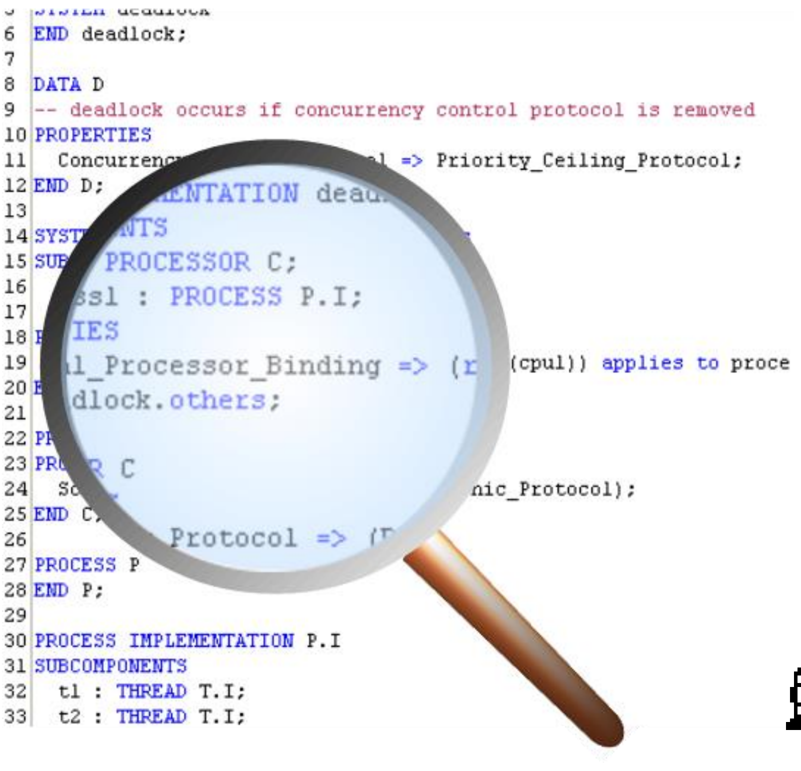
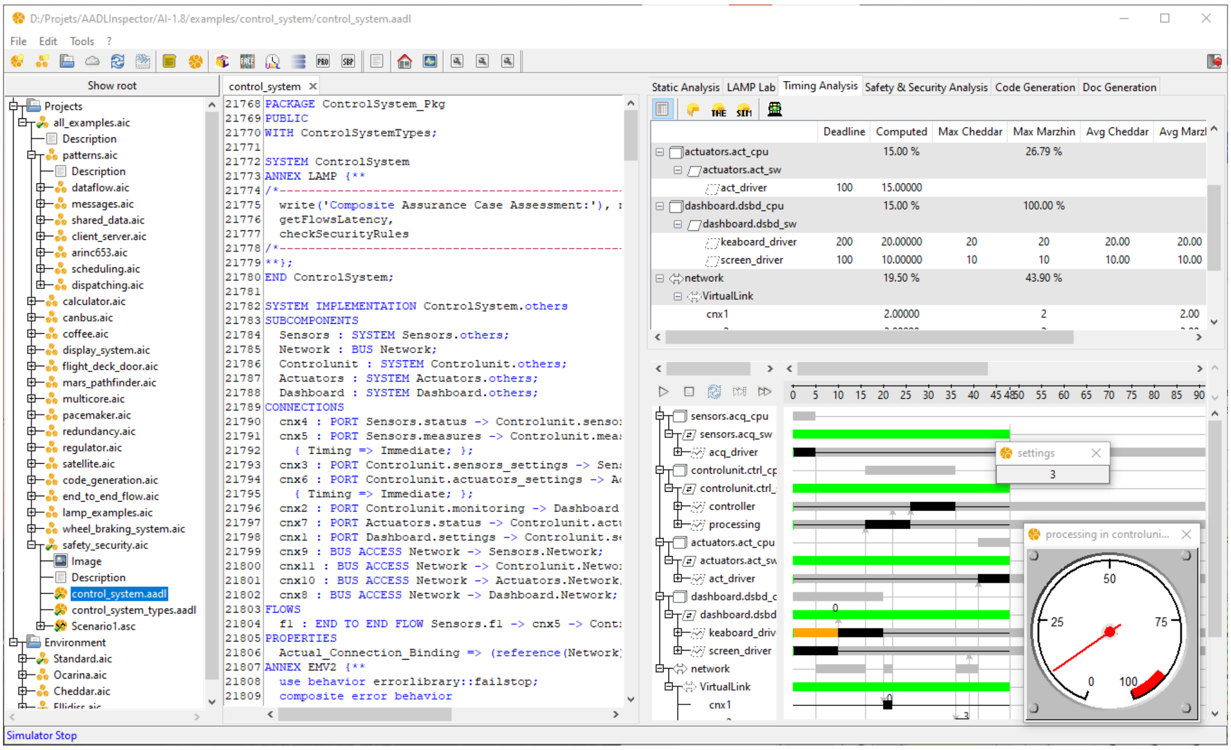
AADL Inspector 1.8

User Manual





Pierre Dissaux
Ellidiss Technologies
http://www.ellidiss.com/
aadl@ellidiss.com

Contents

1 Introduction

AADL Inspector is a model analysis framework for critical and software intensive systems. It focuses on design verification activities of the development life cycle and addresses a variety of topics including static rules checking, timing, safety and security analysis, as well as combination of these in customizable assurance cases. Verification tools are either built-in or user defined thanks to the powerful **LMP** (Logic Model Processing) technology and the **AADL LAMP** annex. The tool operates at architecture model level and does not require the final source code to be available. **AADL Inspector** can process the following kinds of architectures with appropriate abstractions:

- Multi-threaded software (running on a **RTOS**: Real-Time Operating System).
- Multi-partition software (**TSP**: Time and Space Partitioning).
- Multi-processor distributed software with network communication.
- Multi-core architectures with static tasks allocation.

In order to be able to perform advanced model processing in a homogeneous way and to reduce the effort of developing new analysis plug-ins, **AADL Inspector** operates on a common language that can be either the original input or the intermediate result of a prior foreign model transformation. The common language that has been chosen is the Architecture Analysis and Design Language (**AADL**) declarative model. The formal definition of the **AADL** language can be found in the **SAE AS-5506** document that is made available on the official site https://www.sae.org/standards/content/as5506c. More information about this language is available on the **Ellidiss** wiki page: https://www.ellidiss.fr/public/wiki/AADL, and the **OpenAADL** web site www.openaadl.org.

AADL Inspector is packaged in a standalone distribution that minimizes installation and maintenance effort to ease the everyday use of the product on standard personal computers or network servers. The product is available for both **Windows** and **Linux** platforms.

The goal of **AADL Inspector** is to encompass a variety of specialized tools to process a complete **AADL** specification composed of a set of text files. These files can be created within **AADL Inspector** itself, loaded from pre-existing local or remote libraries or automatically generated by an import wizard. **AADL** files can also be organized into hierarchical projects to facilitate the management of large models and the reuse of libraries of components. The processing tools can be used to analyse various facets of the architecture or to offer code generation and documentation capabilities. These processing tools are organized in a modular and extendable way so that they can be customized, and additional ones can be easily included.

The standard installation of **AADL Inspector** 1.8 implements the following model processing tools:

- *Static Analysis* of **AADL** models, using two different frameworks: **LMP** and **Ocarina**. This covers parsing of **AADL** declarative models, verification of standard **AADL** semantic rules (Legality, Consistency and Naming rules) and building the deployed instance model that is required for most purposes. Customized static rules can be added to fit corporate or project specific usage.
- *LAMP Lab*. **LAMP** (Logical **AADL** Model Processing) is a powerful and flexible solution to incorporate online assurance cases within **AADL** specifications. It takes the form of **AADL** Annex subclauses whose sublanguage is standard **Prolog**. The **LAMP**

verification engine checks all the user specified verification goals, supports the definition of reusable libraries of rules and can process analysis results of the Timing Analysis plugin, such as computed response times and simulation events.

- *Timing Analysis* of deployed **AADL** instance models using three complementary approaches: Scheduling theoretical tests and static simulation over the hyper-period with the **Cheddar** analysis kernel, and dynamic simulation with the event based **Marzhin** simulator. Moreover, response time statistics are provided in a table and Scheduling Aware Flow Latency Analysis (**SAFLA**) is also proposed there.
- *Safety & Security Analysis*. This plugin proposes transformations from **AADL** architectures enriched with Error Model annexes into various input models for existing safety analysis tools. Currently, proposed bridger uses the **OpenPSA** language to connect with the Arbre Analyste Fault Tree Analysis (**FTA**) tool.
- *Code Generation* using the **Ocarina** tool and the **PolyORB-HI-Ada** or **PolyORB-HI-C** middlewares.
- *Documentation generation* to keep track of timing analysis results.

The current **AADL** workspace on which the processing tools apply, can managed thanks to a set of advanced functions such as:

- Creating hierarchical projects to facilitate teamwork and reuse of libraries.
- Using predefined **AADL** model templates.
- Importing foreign models (**SysML**, **UML MARTE**, **Capella, FACE**[1]) into **AADL**.
- Loading **AADL** models from remote git repositories.
- Specifying simulation scenarios.
- Identifying the current root of the system instance hierarchy.
- Defining the thread priorities according to predefined ordering algorithms.
- Binding threads to available processors with predefined allocation algorithms.
- Modifying the main thread real-time properties in a spreadsheet.
- Editing textual **AADL** files and applying text formatting rules (autoformat).
- Converting older versions of source text into the most recent version of **AADL**.
- Writing your own online model processing tools with the **LAMP** environment

The current version of **AADL Inspector** supports the following standard definitions. Note that most processing tools only comply with a subset of the standard.

- **AADL** Core v2.2 (AS 5506C)
- **AADL** Behaviour Annex v2.0 (AS 55606/3)
- **AADL** Error Model Annex v2.0 (AS 5506/1A)
- **AADL** Data Model Annex (AS 5506/2)
- **AADL ARINC 653** Annex v2.0 (AS 5506/1A)
- **AADL** Annex for the **FACE** Technical Standard Edition 3.0 (AS 5506/4)

[1] FACE is a trademark of The Open Group

2 Before starting

2.1 Installation

Installation of the product only requires the following easy actions:

- Get a copy of the installation package for the desired platform (**Windows**, or **Linux** 64 bits) from the **Ellidiss** website: http://www.ellidiss.com/

- Run the installation program on **Windows** or uncompress and expand the archive file on **Linux**.

- Launch the `AADLInspector` executable file located in the `bin` subdirectory of the installation directory, or the corresponding desktop shortcut on **Windows**.

Downloaded packages usually come with a temporary trial license that can be used free of charge. If you purchased the product or this temporary license has expired, please contact **Ellidiss** customer support service to get the appropriate license information and installation procedure that fits your situation. A standard installation requires less than 50 Mbytes of free disk space.

2.2 Distribution contents

Once installed on the computer, the **AADL Inspector** installation directory contains the following subdirectories:

- `bin` subdirectory
- `config` subdirectory
- `examples` subdirectory
- `environment` subdirectory
- `include` subdirectory
- `doc` subdirectory

Note that after a first launch of the tool, a directory is created to store temporary files and to be used as a default storage area for generated documentation and code. The actual location of this temporary directory can be customized by the `logDirectory` parameter in the `config/AIConfig.ini` file, or the `-l` command line option. The default location of the temporary directory is within the user's home directory.

2.2.1. Bin subdirectories

These directories contain the executable files for the current platform and **Java** archive files that are shared by all platforms. The only external requirement is the availability of a proper **Java** 1.8 (or higher) Run-time Environment (**JRE**) to run the simulator. These files are:

- `AADLInspector` main executable file
- `AIMonitor` remote process monitoring executable file
- `aadlrev` executable file (**AADL** syntactic analyser)
- `xmlrev` executable file (**XML** syntactic analyser)

- `sbprolog` executable file (**Prolog** engine)
- `cheddarkernel` executable file (**Cheddar** schedulability analyser)
- `ocarina` executable file (**AADL** compiler and code generator)
- `aadl-utils` executable file (**AADL** file splitter)
- `Marzhin`, `VAgent` and `VCore` **Java** archive files for the **Marzhin** simulator

`aadlrev 2.14` is a standalone **AADL** syntactic analyser that is used by the **LMP** (Logic Model Process) plug-ins to convert **AADL** specifications into a list of **Prolog** predicates. This utility tool can analyse textual **AADL** files that comply with **AADL** 2.2 (*SAE AS-5506C*), the **AADL** Error Model v2 (*SAE AS-5506/1A Annex E*), the **AADL** Behaviour Annex (*SAE AS-5506/2 Annex D*), and the **AADL** ARINC 653 Annex (*SAE AS-5506/1A Annex A*). In addition, the previous version of the **AADL** Error Model (future *SAE AS-5506/1 Annex E*) is also supported by `aadlrev`. Most of the **AADL** 1.0 (*SAE AS-5506*), 2.0 (*SAE AS-5506A*) and 2.1 (*SAE AS-5506B*) syntax is also recognized and can be automatically converted into the newest 2.2 format.

`xmlrev 1.2` is a standalone **XML** syntactic analyser that is used by the **LMP** (Logic Model Process) plug-ins to convert **XML** or **XMI** serialized models into a list of **Prolog** predicates. This utility is used by the import wizards to load files having extensions such as `.uml`, `.xml`, `.xmi`, `.ecore`, `.sysml`, `.melodymodeller`, and to convert them into a list of **Prolog** predicates for further processing.

`cheddarkernel 3.2.2.1` is a command-line version of the **Cheddar** v3 schedulability analysis tool. **Cheddar** v3 models (`.xmlv3`) are generated from the **AADL** specification thanks to a dedicated **LMP** model transformation. **Cheddar** outputs (feasibility test reports and static time lines) are displayed by the **AADL Inspector** graphical interface. **Cheddar** is an open source project managed by the University of Brest: http://beru.univ-brest.fr/cheddar

`sbprolog 3.1` is an open source **Prolog** engine that is used by the **LMP** (Logic Model Processing) technology. **AADL Inspector** uses **LMP** to implement the various **AADL** rules checkers and model transformations. **SB-Prolog** was developed by State University of New York at Stony Brook and the University of Arizona.

`marzhin 2.2` is a multi-agent simulator implementing the **AADL** run-time. It consists of three **Java** archive files and requires a **Java** 1.8 Run-time Environment (**JRE**) to operate. No **JRE** is provided with the **AADL Inspector** distribution. **Marzhin** v2 models (`.xml`) are generated from the **AADL** specification thanks to a dedicated **LMP** model transformation. **Marzhin** outputs (dynamic time lines) are displayed in the **AADL Inspector** graphical interface. **Marzhin** is developed in collaboration by **Virtualys** and **Ellidiss Technologies**.

`ocarina 2.0` is an open source **AADL** syntactic and semantic analyser. It embeds various back-ends including **Ada** and **C** code generators using the **polyORB-HI** middleware. **Ocarina** was initially developed by **Telecom ParisTech** and is now maintained by **ISAE** with support of **ESA**: http://www.openaadl.org/ocarina.html

`aadl-utils 1.0` is another standalone **AADL** processing tool. It is used here with command line option `-s` to convert an **AADL** file containing several Packages or Property Sets into a directory of the same name containing on separate file per Package or Property Set. This may be required to interoperate with **OSATE** who enforces this restriction.

2.2.2. Config subdirectory

This directory contains initialization, configuration and license files that are used by the executable files.

The files having a `.sbp` extension contain a binary form of the **LMP** (Logic Model Processing) rules that are used to perform each model processing action. Checkers provide a direct textual output into the **AADL Inspector** window, whereas bridgers perform dedicated model transformations to interface with ancillary tools such as **Cheddar**, **Marzhin or Arbre Analyst**. Activation of these processing rules is performed from within a dedicated service declared in an **AADL Inspector** plugin (see below).

The files having a `.ais` extension contain a description of each **AADL Inspector** plugin. Each plugin defines one or several services that will be available via menu options, buttons or the command line. Each service is described by a sequence of elementary instructions.

The `AIConfig.ini` file contains the declaration of several groups of user variables: *config, projectExplorer, plugins, gantt, accelerators and userConstants*. These options are not supposed to be changed by the end user without assistance from technical support or explicit recommendation provided in user documentation.

The `License` file contains the validation keys that enable the use of the fully featured configuration of the tool in compliance with the terms of end user license. Please refer to chapter 2.3 for more detailed information on that topic.

In the standard distribution, the `config` directory contains the following additional sub-directories and files:

2.2.2.1. plugins

These plugins can be removed and customized. New plugins can also be added there. They are not platform dependent and are located in the `plugins` subdirectory. This section only lists the files that correspond to hardwired features. User customizable features are also available in the various plugins; they are part of the **LAMP** library (`LAMPLib`) and are consequently not mentioned here.

AADL off-line Static Analysis:

Set of predefined analysis rules that apply to current **AADL** model. Some of the rules are defined in **Prolog** and use the **AADL LMP** parser and libraries, others are checked thanks to specific **Ocarina** services. These rules cannot be modified by the end user.

- `1_StaticAnalysis.ais`: plugin description file.
- `metrics.sbp`: **AADL** parse and instantiate with **LMP.**
- `naming.sbp`: **AADL** naming rules checker.
- `legality.sbp`: **AADL** legality rules checker.
- `consistency.sbp`: **AADL** consistency rules checker.
- `arinc653.sbp`: **ARINC 653** rules checker.

AADL on-line Static Analysis (**LAMP** Lab):

Use the same **LMP** technology as above, but the **Prolog** rules can be directly included inside the **AADL** model within dedicated **LAMP** annex subclauses.

- `2_LAMP.ais`: plugin description file.
- `lampchecker.sbp`: check rules defined in **LAMP** annexes in the **AADL** model.
- `lampexec.sbp`: execute the query specified in a dialog box.
- `instances.sbp`: build the main **AADL** instance model predicates.
- `omgumlparser.sbp`: create a **UML 2.5.1** facts base using the **OMG** metamodel.
- `omgsysmlparser.sbp`: create a **SysML 1.5** facts base using the **OMG** metamodel.
- `mdsysmlparser.sbp`: create a **SysML** facts base with **Magic Draw ™** extensions.
- `faceparser.sbp`: create a **FACE 3.0** facts base.

AADL Timing Analysis:

- `3_TimingAnalysis.ais`: plugin description file.
- `chronogram.sbp`: timelines configuration rules.
- `schedulability.sbp`: **AADL** to **Cheddar 3.2** model transformation.
- `marzhinv2.sbp`: **AADL** transformation rules for **Marzhin.**
- `scenario.sbp`: simulation scenario template generator.
- `Marzhin.xml`, `MarzhinLogs.xml`: simulation configuration files.

Safety and Security Analysis:

- `4_SafetySecurityAnalysis.ais:` plugin description file.
- `openpsa.sbp`: generate a fault tree from **AADL EMV2** into **OpenPSA** standard.

Code Generation:

- `5_CodeGenerator.ais`: plugin description file.

Documentation Generation:

- `6_DocGenerator.ais`: plugin description file.

Foreign Model Import

- `Import.ais`: plugin description file.
- `marte.sbp`: **UML MARTE** to **AADL** model transformation.
- `capella.sbp`: **Capella** to **AADL** model transformation.
- `lampimport.sbp`: customizable **SysML** and **FACE** to **AADL** transformations.

AADL Model Templates

- `Templates.ais`: plugin description file.
- `rts.sbp`: template of a multi-thread model.
- `tsp.sbp`: template of a multi-partition model.
- `amp.sbp`: template of a multi-processor model.
- `bmp.sbp`: template of a multi-core model.
- `lamptemplate.sbp`: template of a lamp model.

Miscellaneous:

- `Others.ais`: plugin description file for inline features.
- `Utilities.ais`: plugin description file for helpers and external tools.
- `aadlgen.sbp`: **AADL** generator (unparser).

- `aadlgen2.sbp`: light version of the **AADL** generator (i.e. without **Prolog** libraries).
- `readRTProperties.sbp`: **AADL** real-time properties reader.
- `writeRTProperties.sbp`: **AADL** real-time properties writer.
- `rootselector.sbp`: **AADL** instance model configuration rules.
- `ecore.sbp`: create a **LMP** parser from a metamodel expressed in **Ecore**.
- `emof.sbp`: create a **LMP** parser from a metamodel expressed in **EMOF**.
- `uml.sbp`: create a **LMP** parser from a metamodel expressed in **UML**.

2.2.2.2. images

This directory may contain images that can be referenced in the plugin definition files. It is especially useful to specify a specific icon to launch a customized service or to change the company logo that is included in the generated documentation.

2.2.3. Examples subdirectory

This directory contains a set of **AADL** examples to practice the use of **AADL Inspector**. Five kinds of files are accepted:

- `.aic`: **AADL Inspector** project files containing a list of individual file pathnames or **URL**s, or of sub-project references.
- `.aadl`: individual **AADL** source files. They may contain several Packages and Property Sets.
- `.asc`: **AADL Inspector** simulation scenarios files.
- `.txt`: textual description files.
- image files of various formats.

It is recommended that a project file is loaded rather than individual **AADL** files to ensure all the required **AADL** Packages and Property Sets that are required to activate the analysis tools are opened.

- `all_examples.aic`: loads all the examples. They can be further selected individually in the project browser.
- `patterns.aic`: loads the next seven projects listed below. They illustrate the main communication and scheduling protocols that are supported by **AADL**.
- `dataflow.aic`: dataflow communication between threads. It can be used to observe the effect of *Sampled*, *Immediate* and *Delayed* data port connections.
- `messages.aic`: message-based communication between threads using queued events. It can be used to observe input queue overflow.
- `shared_data.aic`: shared data communication between threads. It can be used to observe the effect of the *Priority_Ceiling_Protocol* to avoid a deadlock.
- `client_server.aic` : subprogram call communication between threads. It can be used to observe the effect of the client-server synchronisation protocols.
- `arinc653.aic`: two-layer hierarchical scheduling. It can be used to investigate time and space partitioned systems with the **AADL ARINC653 Annex**.
- `scheduling.aic`: illustration of the supported scheduling protocols: *Rate Monotonic*, *Deadline Monotonic*, *High Priority First*, *Round Robin* and *Earliest Deadline First*.
- `dispatching.aic`: various thread dispatching protocols. It can be used to compare the behaviour of *Periodic*, *Sporadic*, *Aperiodic*, *Hybrid*, *Timed* and *Background*

threads.

