Modelling Hardware Avionics Architecture with AADL

Julien Delange – julien.delange@esa.int
Jérôme Hugues – jerome.hugues@isae.fr

1 RATIONALE

This document presents an approach for the modelling of hardware concerns of avionics architecture. This is a follow-up of a previous paper (published in the scope of an AADL standardization committee - **see reference 3**) that described our preliminary investigations on avionics architecture modelling. It introduced our overall design method, to go from mission criteria and mission functions definition to system implementation specification.

This new document goes beyond this initial study and details how we can apply our methodology on a real mission (GAIA, an earth observation mission, **see reference 4**). In particular, it explains how we can use models to validate system requirements and automatically produce materials for validation, verification or analysis purposes. In consequence, it focuses on:

1. **Model validation**: validation to be done by appropriate tools to ensure that requirements defined at a functional level are enforced in the implementation.
2. **Model usage**: how models can be used by appropriate tools to produce documentation such as traceability information, etc.

Finally, we also propose some guidelines about potential further work and investigation to be done in the context of avionics system modelling with the AADL.

Next sections constitute a short reminder about the overall design approach and then, introduce the mapping of the GAIA mission to AADL models.

2 DESIGN APPROACH & APPROACH - REMINDER

2.1 Overall presentation

The main goal is to help system designers to asses their architecture, evaluate and find potential errors or requirements trade-offs at the earliest during development, saving time/money (**see reference 5**), improving development process reliability and increasing confidence in produced applications.

Resulting design process (illustrated in Figure 1) focuses on the following phases:

1. **Specify mission criteria and functions** with a dedicated formalism. Mission criteria represent requirements to be met (mission duration, maximum weight, etc.) ; functions specify what is achieved by the system (star tracking, observation, etc.). At this step, implementation concerns are not considered.
2. **Refine the functional architecture** by specifying its implementation using predefined building blocks. These blocks specify their requirements (computation capacity, weight, power, etc.) so that produced models are amenable for analysis.
3. **Validate the implementation against the functional description and mission criteria.** Analysis tools check the models evaluate and compare their requirements. In particular, they check for requirements correctness and constraints enforcement (e.g.: a processor provides sufficient computing capacity for the execution of its associated programs).

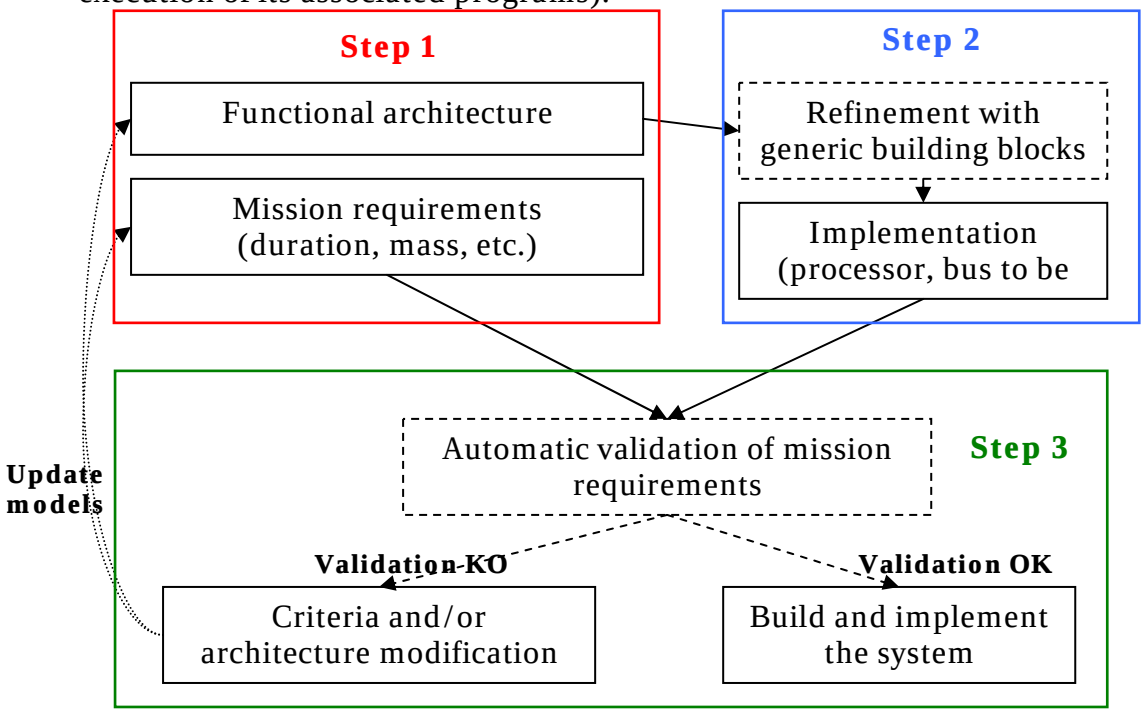


Figure 1 - Design approach

To model mission criteria, the functional architecture and planned implementation, the AADL language is used as the backbone language. It provides a level of abstraction suitable to software and hardware modelling and offers several extensions mechanisms, which ease its adaptation to different modelling approaches. In our context, we tailor the AADL modelling language to our needs by extending it with analysis methods or new properties.

This design approach is iterative, in the sense that designers can describe system architecture in a functional sense, without having to specify all characteristics of the system. Then, during the refinement process, components are replaced by generic building blocks that contain all properties and requirements so that validation tools can analyze and assess system feasibility. The following picture illustrates the relation between the details level and the evolution of the design process.

2.2 Benefits of this approach

By using this methodology, designers would expect:

- Faster design process by reusing predefined building blocks with predefined requirements definition
- Reliable and robust system analysis that relies on automatic analysis tools
- Saving time in system validation by using automatic analysis tools

As this process automates many aspects of system design and so, avoids, all human-related error, designers would expect to gain time and efforts when designing a system. Benefits of such an approach were already discussed in the context of the SAVI project (see **reference 5**).

