e : in parameter int;
  s : out parameter int;
end spB;

subprogram spC
features
  s : out parameter int;
end spC;

subprogram spD
features
  e : in parameter int;
```

```
  s : out parameter int;
end spD;

subprogram implementation spA.impl
properties
  Source_Language => Ada95;
  Source_Name     => "Repository.SpA_Impl"
calls
  seq1 : {spB1 : subprogram spB;};
  seq2 : {spC2 : subprogram spC;
          spB2 : subprogram spB;};
  seq3 : {spD3 : subprogram spD;};
connections
  cnx1 : parameter a -> apB1.e;
  cnx2 : parameter spB1.s -> spD3.e;

  cnx3 : parameter spC2.s -> spB2.e;
  cnx4 : parameter spB2.s -> spD3.e;

  cnx5 : parameter spd3.s -> d;
end spA.impl;
```

The first remark is that the subprogram implementation contains at the same time the `Source_[Language|Name]` (and a possible `Source_Text`) properties and call sequences. The hand-written code describes the algorithm. This algorithm should be able to handle each call sequence as being a block and must be as simple as possible: the user should not know the content of the call sequence.

The generated code for each block (call sequence) is almost identical to the generated code for pure call sequence. For each block, a subprogram is generated. To make things simple for the user, these subprograms have the same signature (one parameter called Status):

```
type SpA_Impl_Status is record
  a, b, c, d : int;
end record;

procedure SpA_Seq1 (in out Status : spA_impl_Status) is
begin
   spB (Status.a, Status.b);
end SpA_Seq1;

procedure SpA_Seq2 (in out Status : spA_impl_Status) is
begin
   spC (Status.c);
   spB (Status.c, Status.b);
end SpA_Seq2;

procedure SpA_Seq3 (in out Status : spA_impl_Status) is
begin
   spD (Status.b, d);
end SpA_Seq3;
```

The generated code for the `spA.impl` subprogram is very simple:

```
procedure SpA_Impl (a : in int; d : out int) is
   Status : spA_impl_Status;
begin
   Status.a := a;
   Repository.SpA_Impl
     (Status,
      SpA_Seq1'Access,
      SpA_Seq2'Access,
```

```
      SpA_Seq3'Access);
   d := Status.d;
end SpA_Impl;
```

The subprogram which describes the algorithm and which should be written by the user is relatively simple, and does not require any knowledge of the call sequences contents:

```
type SpA_Impl_Call_Sequence is access
  procedure (in out Status : spA_impl_Status);

procedure SpA_Impl
  (Status : in out spA_impl_Status,
   seq1   : spA_impl_Call_Sequence,
   seq2   : spA_impl_Call_Sequence,
   seq3   : spA_impl_Call_Sequence)
is
begin
   if Status.a > 4 then
      seq1.all (Status);
   else
      seq2.all (Status);
   end if;
   seq3.all (Status);
end SpA_Impl;
```

7.1.2.5 Data access

If a subprogram has a `requires access` feature to a data, this data is added to the parameters list, with the mode corresponding to data access rights (i.e. `read-only => in`, `write-only => out` and `read-write => in out`).

In the specific case of subprograms requiring protected data access, user should provides different data depending on subprograms' nature.

If the subprogram is a "method" of the protected object (i.e. if it appears in its `features` field), then the user should provides an implementation of the subprogram which take the subprogram access as the first parameter, with the mode chosen following the rule described above. The parameter's name must always be `this`. This parameter type must always be of the protected data type internal type (cf. Section 7.1.1 [Data components mapping], page 35).

If the subprogram is a not "method" of the protected object, user work depends of the accessed data's `Actual_Lock_Implementation` property, which defines shared variables update policy. This policy could be either synchronous (`synchronous_lock`) or asynchronous (`asynchronous_lock`). Default is asynchronous update policy.

The user must write a subprogram implementation complying to the following rules :

- For each *asynchronous policy*-defined data accessed, add an parameter at beginning of the data's protected type.
- For each *synchronous policy*-defined data accessed, add an parameter at beginning of the subprogram's parameter list of the data's protected type internal type.

Note that accessed data (found in the subprogram component's `features` field) must always be parsed in the same order they are declared in the AADL specification. In any case, mode is still chosen accordingly to the rule describe above.

Note that only opaque subprograms currently support synchronous data update policy.

If synchronous policy is chosen for a data update policy, the user should be aware that access protection is ensured by the runtime code (cf. Section 7.1.3 [Thread components mapping], page 48).

Here is an example of data-owned specification of a protected object :

```
data internal_data
properties
  ARAO::data_type => integer;
end internal_data;

data shared_data
features
  method : subprogram update;
properties
  Concurrency_Control_Protocol => Protected_Access;
  ARAO::Actual_Lock_Implementation => Synchronous_Lock;
end shared_data;

data implementation shared_data.i
subcomponents
  Field : data internal_data;
end shared_data.i;

--  subprograms

subprogram update
features
  this : requires data access shared_data.i;
properties
  source_language => Ada95;
  source_name => "Repository";
end update;
```

The user provides :

```
  procedure Update (Field : in out Partition.Internal_Data;
                    I : in Partition.message);

   ------------
   -- Update --
   ------------

   procedure Update (Field : in out Partition.Internal_Data;
                     I : in Partition.message)
   is
      use Partition;
   begin
      Field := Partition.Internal_Data (Integer (Field) + Integer (I));
   end Update;
```

And Gaia will generate the following implementation for the access-protected subprogram :

```
------------
-- update --
------------

procedure Update
  (This : in out Partition.Shared_Data_I;
   I : Partition.Message)
is
begin
   PolyORB.Tasking.Mutexes.Enter
     (This.Mutex);
   Repository.Update
     (Field => This.Field,
      I => I);
   PolyORB.Tasking.Mutexes.Leave
```

```
      (This.Mutex);
end Update;
```

7.1.2.6 AADL Properties support

Available properties for subprogram components can be found in SAE AS5506, in 5.2 page 56 and in Appendix A, pages 197-218.

Actual_Memory_Binding	Not Supported
Actual_Subprogram_Call	Not Supported
Client_Subprogram_Execution_Time	Not Supported
Compute_Deadline	Not Supported
Compute_Execution_Time	Not Supported
Concurrency_Control_Protocol	Not Supported
Queue_Processing_Protocol	Not Supported
Queue_Size	Not Supported
Recover_Deadline	Not Supported
Recover_Execution_Time	Not Supported
Server_Subprogram_Call_Binding	Not Supported
Source_Code_Size	Not Supported
Source_Data_Size	Not Supported
Source_Heap_Size	Not Supported
Source_Stack_Size	Not Supported
Source_Language	Supported (Ada)
Source_Name	Supported
Source_Text	Supported

7.1.3 Thread components mapping

The mapping of thread components is a little bit more complicated than the mapping of data components. Threads are mapped to an Ada95 parameter-less procedure which executes the thread work (periodically or aperiodically depending on the thread nature). For each periodic thread, a middleware thread is created using the API described in Section 7.4 [Description of the Middleware API], page 58. For example~:

```
thread sender
features
  msg_out : out event data port message;
properties
  Dispatch_Protocol => Periodic;
  Period => 1000 Ms;
end sender;
```

7.1.3.1 Servant mapping

If this thread belongs to a process `proc`, and if `th1` is the name of the thread subcomponent of `proc` having the type `sender`, then a package `proc_Servants` is created:

```
package proc_Servants is
   ...
   procedure th1_Ctrler;
   ...
end proc_Servants;
```

In the main subprogram `proc` we find:

```
Aadl_Periodic_Threads.Create_Periodic_Thread
  (TP => sn_Servants.th1_Ctrler'Access);
```

The thread "in" or "in out" ports are mapped in an Ada protected object which allows a protected access to these ports. For each port, a buffer having the port stack size is created, implmented with a cyclic array. Since these ports are the destination of other components requests, for each in port, a PolyORB Reference is created and for each thread containing in ports, a servant is created to handle the incoming requests; Example:

```
thread receiver
features
  msg_in : in event data port message;
end receiver;
```

If this thread belongs to a process `proc`, and if `th2` is the name of the thread subcomponent of `proc` having the type `receiver`, then the following declarations will be generated in the `proc_Servants` package spec:

```
with Partition;

with PolyORB.Components;
with PolyORB.Servants;
with PolyORB.References;

package proc_Servants is
   ...
   procedure th2_Ctrler;

   type th2_Object is new Servant with null record;

   th2_Ref : PolyORB.References.Ref;

   function Execute_Servant
     (Obj : access th2_Object;
      Msg : PolyORB.Components.Message'Class)
     return PolyORB.Components.Message'Class;

   type th2_msg_in_buf_type is array (1 .. 1) of Partition.message;

   protected th2_Ports is
      procedure Put_msg_in (msg_in : Partition.message);
      procedure Get_msg_in (msg_in : out Partition.message);
      procedure Push_Back_msg_in (msg_in : out Partition.message);
   private
      msg_in_Buf : Th2_Msg_In_Buf_Type.Table;
   end th2_Ports;
   ...
end proc_Servants;
```

For each "out" or "in out" port, we declare reference variable for each "in" or "in out" port connected to this port.

7.1.3.2 Shared variables access

In order to comply to the AADL *input-processing-output* algorithm, shared data (either access-protected or not) are not read or written directly, but through temporary variables.

As seen in Section 7.1.4 [Process components mapping], page 51, any thread can access shared variables. In order to ensure protected access when needed, Gaia will declare a local variable in the `thread_controler` function, whose type is the variable internal type (if the variable has the protected access property) or the variable real type.

Each time the thread controller is activated (i.e. each time the related servant is called), the local variable is put to shared variable value by its `Setter` procedure, then processing is done using the proper user-defined procedure. Then the `Getter` is used to update the shared variable.

Note that both `Setter` and `Getter` procedures are generated by Gaia and ensure access protection, as described in Section 7.1.1 [Data components mapping], page 35.

Here is an example of generated code of the `thread_controler` procedure which manage a `mem_sh` variable.

```
procedure Th1_Controller is
   Msg_In : Partition.Message;
   Msg_In_Present : Standard.Boolean;
   Msg_Out : Partition.Message;

   -- local temporary variable definition
   Mem : Partition.Internal_Data;
begin
   --  Read shared data and store it in local variable
   Partition.Get_Field (Sh_Mem, Mem);

   --  Read in IN ports
   Tr_Servants.Th1_IN_Ports.Get_Msg_In
     (Msg_In,
      Msg_In_Present);
   if (True
       and then Msg_In_Present)
   then
      --  Processing local variable
      Repository.Transmit_Message
        (Msg_In => Msg_In,
         Msg_Out => Msg_Out,
         Mem => Mem);

      --  Write in OUT ports
      ARAO.Requests.Emit_Msg
        (Tr_Helpers.To_Any
         (Msg_Out),
         Tr_Th2_Ref,
         "msg_in");
   else
      if Msg_In_Present
      then
         Tr_Servants.Th1_IN_Ports.Push_Back_Msg_In (Msg_In);
      end if;
   end if;

   --  Write back local variable into shared data
   Partition.Set_Field (Sh_Mem, Mem);
end Th1_Controller;
```

7.1.3.3 AADL Properties support

Available properties for thread components can be found in SAE AS5506, in 5.3 page 61 and in Appendix A, pages 197-218.

Activate_Deadline	Not Supported
Activate_Execution_Time	Not Supported
Activate_Entrypoint	Not Supported
Active_Thread_Handling_Protocol	Not Supported
Active_Thread_Queue_Handling_Protocol	Not Supported

Actual_Connection_Binding	Not Supported
Actual_Memory_Binding	Not Supported
Actual_Processor_Binding	Not Supported
Allowed_Connection_Protocol	Not Supported
Client_Subprogram_Execution_Time	Not Supported
Compute_Deadline	Not Supported
Compute_Execution_Time	Not Supported
Concurrency_Control_Protocol	Not Supported
Deactivate_Deadline	Not Supported
Deactivate_Execution_Time	Not Supported
Deactivate_Entrypoint	Not Supported
Deadline	Not Supported
Dispatch_Protocol	Supported (Periodic, Aperiodic)
Finalize_Deadline	Not Supported
Finalize_Execution_Time	Not Supported
Finalize_Entrypoint	Not Supported
Initialize_Deadline	Not Supported
Initialize_Execution_Time	Not Supported
Initialize_Entrypoint	Not Supported
Not_Collocated	Not Supported
Period	Supported
Queue_Size	Not Supported
Recover_Deadline	Not Supported
Recover_Execution_Time	Not Supported
Server_Subprogram_Call_Binding	Not Supported
Source_Code_Size	Not Supported
Source_Data_Size	Not Supported
Source_Heap_Size	Not Supported
Source_Stack_Size	Supported
Source_Name	Not Supported
Source_Text	Not Supported
Source_Language	Not Supported
Synchronized_Component	Not Supported

7.1.4 Process components mapping

The main component in this phase is the `process` component. The distributed application is a set of processes which communicate between each other. Each `process` is mapped to an Ada95 main subprogram which leads to an executable after being compiled.

7.1.4.1 Shared variables declaration and initialization

In the case where a `process` contains shared variables declaration (which should always refers to local `data` components, as Gaia does not support variables shared amongst multiples process), a variable is declared in the '`proc_servant`' body package.

If the shared variable has a protected access property, Gaia will also add a `initialize` procedure to the package, and set it as the package initialization procedure for the middleware, which will ensure that it is ran before any usage of the package. This procedure calls protected type's `Build` interface (cf. Section 7.1.1 [Data components mapping], page 35), initializing middleware's mutexes.

Note that shared variables (either protected or not) are visible from any thread of the process. How those variables are accessed and updated is described in Section 7.1.3 [Thread components mapping], page 48.

Here is a AADL specification for declaring a data shared between two threads, with protected access in a process:

```
--  protected data type declaration

data internal_data
properties
  ARAO::data_type => integer;
end internal_data;

data shared_data
properties
  Concurrency_Control_Protocol => Protected_Access;
end shared_data;

data implementation shared_data.i
subcomponents
  Field : data internal_data;
end shared_data.i;

--  Process declaration

process transmitter_node
features
  msg_in : in event data port message;
  msg_out : out event data port message;
end transmitter_node;

process implementation transmitter_node.complex
subcomponents
  th1 : thread transmitter.simple;
  th2 : thread transmitter.simple;
  sh_mem : data shared_data.i;
connections
  event data port msg_in -> th1.msg_in;
  event data port th1.msg_out -> th2.msg_in;
  event data port th2.msg_out -> msg_out;
  data access sh_mem -> th1.mem;
  data access sh_mem -> th2.mem;
end transmitter_node.complex;
```

and here is the related code generated by Gaia :

```
package body Tr_Servants is

   --  Shared variable declaration

   Sh_Mem : Partition.Shared_Data_I;

   --  Initialization procedure declaration and description

   procedure Initialize;

   ----------------
   -- Initialize --
   ----------------

   procedure Initialize is
   begin
      Partition.Builder
        (Sh_Mem);
   end Initialize;