- `calculator.aic`: integer arithmetics with the **AADL** Behaviour Annex. It can be used to show the math library capabilities and the interaction between the user and the simulator.
- `canbus.aic`: bus communication between processors. It can be used to observe the impact of bus latency on thread scheduling.
- `coffee.aic`: a coffee machine control system. It can be used to show conditional computation with the **AADL** Behaviour Annex.
- `display_system.aic`: a large model (5 processors, 22 processes and 123 threads). It can be used to check the scalability of the tools. Note that full analysis of this model can take a few minutes.
- `flight_deck_door.aic`: access control to a flight deck door. This model can interact with a 3D virtual reality simulation.
- `mars_pathfinder.aic`: several threads with different priority and sharing common data. It can be used to observe the priority inversion problem.
- `multicore.aic`: partitioned scheduling on a dual-core processor. It can be used to practice the automatic thread placement wizard.
- `pacemaker.aic`: ventricular pacemaker simulator.
- `redundancy.aic`: a simplistic Fault Detection Isolation and Recovery system. It uses the **AADL** Behavior Annex to detect erroneous values and isolate the corresponding devices.
- `regulator.aic`: a temperature regulation system. It can be used to illustrate the design and analysis of a discrete control system with the **AADL** Behaviour Annex.
- `satellite.aic`: a model defined in the **AADLib** github repository. It can be used to experiment remote model loading via the internet.
- `code_generation.aic`: a basic test case for **Ada** and **C** code generation with **Ocarina**.
- `end_to_end_flow.aic`: a generic control system with sensors, data processing and actuators that highlights end to end flows. It can be used to perform Scheduling Aware Flow Latency Analysis (**SAFLA**) with **LAMP**.
- `lamp_examples.aic`: two separate test cases to experiment the use of the **LAMP** annex.
- `wheel_braking_system.aic`: a copy of **AADL** files that are part of the **OSATE** examples. It can be used to perform Fault Tree Analysis (**FTA**) with **Arbre Analyst**.
- `safety-security.aic`: a generic distributed control system that can be used to perform Scheduling Aware Flow Latency Analysis (**SAFLA**), Fault Tree Analysis (**FTA**) and check customizable security rules.

In addition, examples of **UML/ SysML, FACE, MARTE,** or **Capella** models are provided in the `Foreign_Models` examples sub-directory to try the import features. In all cases, the corresponding source model is parsed and transformed into a target **AADL** model that is automatically loaded in **AADL Inspector** for further processing. These models can be opened through *File/Import* in the *main menu*.

Note that only those **AADL** files that are explicitly selected will be considered by the various processing tools. When a file is selected, a green tick is shown on its icon. To select or unselect a file, simply click on the corresponding icon or the one of a parent project.

2.2.4. Environment subdirectory

The `environment` subdirectory contains the common **AADL** Property Sets and Packages that are required to properly use the processing tools. They are organized into several projects to isolate the scope of each group of predefined entities and avoid potential conflicts due to assumptions made by some of the processing tools. The proper environment configuration is automatically set by each processing plugin.

- `AIEnvironment.aic`: lists all the environment subprojects to be loaded at launch time. It references the four following ones:
- `Standard.aic`: lists the Property Sets and packages that are explicitly defined in the **AADL** standard and its published annexes.
- `Ocarina.aic`: lists the additional Property Sets that are required by the services offered by **Ocarina**.
- `cheddar.aic`: lists the additional Property Sets that are required by the services offered by **Cheddar**.
- `Ellidiss.aic`: lists the additional common Property Sets and Packages that are used by the examples. The **LAMP** libraries (**LAMP Lib**) are stored there too.

Note that the **AADL** files that are part of the environment cannot be modified directly within the **AADL Inspector** editor. Changes must be done either offline with a remote text editor, or after prior move of the files to a writable workspace.

2.2.5. Include subdirectory

The `include` subdirectory contains libraries that are required by some of the ancillary tools embedded in **AADL Inspector**. Currently, it is only needed for generating code with **Ocarina**.

2.2.6. Doc subdirectory

This directory contains this manual that can be opened from the ?/*Help main menu*. Other documentation volumes provide more details on the use of the processing tools. Note that some of these specialized documentation volumes have not been updated recently, however, most of the provided information still remains valid.

2.2.7. Command line options

AADL Inspector can be launched from a command line. The following optional parameters are available:

- `--help`
  show the list of command line options.
- `-a file1.aic,file2.aadl,file3.asc, …`
  open the specified **AADL Inspector** files at startup.
- `-r dir1, dir2,…`
  open all the **AADL Inspector** files contained in the specified directories.
- `-l logdirname`
  use the specified location to create the temporary files. If used, this information overrides the one specified by the *logDirectory* parameter in the `AIConfig.ini` file.
- `--selectroot id`

set the root of the **AADL** instance hierarchy to the specified model element id.

- `--config configdirname`
  use the specified location to set the pathname to the config directory.
- `--plugin tool.service`
  start a service of a tool as defined in a `.ais` file of the config directory.
- `--result file`
  `--result stdout`
  store the plugin result file in the specified file or in the console (**Unix** only).
  to be used with option `--plugin`
- `--pluginVar variable = value`
  set the value of a variable for further use in a plugin (`@variable`)
  to be used with option `--plugin`
- `--show false`
  launch **AADL Inspector** without showing the graphical interface (batch mode)
  default is `true` (GUI is displayed).
- `--marzhinAddress address`
  set the IP address of a remote **Marzhin** simulator to connect to.
- `--marzhinCmdPort integer`
  set the command socket port number to connect to a remote **Marzhin** simulator.
  can also be used to specify the command port number of the embedded simulator.
- `--marzhinDataPort integer`
  set the data socket port number to connect to a remote **Marzhin** simulator.
  can also be used to specify the data port number of the embedded simulator.
- `--marzhinAcknowledgePort integer`
  set the acknowledge socket port number to connect to a remote **Marzhin** simulator.
- can also be used to specify the acknowledge port number of the embedded simulator.
- `--marzhinScenario ascfilename`
  `--marzhinScenario ascfilename,scenario1,scenario2`
  apply specified scenario file (`.asc`) and optionally select individual scenarios while starting the **Marzhin** simulator.
- `--tickMax integer`
  define the default duration of the **Marzhin** simulation.
- `--debug integer`
  if set to 1 or 2, display debug information to the console.
  if set to 2, display information about the **Marzhin** simulator.
  if set to 0, no console is shown (default).
- `--server true`
  launch **AADL Inspector** in server mode (on **Linux** only).
  when running in server mode, **AADL Inspector** accepts the following commands on its standard input:
    - `loadFile filename`
    - `launchTool tool.service`

An example of use of the command line activation of **AADL Inspector** is to run **Cheddar** on a set of specified **AADL** files and get the results in a specified output file:

```
bin/AADLInspector
   –a examples/dataflow.aic
   --plugin Schedulability.cheddarTheoTest
   --result dataflow.xml
   --show false
```

Such a command will create a file containing the result below (fragment). The detailed description of the **Cheddar** output is provided in a separate annex document.

```
<results>
  <feasibilityTest name="processor utilization factor" …>
    <computation name="base period" reference="all" value="300" …/>
    <computation name="processor utilization factor with deadline"
     reference="all" value="0.78333" …/>
    <computation name="processor utilization factor with period"
     reference="all" value="0.78333" …/>
    …
  </feasibilityTest>
  <feasibilityTest name="worst case task response time" …>
    <computation name="response time"
     reference="root.my_platform.CPU.my_process.T1" value="15" …/>
    <computation name="response time"
     reference="root.my_platform.CPU.my_process.T2" value="10" …/>
    <computation name="response time"
     reference="root.my_platform.CPU.my_process.T3" value="5" …/>
    …
  </feasibilityTest>
</results>
```

2.3 License

A valid license is required to use **AADL Inspector**. Various kind of licences are available, including free of charge evaluation and education licenses. Payment of a license fee is required for commercial or industrial usage of **AADL Inspector**. Please contact your **Ellidiss** sales representative for more details (sales@ellidiss.com).

Since version 1.7, license information is stored in a separate `License` file that must be located inside the `config` directory. Licenses can be attached to a particular computer and limited in time or managed by a license tokens server over the network.

2.3.1. Node locked licenses

When the license is attached to a specific computer, or for temporary evaluation licenses, the information that must be stored inside the License file is provided looks as follows:

```
# Main License

owner <licensee identification>
mac <computer identification>
date <expiration date>
tool AADL Inspector
version 1.8
key <encryption key>
licenseKey <license key>

# End License
```

Note that the complete contents of the `License` file must be provided by **Ellidiss**. None of these fields can be modified by the end user; otherwise the license key will become invalid.

2.3.2. Floating licenses

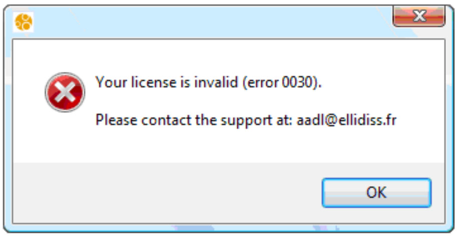
When the licenses are managed by a floating license server over the network (**ETFL**), the local License file must contain the following data:

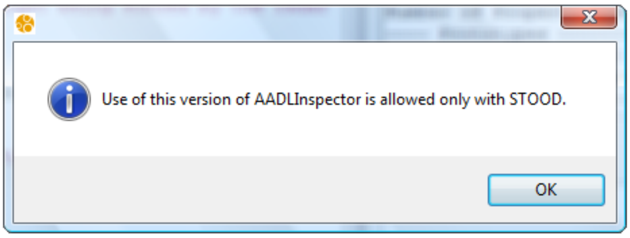
```
# Main License

owner <licensee identification>
licenseServer <server IP address>
licenseServerPort <server port address>

# End License
```

Note these fields must be compliant with the license server installation. Please contact the license server administrator to fill in the local license data.

2.3.3. License errors

In case of a mismatch between the license information and the computer identification or the current date, an error message box is displayed.





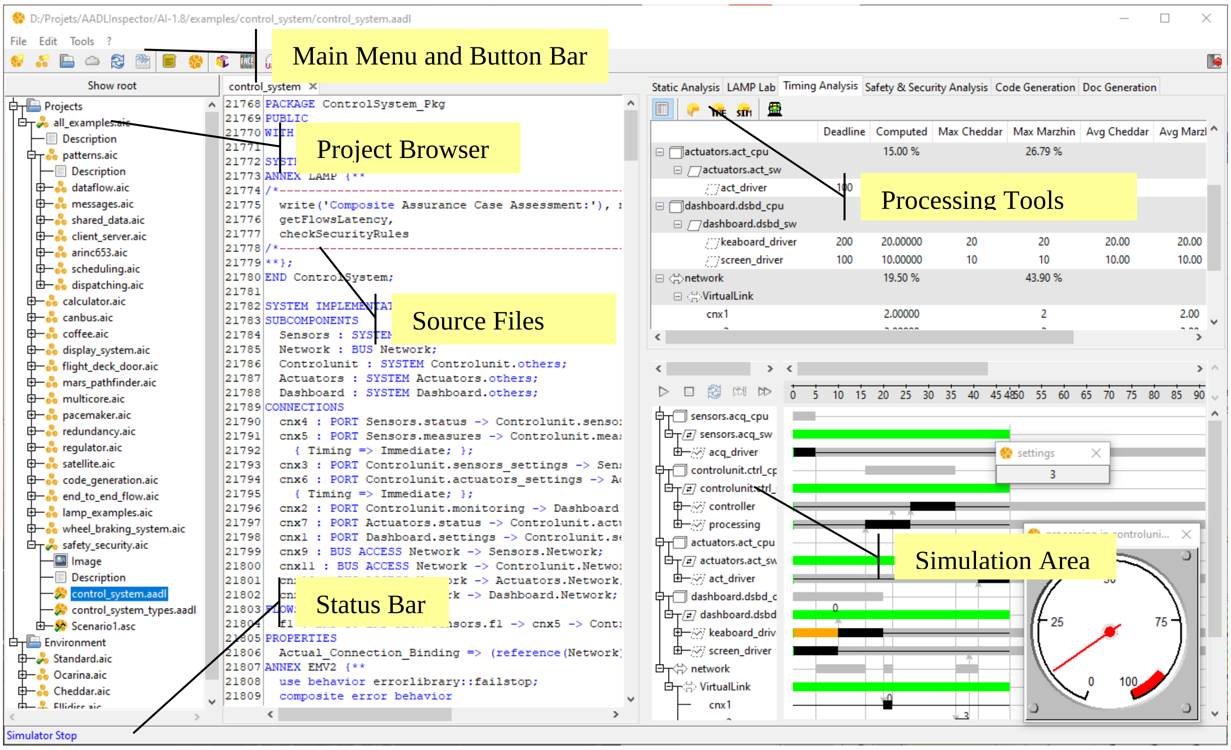
An error number is provided to help identify the license problem. Here are the most usual issues that may occur while installing the license key:

- `0010`: this license has expired
- `0020`: this license date is invalid
- `0030`: this license is attached to another computer
- `0040`: this license is linked to a **Stood** license
- `0050`: this license is not valid for this version of the product
- `0067`: this license is not valid for specified license server path
- `0069`: this license is not valid for this tool

This list of error codes is not exhaustive. Please provide the precise error code when you contact the tool support team (support@ellidiss.com) to solve the issue.

3 Graphical Interface

AADL Inspector opens a single window that encompasses a main menu bar, a button bar, a project browser, a source files area, a processing tools area, a simulation area and a status bar, as shown below:



3.1 Main menu and button bar

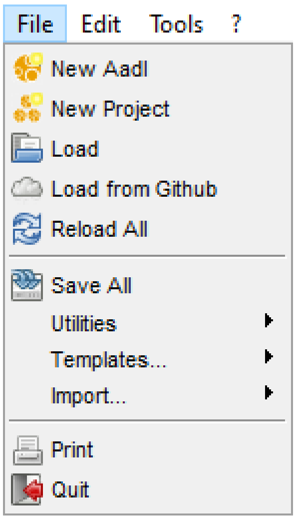
The *Main Menu Bar* contains the following pull-down menus: *File*, *Edit*, *Tools* and *?* (*Help*). The button bar provides shortcuts for frequently used menu options.

3.1.1. File menu

The *File* menu controls all file actions that have a global scope. When a model is loaded, imported or created from this menu, it will appear at the top level in the project browser (i.e. one level below the *Projects* folder). Other file actions with a more restrictive scope are provided by the contextual menus associated with the items of the project browser. The tool can process several files that together define a complete **AADL** specification. The recommended way to manage multiple files is to link them with an **AADL Inspector** project file (`.aic`). There is no particular restriction for the naming and contents of the **AADL** files. In particular, files containing several **AADL** Packages and Property Sets are allowed.

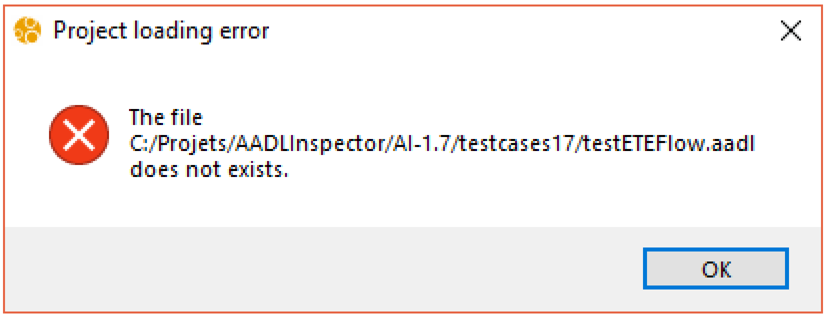
After having been loaded, **AADL** files must be selected to define the boundaries of the model to be processed. A file can be selected on unselected by clicking on its icon in the project browser tree. Files may be selected individually or collectively if the encompassing project is selected. When a file is selected, a small green tick is shown on the corresponding icon.

For most processing actions, all the selected files are concatenated together before being processed by the analysis tools. Please note that load ordering may have an impact on obtained result, especially if the root of the **AADL** instance hierarchy has not been explicitly defined. This ordering may be modified by moving the file items up or down in the project browser tree with the mouse.

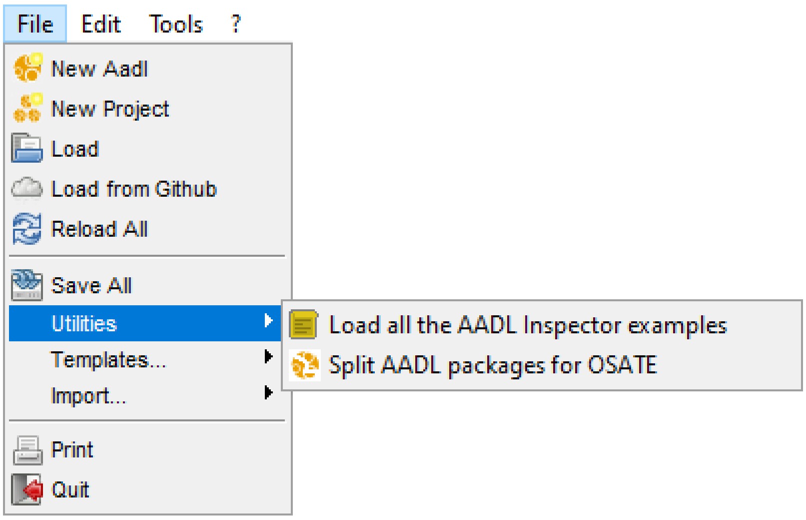


- *New Aadl*: create a new **AADL** file in memory.
- *New Project*: create a new **AADL Inspector** project file in memory.
- *Load*: load the contents of the specified **AADL** files or projects into memory.
- *Load from Github*: load files from remote **AADL** libraries (requires internet access).
- *Reload All*: cancel all the non saved changes in the project browser.
- *Save All*: save to the relevant files all the changes in the project browser.
- *Utilities*: customizable file utilities (cf. 3.1.1.1)
- *Templates*: creates a new **AADL** model applying a predefined template (cf. 3.1.1.2).
- *Import*: convert a foreign model into **AADL** and load it (cf. 3.1.1.3).
- *Print*: build an analysis snapshot of the current project and create a **PDF** file. This feature is specified in the `DocGenerator.ais` plugin definition.
- *Quit*: quit **AADL Inspector**

Note that if a file cannot be found – for instance while fetching it from github and that there is no internet connection – a message is shown in a dialog box:



3.1.1.1. Utilities sun-menu

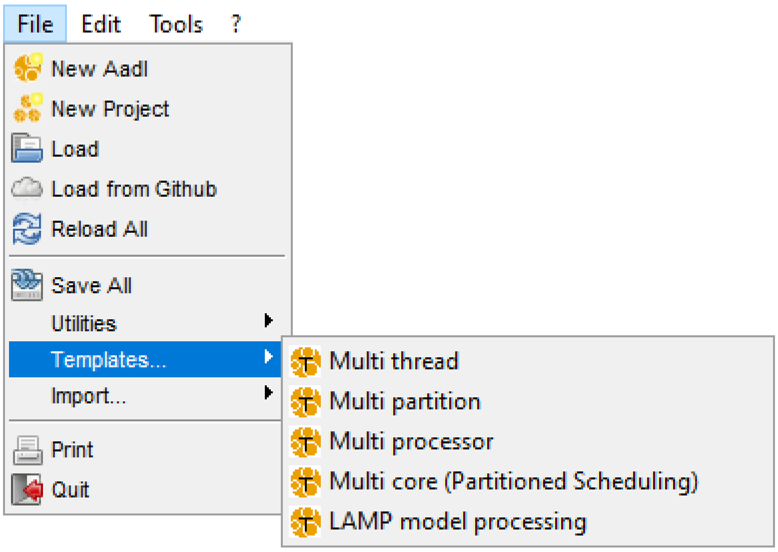


The Utilities sub-menus offer two useful features.

- *Load all the AADL Inspector examples*: shortcut to open all the examples in a single project hierarchy. Same as load `examples/all_examples.aic`.
- *Split AADL packages for OSATE*: modify the current **AADL** file structure of the selected project to ensure that each file contains a single Package or Property Set and copy them to the chosen directory to comply with this **OSATE** restriction.

Note that the contents of this sub-menu can be customized by editing the `Utilities.ais` plugin definition file.

3.1.1.2. Templates sub-menu

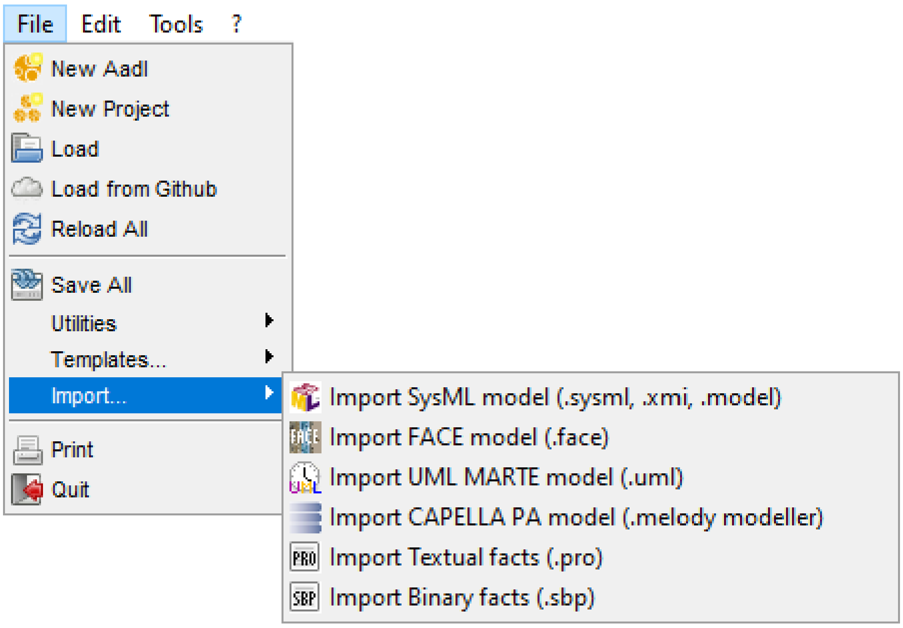


The *Templates* sub-menus can be used to quickly create an **AADL** model of a predefined kind with user parameterization. While selecting one of these sub-menu options, a dialog box is opened to enter the parameters value.