3 APPLY DESIGN APPROACH TO THE GAIA MISSION

We apply our design approach to the GAIA mission. A recent report (**reference 4**) details how to improve the development process of space mission, applying their methodology to the GAIA mission. This report focuses on components reuse, certification/qualification materials production and system validation (enforcement of expected requirements).

This study also depicts the lack of existing tool for the automation of the development process so that designers/engineers still have to design the system by themselves and check requirements manually. Our current investigations demonstrate how system modelling technologies and more specifically AADL can be tailored in that purpose.

In consequence, we reuse this existing study, map system definition and requirements in AADL and discuss how we can automate each development step using the AADL.

Next sections are organized as follow:

1. **Description of the GAIA mission** from the preliminary study
2. **Reminder of our AADL modelling patterns**
3. **Application of our AADL-specific methodology on the GAIA mission**

3.1 GAIA Mission Overview and mission criteria

The GAIA mission consists in taking pictures of stars (1 billion stars). The original report lists the following mission requirements/criteria:

- **Capture and process stars pictures**
- **Store and compress data to be stored in the mass memory**
- **Transmit pictures to the ground with a predictable guaranteed % of collected data**
- **Keep thermal balance of payload cavity**
- **Implement smooth degradation over 5 years mission**

3.1.1 Functional decomposition

To meet mission criteria, the following functions are defined:

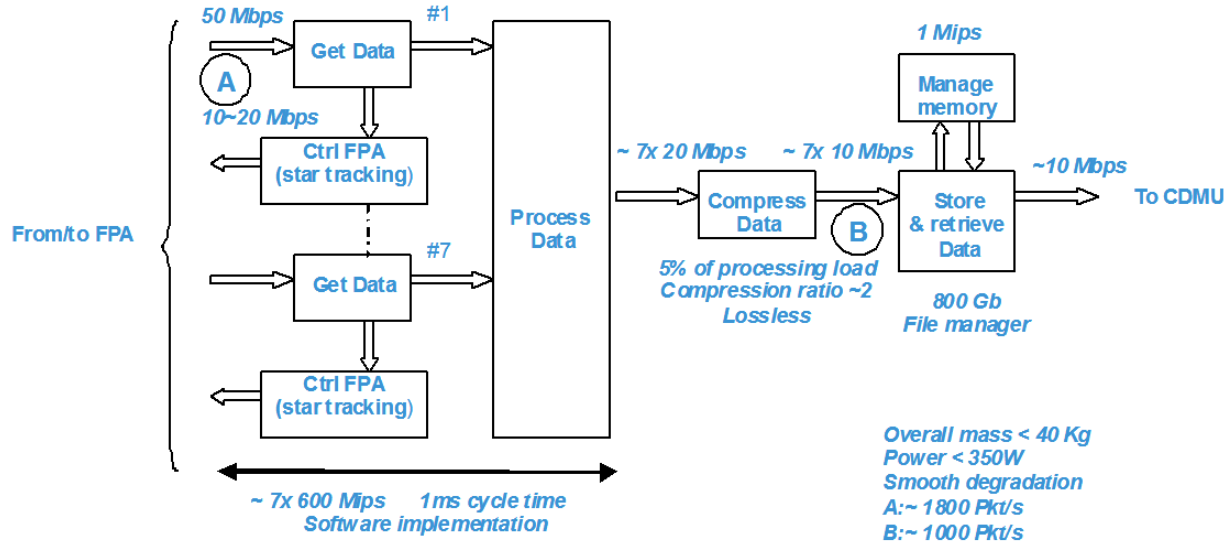
1. **Get data from FPA** (Focal Plane Array, an image-sensing device) (7 raws, 7x50 Mbps)
2. **Control FPA** for star tracking
3. **Process raw data from FPA** (images that are taken) for building packets (data reduction algorithm: 7 x 600 Mbps)
4. **Compress data lossless** (allocation: 5% of overall processing)
5. **Store data and manage memory** (optimal allocation and downlink scheduling)

In addition, the following requirements have to be met by the mission:

1. **Cycle time**: 1ms
2. **Constant thermal load**
3. **Smooth degradation**

The following picture illustrates the decomposition of the system into functions:

1. First, two functions are dedicated to FPA devices:
   a. One function gets data from each device and sends it to a main system that gathers and processes them.
   b. One function controls it (change orientation, etc.)
2. One function receives the data from all FPA and sends them to be compressed.
3. One function performs data compression (in order to reduce the memory required to store the pictures) and send the result to be stored in the mass memory.
4. Two functions care about memory concerns :
   a. One manage the hardware memory itself
   b. One issues commands to store/retrieve data from/to the memory.



3.1.2 First implementation: data-centralized on an OBC platform

In this first implementation, system is divided in three types of blocks (see figure 2):

1. The FPA management (the ones that gather data from the FPA and controls it) for each FPA are implemented in one physical block. This is a specific device that receives data from the camera and has the ability to send specific order to it. There are 7 devices like this one in this architecture.
2. The processing functions (that receive data and compress them) are implemented in one block, an on-board computer that receives data from the FPA handling devices and compress them.
3. The memory-related functions are also allocated to a common hardware component that manages the memory and store/retrieve data.

Figure 2 show the physical allocation from a logical point of view with the definition of the three types of allocation: one related to FPA functions, another to data processing and compression and another that cares about data storage. On the other hand, Figure 3 depicts the deployment of this implementation and the different nodes involved in this design.