   --  Threads-related code
   --  (...)
```

```
   --  Bind initialization function with middleware initialization
begin
   declare
      use PolyORB.Utils.Strings;
      use PolyORB.Utils.Strings.Lists;
   begin
      PolyORB.Initialization.Register_Module
        (PolyORB.Initialization.Module_Info'
         (Name => + "tr_Servants",
          Conflicts => PolyORB.Utils.Strings.Lists.Empty,
          Depends => + "any",
          Provides => PolyORB.Utils.Strings.Lists.Empty,
          Implicit => False,
          Init => Initialize'Access,
          Shutdown => null));
   end;
end Tr_Servants;
```

7.1.4.2 AADL Properties support

Available properties for process components can be found in SAE AS5506, in 5.5 page 77 and in Appendix A, pages 197-218.

Active_Thread_Handling_Protocol	Not Supported
Active_Thread_Queue_Handling_Protocol	Not Supported
Actual_Connection_Binding	Not Supported
Actual_Memory_Binding	Not Supported
Actual_Processor_Binding	Supported
Allowed_Connection_Protocol	Not Supported
Deadline	Not Supported
Load_Deadline	Not Supported
Load_Time	Not Supported
Not_Collocated	Not Supported
Period	Not Supported
Runtime_Protection	Not Supported
Server_Subprogram_Call_Binding	Not Supported
Source_Code_Size	Not Supported
Source_Data_Size	Not Supported
Source_Stack_Size	Supported
Source_Name	Not Supported
Source_Text	Not Supported
Source_Language	Not Supported
Synchronized_Component	Not Supported

7.2 Setup of the application

In order for each executable to work correctly, the middleware must be properly set up. In the case of PolyORB, we used an API named ARAO (AADL Runtime API for Ocarina). the setup consists in two phases :

- adding `with` and `pragma` clauses to initialize the middleware parameters.
- build Portable Object Adapters for each in port.

The nature of these with clauses depends on these factors:

- The number of threads in the node
- The presence or not of periodic threads

The setup is done by including (`with`) static or generated packages. Those packages can be divided into three classes :

- Basic setup package, which are called by all process.
- Tasking package, which are either no_tasking (only one thread in the process) or full_tasking (more than one thread in the process).
- Object Adapter setup package, which can be either static (if no priorities management has been set in AADL description) or generated.

Example:

```
process proc
features
  msg_in : in event data port message;
  msg_out : out event data port message;
end proc;

process implementation proc.simple
subcomponents
  th1 : thread sender.simple;
  th2 : thread receiver.simple;
connections
  event data port msg_in -> th2.msg_in;
  event data port th1.msg_out -> msg_out;
end proc.simple;
```

The process above contains more than one thread, so the Middleware need to be set up in a multitask mode. The execution of a particular node follows this order: first, it put the information concerning its ports in the middleware memory, then collects the information on the other processes (to which it is connected).

The code of the `proc` process is:

```
with PolyORB.Initialization;
with Sn_Servants;
with ARAO.Utils;
with ARAO.Periodic_Threads;
with ARAO.RT_Obj_Adapters;
with PolyORB.Setup;
with PolyORB.ORB;

--  Runtime configuration
with ARAO.Setup.Application;
pragma Warnings (Off, ARAO.Setup.Application);
pragma Elaborate_All (ARAO.Setup.Application);

--  Full tasking mode
with ARAO.Setup.Tasking.Full_Tasking;
pragma Warnings (Off, ARAO.Setup.Tasking.Full_Tasking);
pragma Elaborate_All (ARAO.Setup.Tasking.Full_Tasking);

with ARAO.Periodic_Threads;
with ARAO.RT_Obj_Adapters;

procedure proc is
   use proc_Servants;
begin
   PolyORB.Initialization.Initialize_World;

   --  Link local RT POA to current node, specifing priority

   ARAO.RT_Obj_Adapters.Link_To_Obj_Adapter
```

```
      (new proc_Servants.th2_Object,
       Th2_Ref,
       1);

   --  Collecting the references of the processes to which it's
   --  connected

   ARAO.Utils.Get_GIOP_Ref (tr1_th1_Ref, "127.0.0.1", 4000, 1, "th1", "iiop", 1);

   --  Create a periodic thread

   ARAO.Periodic_Threads.Create_Periodic_Thread
     (TP => proc_Servants.th1_Controller'Access);

   PolyORB.ORB.Run (PolyORB.Setup.The_ORB, May_Poll => True);
end proc;
```

And the code of the generated file `ARAO.Setup.Application` is:

```
with ARAO.Setup.Base;
pragma Warnings (Off, ARAO.Setup.Base);
pragma Elaborate_All (ARAO.Setup.Base);
with PolyORB.Setup.IIOP;
pragma Warnings (Off, PolyORB.Setup.IIOP);
pragma Elaborate_All (PolyORB.Setup.IIOP);
with PolyORB.Setup.Access_Points.IIOP;
pragma Warnings (Off, PolyORB.Setup.Access_Points.IIOP);
pragma Elaborate_All (PolyORB.Setup.Access_Points.IIOP);
--  ORB controller : workers
with PolyORB.ORB_Controller.Workers;
pragma Warnings (Off, PolyORB.ORB_Controller.Workers);
pragma Elaborate_All (PolyORB.ORB_Controller.Workers);
--  Multithreaded no priority mode package
with ARAO.Setup.Ocarina_OA;
pragma Warnings (Off, ARAO.Setup.Ocarina_OA);
pragma Elaborate_All (ARAO.Setup.Ocarina_OA);

package body ARAO.Setup.Application is

   --  No protocol set : default : GIOP/IIOP

   --  No request priority management

end ARAO.Setup.Application;
```

Note that, since no priorities has been set in AADL description, Object Adapter is a generic one.

If thread priorities have been set in AADL description, then ARAO will build a custom Portable Object Adapter. The building of Portable Object Adapter depends of a set of data such has receiver thread priority and stack size, and the number of out ports connected to his thread. A lane will be created for each port, which will contain thread for every connected out port. Lane priority and stack size will be inherited from AADL thread description, or set to default.

Let's modify the previous example by adding priorities to each threads.

```
process proc
features
  msg_in : in event data port message
  msg_out : out event data port message;
end proc;
```

```
process implementation proc.simple
subcomponents
  th1 : thread sender.simple {ARAO::Priority => 1};
  th2 : thread receiver.simple {ARAO::Priority => 32};
connections
  event data port msg_in -> th2.msg_in;
  event data port th1.msg_out -> msg_out;
end proc.simple;
```

Then Ocarina will generate another version of `ARAO.Setup.Application`, which will contain calls to a custom Object Adapter generator in `ARAO.Setup.OA.Multithreaded.Prio`.

```
--  General setup
with ARAO.Setup.Base;
pragma Warnings (Off, ARAO.Setup.Base);
pragma Elaborate_All (ARAO.Setup.Base);

--  Low-level setup packages
with PolyORB.Setup.IIOP;
pragma Warnings (Off, PolyORB.Setup.IIOP);
pragma Elaborate_All (PolyORB.Setup.IIOP);
with PolyORB.Setup.Access_Points.IIOP;
pragma Warnings (Off, PolyORB.Setup.Access_Points.IIOP);
pragma Elaborate_All (PolyORB.Setup.Access_Points.IIOP);

--  ORB controller : workers
with PolyORB.ORB_Controller.Workers;
pragma Warnings (Off, PolyORB.ORB_Controller.Workers);
pragma Elaborate_All (PolyORB.ORB_Controller.Workers);

--  Multithreaded mode package
with ARAO.Setup.OA.Multithreaded.Prio;
pragma Warnings (Off, ARAO.Setup.OA.Multithreaded.Prio);
pragma Elaborate_All (ARAO.Setup.OA.Multithreaded.Prio);