- *Multi thread*: create an **AADL** model of the given name with the given number of threads. This template can be the starting point for new real-time software (**RTS**) architectures. Threads are located on a single process and run on a single processor.

- *Multi partition*: create an **AADL** model of the given name with the given number of partitions. This template can be the starting point for new time and space partitioned (**TSP**) software architectures. Threads are distributed on several processes and run during statically defined time slots on a single processor.
- *Multi processor*: create an **AADL** model of the given name with the given number of processors. This template can be the starting point for new asymmetric multi processor (**AMP**) software architectures. Threads are distributed on several processes and run on different processors connected together by a bus.
- *Multi core*: create an **AADL** model of the given name with the given number of cores. This template can be the starting point for new bound multi processor (**BMP**) software architectures. Threads are located on a single process and run on different cores to which they are statically bound.
- *LAMP model processing*: create an **AADL** model of the given name with pre-set **LAMP** annex place holders.

3.1.1.3. Import sub-menu



The *Import* sub-menus can be used to create a new **AADL** model from “foreign” modelling languages. Proposed foreign models are **SysML**, **FACE**, **MARTE**, **CAPELLA** and **AADL** models expressed as textual or binary facts bases as specified by the **LMP** process.

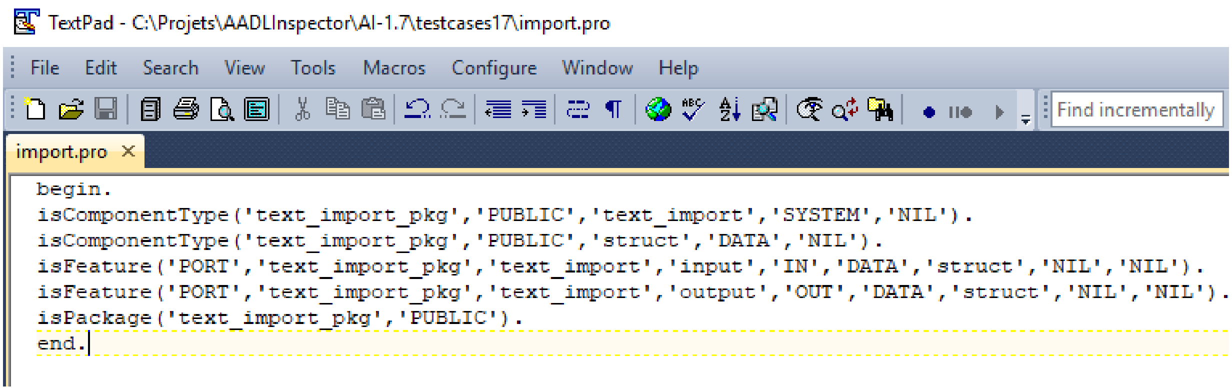
SysML and **FACE** model import features are implemented with **LAMP**, and the corresponding transformation rules are provided in the `LAMPLib`. They can thus be customized as needed.

MARTE and **CAPELLA** model import features are just provided here with a minimal implementation. Please contact **Ellidiss** technical support to adapt these import features to your project.

LMP (Logic Model Processing) was developed by **Ellidiss Technologies** to support advanced model processing tools. Dedicated **LMP** features have been packaged to support the **AADL** language. In particular, **AADL** models can be fully represented by a **LMP Prolog** facts base that can itself be serialized in textual or binary format.

- *Import SysML*: create a new **AADL** model from a foreign model expressed in **SysML** 1.5 with **Magic Draw** ™ extensions. The file navigator asks for a `.sysml`, `.xmi` or `.model` file.
- *Import FACE*: create a new **AADL** model from a foreign model expressed in **FACE** 3.0. The file navigator asks for a `.face` file.
- *Import UML MARTE*: create a new **AADL** model from a foreign model expressed with the **MARTE** profile. The file navigator asks for a `.uml` file.
- *Import Capella model*: create a new **AADL** model from a foreign model representing a **CAPELLA** Physical Architecture. The file navigator asks for a `.melodymodeller` file.
- *Import Textual facts*: create a new **AADL** model from a **LMP Prolog** textual facts base. The file navigator asks for a `.pro` file.
- *Import Binary facts*: create a new **AADL** model from a **LMP Prolog** binary facts base. The file navigator asks for a `.sbp` file.

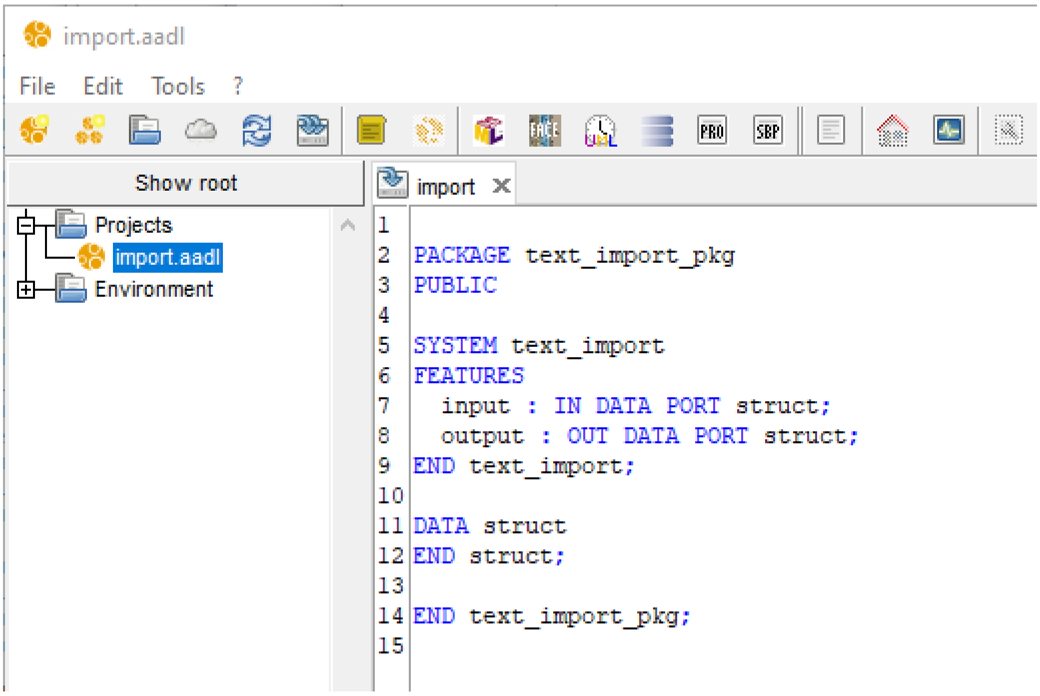
The *Import Textual facts* feature provides a very convenient way to create an **AADL** model without taking care of the statements ordering and syntax. **LMP** predicates can be used to automatically generate the **AADL** specification. These predicates can be included into a `.pro` file with any text editor or generated by a tool. An example of such a list of predicates is shown below:



The exhaustive list of **LMP** predicates is described in the **Ellidiss** technical support website: https://www.ellidiss.fr/public/wiki/aadlDeclarativeModel.

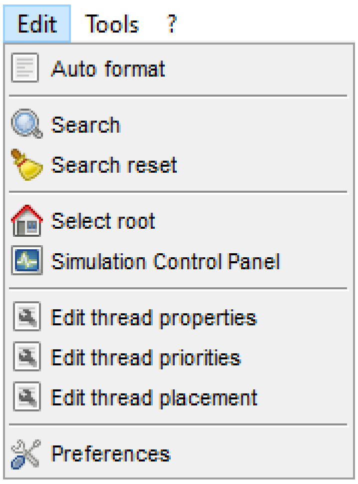
Note that the **LMP** predicates may have their last parameter (line number) or not, and that either the first predicate is `isVersion/4` or two dummy predicates `begin.` and `end.` are inserted at the beginning and at the end of the file.

Then, the use of the *Import Textual facts* menu to load this file will automatically create the corresponding **AADL** specification:



3.1.2. Edit menu

The *Edit* menu provides advanced functions used to perform changes on the input **AADL** specification. When possible, the original source text is not modified, and the changes are applied to an extension of the main system implementation of the project instead.



3.1.2.1. Auto format

This wizard re-writes the current **AADL** file into a normalized form. It impacts the case of identifiers and keywords, the indentation, and the number of blank lines. This feature can also be used to convert older **AADL** files into **AADL** 2.2 syntax, except for some values of v1.0 Property Associations.

```
package math public
data float end float;
data complex end complex;
data implementation complex.impl
subcomponents
re:data float;
im:data float;
end complex.impl;
end math;
```

```
PACKAGE math
PUBLIC

DATA float
END float;

DATA complex
END complex;

DATA IMPLEMENTATION complex.impl
SUBCOMPONENTS
  re : DATA float;
  im : DATA float;
END complex.impl;

END math;
```

The *Auto format* wizard runs the **AADL** parser on the original **AADL** specification as shown on the left-hand side of the picture above, performs an « identity » model transformation and then applies the **AADL** unparser to get a formatted **AADL** specification as shown on the right hand side.

Note that it is possible to customize the format produced by the *Auto format* wizard thanks to dedicated **AADL** properties. These properties can be applied to any **AADL** entity, but we recommend inserting them at the Package level. The currently supported **AADL** unparser properties control the case of identifiers and keywords, as well as the automatic insertion of a header.

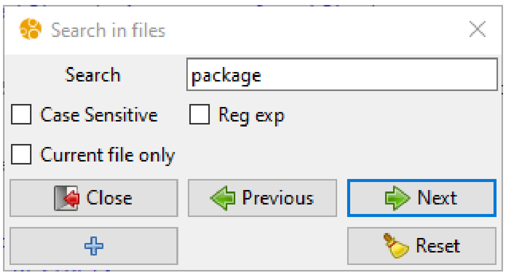
```
PROPERTY SET lmp IS

  unparser_id_case : ENUMERATION (AsIs,Upper,Lower) => Lower
    APPLIES TO (ALL);

  unparser_kw_case : ENUMERATION (AsIs,Upper,Lower) => Upper
    APPLIES TO (ALL);

  unparser_insert_header : ENUMERATION (Yes,No) => No
    APPLIES TO (ALL);

  debug_mode : AADLINTEGER
    APPLIES TO (ALL);

END lmp;
```

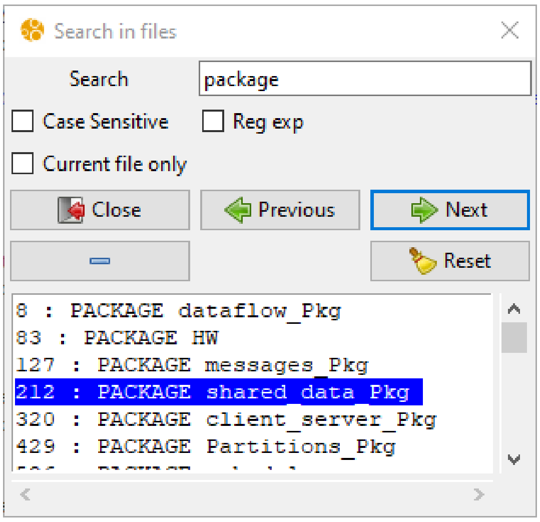
The next picture shows an example of use of these formatting properties.

```
----------------------------------------------------------
-- AADL-2.2
-- aadlrev2.10
-- (c)Ellidiss Technologie:
-- 19Jan2017
------------------------------


package MATH
public

data FLOAT
end FLOAT;

data COMPLEX
end COMPLEX;

data implementation COMPLEX.IMPL
subcomponents
  RE : data FLOAT;
  IM : data FLOAT;
end COMPLEX.IMPL;

properties
  lmp::unparser_id_case => upper;
  lmp::unparser_kw_case => lower;
  lmp::unparser_insert_header => yes;
end MATH;
```

3.1.2.2. *Search*

The *Search* tool can be used to look for all occurrences of the specified text. The scope of the search can be the currently displayed file or the complete set of loaded files. Clicking on the + button opens the list of all the text occurrences that have been found. Select a line in this list to navigate to the corresponding source text editor.



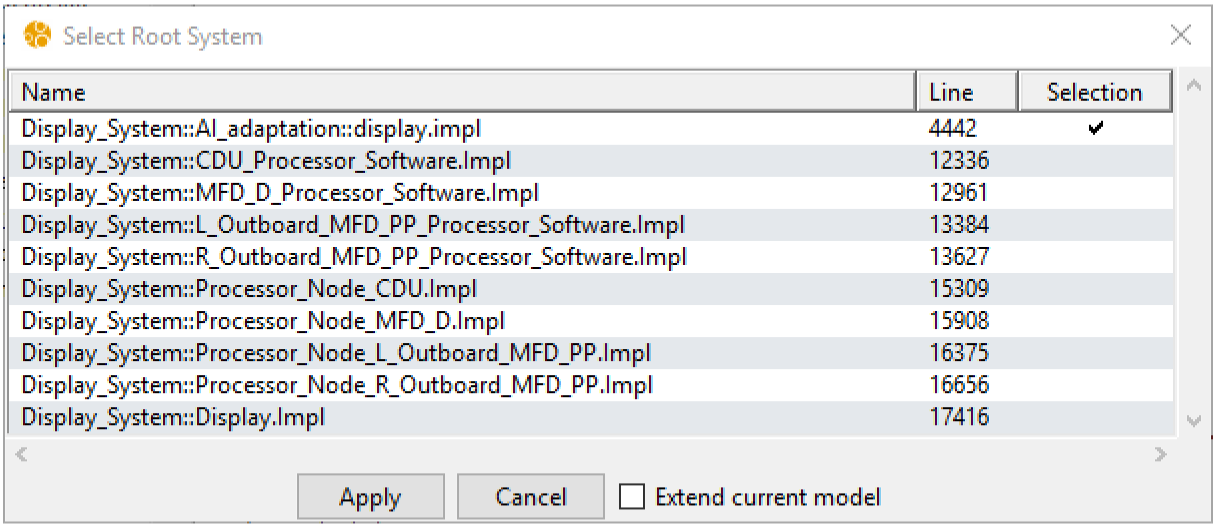


3.1.2.3. *Search reset*

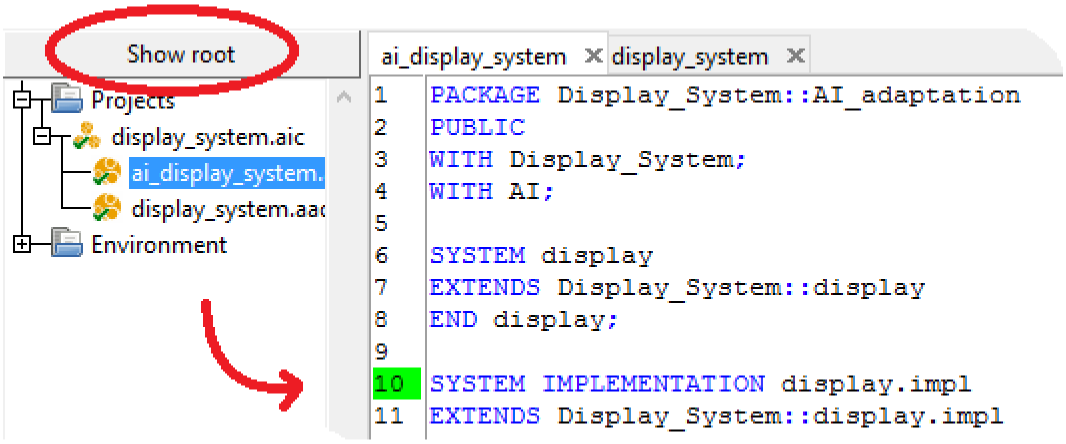
Clean up the *Search* information in the dialog box and the source text editors.

3.1.2.4. *Select root*

The *Select root* wizard shows the **AADL** System Implementation component that has been automatically identified by **AADL Inspector** to be the root of the instance hierarchy and allows the user to change it if needed.



Note that it is also possible to quickly identify the current root System Implementation by clicking on the *Show root* button located on top of the *Projects browser*:



Most of the analysis and processing tools require the **AADL** declarative model to be instantiated and deployed first. **AADL Inspector** does not require this instantiation to be done statically, and the **AADL** instance model is not stored to avoid the risk of processing an outdated model. In practice, the instance model is built on the fly together with the proper model transformation that is required for each processing tool.

However, several instance models can be inferred from a given declarative model. It is thus mandatory to define which System Implementation represents the root of the instance hierarchy (System Instance). The *Select root* wizard provides the list of candidate System Implementations and selects the one to be the root of the **AADL** instance hierarchy.

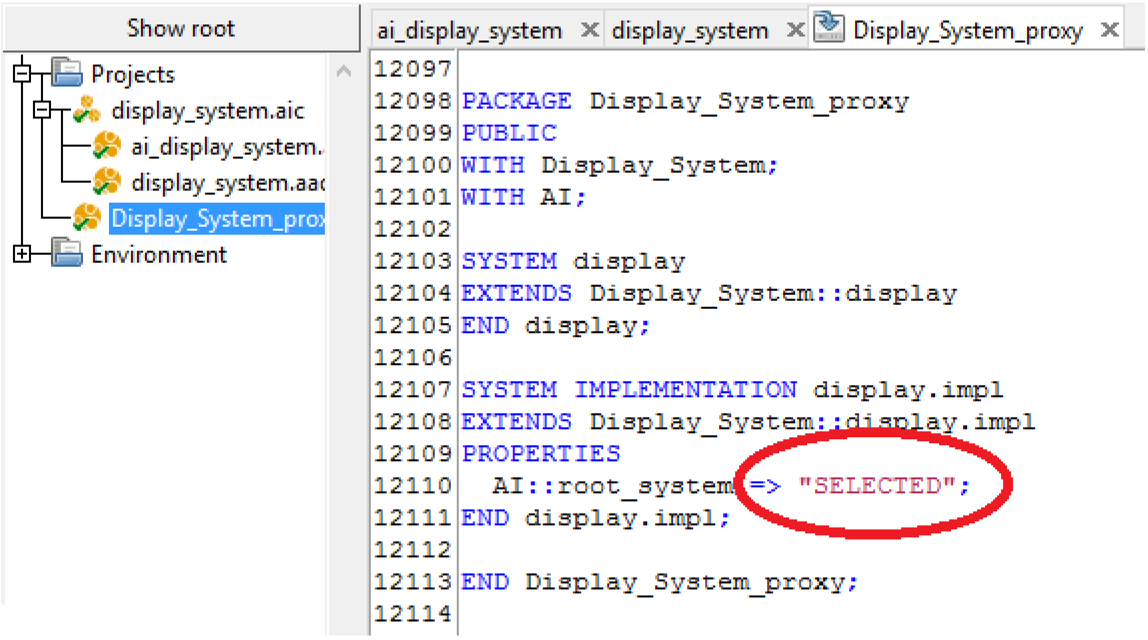
The root system that will be considered by the analysis tools will be (in decreasing priority order):

- the first found System Implementation containing an `AI::Root_System` Property association with the value `"SELECTED";`
- the first found System Implementation containing an `AI::Root_System` Property association with any other value;
- the first found System Implementation containing an `Actual_Connection_Binding` Property association;

- the first found System Implementation containing an `Actual_Processor_Binding` Property association.
- the first found System Implementation containing an `Allowed_Processor_Binding` Property association.
- the first found System Implementation that is not instantiated as a Subcomponent in the scope of the current Project.

If another root is selected in the *Select Root System* dialog box, two options are possible: either create an extended root system to avoid altering the existing files or directly modify the current model. These options are controlled by the tick box *Extend current model* in the dialog box.

When the *Extend current model* box is ticked, a new system component is created in memory only and is located in a new proxy package. The newly created system extends the one in the existing model and contains an `AI::Root_System => "SELECTED"` property association so that it becomes the new current root system.

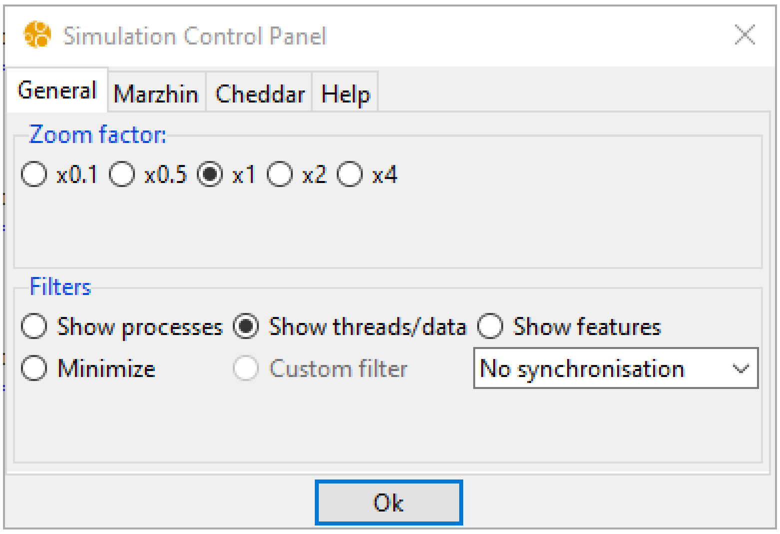


When the *Extend current model* box is not ticked (default), an `AI::Root_System => "SELECTED"` property association is directly added to the chosen system component in the original model. Note that the formatting of the original file (characters case, line returns and indentation) may be modified in that case.