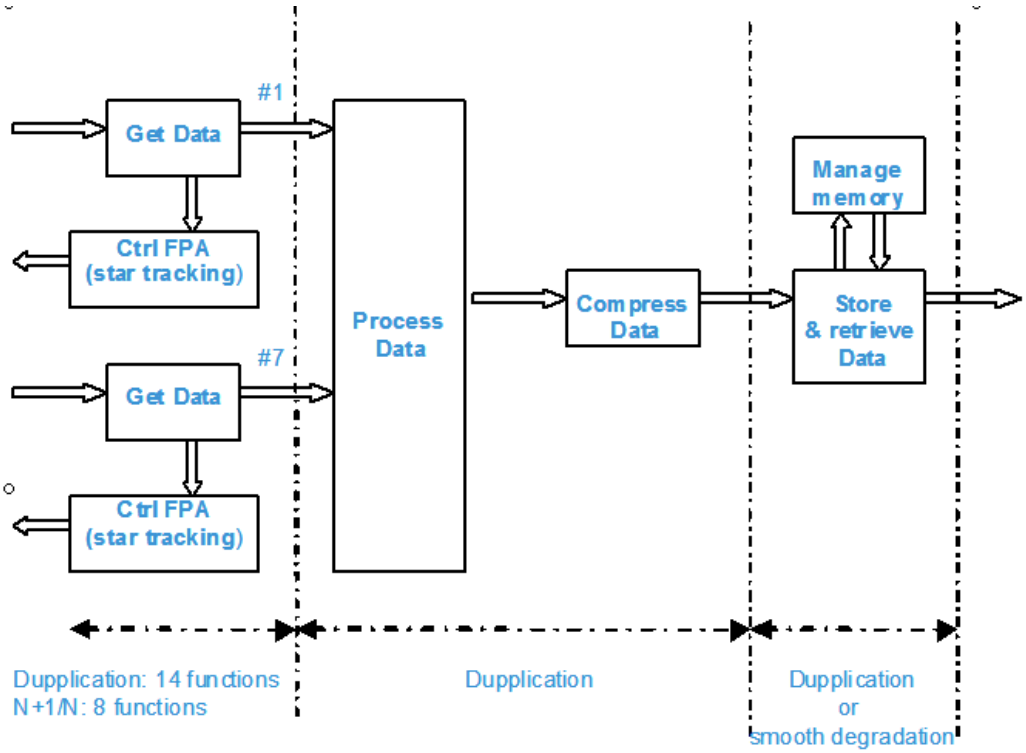


Figure 2 - Physical allocation of the first implementation - type of components

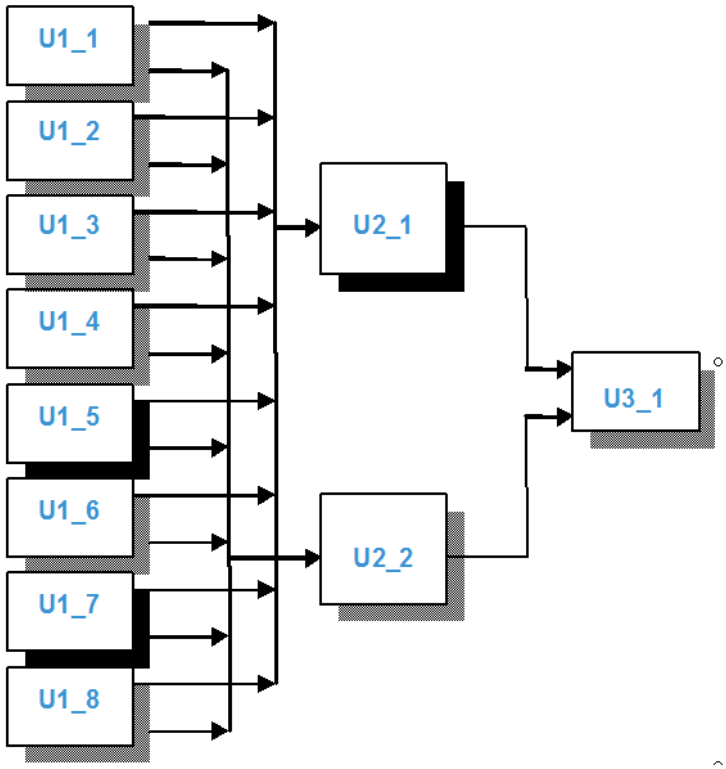


Figure 3 - Physical allocation of the first implementation - deployment

3.1.3 *Second implementation: processing separation on FPA devices*

The second implementation is an optimization of the first one, removing the on-board computer that processes and compresses data from the FPA devices. Instead, each device that handles FPA data process and compress the data it received by itself. By doing so, it removes a bottleneck of the system (the on-board computer that had to process all the data from all FPA) and distribute computing charge over all FPA-dedicated devices. Finally, a single component handle all memory-related functions (store and retrieve pictures taken by the FPA). Figure 4 illustrates the types of components used in this implementation while Figure 5 shows the deployment of these components with their interfaces and connections.