--  priorites-related packages
with PolyORB.Types;
with ARAO.Threads;
with PolyORB.Setup.OA.Basic_RT_Poa;
with ARAO.Setup.OA.Multithreaded;

--  Initialization-related packages
with PolyORB.Initialization;
with PolyORB.Utils.Strings;
with PolyORB.Utils.Strings.Lists;

package body ARAO.Setup.Application is

   --  No protocol set : default : GIOP/IIOP

   Threads_Array_ : constant ARAO.Threads.Threads_Properties_Array :=
     ((Standard.Natural
        (1),                --  thread th1 Priority
      Standard.Natural
        (0),
      PolyORB.Types.To_PolyORB_String
        ("th1"),
      Standard.Natural
        (0)),
      (Standard.Natural
        (32),               --  thread th2 Priority
      Standard.Natural
```

```
        (0),
      PolyORB.Types.To_PolyORB_String
        ("th2"),
      Standard.Natural
        (2)));

   package Priority_Manager is
     new ARAO.Setup.OA.Multithreaded.Prio
       (Threads_Array_);

   procedure Initialize;

end ARAO.Setup.Application;
```

7.3 Node positioning

Node (process) location is done via a native mechanism of PolyORB. By overloading the abstract function Get_Conf of PolyORB.Parameters, we can assign a specific location to a node.

For each process, Gaia will generate a package `PolyORB.Parameters.Partition` which will contains a static array and a Get_Conf function definition linking the current node location to PolyORB local data. When PolyORB will Initialize itself, this function will be called as it's registered in the Initialize hierarchy.

Example :

```
system implementation position.impl
subcomponents
  proc : process sender_node.simple {ARAO::port_number => 3200;};
  proc_1 : processor a_processor {ARAO::location => "127.0.0.1"};
properties
  actual_processor_binding => reference proc_1 applies to proc;
end position.impl;
```

```
package body PolyORB.Parameters.Partition is

   type Parameter_Entry is
      record
         Key : PolyORB.Utils.Strings.String_Ptr;
         Val : PolyORB.Utils.Strings.String_Ptr;
      end record;

   Conf_Table : constant array (1 .. 2)
     of Parameter_Entry :=
     ((new Standard.String'
        ("polyorb.protocols.iiop.default_addr"),
      new Standard.String'
        ("127.0.0.1")),
      (new Standard.String'
        ("polyorb.protocols.iiop.default_port"),
      new Standard.String'
        ("3200")));

   type Partition_Source is
     new PolyORB.Parameters.Parameters_Source with null record;

   The_Partition_Source : aliased Partition_Source;

   function Get_Conf
     (Source : access Partition_Source;
```

```
      Section : Standard.String;
      Key : Standard.String)
     return Standard.String;
   --  Called by PolyORB Initialization
   --  return the configuration data as in the Conf_Table array

   procedure Initialize;
   --  Initialize PolyORB by registerering Get_Conf function

end PolyORB.Parameters.Partition;
```

7.4 Description of the Middleware API

ARAO, the middleware API, contains package to use and configure the PolyORB middleware.

7.4.1 API to manipulate PolyORB

7.4.1.1 ARAO.Obj_Adapters

This package defines the following subprograms:

Link_To_Obj_Adapter: This procedure performs the link between the object reference (used by a client to send a request) and the servant who does the job specified by the request. This procedure assumes that the middlware is correctly set up and that a object adapter is created.

```
procedure Link_To_Obj_Adapter
     (T_Object :     PolyORB.Servants.Servant_Access;
      Ref      : out PolyORB.References.Ref);
```

7.4.1.2 ARAO.RT_Obj_Adapters

This package defines the following subprograms:

Link_To_Obj_Adapter: This procedure performs the link between the object reference (used by a client to send a request) and the servant who does the job specified by the request. This procedure assumes that the middlware is correctly set up and that a real-time object adapter is created for that Servant (instead of for the whole node as in ARAO.Obj_Adapter).

```
procedure Link_To_Obj_Adapter
     (T_Object   : PolyORB.Servants.Servant_Access;
      Ref        : out PolyORB.References.Ref;
      Thread_Name : Standard.String;
      Priority   : Integer := System.Default_Priority);
```

7.4.1.3 ARAO.Periodic_Threads

This package defines the following subprograms:

Create_Periodic_Thread: This procedure creates a periodic thread. The fact that the thread is periodic is handled in the TP procedure. Also, we assume that the PolyORB thread pool was properly created during the setup phase. Storage_size 0 is default size (not really 0 bit).

```
procedure Create_Periodic_Thread
     (TP           : Parameterless_Procedure;
      Priority     : System.Any_Priority := System.Default_Priority;
      Storage_Size : Integer := 0);
```

7.4.1.4 ARAO.Requests

This package defines the following subprograms:

Emit_Msg: This procedure creates a request whose target is the reference Ref. The PortName argument is used to distinguish the different port of one single thread. The data sent by the request (Item) must be of og the PolyORB neutral type (Any).

```
procedure Emit_Msg
     (Item     : PolyORB.Any.Any;
      Ref      : PolyORB.References.Ref;
      PortName : String);
```

7.4.1.5 ARAO.Utils

This package defines the following subprograms:

Put_Ref: Put the IOR reference Ref in the file called Filename

```
procedure Put_Ref
     (Ref      : PolyORB.References.Ref;
      Filename : String);
```

Get_Ref: Get the IOR reference Ref from the file called Filename

```
procedure Get_Ref
     (Ref      : in out PolyORB.References.Ref;
      Filename :        String);
```

Get_Ref: Get the reference Ref from the properties of the remote servants.

```
procedure Get_Ref
     (Ref           : in out PolyORB.References.Ref;
      Host_Location : String;
      Port_Number   : Positive;
      Servant_Index : Natural;
      Protocol      : String);
```

Get_GIOP_Ref: Get the reference Ref from the properties of the remote servants for IIOP profiles.

```
procedure Get_GIOP_Ref
     (Ref           : in out PolyORB.References.Ref;
      Ior_Ref       : String);
```

7.4.2 PolyORB Setup files

7.4.2.1 ARAO.Setup.Ocarina_OA

Set up the Ocarina Object Adapter

This package defines no subprogram

7.4.2.2 ARAO.Setup.OA.Multithreaded

Set up translation procedures for PolyORB priorities. Needed by RTPOA setup. has to be called before ARAO.Setup.OA.Multithreaded.Prio.Initialize.

This package defines no subprogram

7.4.2.3 ARAO.Setup.OA.Multithreaded.Prio

Setup an object adapter for multithread processes with request priority management.

This package defines the following subprograms:

Initialize: Create a Real-Time Object Adapter (RTPOA) for each IN port of the caller process. This procedure assumes that PolyORB was correctly setup, and particulary that PolyORB.RT_POA was previously withed. The RTPOAs will be created with respect to in port thread priority, stack size and number of connected out ports.