3.1.2.5. Simulation Control Panel

The *Simulation Control Panel* is used to edit the various simulation parameters that can be controlled by the user. This dialog box can be opened from the main menu or button bar and is also automatically opened when the **Marzhin** simulator is started. It is composed of four tabs that can be used to control the display and behaviour of the time simulators.

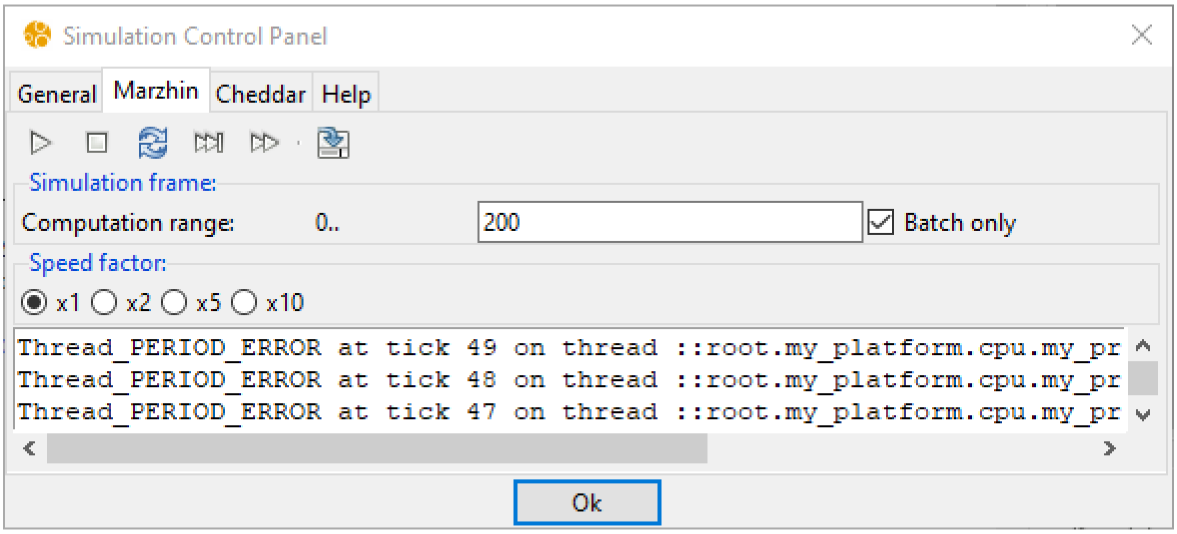
The timing analysis tools are using a virtual time scale whose unit is a *tick*. Correspondence with the actual time units that are used in the **AADL** model is given by the *reference time unit*. The *reference time unit* is the smallest time unit found in all the `Period` property associations.



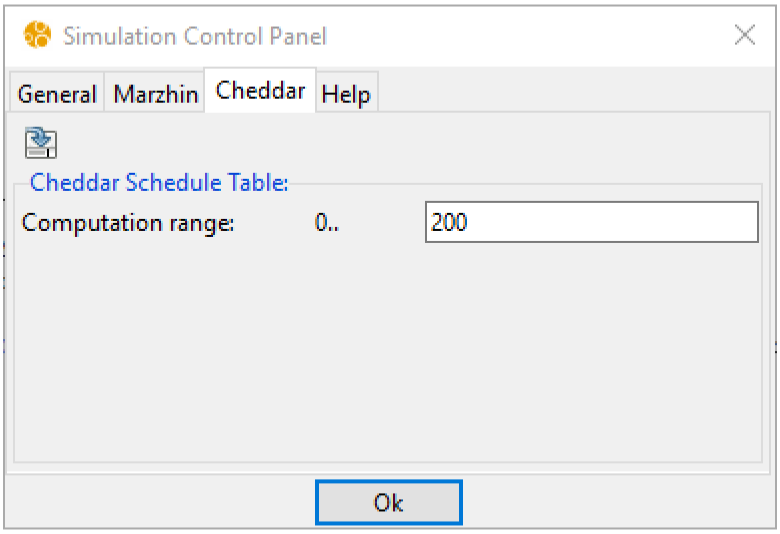
The *General* tab controls the appearance of the timelines frame. The horizontal axis (time) can be squeezed of extended with the *Zoom factor*. Note that the zoom factors can be customized in the `AIConfig.ini` configuration file. The vertical axis (model entities) can be selectively deployed thanks to the display *Filters*. The effect of these filters is described below:

- *Minimize*: only displays the Processors and the Buses.
- *Show processes*: adds a time line for each Process.
- *Show threads/data*: adds a time line for each Thread and shared Data subcomponent.
- *Show features*: adds a time line for each port, data access and subprogram access feature.
- *Custom filter*: this option is selected when the display filters are directly controlled from within the simulation display area.

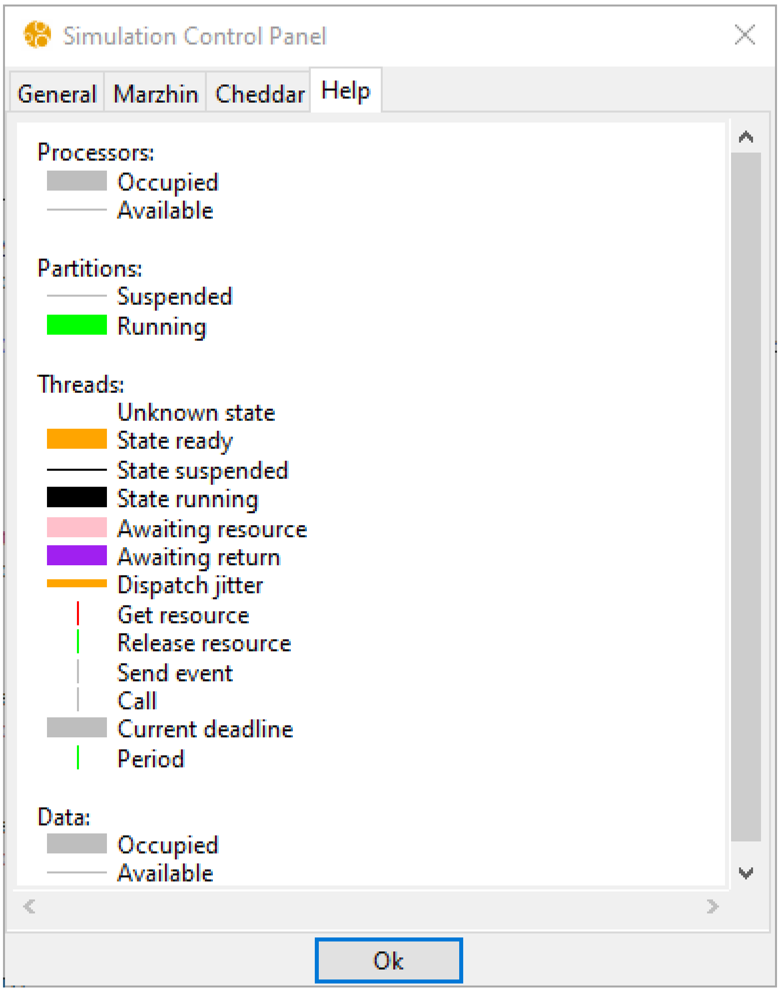
The selected filter applies to both the **Cheddar** schedule table and the **Marzhin** simulation trace. However, when a custom filter is set, it is possible to decide if the changes done on one simulation frame will not be mirrored on the other one (*No synchronization*), or it will be propagated to the other one (*Sync on Marzhin* or *Sync on Cheddar ST*).



The *Marzhin* tab is used to interact with the **Marzhin** simulator. It contains a remote panel of the simulator main commands (*start/pause/resume*, *stop*, *refresh*, *go to last tick* and *optimize*) that are described in section 3.5.1, a *save as VCD…* command to store the current simulation trace in a file, a *Speed factor* selector and a display area where message issued by the simulator are shown. The *Computation range* box can be used to limit the simulation duration to the specified number of ticks. Its default value can be specified in the `AIConfig.ini` file. By default, this limitation only applies when the simulator is run in batch mode (e.g., for assurance cases depending on simulation outputs or document generation). If this limitation must also apply to interactive simulations, the *Batch only* box must not be ticked.



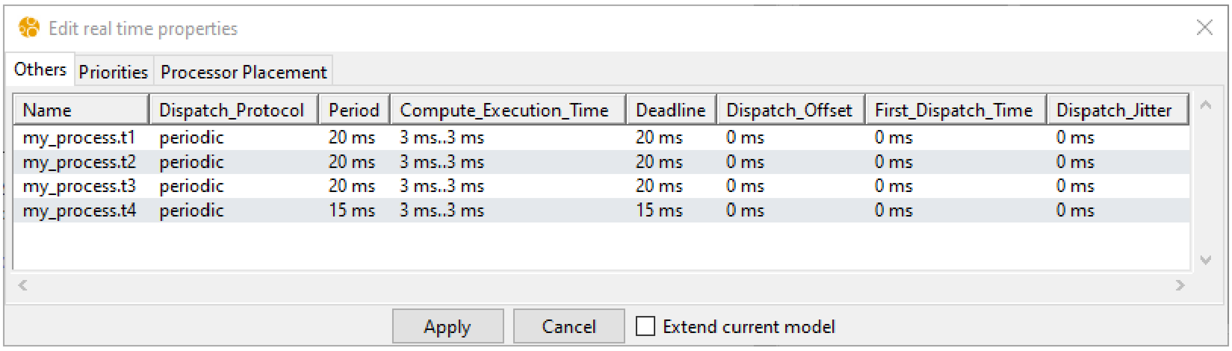
The *Cheddar* tab can be used to define the time window for computing the **Cheddar** static simulation (*Cheddar Schedule Table*). Minimizing the *Computation range* can significantly reduce the computation time on large models. Its default value can be specified in the `AIConfig.ini` file. This tab also contains a *save as VCD…* command to store the current simulation trace in a file.



The *Help* tab provides a caption for the colour code that is associated with the various states of the modelling entities that are observed during the simulation. The default values are explained in section 3.4.2. Note that this colour code can be customized in the `AIConfig.ini` configuration file.

3.1.2.6. *Edit thread properties*

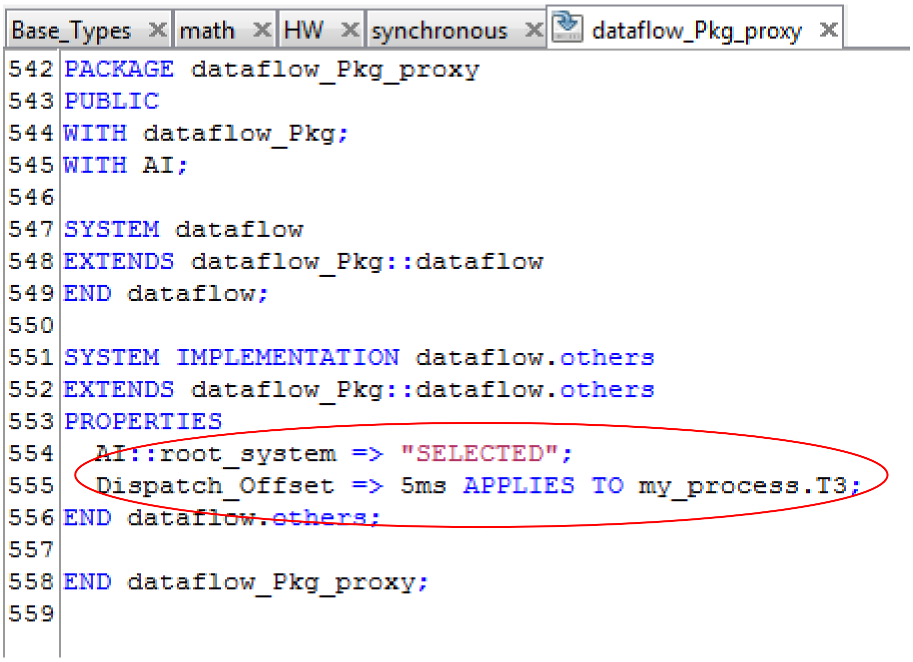
This wizard opens a spreadsheet to edit usual real-time Properties and apply them to the current model. The current Property values that are found in the selected **AADL** files are shown.



When these values have been modified, the corresponding **AADL** Property associations are either directly changed inside the current model or declared as contained Properties of an extension of the current root System Implementation. The extended root System is created in memory only and is located in a new proxy Package. The newly created System contains an

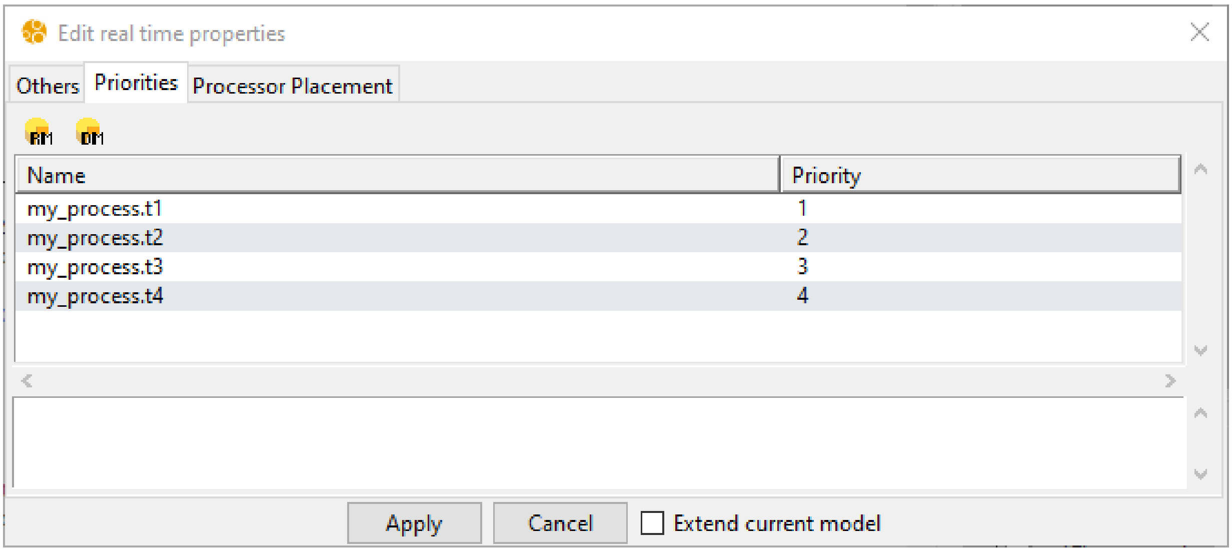
AI::Root_System Property association so that it becomes the new current root System to ensure that the new Property values are used.

The *Extend current model* tick box is used to control whether the current model is modified (default case) or an extended root System is created. Note that the formatting of the original file (characters case, line returns and indentation) may be modified in the former case.



3.1.2.7. *Edit thread priorities*

This wizard opens a spreadsheet to manually specify or automatically compute the Threads priority according to rate monotonic (*RM*) or deadline monotonic (*DM*) algorithms.



When priorities have been modified, the corresponding **AADL** Property associations are either directly changed inside the current model or declared as contained Properties of an extension of the current root System Implementation. The extended root System is created in memory only and is located in a new proxy Package. The newly created System contains an AI::Root_System Property association so that it becomes the new current root System to ensure that the new Property values are used.

The *Extend current model* tick box is used to control whether the current model is modified (default case) or an extended root System is created. Note that the formatting of the original file (characters case, line returns and indentation) may be modified in the former case.

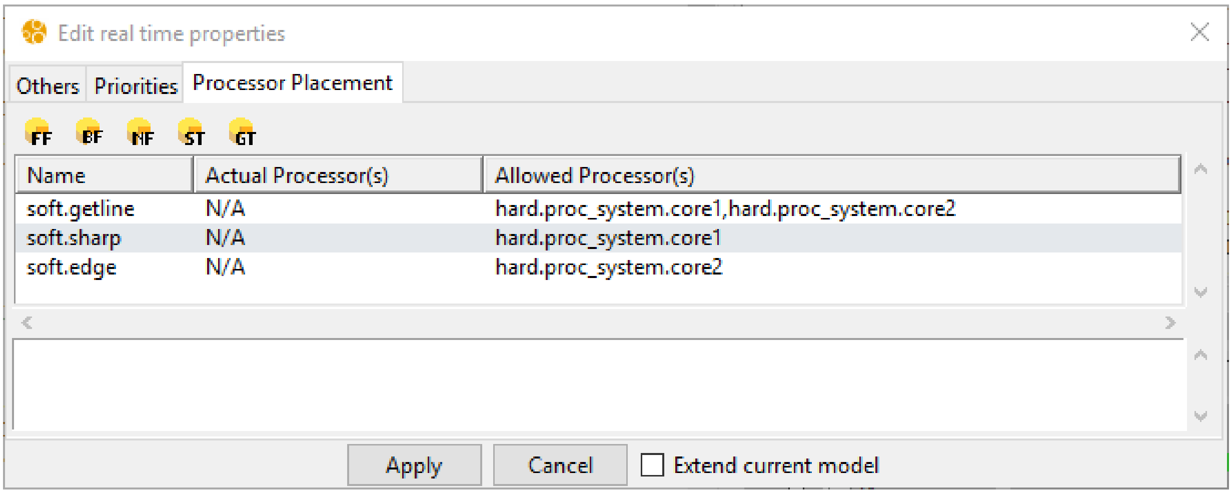
3.1.2.8. *Edit thread placement*

This wizard opens a spreadsheet to automatically compute the Threads placement onto the available Processors according to various placement algorithms. Typical use of this tool is to statically allocate Threads on a multi-core architecture.

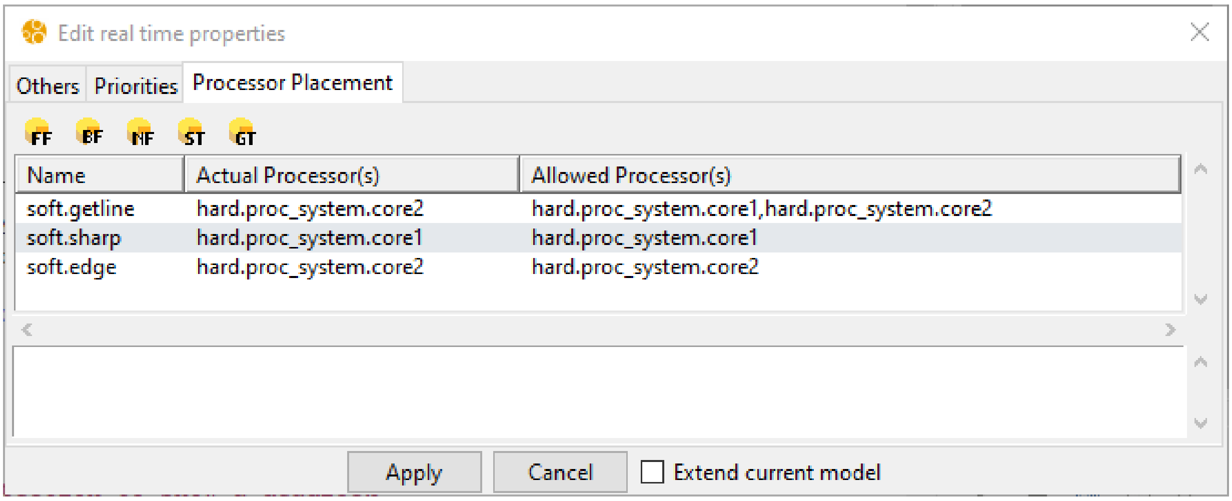
Note that global schedulers implying dynamic Thread migration between Processors (cores) are not supported yet.

```
SYSTEM IMPLEMENTATION product.impl
SUBCOMPONENTS
  hard : SYSTEM soc_leon4::soc.asic_leon4;
  soft : PROCESS edgeDetection.impl;
PROPERTIES
  allowed_processor_binding => (
    REFERENCE(hard.Proc_System.Core1),
    REFERENCE(hard.Proc_System.Core2) ) APPLIES TO soft.getLine;
  allowed_processor_binding => (
    REFERENCE(hard.Proc_System.Core1) ) APPLIES TO soft.sharp;
  allowed_processor_binding => (
    REFERENCE(hard.Proc_System.Core1) ) APPLIES TO soft.edge;
END product.impl;
```

As shown above, the original model must contain a set of Threads located in a global Process that is bound to a group of Processors with `Allowed_Processor_Binding` Property associations. This initial situation is reflected in the Processor Placement wizard. As follows:



Then, it is possible either to allocate an actual processor to each thread manually, or to apply one of the placement algorithms that are proposed by **Cheddar**: first fit (*FF*); best fit (*BF*); next fit (*NF*); small task (*ST*) or general task (*GT*).



When the proposed placement is accepted (*Apply* button), the wizard generates corresponding **AADL** `Actual_Processor_Binding` Property associations. These Properties are either directly inserted inside the current model, or declared as contained Properties of an extension of the current root System Implementation. The extended root System is created in memory only and is located in a new proxy Package. The newly created System contains an `AI::Root_System` Property association so that it becomes the new current root System to ensure that the new Property values are used.

The *Extend current model* tick box is used to control whether the current model is modified (default case) or an extended root System is created. Note that the formatting of the original file (characters case, line returns and indentation) may be modified in the former case.