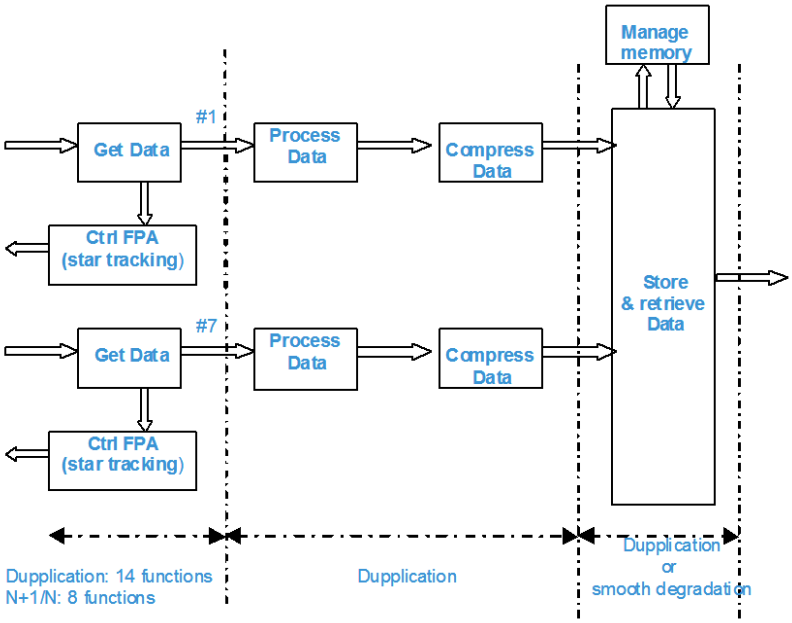


Figure 4 - Physical allocation of the second implementation - type of components

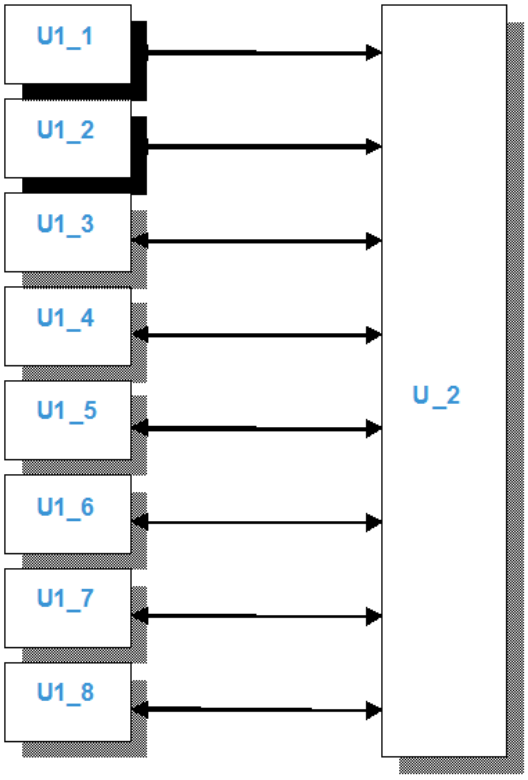


Figure 5 - Physical allocation - deployment of the second implementation

3.2 Introduction/reminder to AADL modelling patterns

Following sections remind the modelling patterns used to capture mission criteria, functional concerns and implementation specification of the system.

3.2.1 Mission criteria with AADL system and properties

System requirements are described in the AADL root component, a `system` component. Each requirement or criteria is specified by associating an AADL property to this component.

3.2.2 Functional blocks modelling with abstract and properties

Functional blocks are specified using an AADL `system` or `abstract` component. These are declared as sub-component of the AADL root system that represents the main mission component. As for mission criteria, these components describe their requirements using AADL properties. To model required communication and interfaces between functions, components define AADL features. Finally, please note that specialized software/hardware component cannot be introduced at that point.

3.2.3 Refinement into implementation with specialized AADL components

A generic building block corresponds to one AADL component with a specialized type (e.g: `device`, `processor`, `process`, `memory`, etc.). The component indicates the nature of the implementation (hardware or software) with its associated requirements (type of bus to be used as connection point, etc.). This specialized component refines the corresponding functional component (abstract or system) and redefines the buses connections to be used: the implementation component expresses its buses requirements by referencing real bus, not generic ones.

3.3 Mapping mission criteria into AADL models

First, we map mission criteria into an AADL models. As described in our AADL modelling patterns, it consists in the definition of one AADL system with appropriate properties that describe mission criteria. In the following system, we map these requirements:

- Bandwidth capacity for the connection to the earth (property `Mission_Properties::Bandwidth_To_Earth`)
- Required power to be provided to the system (property `Physical_Properties::Total_Power`)
- Maximum mass of all system equipments (property `Physical_Properties::Max_Mass`)

3.4 Functional design with AADL

Each function is specified using an AADL `abstract` component:

- **`fpa_data_get`** for image acquisition with FPA device
- **`fpa_control`** for managing and controlling FPA device
- **`process_data`** that processes data from `fpa` devices
- **`compress_data`** encodes image data received from the processing function.
- **`store_data`** interacts with the compression function and communicate with the data management function to store and retrieve data from the mass memory.
- **`manage_memory`** controls the memory device and take care of all low-level operations

Finally, functions and their interactions are specified using a global AADL `system` component (`Gaia.Functional`). It inherits the main system (the one related to mission criteria) and contains the following components:

- **7 `fpa_data_get` (one for each FPA function)**
- **7 `fpa_control`**
- **1 `process_data`**
- **1 `compress_data`**
- **1 `store_data`**

Then, the `connections` section of the `Gaia.Functional` system details the interactions between each function.

3.5 Implementations: assembling generic blocks

3.5.1 Generic blocks definition

First of all, and before defining or modelling any implementation, we define generic components using AADL. These components would be defined by system designers when creating a new device/software that could be integrated into a mission. Then, they are reused by users: composition/aggregation of such predefined components constitute the system architecture by reusing existing software/hardware components.

In the scope of our study, we define components to be reused in two packages (textual definition available in section 8.3):

1. **`The library package`** contains generic components that are not specific to any domain (bus to be used in different domain and systems, etc.)
2. **`The blocks package`** contains components specific to a particular domain (in our case, the space domain) but that can be reused on several missions (on-board software/computer that can be used on different system implementation, etc.).

The library package defines the following components:

- **`generic bus`**: corresponds to a simple bus that can interconnect function. This type of bus is useful when designers want to interconnect component without specifying the type of bus to be used.
- **`spacewire bus`**: corresponds to the definition of a SpaceWire bus (used in the space domain).
- **`mil1553 bus`**: corresponds to the specification of a MIL1553 bus.
- **`can bus`**: corresponds to the definition of a CAN bus (mainly used in the automotive domain).
- **`Ethernet bus`** with highspeed and lowspeed implementations

The blocks package defines the following components:

- **`fpa`** device: corresponds to a camera device that captures images of stars.
- **`fpa_control`** device: controls the camera (takes pictures, etc.)
- **`fpa_block`** system: assembles the fpa device and its associated controller. This component is a generic one; it is then available in two implementations: one with a runtime (on-board software) and another without any runtime.
- **`fpa_block_without_runtime`**: extends the generic `fpa_block` system and does not make any processing on the pictures. This system is only used to send/receive data. It would be used in the first implementation of the architecture.
- **`fpa_block_with_runtime`**: extends the generic `fpa_block` system. It also processes and compresses data acquired from the focal planes. It would be used in the second implementation of the system.
- **`compress_runtime`**: processor that process and compress the incoming RAW data from an FPA device. Two implementations of this components exist:

1. **Compress_runtime_lc** that has a low computation capacity. This version would be used in a **fpa_block** that embeds a runtime (second implementation of the architecture).
2. **Compress_runtime_hc** that has a high computation capacity. This version would be used in a separate on-board computer to process data that is coming from all FPA devices.