```
procedure Initialize;
```

8 Petri Net Mapping Rules

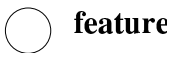
Ocarina lets you generate Petri nets from AADL descriptions. This way, it is possible to achieve verification on AADL architectures before generating the corresponding source code.

8.1 Mapping Patterns

The AADL elements to map into Petri nets are the software components. Indeed, execution platform components are used to model the deployment of the software components; such deployment information is not to in the scope of Petri nets. AADL threads and AADL subprograms are the most important components, since they can host subprogram call sequences: they describe the actual execution flows in the architecture. AADL processes and systems are actually boxes containing threads or other components, hence they do not provide any "active" semantics; data components are not active components either. The following table lists the main rules of the mapping:

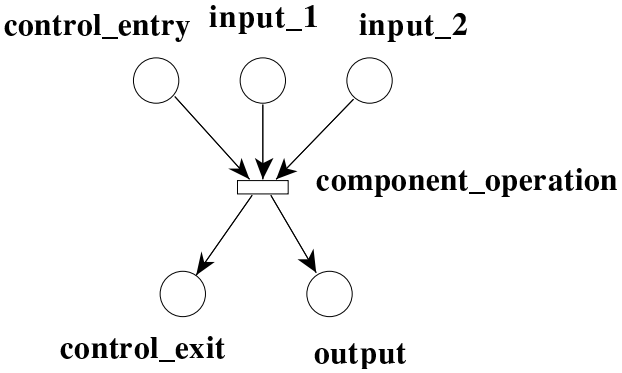
8.1.1 Component Features

```
feature : in data port;
```



8.1.2 Subprograms

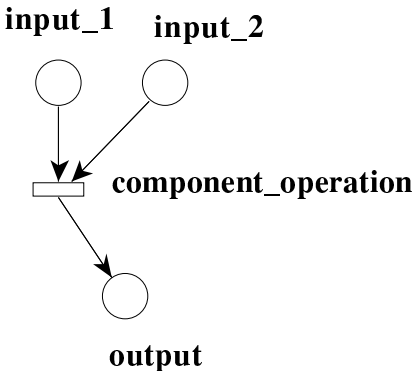
```
subprogram a_subprogram
features
  input_1 : in parameter;
  input_2 : in parameter;
  output : out parameter;
end a_subprogram;
```



8.1.3 Other Components

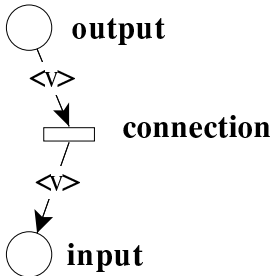
```
process a_process
features
  input_1 : in data port;
  input_2 : in data port;
  output : out data port;
```

```
end a_process;
```



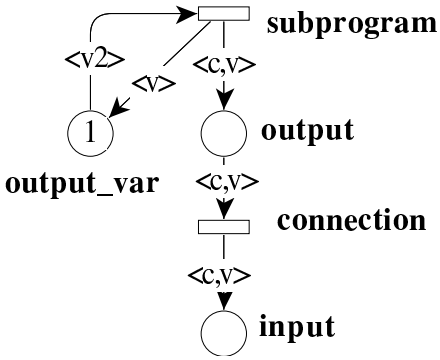
8.1.4 Connections

```
connection : data port output -> input;
```



8.1.5 Subprogram Connections

```
connection : parameter output -> input;
```



This mapping mainly consists of translating the AADL execution flows into Petri nets. Components that do not have any subcomponents nor call sequences are modeled by a transition that consumes inputs and produces outputs. Component features are modeled by places. Tokens stored in input features are to be consumed by the components or connections; tokens produced by component or connection transitions are stored in output features. Components that have subcomponents are modeled by merging the transition with the models of the subcomponents.

We model a place per feature. This systematic approach help the user identify the translation between AADL models and corresponding Petri nets. In addition, it facilitates the expansions of the feature places. For example, we might want to describe the queue protocols defined by the AADL properties: in this case we would replace each place by Petri nets modeling FIFOs or whatever type of queue is specified by the AADL properties.

Connections between features are modeled by transitions. We distinguish connections between subprograms parameters and between other component ports.

If an AADL port is connected to several other ports at a time, the Petri net transition shall be connected to all the corresponding places: a token will be sent to each target place, thus modeling the fact that each destination port receives the output of the initial port.

Connections of subprograms parameters are slightly more complex. Indeed, output places of subprograms model variables, that can be read many times. Therefore, a subprogram produces two tokens: one that carries the output value and the control information, and an extra one that only carries the value. The extra token is stored in a place where subprograms from other call sequences can get it—the transition shall put the token back into the place. In order to ensure the correct replacement of the token whenever a new value is produced by the subprogram, the subprogram itself consumes its old extra output token to produce the new one.

Call sequences are made of subprograms that are connected. We use an extra token to model the execution control. There is a single execution control token in each thread or subprogram, thus reflecting the fact that there is no concurrency in call sequences, and in threads and subprograms in general.

8.2 Examples

Like code generation, the Petri net mapping does not directly handle behavioral description: such descriptions must be provided using AADL properties and then merged with the generated net.

Here is an example of an AADL model:

```
thread micro_broker end micro_broker;

thread implementation micro_broker.receiver
subcomponents
  buffer : data data_buffer;
calls
  listen : {get_data : subprogram protocol.simple;};
  compute : {execute : subprogram execution.simple;};
connections
  data access buffer -> get_data.buffer;
  data access buffer -> execute.buffer;
properties
  Dispatch_Protocol => background;
  Ocarina::formal_implementation => "broker.pn";
end micro_broker.receiver;

data data_buffer end data_buffer;

subprogram protocol
features
  buffer : requires data access data_buffer;
end protocol;

subprogram implementation protocol.simple
calls
  {get_data : subprogram transport;};
connections
  data access buffer -> get_data.buffer;
end protocol.simple;

subprogram transport
features
  buffer : requires data access data_buffer;
end transport;
```

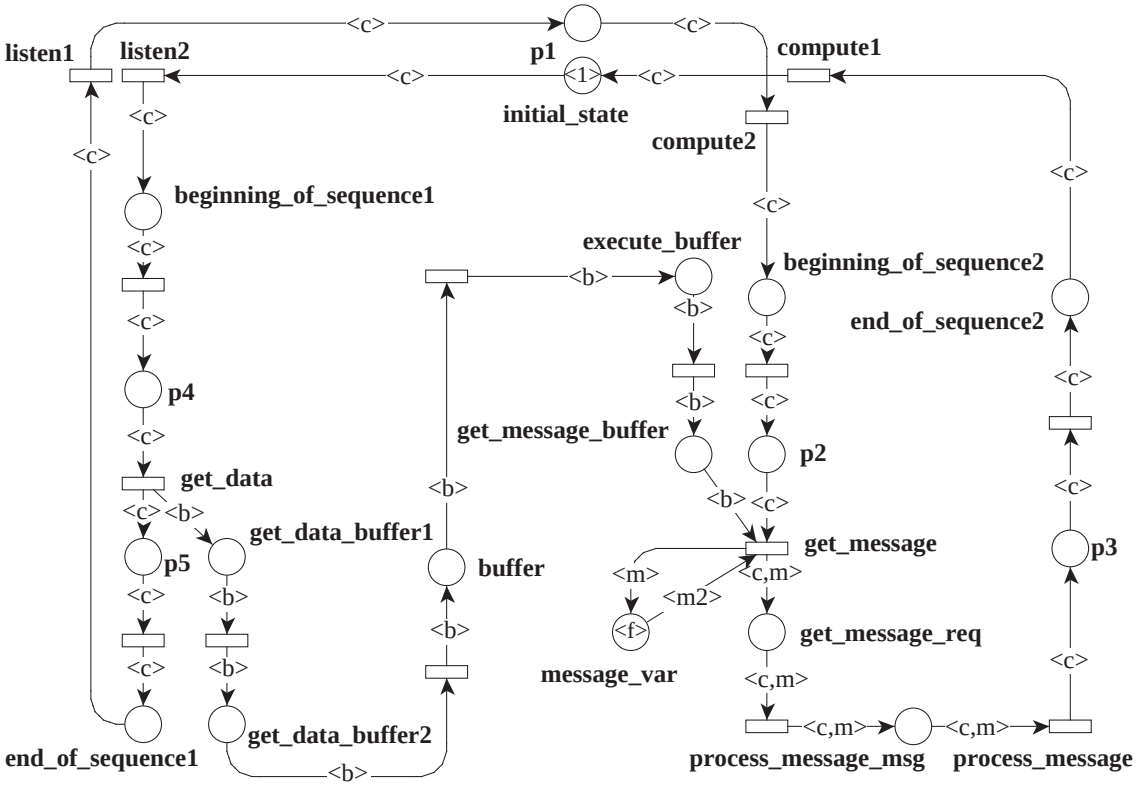
```
subprogram execution
features
  req : in parameter message;
end execution;

subprogram implementation execution.simple
calls
  {get_message : subprogram arguments;
    process_message : subprogram application;};
connections
  data access buffer -> get_message.buffer;
  parameter get_message.req -> process_message.msg;
end execution.simple;

subprogram arguments
features
  req : out parameter message;
  buffer : requires data access data_buffer;
end arguments;

subprogram application
features
  msg : in parameter message;
end application;
```

By applying the mapping, we obtain the following Petri net:



9 AADL modes for Emacs and vim

The AADL modes for Emacs and vim provide syntax coloration and automatic indentation features when editing AADL files.

9.1 Emacs

To load the AADL mode for Emacs, you need to add the following line to your emacs configuration file (usually located in '`~/.emacs`') :