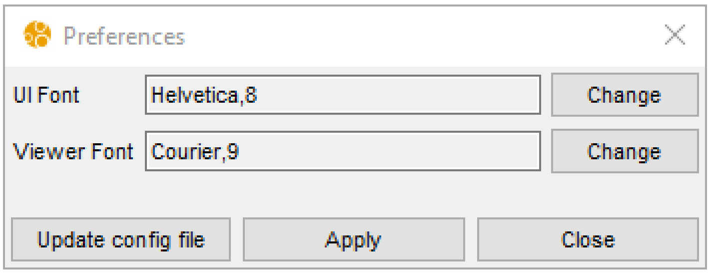
Note that the current wizard does not check that the actual binding matches the allowed bindings list.

app ✕ app_proxy ✕

```

PACKAGE app_proxy
PUBLIC
WITH app;
WITH AI;

SYSTEM product
EXTENDS app::product
END product;

SYSTEM IMPLEMENTATION product.impl
EXTENDS app::product.impl
PROPERTIES
  AI::root_system => "SELECTED";
  Actual_Processor_Binding =>
     (reference(hard.proc_system.core2))
     APPLIES TO soft.getline;
  Actual_Processor_Binding =>
     (reference(hard.proc_system.core1))
     APPLIES TO soft.sharp;
  Actual_Processor_Binding =>
     (reference(hard.proc_system.core2))
     APPLIES TO soft.edge;
END product.impl;

END app_proxy;
```

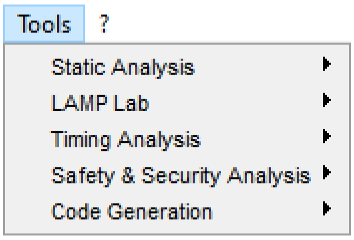
3.1.2.9. Preferences

The *Preferences* menu opens a dialog box to change the fonts used by the application. Two fonts are used by the tool. The *UI Font* applies to all menu items, tab names and the project explorer elements. The *Viewer Font* is used to display text in the editing area as well as in the analysis report areas. The latter one is intended to be a monospaced font.



Note that the default values are defined in the `AIConfig.ini` file. It is possible to update these values using the *Update config file* button.

3.1.3. Tools menu

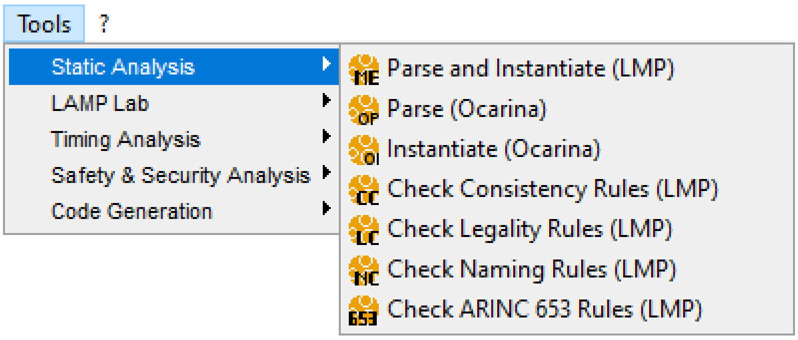


The *Tools* menu provides access to the processing tools and services that are defined in the `.ais` files located in the `config` directory. Six tools are available with the standard distribution: *Static Analysis*, *LAMP Lab*, *Timing Analysis*, *Safety &Security Analysis*, *Code Generation*, and *Utilities*. Each menu item opens a submenu that gives access to the services offered by the corresponding tool.

Except for *Utilities*, each item of the *Tools* menu corresponds to a tab in the *Processing tools area* in the left-hand side part of the main window, and each submenu is associated with a button of the corresponding tab (cf. 3.4).

3.1.3.1. Static Analysis

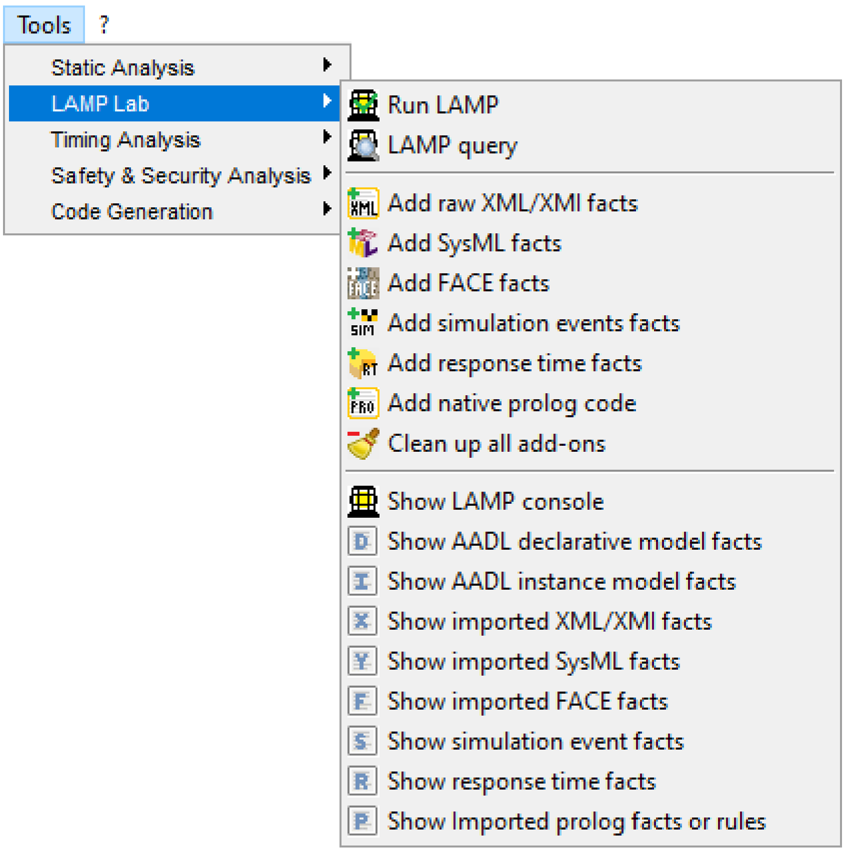
The static analysis services make use of two different and complementary technologies. One is based on the Logic Model Processing (**LMP**) toolbox and the other one is provided by calls to the **Ocarina** tool.



- *Parse and Instantiate (LMP)*: parse the selected **AADL** files, instantiate the model from the root System instance (cf. 3.1.2.3), perform quick consistency analysis and provide statistics about both the instance and the declarative **AADL** models.
- *Parse (Ocarina)*: parse the selected **AADL** files and check the consistency, legality and naming rules defined by the standard, with a call to **Ocarina** –p.
- *Instantiate (Ocarina)*: instantiate the **AADL** model with a call to **Ocarina** –i.
- *Check Consistency Rules (LMP)*: verify the consistency rules defined by the standard.
- *Check Legality Rules (LMP)*: verify the legality rules defined by the standard.
- *Check Naming Rules (LMP)*: verify the naming rules defined by the standard.
- *Check ARINC 653 Rules (LMP)*: verify rules for partitioned systems.

3.1.3.2. LAMP Lab

LAMP stands for Logic **AADL** Model Processing. It is an online processing language that can be directly included within **AADL** Packages and Components as Annex sub-clauses. This language is the same as the one that is used for the definition of the predefined plug-ins and wizards (**LMP**). **LMP** consists of a set of parsers, a **Prolog** engine and libraries to access and process model elements. These features are available to create customized assurance cases functions that can be modified interactively. The **LAMP** services are organized in three groups as shown below:



The first group of services control the execution of the **LAMP** engine:

- *Run LAMP*: load the contents of all the **LAMP** annexes that are found in the selected **AADL** user files and environment libraries and run the included queries (goals).
- *LAMP query*: same as above but ignore the goals that are included inside the **LAMP** annexes and ask for a query in a dialog box instead.

The second group of services enables addition of information that are not present within the **LAMP** annexes. All these additions are inclusive. This is especially useful to perform cross-model processing.

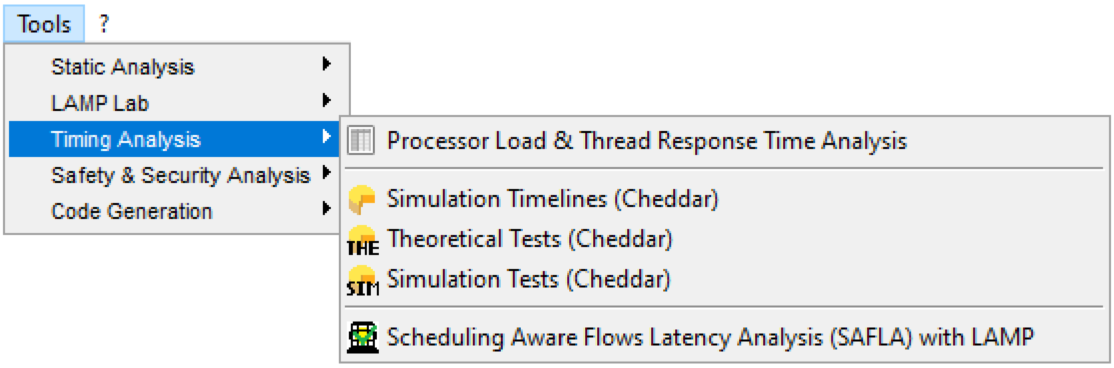
- *Add raw XML/XMI facts*: parse specified **XML** file and load corresponding **Prolog** facts before next execution of the **LAMP** queries.
- *Add SysML facts*: parse specified **XMI** file, interpret it according to the **UML** and **SysML** metamodels and load corresponding **Prolog** facts before next execution of the **LAMP** queries.
- *Add FACE facts*: parse specified **XML** file, interpret it according to the **FACE** metamodel and load corresponding **Prolog** facts before next execution of the **LAMP** queries.
- *Add simulation events facts*: run **Marzhin** simulator and load corresponding **Prolog** facts before next execution of the **LAMP** queries.
- *Add response time facts*: run the **AADL** Threads response time computation wizard and load corresponding **Prolog** facts before next execution of the **LAMP** queries.
- *Add native prolog code*: load selected **Prolog** code before next execution of the **LAMP** queries.
- *Clean up all add-ons*: remove all previously added **Prolog** extensions before next execution of the **LAMP** queries.

The third group of services show the various available sources of information in the display area. Only one source of information is shown at a time.

- *Show LAMP console*: display output produced by the last execution of **LAMP**.
- *Show AADL declarative model facts*: show the list of **Prolog** predicates that represent the current **AADL** declarative model.
- *Show AADL instance model facts*: show the list of **Prolog** predicates that represent the current **AADL** instance model.
- *Show imported XML/XMI facts*: show the list of **Prolog** predicates generated from previously added raw **XML** or **XMI** file.
- *Show imported SysML facts*: show the list of **Prolog** predicates generated from previously added **SysML** file.
- *Show imported FACE facts*: show the list of **Prolog** predicates generated from previously added **FACE** file.
- *Show simulation events facts*: show the list of **Prolog** predicates that represent the logged **Marzhin** simulation events.
- *Show response time facts*: show the list of **Prolog** predicates that represent the computed Thread response time by **Cheddar** and **Marzhin**.
- *Show imported prolog facts or rules*: show the list of **Prolog** predicates that were previously added.

3.1.3.3. Timing Analysis

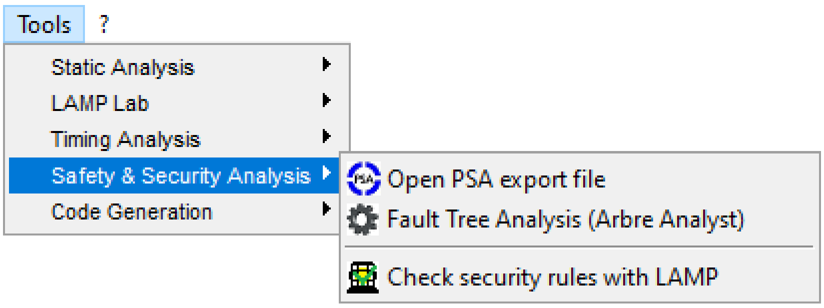
The timing analysis services make use of two different and complementary tools. One is based on the **Cheddar** scheduling analysis tool and the other one is provided by the **Marzhin** simulator. These services make use of standard **AADL** real-time Properties as well as a subset of the **AADL** Behavior Annex.



- *Processor Load & Thread Response Time Analysis*: compute statistics for processor load and thread response time from the various outputs given by **Cheddar** and **Marzhin**, and show them in a spreadsheet for comparison.
- *Simulation Timelines (Cheddar)*: static simulation computed by **Cheddar**.
- *Theroritical Tests (Cheddar)*: set of feasibility tests checked by **Cheddar**.
- *Simulation Tests (Cheddar)*: set of tests based on the static simulation computed by **Cheddar**.
- *Scheduling Aware Flows Latency Analysis (SAFLA) with LAMP*: associate response time computation done by **Cheddar** and **Marzhin** with **AADL** Flows analysis done by **LAMP** to provide an estimate of End-to-End Flows latency.

3.1.3.4. Safety & Security Analysis

This plugin groups both safety and security analysis services.



The safety analysis services aim at interfacing external tools that support model driven safety analysis. These model transformations make use of the **AADL** Error Model Annex (**EMV2**) and are currently focusing on Fault Tree Analysis (**FTA**).

- *Open PSA export file*: generate a file complying with the **Open PSA** model exchange format to export fault trees from **EMV2** declarations.
- *Fault Tree Analysis (Arbre Analyst)*: generate an **Open PSA** file as above and launch the **Arbre Analyst** tool to display a graphical fault tree. Note that the **Arbre Analyst** tool is not included into the **AADL Inspector** distribution. This tool can be found at the following address: https://www.arbre-analyste.fr/en.html

Note that once installed onto your computer and checked the terms of the license, you need to update the corresponding file pathname in the `AIConfig.ini` file before being able to use this service, for instance:

```
variable userConstants { \
        "FTAToolPath" "{C:/Projets/AADLInspector/Safety/arbre_analyste-
2.3.2-win32/Arbre Analyst.exe}" \
```

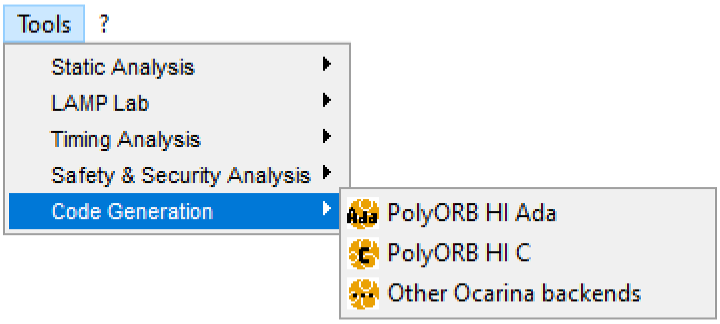
The security analysis service makes use of customizable LAMP rules:

- *Check security rules with LAMP*: execute the **LAMP** query `checkSecurityRules` that is defined in `LAMPLib`. It is based on a simplistic user defined security model with a single **AADL** Property defining the security level associated with a Data classifier.

Note that these security model and rules can be customized to fit specific security policies. As the rules defined in `LAMPLib` are read-only, it is necessary to either move the file `LAMPSecurity` to a writable workspace before editing it. An alternate solution is to edit it with another text editor, however **AADL Inspector** will need to be restarted to take changes into account in that case.

3.1.3.5. Code Generation

The code generation services are provided by **Ocarina** back-ends. Please refer to the **Ocarina** documentation for detailed explanations about the use of these features.

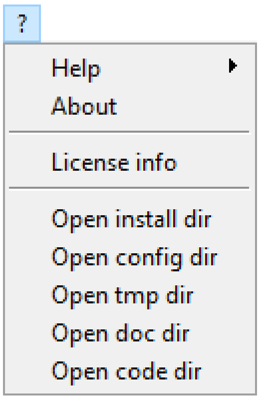


- *PolyORB HI Ada*: generate **Ada** source code files for the **PolyORB-HI-Ada** middleware. A dialog box asks about the location of the generated code. A default location is proposed in the **AADL Inspector** temporary directory.
- *PolyORB HI C*: static generate **C** source code files for the **PolyORB-HI-C** middleware. A dialog box asks about the location of the generated code. A default location is proposed in the **AADL Inspector** temporary directory.
- *Other Ocarina backends*: gives access to the other available **Ocarina** back-ends. The actual back-end to use can be selected in a dialog box.

Note that **Ocarina** generates the source code architecture and glue code with the Operating System. However, it requires the applicative functional code to be made available for a complete build of the software. Access to the functional code can be specified by `Source_Text` **AADL** Properties.

3.1.4. Help menu

The *?* menu provides information about **AADL Inspector**.



- *Help*: open the help files. Note that the name of the help file directory and the application that is used to open it can be customized in the `AIConfig.ini` file. By default, this application will be the default one for `.pdf` files on **Windows** and `xpdf` on **Linux**.
- *About*: display the version of the software.
- *License info*: provide information about the license.
- *Open install dir*: open the installation directory.
- *Open config dir*: open the configuration directory.
- *Open tmp dir*: open the temporary directory.
- *Open doc dir*: open the default documentation directory.
- *Open code dir*: open the default code generation directory.

3.1.5. Button bar

The *Main Button Bar* provides another entry point for menu actions.

The effect of these actions is described in the corresponding menu section. Button association with menu bar items is given below from left to right:

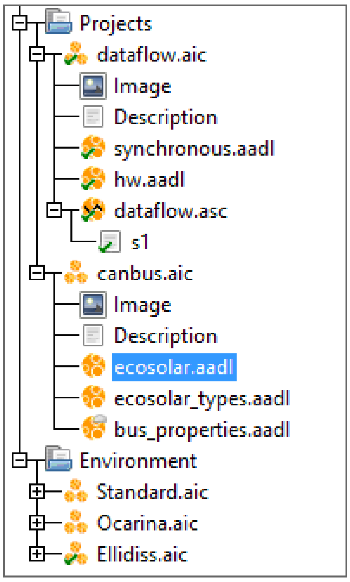
- *File/New Aadl*
- *File/New Project*
- *File/Load*
- *File/Load from Github*
- *File/Reload*
- *File/Save All*
- *File/Utilities/Load all the AADL Inspector examples*
- *File/Utilities/Split AADL packages for OSATE*
- *File/Import/Import SysML model (.sysml, .xmi, .model)*
- *File/Import/Import FACE model (.face)*
- *File/Import/Import UML MARTE model (.uml)*
- *File/Import/Import CAPELLA PA model (.melodymodeller)*
- *File/Import/Import Textual facts (.pro)*

- *File/Import/Import Binary facts (.sbp)*
- *Edit/Auto format*
- *Edit/Select root*
- *Edit/Simulation Control Panel*
- *Edit/Edit thread properties*
- *Edit/Edit thread priorities*
- *Edit/Edit thread placement*
- *Edit preferences*
- *File/Quit*

3.2 Project browser

The *Project Browser* offers advanced structuring and navigation features to manage **AADL** projects. **AADL Inspector** projects are organized hierarchically and can contain several kinds of files. **AADL Inspector** projects contents are defined in `.aic` files.

The *Project Browser* has two main sections: *Projects*, where user defined **AADL** Packages and Property Sets can be loaded or created, and *Environment*, where standard or tool dependent **AADL** Packages and Property Sets are stored. Contents of the latter cannot be modified from the **AADL Inspector** user interface.



Terminal items in the **AADL Inspector** project hierarchy can be:

- *AADL files*: containing standard textual **AADL** declarations (`.aadl`).
- *Scenarios files*: defining inputs values and time for the simulator (`.asc`).
- *Description files*: allowing for a textual documentation of the project (`.txt`).
- *Image files*: read-only illustration associated with the project (`.jpg;.jpeg:.xbm;.bmp;.png;.gif`).

Note that a single description file and a single image file can be inserted within a project.

The items of the *Project Browser* may be in different non-exclusive states that are indicated by a change of the corresponding icon or colour of the text label:

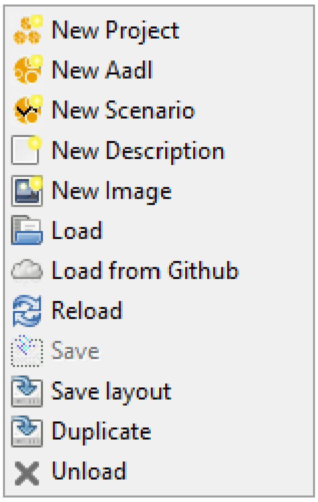
- loaded project file (icon)
- selected project file (icon)
- canbus.aic modified project file (label)
- loaded **AADL** file (icon)
- read-only **AADL** file loaded from a remote git repository (icon)
- selected **AADL** file for processing (icon)
- ecosolar.aadl currently displayed **AADL** file (label)
- ecosolar.aadl modified **AADL** file (label)
- loaded scenarios file (icon)
- selected scenarios file (icon)
- dataflow.asc modified scenarios file (label)

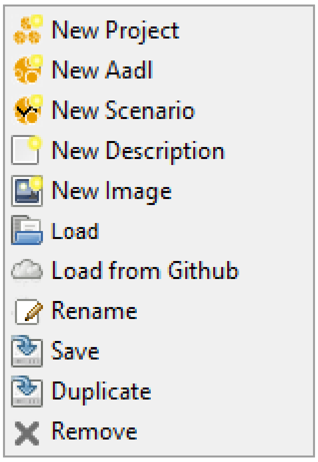
Note that the scenarios files are not terminal nodes in the browser tree. Indeed, individual scenarios are shown as sub-items in the hierarchy although they are all included in the same file. They can be selected individually if needed.

A contextual menu is associated with each kind of item and is updated according to its states to only offer the valid actions in each case.

3.2.1. Project file contextual menu

When a project is selected in the browser, the following contextual menu options are available:



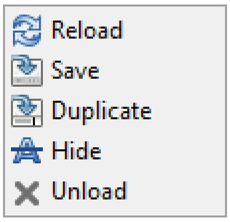


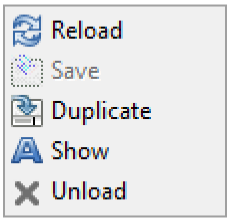
- *New Project*: create a new sub-project slot in memory.
- *New Aadl*: create a new **AADL** model slot in memory.
- *New Scenario*: create a new scenario template in memory. Note that *scenarios* can be created on instance models only. If not done yet, select the project (green tick) and use the *Show root* button on top of *the Project Browser*.
- *New Description*: create a new textual description in memory.
- *New Image*: create a new image slot in memory.
- *Load*: open a file navigator to load any of the accepted file types.
- *Load from Github*: open a dialog to load an **AADL** file from a registered server.
- *Reload*: reload the project.
- *Rename*: if newly created project has not been saved yet, rename it.
- *Save*: save the project file and its contents.
- *Save layout*: save the *selected* and *opened* status of each file contained in the project.
- *Duplicate*: create a copy of the project in memory.