- **obc**: an on-board computer that processes and compresses data received from FPA devices. This component is used in the first implementation of the system.
- **Mass_memory**: corresponds to the physical implementation of the memory itself (device that contains electronic component for data storage).
- **Memory_runtime**: defines a processor and an environment to handle data storage/retrieval requests and is connected to the physical mass memory.
- **Memory_management**: contains all the components to manage and store data. This component is available in two versions:
  1. **Memory_management_eight_links**: can be interfaced using eight links to a network. This version of the memory management subsystem is used in the second implementation of the system (connection to each **fpa_block** subsystem).
  2. **Memory_management_two_links**: can be interfaced using two links to a network. This component would be used in the specification of the first implementation of our system, to connect the on-board computer (**obc** component) to the memory subsystem.

3.5.2 First implementation

The first implementation of the gaia mission (described in section 3.1.2) is specified in the `gaia.first_architecture` AADL component implementation (see section 8.4). It relies on generic building blocks defined in the AADL components library (see section 8.3). Then, the designer/user has to reuse predefined components that already specify their requirements/properties.

3.5.3 Second implementation

The second implementation (see section 3.1.3) is defined in the `gaia.second_architecture` AADL component implementation (see section 8.4). As for the first implementation, it uses predefined components from the library defined before (see section 8.3).

4 MODEL VALIDATION & DOCUMENT GENERATION

4.1 Validation

Figure 6 and Figure 7 show the execution of the validation theorems on both implementations. System architectures are processed and analyzed using two theorems: one for the power consumption, another for the mass. Next sections detail the execution of these theorems on each implementation.

```
[julien@minerva]/home/julien/wip/aadl/models/aram-gaia#make check-real-second-impl    11:37:07
ocarina -aadlv2 -y -I.. -g real_theorem -real_lib ./lib.real -r gaia.second_architecture gaia-i
mplementations.aadl
gaia-implementations.aadl:180:05: warning: transportlayer references a component type
ocarina: Total: 0 error and 1 warning
check_all execution
Content of set system (gaia.aadl:34:17) is
  gaia.second_architecture: 6 component instance gaia-implementations.aadl:159:03
   Evaluating verif_mass
   Evaluating total_mass
Content of set subsystems (lib.real:10:20) is
  gaia.second_architecture_u1_1: 26 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_2: 245 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_3: 424 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_4: 603 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_5: 782 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_6: 961 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_7: 1140 component instance blocks.aadl:103:03
  gaia.second_architecture_u2_1: 1319 component instance blocks.aadl:308:03
     -> value for system_mass is 36.0
   value for total_mass after evaluating compute_total_mass is 3.60000E+01
     -> value for system_max_mass is 40
     -> value for mass_ok is 0.0
   value for verif_mass after evaluating check_mass is 0.00000E+00
   Evaluating verif_power
   Evaluating total_power_consume
Content of set subsystems (lib.real:51:20) is
  gaia.second_architecture_u1_1: 26 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_2: 245 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_3: 424 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_4: 603 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_5: 782 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_6: 961 component instance blocks.aadl:103:03
  gaia.second_architecture_u1_7: 1140 component instance blocks.aadl:103:03
  gaia.second_architecture_u2_1: 1319 component instance blocks.aadl:308:03
     -> value for system_power_consume is 290.0
   value for total_power_consume after evaluating compute_total_power_consume is 2.90000E+02
     -> value for system_total_power is 350
     -> value for power_ok is 0.0
   value for verif_power after evaluating check_power_consume is 0.00000E+00
theorem check_all is: TRUE

[julien@minerva]/home/julien/wip/aadl/models/aram-gaia#                            11:37:07
```

Figure 6 - Using REAL for the validation of the first architecture

```
[julien@minerva]/home/julien/wip/aadl/models/aram-gaia#make check-real-first-impl      11:38:24
ocarina -aadlv2 -y -I.. -g real_theorem -real_lib ./lib.real -r gaia.first_architecture gaia-imp
lementations.aadl
gaia-implementations.aadl:70:05: warning: transportlayer references a component type
ocarina: Total: 0 error and 1 warning
check_all execution
Content of set system (gaia.aadl:34:17) is
  gaia.first_architecture: 6 component instance gaia-implementations.aadl:38:03
   Evaluating verif_mass
   Evaluating total_mass
Content of set subsystems (lib.real:10:20) is
  gaia.first_architecture_u1_1: 26 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_2: 156 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_3: 257 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_4: 358 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_5: 459 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_6: 560 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_7: 661 component instance blocks.aadl:73:03
  gaia.first_architecture_u2_1: 762 component instance blocks.aadl:199:03
  gaia.first_architecture_u2_2: 1029 component instance blocks.aadl:199:03
  gaia.first_architecture_u3_1: 1284 component instance blocks.aadl:295:03
     -> value for system_mass is 47.0
   value for total_mass after evaluating compute_total_mass is 4.70000E+01
     -> value for system_max_mass is 40
     -> value for mass_ok is 1.0
   value for verif_mass after evaluating check_mass is 1.00000E+00
   Evaluating verif_power
   Evaluating total_power_consume
Content of set subsystems (lib.real:51:20) is
  gaia.first_architecture_u1_1: 26 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_2: 156 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_3: 257 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_4: 358 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_5: 459 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_6: 560 component instance blocks.aadl:73:03
  gaia.first_architecture_u1_7: 661 component instance blocks.aadl:73:03
  gaia.first_architecture_u2_1: 762 component instance blocks.aadl:199:03
  gaia.first_architecture_u2_2: 1029 component instance blocks.aadl:199:03
  gaia.first_architecture_u3_1: 1284 component instance blocks.aadl:295:03
     -> value for system_power_consume is 265.0
   value for total_power_consume after evaluating compute_total_power_consume is 2.65000E+02
     -> value for system_total_power is 350
     -> value for power_ok is 0.0
   value for verif_power after evaluating check_power_consume is 0.00000E+00
gaia.aadl:44:10 Backends: error : Property is false for instance 6 (gaia.first_architecture)
theorem check_all is: FALSE

[julien@minerva]/home/julien/wip/aadl/models/aram-gaia#                      11:38:25
```

Figure 7 - REAL for the validation of the second architecture

4.1.1 Electric consumption

When evaluating the electric consumption of the system, the theorem reports a power consumption of 265W for the first architecture (20W for each `fpa` device, 40W for each on-board computer and 45W for the mass memory – total power consumption is 7 * 20 + 2 * 40 + 45 = 265). Consequently, criteria (power consumption under 265W) are met.