```
(load "/path/to/this/file.el")
```

For more details on this mode, please refer to the emacs contextual help.

9.2 vim

The AADL mode for vim is made of two files '`aadl.vim`': one for syntactic coloration, and the other for indentation. The file for indentation must be placed into '`~/.vim/indent/`' while the one for syntactic coloration must be placed into '`~/.vim/syntax/`'

To load the AADL mode whenever you edit AADL files, create a file named '`~/.vim/filetype.vim`', in which you write:

```
augroup filetypedetect
        au BufNewFile,BufRead *.aadl    setf aadl
augroup END
```

For more details, please read the documentation of vim.

10 Dia editor & AADL

The documentation on the Dia/AADL plugin will appear in a future revision of Ocarina.

Appendix A Standard AADL property files

A.1 AADL Project

```
--*****************************************************
--   AADL Standard AADL_V1.0
--  Appendix A (normative)
--  Predeclared Property Sets
--  03Nov04
--  Revised 14May06
--*****************************************************

property set AADL_Project is

  Default_Active_Thread_Handling_Protocol : constant
     Supported_Active_Thread_Handling_Protocols => abort;
  -- one of the choices of Supported_Active_Thread_Handling_Protocols.

  Supported_Active_Thread_Handling_Protocols: type enumeration
    (abort,
     complete_one_flush_queue,
     complete_one_transfer_queue,
     complete_one_preserve_queue,
     complete_all);
  -- a subset may be supported.

  Supported_Connection_Protocols: type enumeration
    (HTTP,
     HTTPS,
     UDP,
     IP_TCP);
  -- The following are example protocols.
  -- (HTTP, HTTPS, UDP, IP_TPC);

  Supported_Concurrency_Control_Protocols: type enumeration
    (NoneSpecified,
     Protected_Access,
     Priority_Ceiling);
  -- phf : NoneSpecified instead of None
  -- The following are example concurrency control protocols.
  -- (Interrupt_Masking, Maximum_Priority, Priority_Inheritance,
  --  Priority_Ceiling)

  Supported_Dispatch_Protocols: type enumeration
    (Periodic,
     Aperiodic,
     Sporadic,
     Background);
  -- The following are protocols for which the semantics are defined.
  -- (Periodic, Sporadic, Aperiodic, Background);

  Supported_Hardware_Source_Languages: type enumeration
    (VHDL);
  -- The following is an example hardware description language.
  -- (VHDL)

  -- phf A26: added
  Supported_Queue_Processing_Protocols: type enumeration
    (FIFO);
  -- The Supported_Queue_Processing_Protocols property enumeration
  -- type specifies the set of queue processing protocols that are
  -- supported.
```

```
  Supported_Scheduling_Protocols: type enumeration
    (PARAMETRIC_PROTOCOL,
     EARLIEST_DEADLINE_FIRST_PROTOCOL,
     LEAST_LAXITY_FIRST_PROTOCOL,
     RATE_MONOTONIC_PROTOCOL,
     DEADLINE_MONOTONIC_PROTOCOL,
     ROUND_ROBIN_PROTOCOL,
     TIME_SHARING_BASED_ON_WAIT_TIME_PROTOCOL,
     POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL,
     D_OVER_PROTOCOL,
     MAXIMUM_URGENCY_FIRST_BASED_ON_LAXITY_PROTOCOL,
     MAXIMUM_URGENCY_FIRST_BASED_ON_DEADLINE_PROTOCOL,
     TIME_SHARING_BASED_ON_CPU_USAGE_PROTOCOL,
     NO_SCHEDULING_PROTOCOL,
     HIERARCHICAL_CYCLIC_PROTOCOL,
     HIERARCHICAL_ROUND_ROBIN_PROTOCOL,
     HIERARCHICAL_FIXED_PRIORITY_PROTOCOL,
     HIERARCHICAL_PARAMETRIC_PROTOCOL);
  -- The following are example scheduling protocols.
  -- (RMS, EDF, Sporadicserver, SlackServer, ARINC653)

  Supported_Source_Languages: type enumeration
    (Ada95,
     C,
     Simulink_6_5);
  -- The following are example software source languages.
  -- ( Ada95, C, Simulink_6_5 )

  Max_Aadlinteger: constant aadlinteger => 2#1#e32;

  Max_Base_Address: constant aadlinteger => 512;

  Max_Memory_Size: constant aadlinteger Size_Units => 2#1#e32 B;

  Max_Queue_Size: constant aadlinteger => 512;

  Max_Thread_Limit: constant aadlinteger => 32;

  Max_Time: constant aadlinteger Time_Units => 1000 hr;

  Max_Urgency: constant aadlinteger => 12;

  Max_Word_Count: constant aadlinteger => 2#1#e32;

  Max_Word_Space: constant aadlinteger => 64;

  Size_Units : type units (
    Bits,
    B     => Bits  * 8,
    KB    => B     * 1000,
    MB    => KB    * 1000,
    GB    => MB    * 1000);

  Time_Units : type units (
    ps,
    Ns    => ps  * 1000,
    Us    => Ns  * 1000,
    Ms    => Us  * 1000,
    Sec   => Ms  * 1000,
    Min   => Sec * 60,
    Hr    => Min * 60);

end AADL_Project;
```

A.2 AADL Properties

```
--*****************************************************
--   AADL Standard AADL_V1.0
--  Appendix A (normative)
--  Predeclared Property Sets
--  03Nov04
--  Update to reflect current standard on 28Mar06
--*****************************************************

property set AADL_Properties is

  ---------------------------------------------------
  ---------------------------------------------------
  Activate_Deadline: Time
  applies to (thread);
  ---------------------------------------------------

  Activate_Execution_Time: Time_Range
  applies to (thread);
  ---------------------------------------------------

  Activate_Entrypoint: aadlstring
  applies to (thread);
  ---------------------------------------------------

  Active_Thread_Handling_Protocol: inherit
    Supported_Active_Thread_Handling_Protocols
  => value(Default_Active_Thread_Handling_Protocol)
  applies to (thread, thread group, process, system);
  ---------------------------------------------------

  Active_Thread_Queue_Handling_Protocol: inherit enumeration
    (flush,
     hold)
  => flush
  applies to (thread, thread group, process, system);
  ---------------------------------------------------

  Actual_Connection_Binding: inherit list of reference
    (bus,
     processor,
     device)
  applies to (port connections, thread, thread group, process, system);
  ---------------------------------------------------

  Actual_Latency: Time
  applies to (flow);
  ---------------------------------------------------

  Actual_Memory_Binding: inherit reference (memory)
  applies to (thread, thread group, process, system, processor,
  data port, event data port, subprogram);
  ---------------------------------------------------