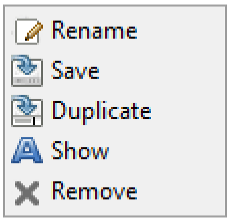
- *Unload*: remove the project and its contents from memory.

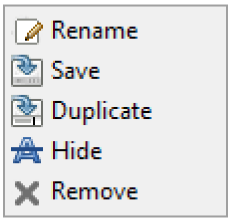
3.2.2. AADL file contextual menu

When an **AADL** file is selected in the browser, the following contextual menu options are available (right mouse button click):





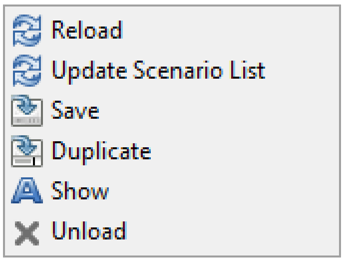


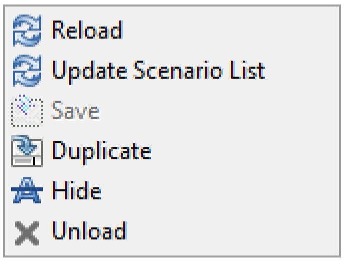


- *Reload*: reload the **AADL** file.
- *Rename*: if newly created **AADL** file has not been saved yet, rename it.
- *Save*: save the **AADL** file.
- *Duplicate*: create a copy of the **AADL** file in memory.
- *Show/Hide*: open or close a corresponding editor in the *Source File Area*.
- *Unload/Remove*: remove the **AADL** loaded/new file from memory.

3.2.3. Scenario file contextual menu

When a scenario is selected in the browser, the following contextual menu options are available:

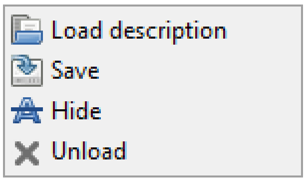


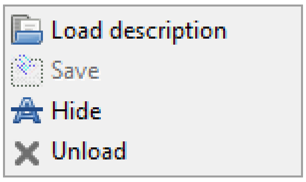


- *Reload*: reload the scenario file.
- *Update Scenario List*: update the scenario contents after editing in the *Source File Area*.
- *Save*: save the scenario file.
- *Duplicate*: create a copy of the scenario file in memory.
- *Show/Hide*: open or close a corresponding editor in the *Source File Area*.
- *Unload/Remove*: remove the loaded/new scenario file from memory.

3.2.4. Description file contextual menu

When a description file is selected in the browser, the following contextual menu options are available:





- *Load description*: open a file navigator to load a `.txt` file.
- *Save*: save the description file.
- *Show/Hide*: open or close a corresponding editor in the *Source File Area*.
- *Unload/Remove*: remove the load/new description file from memory.

3.2.5. Image file contextual menu

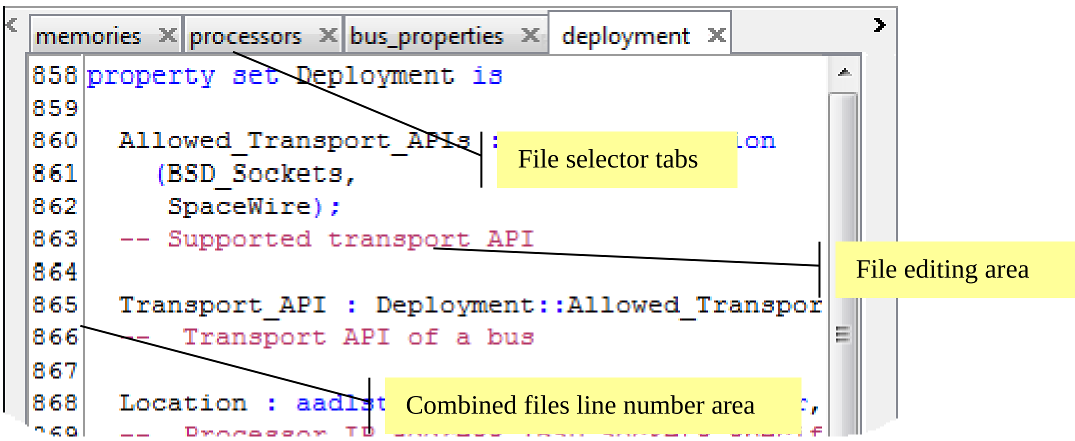
When an image file is selected in the browser, the following contextual menu options are available:

- *Load image*: open a file navigator to load a `.jpg .jpeg .xbm .bmp .png` or `.gif` file.
- *Show/Hide*: open or close a corresponding viewer in the *Source File Area*.
- *Unload*: remove the image file from memory.

3.3 Source files area

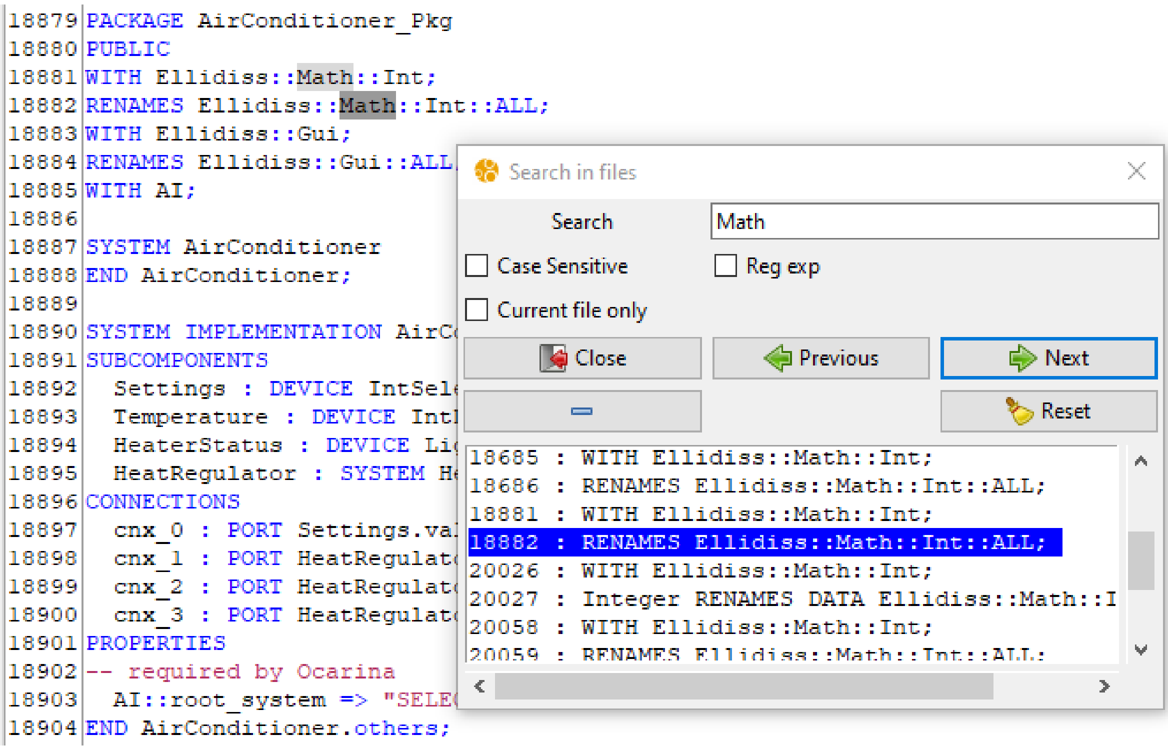
After having been loaded in the *Project browser*, the files can be opened in the *Source file area*. Closing an editor in the *Source file area* does not unload the corresponding file from the browser. The source file area is composed of:

- a set of *file selector tabs*
- a *file editing area*
- a *line number area*

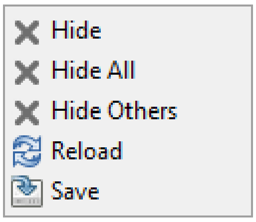


To load a file in the editing area a drag and drop action is possible instead of using the *File/Load* menu: open the appropriate directory, select the desired file, depress and hold the left mouse button then drag the mouse until the **AADL Inspector** window is reached.

To find all the occurrences of a word in the displayed file, select the desired word and press the `Ctrl-F` key to open the search dialog box. Note that the *Next* button must be pressed to start the search.

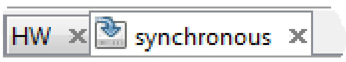


A contextual menu (right mouse button click) is associated with the current *file selector tab*.



- *Hide*: closes the currently selected tab.
- *Hide All*: closes all the opened tabs.
- *Hide Others*: closes all the opened tabs but the current one.
- *Reload*: reload last saved version of the file.
- *Save*: update the file with current contents of the editor.

When a file has been modified, an icon appears on the tab to indicate that the changes have not been saved. Clicking on the *save* icon of the tab will save the file.



Note that clicking on the grey *cross* at the right-hand side of a tab also closes the tab.

Files that can be edited in the Source files area are:

- textual AADL files: `.aadl`.

- simulator scenario files: `.asc`.
- textual description files: `.txt`.
- image files (read-only): `.jpg .jpeg .xbm .bmp .png` or `.gif`

Note that images cannot be edited in the *Source files area*. No specific action is proposed for image files that can only be loaded and displayed.

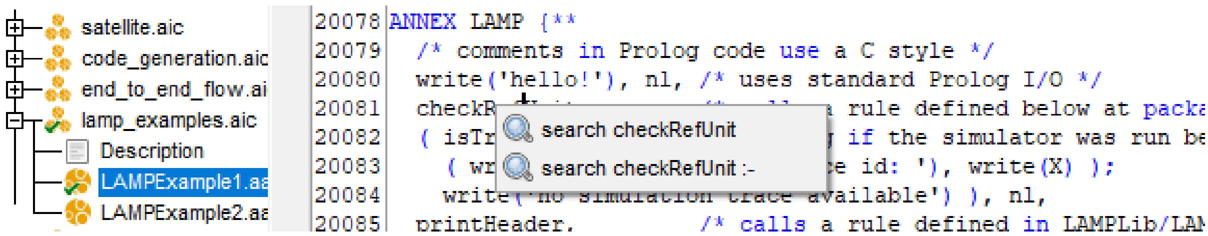
3.3.1. Editing AADL files

The textual contents of a file editor associated with an **AADL** file must comply with the syntax defined by the standard. No verification is done on text input before an analysis tool is launched.

AADL Inspector accepts **AADL** files that encompass several packages and property sets. However, the user must be aware that other **AADL** tools may have a more restrictive policy, such as enforcing the single package or property set per file rule.

When an **AADL** model is edited, line numbering is activated. Line numbers correspond to those of a virtual file that would be the concatenation of all the actual **AADL** files that are selected in the *Project browser*.

Note that a cross-reference contextual menu opens the search dialog box on the identifier pointed by the mouse. This is especially useful when editing **Prolog** code inside **LAMP AADL** annex subclauses:



3.3.2. Editing Simulator Scenario files

The textual contents of a file editor associated with a scenario file must comply with a specific **XML** syntax. No verification is done on text input before the scenario is saved.

The structure of a scenario file is as follows:

```
<scenarii>
    <interface>
        <feature type="data" id="input"
           aadlID="my_platform.cpu.my_process.t1.input"/>
        <feature type="data" id="output"
           aadlID="my_platform.cpu.my_process.t1.output"/>
        …
    </interface>
    <scenario name="s1" description="">
        <probes>
            <probe ref="output"/>
            …
        </probes>
```

```
        <tick value="0" next=”tick+10”>
            <action ref="input" value="1"/>
            …
        </tick>
        …
    </scenario>
</scenarii>
```

When a new scenario file is created from the *Project browser* (project contextual menu), its contents is initialized with the list of ports that can be triggered within the scenarios. This list is provided in the `<interface>` section and corresponds to all the input ports of the threads that are found in the current set of selected **AADL** files. A short name is given for each port so that it can be easily reused in the scenario specification.

A list of independent scenarios can then be added. Each scenario can be selected individually in the *Project Browser*. A scenario is defined by an optional `<probes>` section and a list of `<tick>` sections.

The `<probes>` section can be used to open a visualisation probe on the specified port when the scenario starts. Probes can also be opened at any time while the simulation is running. Probes may be attached to input or output ports.

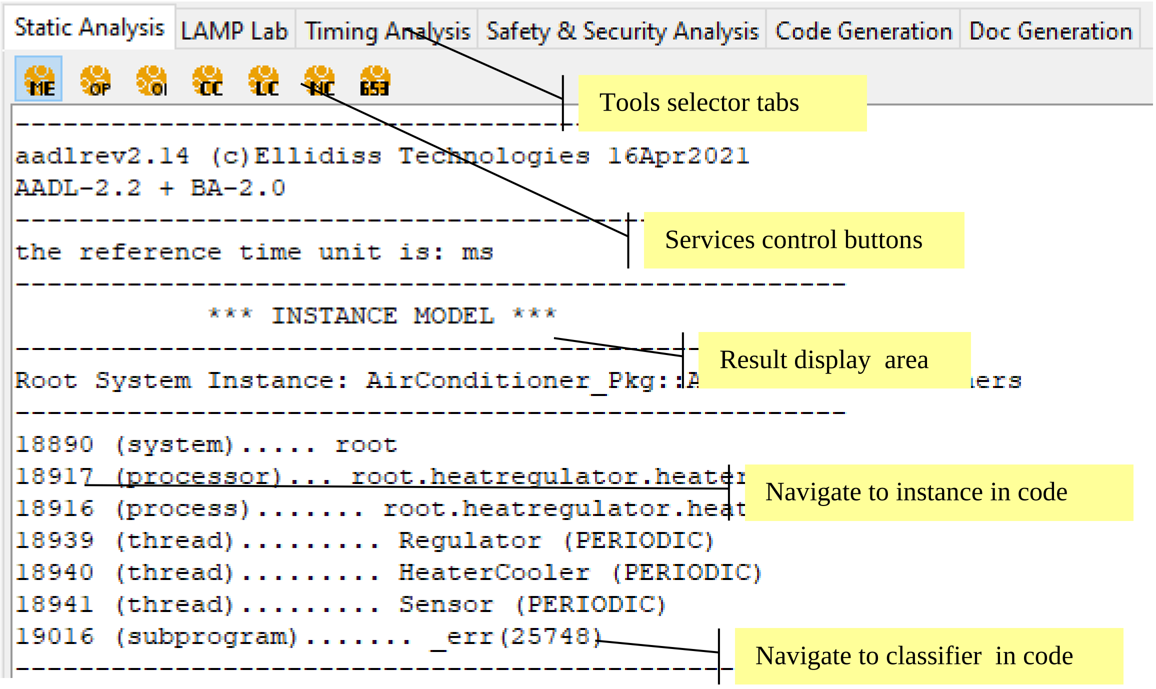
The `<ticks>` sections indicated what value that is inserted automatically into an input port variable at the instant denoted by the tick value. In case of an input event port, no value is needed. It is also possible to specify a sequence of ticks thanks to the `next` attribute which may contain an arithmetic formula to define the value of the next tick. For instance, a periodic activation of an event port will be obtained by the following statement:

```
<tick value="0" next=”tick+10”>
  <action ref="input"/>
</tick>
```

3.4 Processing tools area

The *Processing Tools Area* allows for selecting the processing tool to be applied to the set of **AADL** files that are selected in the *Project Browser* and display the corresponding execution result. This area is composed of:

- a set of *tool selector tabs*
- one or several *service control buttons*
- a read-only *result display area*



- *Tools selector tabs* can be configured by adding or removing tool description files (`.ais` files) in the `config` subdirectories of the installation directory

If one of the analysed files is modified, the background colour of the result display area becomes gray to indicate that the information is potentially out of date.

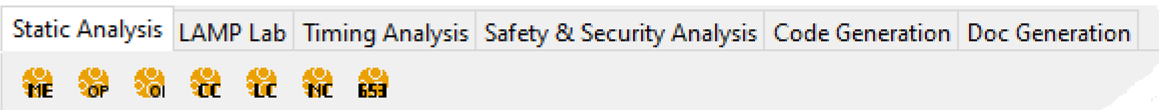
When the selected analysis tool can not be executed normally for the current **AADL** specification or if the **AADL** syntax is not correct, the corresponding error message will appear in an additional temporary *Report* tab.

When line numbers are shown in the generated report, clicking on them will highlight the corresponding lines in the *Source Files Area*.

Note that while working on large **AADL** projects, processing actions may take a significant time (up to a few minutes). Depending on the processing tool that is running, other user actions may not be allowed, and the display may not be refreshed during that time.

3.4.1. Static Analysis

The *Static Analysis* tool encompasses a set of independent rules checkers that verify various facets of the semantic correctness of the source **AADL** specification. Each rules checker is implemented as a service of the static analysis tool and can be activated by pressing the correspondent button:



- ME call the **AADL** parse and instantiate **LMP** service.
- OP call the **AADL** parse and verify **Ocarina** service.
- OI call the **AADL** instantiate **Ocarina** service
- CC call the **AADL** Consistency rules **LMP** checker.

- call the **AADL** Legality rules **LMP** checker.
- call the **AADL** Naming rules **LMP** checker.
- call the **ARINC 653** rules **LMP** checker.

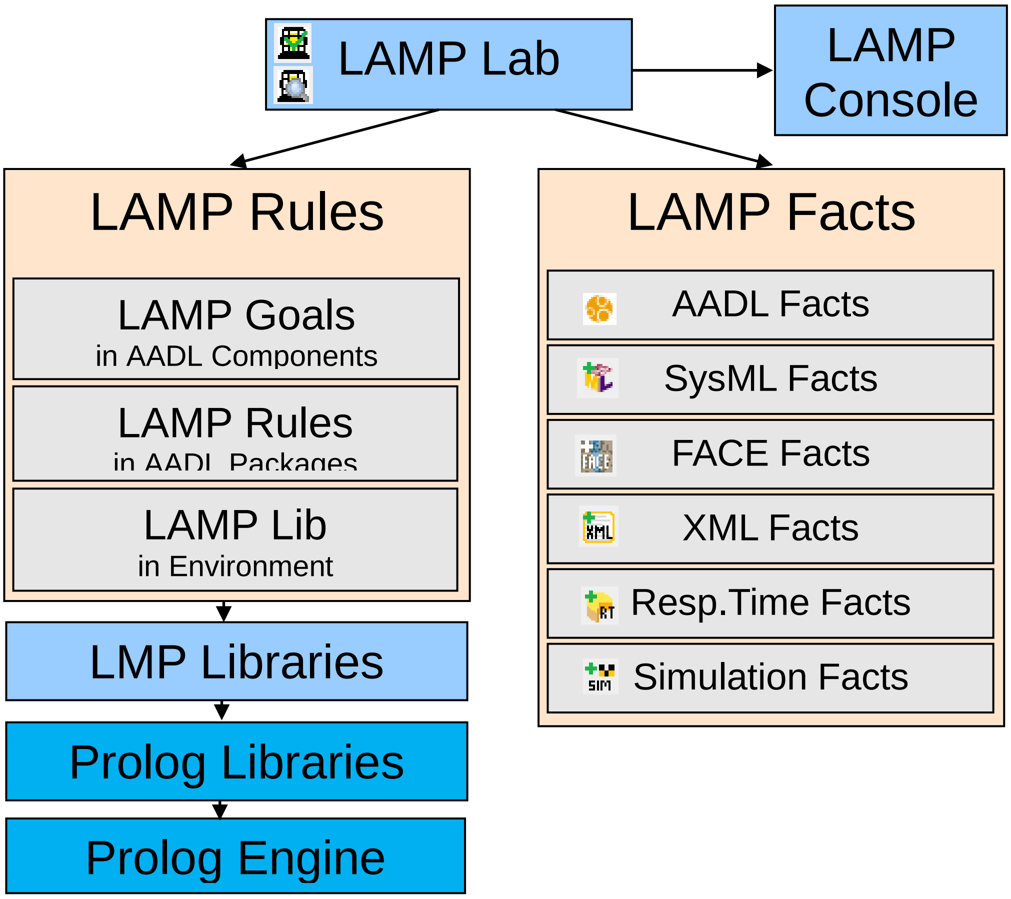
When an error, warning or information message is displayed by a processing tool, the line number of the corresponding **AADL** code is shown in the *Processing Tools Area*. Clicking on a line number updates the display of the *Source Files Area* to make the relevant line visible.

More detailed explanations about the scope of each of these checkers can be found in separate documentation.

3.4.2. LAMP Lab

The *LAMP Lab* tool can be used to experiment the use of **LAMP** and create customized assurance cases that may be modified interactively and can take heterogenous inputs including **SysML**, **FACE** and any other **XML** based models. **LAMP** stands for Logic **AADL** Model Processing. It is an online processing language that can be directly included within **AADL** Packages and Components as Annex sub-clauses. This language is the same as the one that is used for the definition of the predefined plug-ins and wizards (**LMP**). **LMP** consists of a set of parsers, a **Prolog** engine and libraries to access and process model elements.