The validation of the second implementation reports a power consumption of 290W (35*7 + 45). Indeed, each **`fpa`** device and its associated runtime consume 35W while the mass memory consumes 45W. So, the validation theorem reports that mission criteria (power consumption under 350W) are met and does not report any error.

4.1.2 Mass budget

When evaluating the mass budget, validation of the first architecture fails. Indeed, the system is composed of 7 FPA devices (without runtimes) with a mass of 2Kg, two on-board computers with a mass of 9Kg and one mass memory unit with a mass of 15Kg. In other word, the total mass of the system has a mass of 47 Kg, which is too important regarding mission requirements (maximum mass of 40 Kg). In that case, our analysis tool (REAL) reports an error, indicating that mission requirement are not met.

The validation of the second implementation is successful: indeed, this deployment does not use on-board computer and use only 7 `fpa` devices, each one processes their data (3Kg for each) and send them to the mass memory subsystem (15Kg). Consequently, the mass of the system is 36Kg (7 * 3 + 15), which is less than mission requirements.

4.2 Documentation & qualification/certification materials generation

The document that describes the GAIA mission (**reference 4**) explains how to produce materials for certification/qualification purposes. Due to a lack of tools, production of such documents is still done manually by engineers. This section shows the accuracy of AADL to automate qualification and/or certification materials production. We illustrate that by demonstrating that documents produced in the initial report can be automatically generated from AADL models, ensuring specifications compliance.

4.2.1 Connectivity matrix generation

First of all, we introduce a connectivity matrix generator. It consists in the generation of tables that shows the connection between each subsystem. By inspecting such documents, designers and developers can evaluate, assess and optimize their architecture.

The document that describes the GAIA mission (**reference 4**) proposes a connectivity matrix for the second implementation of the system. On our side, we create a matrix generator that creates this matrix from AADL models (see Figure 8 for the matrix of the first implementation and Figure 9 for the matrix of the second implementation).

From these generated matrixes we can see that the second architecture would be more accurate than the second: it would require less bandwidth and use less connections from one component to another.

Connectivity Matrix for System gaia.first_architecture

	fpa1	fpa2	fpa3	fpa4	fpa5	fpa6	fpa7	fpa8	obc1	obc2	mm
fpa1									50000000bitsps (genericbus)	50000000bitsps (genericbus)	
fpa2									50000000bitsps (genericbus)	50000000bitsps (genericbus)	
fpa3									50000000bitsps (genericbus)	50000000bitsps (genericbus)	
fpa4									50000000bitsps (genericbus)	50000000bitsps (genericbus)	
fpa5									50000000bitsps (genericbus)	50000000bitsps (genericbus)	
fpa6									50000000bitsps (genericbus)	50000000bitsps (genericbus)	
fpa7									50000000bitsps (genericbus)	50000000bitsps (genericbus)	
fpa8									50000000bitsps (genericbus)	50000000bitsps (genericbus)	
obc1											70000000bitsps (genericbus)
obc2											70000000bitsps (genericbus)
mm											

Buses used

- **genericbus** (*bus properties are not declared*)

Figure 8 - Connectivity Matrix generated from AADL models for the first implementation

Connectivity Matrix for System gaia.second_architecture

	fpa1	fpa2	fpa3	fpa4	fpa5	fpa6	fpa7	fpa8	mm
fpa1									10000000bitsps (genericbus)
fpa2									10000000bitsps (genericbus)
fpa3									10000000bitsps (genericbus)
fpa4									10000000bitsps (genericbus)
fpa5									10000000bitsps (genericbus)
fpa6									10000000bitsps (genericbus)
fpa7									10000000bitsps (genericbus)
fpa8									10000000bitsps (genericbus)
mm									

Buses used

- **genericbus** (*bus properties are not declared*)

Figure 9 - Connectivity matrix generated from AADL models for the second implementation

4.2.2 Function implementation coverage

To provide the ability to trace the implementation components with the functional view, we introduce a traceability matrix between the implementation models and the functional models. This matrix establishes a link between components of the implementation and components from the functional view. By doing so, it provides a convenient view of the system and gives the ability to detect unimplemented function or components from the implementation that corresponds to nothing.

However, at this time, this traceability matrix is not fully complete. In particular, the specification of the binding between the implementation and the functional view is not clear. And so, the generation rules of this traceability matrix remain unclear. In particular, when a component implements a particular function, how should be considered its parent component? Shall the matrix generator consider that the parent component implicitly implements the function or is this function explicitly specified in the parent component by the system designer? Such generation or modelling rules would have an impact on the modelling approach and its associated tools.