  Actual_Processor_Binding: inherit reference (processor)
  applies to (thread, thread group, process, system);
  ---------------------------------------------------

  Actual_Subprogram_Call: reference (server subprogram)
  applies to (subprogram);
  ---------------------------------------------------

  Actual_Subprogram_Call_Binding: inherit list of reference
```

```
  (bus,
   processor,
   memory,
   device)
applies to (subprogram);
-------------------------------------------------

Actual_Throughput: Data_Volume
applies to (flow);
-------------------------------------------------

Aggregate_Data_Port: aadlboolean => false
applies to (port group);
-------------------------------------------------

Allowed_Access_Protocol: list of enumeration
  (Memory_Access,
   Device_Access)
applies to (bus);
-------------------------------------------------

Allowed_Connection_Binding: inherit list of reference
  (bus,
   processor,
   device)
applies to (port connections, thread group, process, system);
-------------------------------------------------

Allowed_Connection_Binding_Class: inherit list of classifier
  (processor,
   bus,
   device)
applies to (port connections, thread, thread group, process, system);
-------------------------------------------------

Allowed_Connection_Protocol: list of enumeration
  (Data_Connection,
   Event_Connection,
   Event_Data_Connection,
   Data_Access_Connection,
   Server_Subprogram_Call)
applies to (bus, device);
-------------------------------------------------

Allowed_Dispatch_Protocol: list of Supported_Dispatch_Protocols
applies to (processor);
-------------------------------------------------

Allowed_Memory_Binding: inherit list of reference
  (memory,
   system,
   processor)
applies to (thread, thread group, process, system, device, data port,
event data port, subprogram, processor);
-------------------------------------------------

Allowed_Memory_Binding_Class: inherit list of classifier
  (memory,
   system,
   processor)
applies to (thread, thread group, process, system, device, data port,
event data port, subprogram, processor);
-------------------------------------------------

Allowed_Message_Size: Size_Range
```

```
applies to (bus);
-------------------------------------------------

Allowed_Period: list of Time_Range
applies to (processor, system);
-------------------------------------------------

Allowed_Processor_Binding: inherit list of reference
  (processor,
   system)
applies to (thread, thread group, process, system, device);
-------------------------------------------------

Allowed_Processor_Binding_Class: inherit list of classifier
  (processor,
   system)
applies to (thread, thread group, process, system, device);
-------------------------------------------------

Allowed_Subprogram_Call: list of reference
  (server subprogram)
applies to (subprogram);
-------------------------------------------------

Allowed_Subprogram_Call_Binding: inherit list of reference
  (bus,
   processor,
   device)
applies to (subprogram, thread, thread group, process, system);
-------------------------------------------------

Assign_Time: Time
applies to (processor, bus);
-------------------------------------------------

Assign_Byte_Time: Time
applies to (processor, bus);
-------------------------------------------------

Assign_Fixed_Time: Time
applies to (processor, bus);
-------------------------------------------------

Available_Memory_Binding: inherit list of reference
  (memory,
   system)
applies to (system);
-------------------------------------------------

Available_Processor_Binding: inherit list of reference
  (processor,
   system)
applies to (system);
-------------------------------------------------

Base_Address: access aadlinteger 0 .. value(Max_Base_Address)
applies to (memory);
-------------------------------------------------

Client_Subprogram_Execution_Time: Time
applies to (subprogram);
-------------------------------------------------