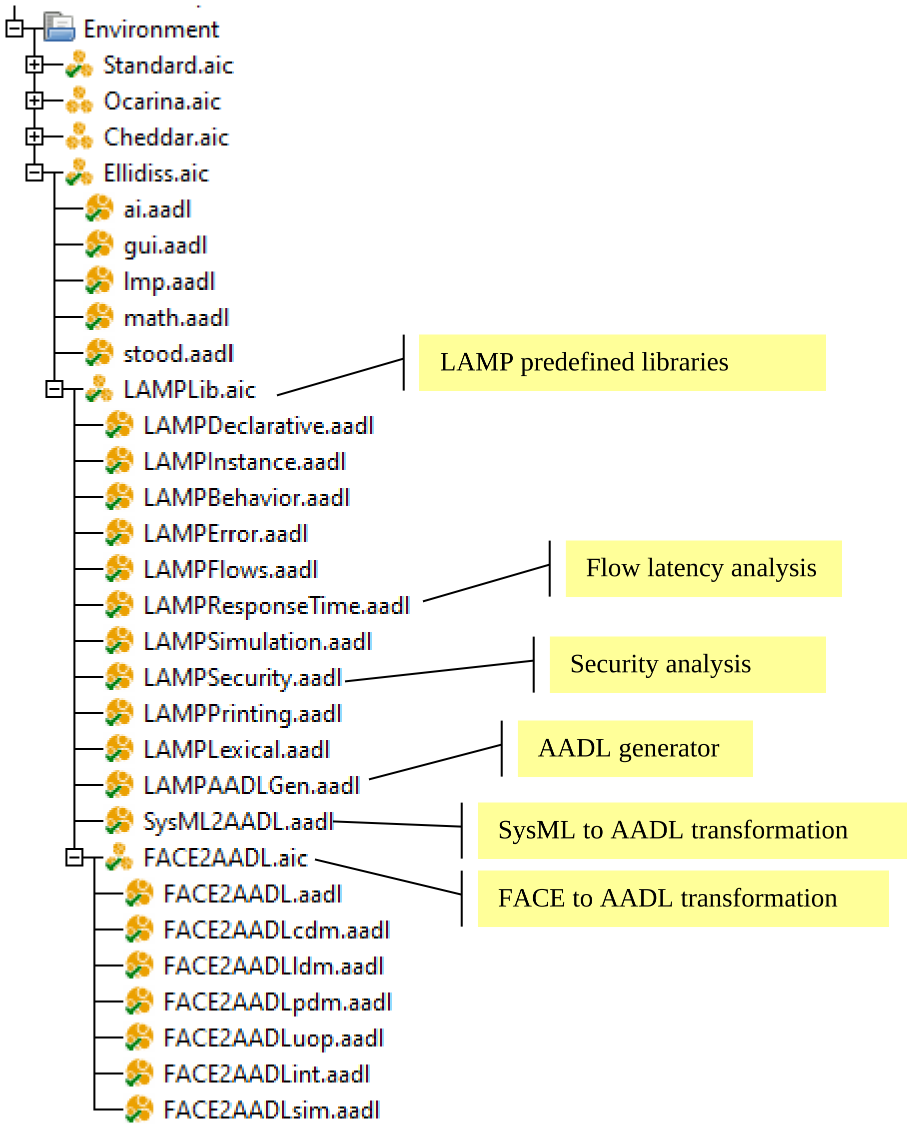
An overview of the *LAMP Lab* tool is provided by the picture below:



The **LAMP** services are organized as follows:

- : load the contents of all the **LAMP** annexes that are found in the selected **AADL** user files and environment libraries and run the corresponding queries (goals).
- : do the same as above but ignore the goal definitions found in the **LAMP** annexes and ask for a query in a dialog box instead.
- : parse specified **XML** file and load corresponding **Prolog** facts for next executions of the **LAMP** queries.
- : parse specified **XMI** file, interpret it according to the **UML** and **SysML** metamodels and load corresponding **Prolog** facts for next executions of the **LAMP** queries.
- parse specified **XML** file, interpret it according to the **FACE** metamodel and load corresponding **Prolog** facts for next executions of the **LAMP** queries.
- run **Marzhin** simulator and load corresponding **Prolog** facts for next executions of the **LAMP** queries.
- run the **AADL** Threads response time computation wizard and load corresponding **Prolog** facts for next executions of the **LAMP** queries.
- load selected **Prolog** code for next executions of the **LAMP** queries.
- remove all previously added **Prolog** extensions for next executions of the **LAMP** queries.
- display output produced by the last execution of **LAMP**.
- show the list of **Prolog** predicates that represent the current **AADL** declarative model. The definition of these predicates can be found on the Ellidiss wiki: https://www.ellidiss.fr/public/wiki/aadlDeclarativeModel
- show the list of **Prolog** predicates that represent the current **AADL** instance model.
- show the list of **Prolog** predicates generated from previously added raw **XML** or **XMI** file.
- show the list of **Prolog** predicates generated from previously added **SysML** file.
- show the list of **Prolog** predicates generated from previously added **FACE** file.
- show the list of **Prolog** predicates that represent the logged **Marzhin** simulation events.
- show the list of **Prolog** predicates that represent the computed Thread response time by **Cheddar** and **Marzhin**.
- show the list of **Prolog** predicates that were previously added.

LAMP annex sub-clauses that are defined at an **AADL** Package level specify processing rules libraries. Predefined **LAMP** libraries are provided in the *Environment* section of the *Project Browser*. Predefined libraries provide a complete access to all the **AADL** modelling elements (declarative and instance model, Behavior annex and Error Model V2 annex), as well as various utility and processing rules (**AADL** generator, security and flow analysis, **SysML** to **AADL** and **FACE** to **AADL** transformations). User defined **LAMP** libraries can be added inside standard **AADL** files belonging to the project. Predefined libraries are always implicitly selected whereas user defined libraries must be explicitly selected to be usable.



LAMP annex sub-clauses that are inserted at an **AADL** Component level specify goals that control the execution of the **LAMP** processing engine. All the goals found within the selected set of **AADL** files will be executed in sequence, except if the **LAMP** query is explicitly defined in a predefined menu or a dialog box.

Both rules and goals use the same standard **Prolog** language syntax and semantics with a few **SB-Prolog** specific features and behaviors. However, other restrictions apply while being used inside a **LAMP** annex:

- If it exists, a **LAMP** annex within an **AADL** Component (goal) cannot be empty and must not end with a dot.
- The size of a **LAMP** annex subclause cannot exceed 4096 characters. However, it is possible to add several annexes within the same Component or Package.

An example of use of a user-defined **LAMP** program using pre-defined `LAMPLib` rules is shown below:

```
PACKAGE lamp_pkg
PUBLIC

SYSTEM lamp
END lamp;

SYSTEM IMPLEMENTATION lamp.i
SUBCOMPONENTS
  hw : PROCESSOR hw;
  sw : PROCESS sw;
PROPERTIES
  SCHEDULING_PROTOCOL => (Rate_Monotonic_Protocol) APPLIES TO hw;
  ACTUAL_PROCESSOR_BINDING => (REFERENCE(hw)) APPLIES TO sw;
ANNEX LAMP {**
  /* goal */
  printProperties
**};
END lamp.i;
```

LAMP goal definition in an AADL Component

```

ANNEX LAMP {**
  /* user defined rules */
  printProperties :-
    getClassProperties('',P,V,O), printProperty(P,V,O),
    fail.
  printProperties.
**};

END lamp_pkg;
```

LAMP rule definition in an AADL Package

LAMP rule references in predefined LAMP libraries

Static Analysis | LAMP Lab | Timing Analysis | Safety & Security Analysis | Code Generation | Doc Generation

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*\
|           LAMP console            |
|  (c) Ellidiss Technologies, 2021  |
\*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

[x] AADL facts base loaded.
[ ] no XML facts base loaded.
[ ] no SysML facts base loaded.
[ ] no FACE facts base loaded.
[ ] no Simulation facts base loaded.
[ ] no Response Time facts base loaded.
[ ] no Native Prolog facts base loaded.
[x] LAMP rules base loaded.
[x] LAMP queries loaded.

LAMP> execution started.

(Rate_Monotonic_Protocol) => SCHEDULING_PROTOCOL APPLIES TO hw
(REFERENCE(hw)) => ACTUAL_PROCESSOR_BINDING APPLIES TO sw

LAMP> execution completed.
```

Result of LAMP execution

The following sub-sections provide more details about some of the proposed processing rules in `LAMPLib`. Note that corresponding source code is read-only when accessed from within the **AADL Inspector** text editor. To customize these rules, apply one of the three possible solutions:

- Create a copy of the relevant `LAMPLib` files into a writable workspace and take care to rename all the declared rules not to interfere with `LAMPLib` ones. There is no need to restart **AADL Inspector** to execute the modified rules. This is the recommended solution.
- Move the relevant `LAMPLib` files to a writable workspace, restart **AADL Inspector**, do your changes, test them interactively and then replace the modified files in the `LAMPLib` area.
- Edit the relevant `LAMPLib` files with a separate text editor and restart **AADL Inspector** each time you need to execute the modified rules.

3.4.2.1. Flow latency analysis

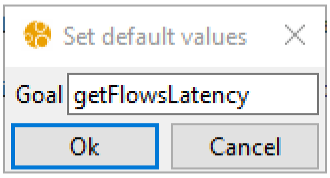
The `getFlowsLatency` query performs Scheduling Aware Flow Latency Analysis (**SAFLA**). This rule finds all the End-to-End flows in the current root system, compute their maximum latency using **Marzhin** simulation, and prints the result in the **LAMP** console. The source code is available in file:

`Environment/Ellidiss/LAMPLib/LAMPResponseTime.aadl`.

There are three ways to activate this analysis tool. The first one consists in adding a **LAMP** goal within the **AADL** specification to be processed and then to press the *Run LAMP* button of the *LAMP Lab* button bar. This solution is used in the examples `end_to_end_flow.aic` and `safety_security.aic`.

```
abstract lamp_goal
annex lamp  {** getFlowsLatency **};
end lamp_goal;
```

The second way to launch this service is to use the **LAMP** query button:



The last one fully hides the **LAMP** machinery and is available via a dedicated button in the *Timing Analysis* tab:

3.4.2.2. Security analysis

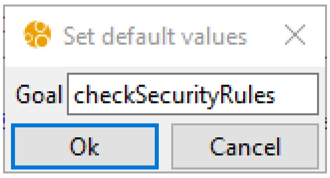
The `checkSecurityRules` query performs security analysis. As the **AADL** Security Annex has not been published yet at the time this feature was developed, it uses a simplistic user defined security model with a single property defining the security level associated with Data classifiers and a few examples of possible corresponding verifications. The source code is available in file:

`Environment/Ellidiss/LAMPLib/LAMPSecurity.aadl.`

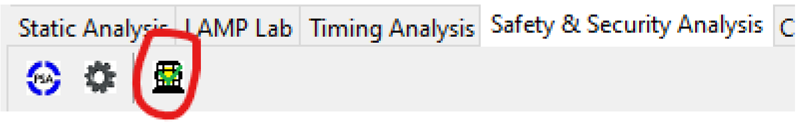
There are three ways to activate this analysis tool. The first one consists in adding a **LAMP** goal within the **AADL** specification to be processed and then to press the *Run LAMP* button of the *LAMP Lab* button bar. This solution is used in the example `safety_security.aic.`

```
abstract lamp_goal
annex lamp  {** checkSecurityRules **};
end lamp_goal;
```

The second way to launch this service is to use the **LAMP** query button:



The last one fully hides the **LAMP** machinery and is available via a dedicated button in the *Safety & Security Analysis* tab:



3.4.2.3. SysML to AADL

The `sysml2aadl` query performs a model transformation between an input **SysML Prolog** facts base and an output **AADL Prolog** facts base. The input facts must be imported at first. The output facts must be post-processed with the `runAADLgen` **LAMP** query to generate a proper **AADL** file. The source code of the mapping rules between the two languages is available in file:

`Environment/Ellidiss/LAMPLib/SysML2AADL.aadl`

There are two ways to activate this transformation tool. The first one consists in adding a **LAMP** goal within an **AADL** specification, manually load the **SysML** model thanks to the *Add SysML facts* button of the *LAMP Lab* button bar and then to press the *Run LAMP* button of the same *LAMP Lab* button bar.

```
abstract lamp_goal
annex lamp  {** sysml2aadl, runAADLGen **};
end lamp_goal;
```

The second way fully hides the **LAMP** machinery and is available via a dedicated button in the *File/Import/Import SysML model (.sysml, .xmi, .model)* menu, or corresponding button of the main button bar:

3.4.2.4. *FACE to AADL*

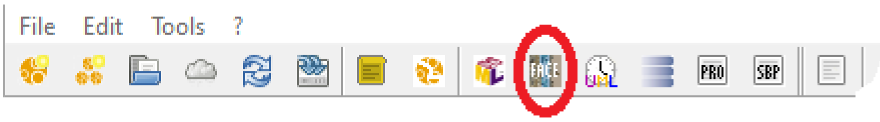
The `face2aadl` query performs a model transformation between an input **FACE Prolog** facts base and an output **AADL Prolog** facts base. The input facts must be imported at first. The output facts must be post-processed with the `runAADLgen` **LAMP** query to generate a proper **AADL** file. The source code of the mapping rules between the two languages is available in directory:

`Environment/Ellidiss/LAMPLib/FACE2AADL/`

There are two ways to activate this transformation tool. The first one consists in adding a **LAMP** goal within an **AADL** specification, manually load the **FACE** model thanks to the *Add FACE facts* button of the *LAMP Lab* button bar and then to press the *Run LAMP* button of the same *LAMP Lab* button bar.

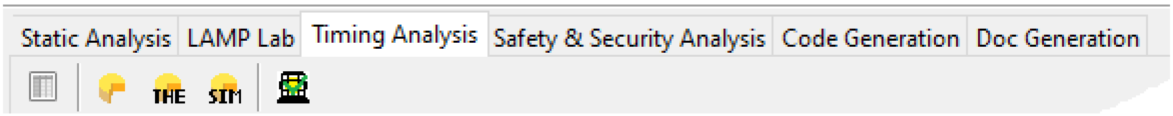
```
abstract lamp_goal
annex lamp {** face2aadl, runAADLGen **};
end lamp_goal;
```

The second way fully hides the **LAMP** machinery and is available via a dedicated button in the *File/Import/Import FACE model (.face)* menu, or corresponding button of the main button bar:



3.4.3. Timing Analysis

When the *Timing Analysis* tab is selected, five buttons are presented to activate timing analysis services.

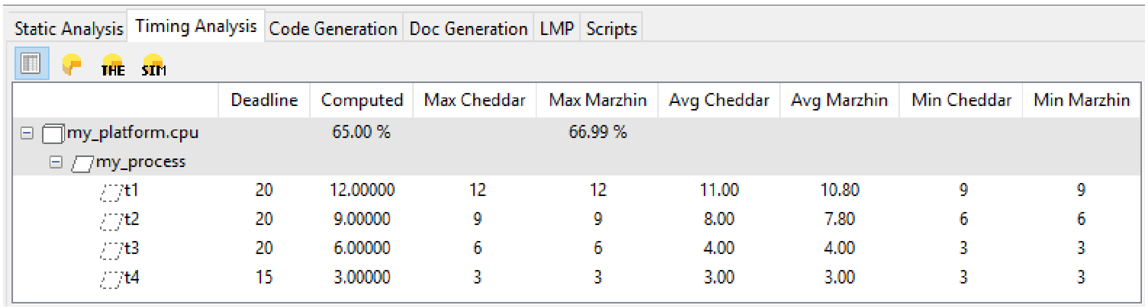


- compute statistics for processor load and thread response time from the various outputs given by **Cheddar** and **Marzhin**, and show them in a spreadsheet for comparison.
- static simulation computed by **Cheddar**.
- THE set of feasibility tests checked by **Cheddar**.
- SIM set of tests based on the static simulation computed by **Cheddar**.
- Scheduling Aware Flows Latency Analysis (**SAFLA**): associate response time computation done by **Cheddar** and **Marzhin** with **AADL** Flows analysis done by **LAMP** to provide an estimate of End-to-End Flows latency.

These features are detailed in the next sub-sections:

3.4.3.1. *Processor load and Thread response time*

This service shows a summary of the *Timing Analysis* in a single table. For each Processor, the maximum load rates that are computed by **Cheddar**, and estimated by the **Marzhin** simulator are provided. For each Thread, the minimum, average and maximum response time computed by **Cheddar** and estimated by the **Marzhin** simulator are also provided and can be compared with the deadline.

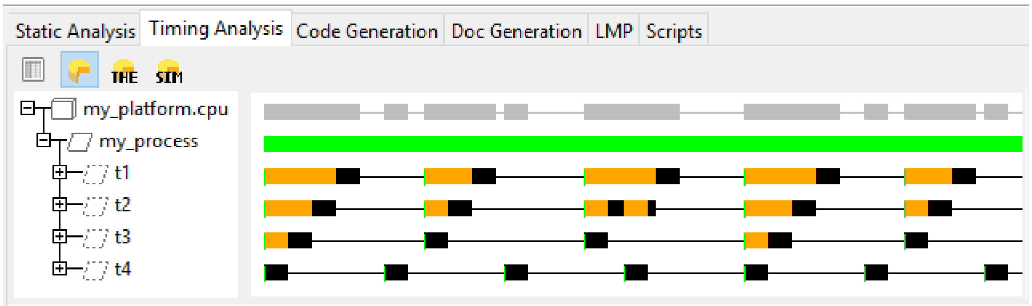


	Deadline	Computed	Max Cheddar	Max Marzhin	Avg Cheddar	Avg Marzhin	Min Cheddar	Min Marzhin
my_platform.cpu		65.00 %		66.99 %				
my_process								
t1	20	12.00000	12	12	11.00	10.80	9	9
t2	20	9.00000	9	9	8.00	7.80	6	6
t3	20	6.00000	6	6	4.00	4.00	3	3
t4	15	3.00000	3	3	3.00	3.00	3	3

Note that this table may contain empty cell if the corresponding tool or service has not been launched or cannot provide relevant data. The table is dynamically updated when the Marzhin simulator is running.

3.4.3.2. *Cheddar simulation timelines*

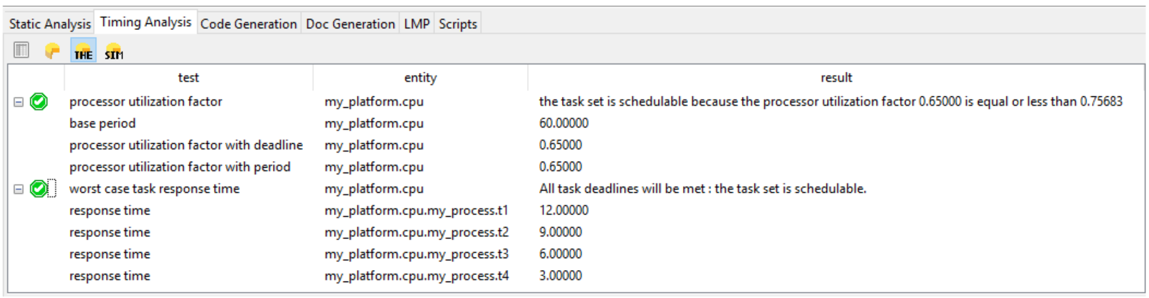
Cheddar can produce a graphical representation of the timing behaviour of the real-time system being analysed. This graphical schedule table is a result of the static simulation and may not be available on every kind of system.



Timelines are displayed for each Processor, Process, Thread, shared Data and Bus subcomponent in the current root System. The time scale and meaning of each used colour is shared with the dynamic simulator which is described below.

3.4.3.3. *Scheduling Theoretical Tests*

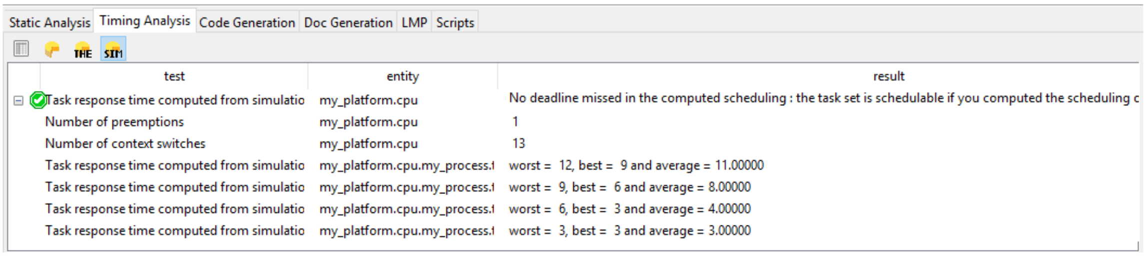
Theoretical tests compute the processor utilization factor and threads response time when the corresponding conditions are met. This service is provided by **Cheddar**.



test	entity	result
processor utilization factor	my_platform.cpu	the task set is schedulable because the processor utilization factor 0.65000 is equal or less than 0.75683
base period	my_platform.cpu	60.00000
processor utilization factor with deadline	my_platform.cpu	0.65000
processor utilization factor with period	my_platform.cpu	0.65000
worst case task response time	my_platform.cpu	All task deadlines will be met : the task set is schedulable.
response time	my_platform.cpu.my_process.t1	12.00000
response time	my_platform.cpu.my_process.t2	9.00000
response time	my_platform.cpu.my_process.t3	6.00000
response time	my_platform.cpu.my_process.t4	3.00000

3.4.3.4. *Scheduling Simulation Tests*

Simulation tests provide information about the number of pre-emption and context switches as well as threads response time. This static simulation can only be run for periodic systems. This service is provided by **Cheddar**.



test	entity	result
Task response time computed from simulatio	my_platform.cpu	No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling c
Number of preemptions	my_platform.cpu	1
Number of context switches	my_platform.cpu	13
Task response time computed from simulatio	my_platform.cpu.my_process.t	worst = 12, best = 9 and average = 11.00000
Task response time computed from simulatio	my_platform.cpu.my_process.t	worst = 9, best = 6 and average = 8.00000
Task response time computed from simulatio	my_platform.cpu.my_process.t	worst = 6, best = 3 and average = 4.00000
Task response time computed from simulatio	my_platform.cpu.my_process.t	worst = 3, best = 3 and average = 3.00000

More detailed explanations about the scope of each of these tests can be found in a separate user document.

3.4.3.5. *Scheduling Aware Flows Latency Analysis (SAFLA) with LAMP*

Ask for the duration of the **Marzhin** simulation and run it, then apply the `getFlowsLatency` **LAMP** query. The source code of this **Prolog** rule is available in file:
`Environment/Ellidiss/LAMPLib/LAMPResponseTime.aadl`.

3.4.4. Safety & Security Analysis

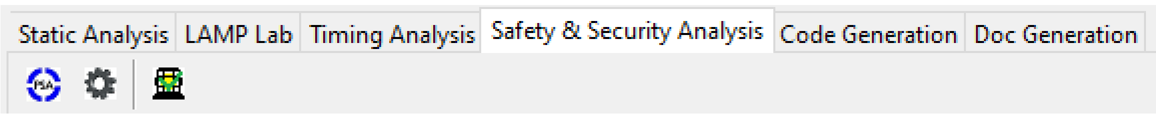
The *Safety & Security Analysis* tool aims at interfacing external programs that support model driven safety analysis as well as checking security rules. The safety related model transformation makes use of the **AADL** Error Model Annex (**EMV2**). The security related model processing is based on `LAMPLib`.

The safety analysis tool that is currently supported is **Arbre Analyst**. This tool is not included within the **AADL Inspector** distribution. It can be found at the following address: https://www.arbre-analyste.fr/en.html

Note that once installed onto your computer and checked the terms of the license, you need to update the corresponding file pathname in the `AIConfig.ini` file before being able to use this service, for instance:

```
variable userConstants { \
        "FTAToolPath" "{C:/Projets/AADLInspector/Safety/arbre_analyste-
2.3.1-win32/Arbre Analyst.exe}" \
```

Arbre Analyste can load models that are expressed with the **Open PSA** format. The *Safety & Security Analysis* tool thus provides the following services:



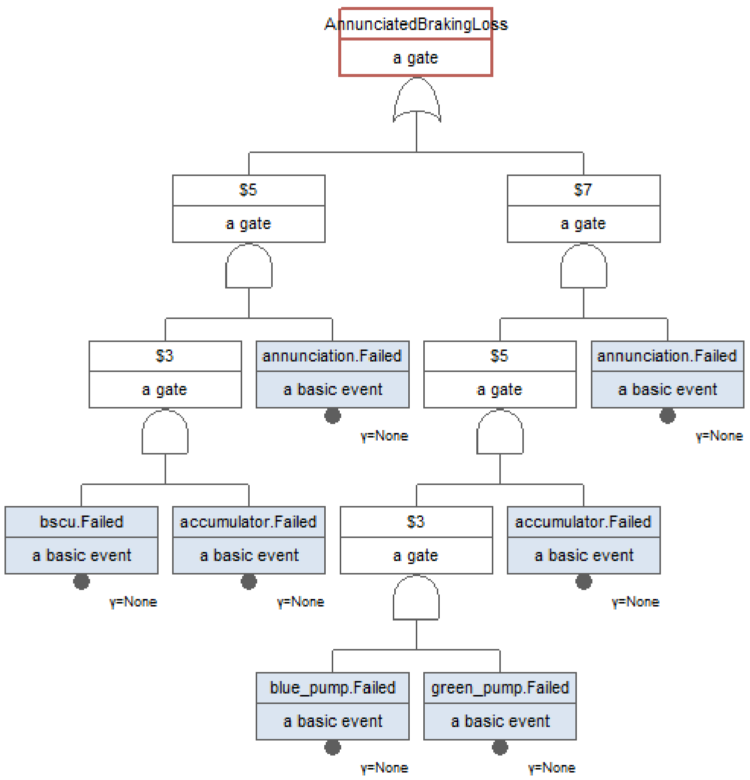
–  generate a file complying with the **Open PSA** model exchange format.

- generate an **Open PSA** file as above and launch the **Arbre Analyste** tool to display a graphical fault tree.
- apply the `checkSecurityRules` **LAMP** query. The source code of this **Prolog** rule is available in file:

  `Environment/Ellidiss/LAMPLib/LAMPSecurity.aadl`.

An example of use of the *Safety & Security Analysis* tool can be found in the `safety_security.aic` example. Use of **Arbre Analyste** is presented below. It first shows a fragment of the **AADL** model and then the graphical representation of the corresponding Fault Tree that is generated by **Arbre Analyste**.

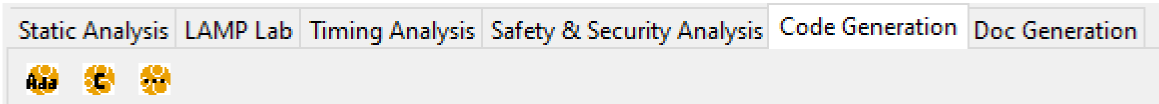
```
annex EMV2 {**
  use types error_library;
  use behavior error_library::wbs;
  composite error behavior
    states
    [ bscu.Failed
      and accumulator.Failed
      and annunciation.Failed ]-> AnnunciatedBrakingLoss;
    [ blue_pump.Failed
      and green_pump.Failed
      and accumulator.Failed
      and annunciation.Failed ]-> AnnunciatedBrakingLoss;
    [ bscu.Failed
      and accumulator.Failed
      and annunciation.Failed ]-> UnannunciatedBrakingLoss;
    [ blue_pump.Failed
      and green_pump.Failed
      and accumulator.Failed
      and annunciation.Failed ]-> UnannunciatedBrakingLoss;
  end composite;
**};
```



Similar connection to other safety analysis tools can be added to **AADL Inspector** if required. Please contact the technical support if you wish to add another connector.

3.4.5. Code Generation

The code generation services are provided by **Ocarina** back-ends. Please refer to the **Ocarina** documentation or the **OpenAADL** web site www.openaadl.org for detailed explanations about the use of these features.

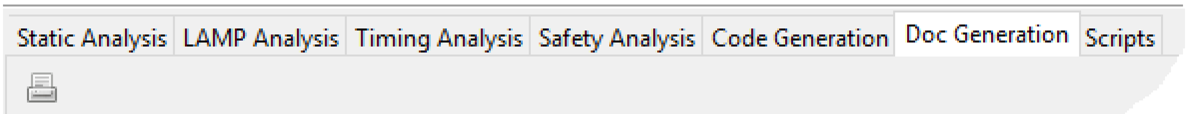


- : generate **Ada** source code files for the **PolyORB-HI-Ada** middleware. A dialog box asks about the location of the generated code. A default location is proposed in the **AADL Inspector** temporary directory.
- : generate **C** source code files for the **PolyORB-HI-C** middleware. A dialog box asks about the location of the generated code. A default location is proposed in the **AADL Inspector** temporary directory.
- : gives access to the other available **Ocarina** back-ends. The actual back-end to use can be selected in a dialog box.

Note that **Ocarina** generates the source code architecture and glue code with the Operating System. However, it requires the applicative functional code to be made available for a complete build of the software. Access to the functional code can be specified by `Source_Text` **AADL** Properties.

3.4.6. Doc Generation

A standard analysis report can be automatically generated thanks to the documentation generator.



- : The documentation generator can also be activated from the *File/Print* menu and applies to the current **AADL** system instance. It produces a pre-formatted report that contains the following sections:

- The output of the *Metrics* static analysis tool that recalls the **AADL** scope of the report.
- The description of the scenarios that are selected.
- A snapshot of the simulation time lines from tick 0 to tick 100.
- The timing analysis summary table.

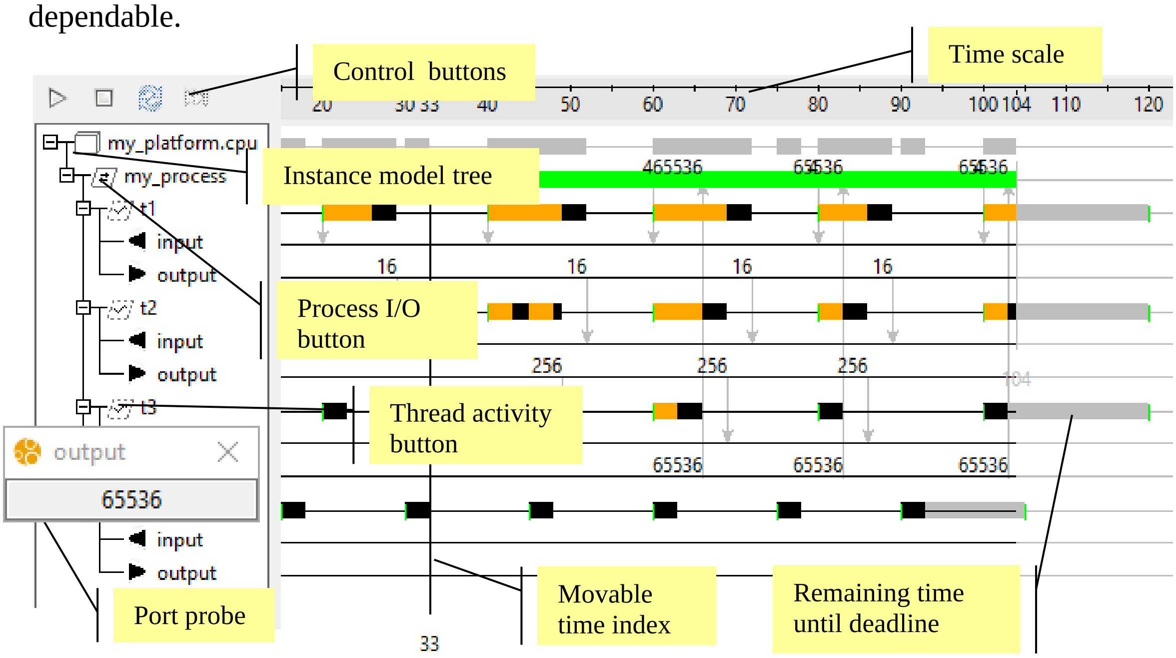
Note that the graphical sections that are inserted into the documentation depend on the actual layout of the tool window on the screen. Take care to properly resize the window before starting the documentation generator, so that the corresponding elements are sufficiently visible.

To open the generated document, use the *?/Open doc dir* menu and select the most recent `.pdf` file that has been generated.

To customize the contents of the generated report, for instance to modify the size of the printed time lines, it is necessary to edit the plugin configuration file: use the *?/Open config dir* and edit the file `plugins/DocGeneration.ais`.

3.5 Simulation area

The *Simulation Area* is dedicated to controlling and displaying the output of the **Marzhin** dynamic simulator. This simulator complements the static simulator provided by **Cheddar** but is event-driven and can analyse a wider variety of real-time systems. The counter part is that the obtained timelines are not the result of mathematical computations and are thus less dependable.



The simulation area is composed of:

- a set of control buttons (same as in the Simulation Control Panel).
- a time scale (shared with the **Cheddar** Schedule Table).
- a deployable tree showing the **AADL** instance hierarchy.
- an external I/O button on each Process that has connected ports.
- an activity button on each Thread to open a tachymeter.
- probes on input and output ports.
- a simulator output area showing timelines for each Processor, Process, Thread, shared Data and Bus subcomponent in the current root System.

In addition, the *Simulation Control Panel* dialog can be used to set up the time scale and filter the entities displayed for the simulation. This feature can be activated from the *Edit* menu or the corresponding button in the *Main buttons Bar*. Refer to section 3.1.2.4 for more details.

3.5.1. Simulator action buttons

The simulator toolbar is composed of the following buttons:

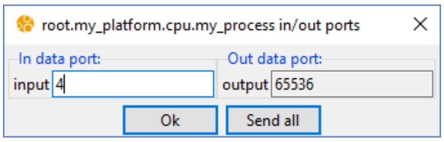
	start the simulator
	pause the simulator
	stop the simulator
	refresh the simulation input
	go to the current tick
	toggle optimized mode (see below)

Since version 1.7, **AADL Inspector** includes an optimized mode for **Marzhin**. When this mode is set (default case), the simulator automatically jumps to the next significant event. Note that this mode is automatically unset when a scenario has been selected.

3.5.2. External I/O

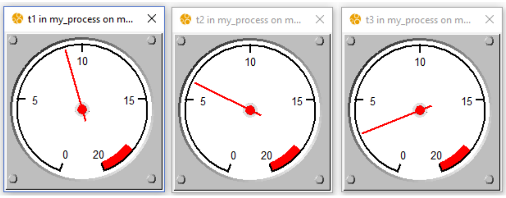
When a Process has ports that are connected downstream in the instance hierarchy, they can be displayed in a specific dialog box to allow the user to send in data and events and to show the result of out data and events. This dialog box can be opened by pressing the I/O button .Note that the value that is displayed for an out event port is the time of its last update.

```
PROCESS my_process
FEATURES
  input : IN DATA PORT int;
  output : OUT DATA PORT int;
END my_process;
```



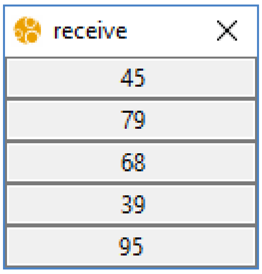
3.5.3. Thread activity

A graphical tachymeter can be associated with each running Thread thanks to the activity button in the instance tree. Each indicator shows the instant response time of the Thread and is updated at each period.



3.5.4. Port probe

A probe can be attached to in and out ports to show the current value that is stored in the port variable. For event and event data ports, a table shows the contents of the port **FIFO**, according to the specified Queue_Size property (default value is 1).



A probe can be opened by clicking on a port while the simulator is running, or preset in a scenario file (for input ports only)

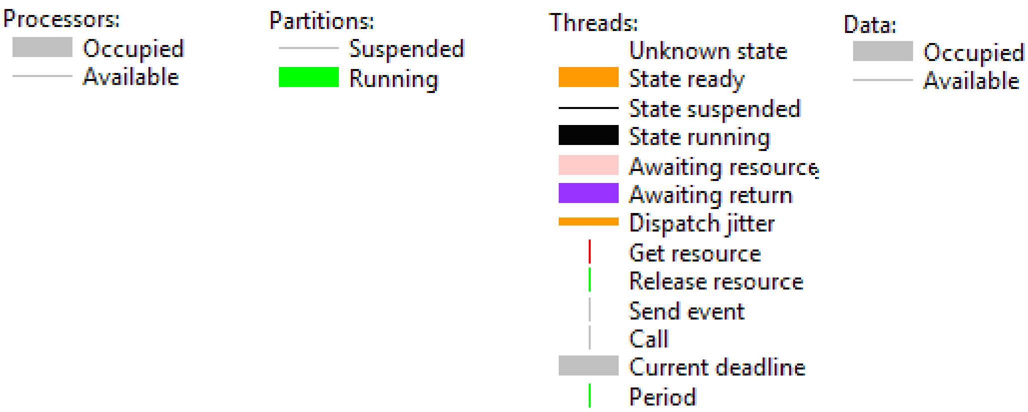
```
<scenarii>
  <interface>
    <feature type="eventdata" id="FIFO"
      aadlID="my_platform.cpu.my_process.receiver.receive"/>
  </interface>
  <scenario name="default" description="">
    <probes>
      <probe ref="FIFO"/>
    </probes>
  </scenario>
</scenarii>
```

3.5.5. Simulation timelines

A separate timeline is shown for each Processor, each partition (Process), each Thread, each shared Data component, as well as for each Bus, each Bus channel and each Bus message. The colour code that is used for the timelines can be configured in the `AIConfig.ini` file and displayed in the *help* tab of the *Simulation control panel*. Timelines can be saved in **VCD** format (cf. 3.1.2.5).

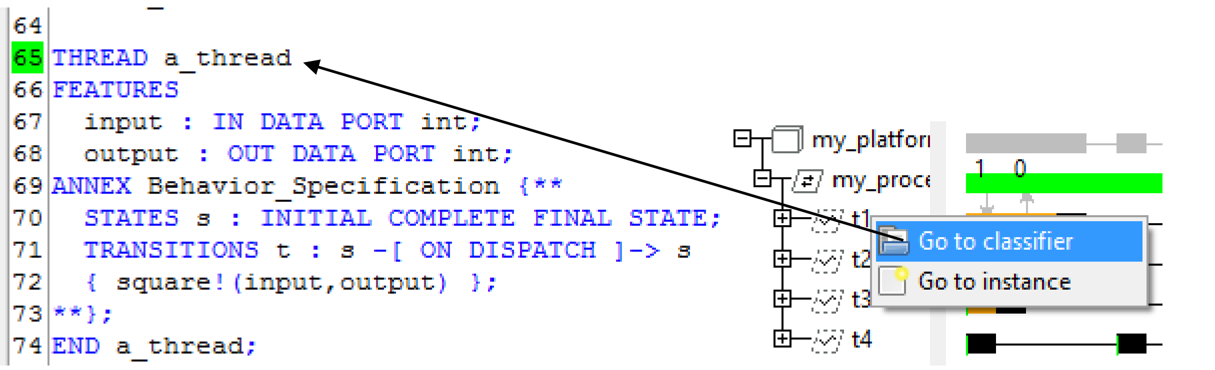
Note that the same representation is used for respectively Processors and Buses, Processes and Bus channels and Threads and Bus messages.

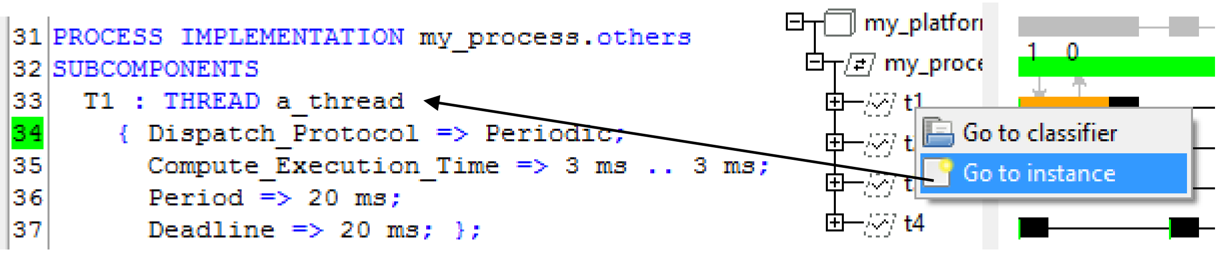
Default time lines colour mapping is as follows:



3.5.6. Navigation to the AADL source code

There is a contextual menu (right mouse button) associated with the entities of the instance model tree. It allows direct access to the corresponding classifier and instance declarations in the **AADL** source text.

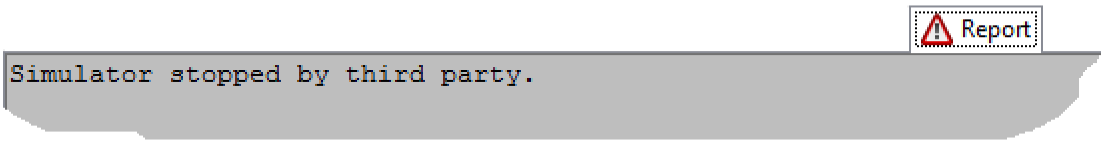




3.6 Status bar and Error Report

The status bar located in the lower part of the window shows various informational or error messages generated by **AADL Inspector**:

When relevant, detailed error messages are displayed in the *Report* tab.



4 Used Key Words and Acronyms

AADL	Architecture Analysis and Design Language: SAE AS-5506 (more)
AADL Inspector	An AADL centric model analysis framework (more)
AADLib	Repository of AADL resources (more)
AMP	Asymmetric Multi Processor
ARINC 653	Avionics application software standard interface (more)
Ada	A programming language (more)
Arbre Analyst	A Fault Tree Analysis tool (more)
BMP	Bound Multi Processor
C	A programming Language (more)
Capella	A Model Based System Engineering tool (more)
Cheddar	A timing analysis tool (more)
DM	Deadline Monotonic
EMOF	Essential Meta-Object Facility (more)
EMV2	Error Modeling AADL annex v2 (more)
ETFL	Ellidiss Technologies Floating License
Ecore	Eclipse Modeling Framework metamodel language (more)
Ellidiss Technologies	A company editing AADL and HOOD tools (more) (again more)
ESA	European Space Agency (more)
FACE™	Future Airborne Capability Environment (more)
FIFO	First In First Out
FTA	Fault Tree Analysis
HOOD	Hierarchical Object Oriented Design (more)
ISAE	Institut supérieur de l'aéronautique et de l'espace (more)
JRE	Java Runtime Environment
Java	A programming language (more)
LAMP	Logical AADL Model Processing (more)
LMP	Logic Model Processing (more)
Linux	An Operating System
MARTE	Modeling and Analysis of Real-Time Embedded systems (more)
Magic Draw	A SysML modeling tool (more)
Marzhin	An AADL runtime simulator
OMG	Object Management Group (more)
OSATE	Open Source AADL Tool Environment (more)
Ocarina	A stand-alone AADL model processor (more)
OpenAADL	AADL resourses web site (more)
OpenPSA	Open initiative for Probabilistic Safety Assessment (more)
PDF	Portable Document Format (more)
PolyORB-HI-Ada	High-integrity middleware for Ocarina Ada code generator (more)
PolyORB-HI-C	High-integrity middleware for Ocarina C code generator (more)
Prolog	A programming language (more)
RM	Rate Monotonic
RTOS	Real Time Operating System
RTS	Real Time System
SAE AS-5506	A SAE International standard: AADL (more)
SAFLA	Scheduling Aware Flow Latency Analysis
SB-Prolog	A prolog engine (more)
Stood	A HOOD and AADL software design tool (more)
SysML	Systems Modeling Language (more)

TSP	Time and Space Partitioning
Telecom ParisTech	An engineering school (more)
UML	Unified Modeling Language (more)
VCD	Value Change Dump format (more)
Virtualys	An Ellidiss Technologies partner company (more)
Windows	An Operating System (more)
XMI	XML Metadata Interchange (more)
XML	Extensible Markup Language (more)

Ellidiss Technologies
24 quai de la douane
29200 Brest
Brittany
France

http://www.ellidiss.com

aadl@ellidiss.com
+33 298 451 870