Traceability of functions implementation for System Gaia.Validation

	get1	get2	get3	get4	get5	get6	get7	ctrl1	ctrl2	ctrl3	ctrl4	ctrl5	ctrl6	ctrl7	prs_data	compress	store	mem
Impl.U1_1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_1.datapart	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_1.ctrlpart	X	X	X	X	X	X	X	O	X	X	X	X	X	X	X	X	X	X
Impl.U1_2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_2.datapart	X	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_2.ctrlpart	X	X	X	X	X	X	X	X	O	X	X	X	X	X	X	X	X	X
Impl.U1_3	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_3.datapart	X	X	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_3.ctrlpart	X	X	X	X	X	X	X	X	X	O	X	X	X	X	X	X	X	X
Impl.U1_4	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_4.datapart	X	X	X	O	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_4.ctrlpart	X	X	X	X	X	X	X	X	X	X	O	X	X	X	X	X	X	X
Impl.U1_5	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_5.datapart	X	X	X	X	O	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_5.ctrlpart	X	X	X	X	X	X	X	X	X	X	X	O	X	X	X	X	X	X
Impl.U1_6	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_6.datapart	X	X	X	X	X	O	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_6.ctrlpart	X	X	X	X	X	X	X	X	X	X	X	X	O	X	X	X	X	X
Impl.U1_7	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_7.datapart	X	X	X	X	X	X	O	X	X	X	X	X	X	X	X	X	X	X
Impl.U1_7.ctrlpart	X	X	X	X	X	X	X	X	X	X	X	X	X	O	X	X	X	X
Impl.U2_1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U2_1.computer1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	O	X	X
Impl.U2_1.computer2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	O	X	X
Impl.U2_2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U2_2.computer1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	O	X	X
Impl.U2_2.computer2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	O	X	X
Impl.U3_1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Impl.U3_1.mm	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	O
Impl.U3_1.runtime	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	O	X
Impl.transportlayer	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 10 - Functions implementation traceability for the first implementation

5 FURTHER WORK AND PERSPECTIVES

This section summarizes the modelling approach we propose in our approach and provides some perspectives and next steps to improve this development approach.

5.1 Documentation & qualification/certification materials generation

First of all, system validation (theorems to be checked by tools) as well as qualification/certification materials generation can be improved. In particular, other requirements can be checked from AADL models. This would be addressed through the definition of more theorems that can be processed by REAL.

Certification/qualification materials could also be improved and more documents would be produced from AADL models. This would help system designers and engineers by automating the production of such documents and ensuring their consistency with specifications.

5.2 Interface with domain-specific tools

Another concern is the validation of the system for some specific requirements. In some case, validation approach such as REAL is not sufficient and it is necessary to interface the AADL models with domain-specific tools to analyze the system and check its requirements.

For example, in the case of bus load analysis and bandwidth analysis, a precise validation of these aspects would require interfacing the specifications with dedicated analysis tools. Indeed, such validation requires to precisely knowing the behaviour of the bus (priority, bus sharing policy, etc) and cannot be done using a theorem-based approach.

However, interfacing AADL models with domain-specific tools would be easily feasible since all required information are available in the specifications (models) and would only need to be exported to an appropriate representation format to be processed by specific tools.

6 CONCLUSION

This white paper is a follow-up of our previous study for the specification of avionics architecture (reference 3). The overall approach is then quite stable: designers first specify mission requirements and criteria. Then, they define system functions and their interaction. Finally, this function view of the system is refined into an implementation using specialized components that precisely describe system requirements (required bandwidth, computation capacity to be consumed, etc.). Finally, these models are processed by tools to (1) validate several requirements and (2) automatically create documents for qualification/certification.

Using our methodology to design a real case-study (the GAIA mission, see reference 4) leads us to design tools that demonstrate the relevancy of AADL models for avionics system modelling and the potential automation of documents generation for qualification/certification. It also shows that requirements can be validated from AADL models. However, other system properties are very specific and would require to be processed by dedicated tools. In that case, it would be possible to export AADL specifications to an appropriate abstraction level to be analyzed and processed by these domain-specific tools.

7 REFERENCES

1. *"A Practice Framework for Model-Based Analysis Using the Architecture Analysis & Design Language (AADL)"* – NASA – 13/02/2009
2. *"AADL Practice Framework: Preliminary Version"* – Embry-Riddle – Technical Report – 09/2006
3. *"Modelling Hardware Avionics Architecture with AADL"*. Julien Delange. White paper for the AADL committee, January 2011.
4. *"Guidelines for the selection of architectures"*. Jean-François Soucaille & Luc Planche. Technical report, ASTRIUM, 12/2010.
5. *"System Architecture Virtual Integration: A Case Study"*. P. Feiler, L. Wrage and J. Hansson. In ERTSS2010.

8 TEXTUAL VERSION OF AADL MODELS

8.1 Mission requirements and criteria

```
system Gaia
properties
  Mission_Properties::Bandwidth_To_Earth => 10 Mbytesps;
  Physical_Properties::Total_Power       => 350 W;
  Physical_Properties::Max_Mass          => 40 Kg;
end Gaia;
```