Clock_Jitter: Time
applies to (processor, system);
```

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

Clock_Period: Time
applies to (processor, system);
-------------------------------------------------

Clock_Period_Range: Time_Range
applies to (processor, system);
-------------------------------------------------

Compute_Deadline: Time
applies to (thread, device, subprogram, event port, event data port);
-------------------------------------------------

Compute_Entrypoint: aadlstring
applies to (thread, subprogram, event port, event data port);
-------------------------------------------------

Compute_Execution_Time: Time_Range
applies to (thread, device, subprogram, event port, event data port);
-------------------------------------------------

Concurrency_Control_Protocol: Supported_Concurrency_Control_Protocols
  => NoneSpecified
applies to (data);
-------------------------------------------------

Connection_Protocol: Supported_Connection_Protocols
applies to (connections);
-------------------------------------------------

Data_Volume: type aadlinteger 0 bitsps .. value(Max_Aadlinteger)
units
  (bitsps,
   Bps    => bitsps * 8,
   Kbps   => Bps    * 1000,
   Mbps   => Kbps   * 1000,
   Gbps   => Mbps   * 1000 );
-------------------------------------------------

Deactivate_Deadline: Time
applies to (thread);
-------------------------------------------------

Deactivate_Execution_Time: Time_Range
applies to (thread);
-------------------------------------------------

Deactivate_Entrypoint: aadlstring
applies to (thread);
-------------------------------------------------

Deadline: inherit Time => value(Period)
applies to (thread, thread group, process, system, device);
-------------------------------------------------

Dequeue_Protocol: enumeration
  (OneItem,
   AllItems)
   => OneItem
applies to (event port, event data port);
-------------------------------------------------

Device_Dispatch_Protocol: Supported_Dispatch_Protocols
  => Aperiodic
```

```
applies to (device);
----------------------------------------

Device_Register_Address: aadlinteger
applies to (port, port group);
-------------------------------------------------

Dispatch_Protocol: Supported_Dispatch_Protocols
applies to (thread);
-------------------------------------------------

Expected_Latency: Time
applies to (flow);
-------------------------------------------------

Expected_Throughput: Data_Volume
applies to (flow);
-------------------------------------------------

Finalize_Deadline: Time
applies to (thread);
-------------------------------------------------

Finalize_Execution_Time: Time_Range
applies to (thread);
-------------------------------------------------

Finalize_Entrypoint: aadlstring
applies to (thread);
-------------------------------------------------

Hardware_Description_Source_Text: inherit list of aadlstring
applies to (memory, bus, device, processor, system);
-------------------------------------------------

Hardware_Source_Language: Supported_Hardware_Source_Languages
applies to (memory, bus, device, processor, system);
-------------------------------------------------

Initialize_Deadline: Time
applies to (thread);
-------------------------------------------------

Initialize_Execution_Time: Time_Range
applies to (thread);
-------------------------------------------------

Initialize_Entrypoint: aadlstring
applies to (thread);
-------------------------------------------------

Latency: Time
applies to (flow, connections);
-------------------------------------------------

Load_Deadline: Time
applies to (process, system);
-------------------------------------------------

Load_Time: Time_Range
applies to (process, system);
-------------------------------------------------

Memory_Protocol: enumeration
  (read_only,
```

```
   write_only,
   read_write)
   => read_write
applies to (memory);
-------------------------------------------------

Not_Collocated: list of reference
  (data,
   thread,
   process,
   system,
   connections)
applies to (data, thread, process, system, connections);
-------------------------------------------------

Overflow_Handling_Protocol: enumeration
  (DropOldest,
   DropNewest,
   Error)
   => DropOldest
applies to (event port, event data port, subprogram);
-------------------------------------------------

Period: inherit Time
applies to (thread, thread group, process, system, device);
-------------------------------------------------

Process_Swap_Execution_Time: Time_Range
applies to (processor);
-------------------------------------------------

Propagation_Delay: Time_Range
applies to (bus);
-------------------------------------------------

Provided_Access : access enumeration
  (read_only,
   write_only,
   read_write,
   by_method)
   => read_write
applies to (data, bus);
-------------------------------------------------

Queue_Processing_Protocol: Supported_Queue_Processing_Protocols
  => FIFO
applies to (event port, event data port, subprogram);
-------------------------------------------------

Queue_Size: aadlinteger 0 .. value(Max_Queue_Size)
  => 0
applies to (event port, event data port, subprogram);
-------------------------------------------------

Read_Time: list of Time_Range
applies to (memory);
-------------------------------------------------

Recover_Deadline: Time
applies to (thread, server subprogram);
-------------------------------------------------

Recover_Execution_Time: Time_Range
applies to (thread, server subprogram);
-------------------------------------------------
```

```
Recover_Entrypoint: aadlstring
applies to (thread);
-------------------------------------------------

Required_Access : access enumeration
  (read_only,
   write_only,
   read_write,
   by_method)
   => read_write
applies to (data, bus);
-------------------------------------------------

Required_Connection : aadlboolean => true
applies to (port);
-------------------------------------------------

Runtime_Protection : inherit aadlboolean => true
applies to (process, system);
-------------------------------------------------

Scheduling_Protocol: list of Supported_Scheduling_Protocols
applies to (processor);
-------------------------------------------------

Server_Subprogram_Call_Binding: inherit list of reference
  (thread,
   processor)
applies to (subprogram, thread, thread group, process, system);
-------------------------------------------------
Size: type aadlinteger 0 B .. value (Max_Memory_Size) units Size_Units;

--  OLD DECLARATION:
--  Size: type aadlinteger 0 B .. value (Max_Memory_Size);
--  This is wrong according to the AADL standard 1.0 page 150:

--  "An aadlinteger property type represents an integer value or an
--  integer value and its measurement unit.  If an units clause is
--  present, then the value is a pair of values, and unit may only
--  be one of the enumeration literals specified in the units
--  clause. *IF AN UNITS CLAUSE IS ABSENT, THEN THE VALUE IS AN
--  INTEGER VALUE*. If a simple range is present, then the integer
--  value must be an element of the specified range"

-------------------------------------------------
Size_Range: type range of Size;
-------------------------------------------------

Source_Code_Size: Size
applies to (data, thread, thread group, process, system, subprogram,
processor, device);
-------------------------------------------------

Source_Data_Size: Size
applies to (data, subprogram, thread, thread group, process, system,
processor, device);
-------------------------------------------------

Source_Heap_Size: Size
applies to (thread, subprogram);
-------------------------------------------------

Source_Language: inherit Supported_Source_Languages
applies to (subprogram, data, thread, thread group, process, bus,
```

```
device, processor, system);
-------------------------------------------------

Source_Name: aadlstring
applies to (data, port, subprogram, parameter);
-------------------------------------------------

Source_Stack_Size: Size
applies to (thread, subprogram, processor, device);
-------------------------------------------------

Source_Text: inherit list of aadlstring
applies to (data, port, subprogram, thread, thread group, process,
system, memory, bus, device, processor, parameter, port group);
-------------------------------------------------

Startup_Deadline: inherit Time
applies to (processor, system);
-------------------------------------------------

Subprogram_Execution_Time: Time_Range
applies to (subprogram);
-------------------------------------------------

Supported_Source_Language: list of Supported_Source_Languages
applies to (processor, system);
-------------------------------------------------

Synchronized_Component: inherit aadlboolean => true
applies to (thread, thread group, process, system);
-------------------------------------------------

Thread_Limit: aadlinteger 0 .. value(Max_Thread_Limit)
  => value(Max_Thread_Limit)
applies to (processor);
-------------------------------------------------

Thread_Swap_Execution_Time: Time_Range
applies to (processor, system);
-------------------------------------------------

Throughput: Data_Volume
applies to (flow, connections);
-------------------------------------------------

Time: type aadlinteger 0 ps .. value(Max_Time) units Time_Units;

--  OLD DECLARATION:
--  Time: type aadlinteger 0 ps .. value(Max_Time);
--  This is wrong according to the AADL standard 1.0 page 150:

--  "An aadlinteger property type represents an integer value or an
--  integer value and its measurement unit.  If an units clause is
--  present, then the value is a pair of values, and unit may only
--  be one of the enumeration literals specified in the units
--  clause. *IF AN UNITS CLAUSE IS ABSENT, THEN THE VALUE IS AN
--  INTEGER VALUE*. If a simple range is present, then the integer
--  value must be an element of the specified range"

-------------------------------------------------

Time_Range: type range of Time;
-------------------------------------------------

Transmission_Time: list of Time_Range
```

```
  applies to (bus);
  ----------------------------------------------------

  Type_Source_Name: aadlstring
  applies to (data, port, subprogram);
  ----------------------------------------------------

  Urgency: aadlinteger 0 .. value(Max_Urgency)
  applies to (port);
  ----------------------------------------------------

  Word_Count: aadlinteger 0 .. value(Max_Word_Count)
  applies to (memory);
  ----------------------------------------------------

  Word_Size: Size => 8 bits
  applies to (memory);
  ----------------------------------------------------

  Word_Space: aadlinteger 1 .. value(Max_Word_Space) => 1
  applies to (memory);
  ----------------------------------------------------

  Write_Time: list of Time_Range
  applies to (memory);
  ----------------------------------------------------
  ----------------------------------------------------
end AADL_Properties;
```

Appendix B Ocarina AADL property files

B.1 ARAO

```
-- Property set for ARAO features

property set ARAO is

  simple_type   : type enumeration
   (boolean,
    character,
    float,
    fixed,
    integer,
    null,
    string,
    wide_character,
    wide_string);
  -- Supported data types

  data_type     : simple_type applies to (data);
  -- Available data type

  data_digits   : aadlinteger applies to (data);
  data_scale    : aadlinteger applies to (data);
  -- Properties for fixed point types

  max_length    : aadlinteger applies to (data);
  -- Max length for string data

  symbolic_values : list of aadlstring applies to (data);
  -- Special values for data defined following AADL standard specs

  location    : aadlstring applies to (processor);
  -- Processor network's address

  port_number : aadlinteger applies to (process);
  -- Port number used by a node

  process_id : aadlinteger applies to (process);
  -- Identifier of the process (used by SpaceWire)

  channel_address : aadlinteger applies to (process);
  -- SpaceWire channel address

  protocol_type : type enumeration (iiop, diop);
  -- Available communication protocol implementations

  protocol      : protocol_type applies to (system);
  -- Protocol implementation used for communications
  -- currently only support GIOP implementations

  priority_type : type aadlinteger 0 .. 255;

  priority      : priority_type applies to (data, thread);
  -- Thread and data component priority

  Level_A : constant priority_type => 250;
  Level_B : constant priority_type => 190;
  Level_C : constant priority_type => 130; --  Default
  Level_D : constant priority_type => 70;
  Level_E : constant priority_type => 10;
  -- Some predefined priorities
```

```
  Allowed_Execution_Platform : type enumeration
   (Native,
    LEON,
    ERC_32);
  -- Available platform to which distributed applications can be
  -- generated.

  Execution_Platform : Allowed_Execution_Platform applies to (processor);
  -- The execution platform of a distributed application

  Allowed_Transport_Layers : type enumeration
    (Ethernet,
     SpaceWire,
     Serial);
  -- Available transport layers

  Transport_Layer : Allowed_Transport_Layers applies to (bus);
  -- The transport layer of a bus component

  IP_Address : access aadlstring applies to (bus);
  -- IPv4 address of a network card

  MAC_Address : access aadlstring applies to (bus);
  -- MAC id. of a network card

  Memory_Size : Size applies to (memory);
  -- Memory size

end ARAO;
```

Appendix C Conformance to standards

The documentation on conformance to standards will appear in a future revision of Ocarina.

Index

A

AADL Scenario Files 6

C

Conventions 1

O

`ocarina` 7
`ocarina-config` 7
`ocarina_sh` 6

T

Typographical conventions 1