8.2 Functions specification

```
package GAIA::Functions
public

  with ARAM_Properties;
  with Physical_Properties;
  with Processor_Properties;
  with Bus_Properties;

  with GAIA;
  with Data_Types;

  ------------------------
  -- Mission functions --
  ------------------------

  -- These abstract component types define functional blocks. These
  -- use the AADLv2 abstract component type, as these are to be later
  -- refined as either software or hardware blocks. This is to be
  -- decided at implementation time.

  abstract fpa_data_get
  features
    dataout     : out data port Data_Types::fpa_data;
    ctrlout     : out data port Data_Types::fpa_ctrl;
  end fpa_data_get;

  abstract fpa_control
  features
    ctrlin      : in data port Data_Types::fpa_ctrl;
    ctrlout     : out data port Data_Types::fpa_ctrl;
  end fpa_control;
```

```
abstract process_data
features
  fpadata1    : in data port Data_Types::fpa_data;
  fpadata2    : in data port Data_Types::fpa_data;
  fpadata3    : in data port Data_Types::fpa_data;
  fpadata4    : in data port Data_Types::fpa_data;
  fpadata5    : in data port Data_Types::fpa_data;
  fpadata6    : in data port Data_Types::fpa_data;
  fpadata7    : in data port Data_Types::fpa_data;
  output      : out data port Data_Types::processed_data;
end process_data;

abstract compress_data
features
  input    : in data port Data_Types::processed_data;
  output   : out data port Data_Types::compressed_data;
end compress_data;

abstract store_data
features
  input    : in  data port Data_Types::compressed_data;
  output   : out data port Data_Types::compressed_data;
properties
  ARAM_Properties::Required_Memory => 800 GByte;
end store_data;

abstract manage_memory
features
  input : in data port Data_Types::compressed_data;
properties
  Processor_Properties::MIPS => 1;
end manage_memory;
-- Gaia functional design
--
-- In this model, we propose a model that supports the function view
-- of the Gaia mission.
--
-- The Gaia system component type extends Mission_Criteria::Gaia,
-- and thus inherits its requirements, and also the validation rules
-- to be performed.
--
-- The Gaia.Functionnal component implementation details how
-- functional blocks are to be used in this variant of the model.

system Gaia extends GAIA::Gaia
end Gaia;

system implementation Gaia.Functional
subcomponents
  get1        :  abstract fpa_data_get;
  get2        :  abstract fpa_data_get;
  get3        :  abstract fpa_data_get;
  get4        :  abstract fpa_data_get;
  get5        :  abstract fpa_data_get;
  get6        :  abstract fpa_data_get;
  get7        :  abstract fpa_data_get;
```

```
  ctrl1       :  abstract fpa_control;
  ctrl2       :  abstract fpa_control;
  ctrl3       :  abstract fpa_control;
  ctrl4       :  abstract fpa_control;
  ctrl5       :  abstract fpa_control;
  ctrl6       :  abstract fpa_control;
  ctrl7       :  abstract fpa_control;

  prs_data    :  abstract process_data;
  compress    :  abstract compress_data;
  store       :  abstract store_data;

  mem         :  abstract manage_memory;

connections
  port get1.ctrlout -> ctrl1.ctrlin
  {Bus_Properties::Required_Bandwidth => 20_000_000 bitsps;};
  port get2.ctrlout -> ctrl2.ctrlin
  {Bus_Properties::Required_Bandwidth => 20_000_000 bitsps;};
  port get3.ctrlout -> ctrl3.ctrlin
  {Bus_Properties::Required_Bandwidth => 20_000_000 bitsps;};
  port get4.ctrlout -> ctrl4.ctrlin
  {Bus_Properties::Required_Bandwidth => 20_000_000 bitsps;};
  port get5.ctrlout -> ctrl5.ctrlin
  {Bus_Properties::Required_Bandwidth => 20_000_000 bitsps;};
  port get6.ctrlout -> ctrl6.ctrlin
  {Bus_Properties::Required_Bandwidth => 20_000_000 bitsps;};
  port get7.ctrlout -> ctrl7.ctrlin
  {Bus_Properties::Required_Bandwidth => 20_000_000 bitsps;};

  port get1.dataout -> prs_data.fpadata1
  {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;};
  port get2.dataout -> prs_data.fpadata2
  {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;};
  port get3.dataout -> prs_data.fpadata3
  {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;};
  port get4.dataout -> prs_data.fpadata4
  {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;};
  port get5.dataout -> prs_data.fpadata5
  {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;};
  port get6.dataout -> prs_data.fpadata6
  {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;};
  port get7.dataout -> prs_data.fpadata7
  {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;};

  port prs_data.output -> compress.input
  {Bus_Properties::Required_Bandwidth => 140_000_000 bitsps;};

  port compress.output -> store.input
  {Bus_Properties::Required_Bandwidth => 70_000_000 bitsps;};

  port store.output -> mem.input;
properties
  Period                    => 1 ms;
  --  XXX What is the meaning of this period?
```

```
    Physical_Properties::Mass => 40 Kg;
  end Gaia.Functional;

end GAIA::Functions;
```

8.3 Generic components

```
-- This package models a set of generic reusable elements, outside of
-- the space domain.

-- XXX to be extended/corrected so that these elements match actual
-- hardware/software elements.

package Library
public
  with Bus_Properties;

  -----------
  -- Buses --
  -----------

  bus generic
  end generic;

  bus genericbus extends generic
  end genericbus;

  --  SpaceWire bus

  bus spacewire extends genericbus
  end spacewire;

  bus implementation spacewire.i
  properties
    Bus_Properties::Bus_Type     => ptp;
    Bus_Properties::Bandwidth    => 100_000_000 bitsps;
    Bus_Properties::Max_Latency  => 100 ms;
  end spacewire.i;

  --  MIL-STD 1553 bus

  bus mil1553 extends genericbus
  properties
    Bus_Properties::Bus_Type     => mtp;
    Bus_Properties::Bandwidth    => 1_000_000 bitsps;
    Bus_Properties::Max_Latency  => 100 ms;
  end mil1553;

  bus implementation mil1553.i
  end mil1553.i;
```

```
  --  CAN Bus

  bus can extends genericbus
  properties
    Bus_Properties::Bus_Type     => mtp;
    Bus_Properties::Bandwidth    => 600_000 bitsps;
    Bus_Properties::Max_Latency  => 100 ms;
  end can;

  bus implementation can.i
  end can.i;

  --  Ethernet

  bus ethernet extends genericbus
  end ethernet;

  bus implementation ethernet.highspeed
  properties
    Bus_Properties::Bus_Type     => mtp;
    Bus_Properties::Bandwidth    => 100_000_000 bitsps;
    Bus_Properties::Max_Latency  => 100 ms;
  end ethernet.highspeed;

  bus implementation ethernet.lowspeed
  properties
    Bus_Properties::Bus_Type     => mtp;
    Bus_Properties::Bandwidth    => 10_000_000 bitsps;
    Bus_Properties::Max_Latency  => 100 ms;
  end ethernet.lowspeed;

end Library;


-- This package models reusable functional blocks for space missions.

package Blocks
public

  with ARAM_Properties;
  with Physical_Properties;
  with Bus_Properties;

  with Library;
  with GAIA::Functions;
  with Data_Types;

  ---------
  -- FPA --
  ---------

  device FPA
    --  The FPA (Focal Plane Arrays) camera device corresponds to a
    --  device that acquires/captures images of stars.
```

```
  features
    dataout : out data port Data_Types::FPA_data;
    ctrlout : out data port Data_Types::FPA_ctrl;
  properties
    ARAM_Properties::Realizes => (classifier
(GAIA::Functions::FPA_data_get));
  end FPA;

  device implementation FPA.i
  end FPA.i;

  -----------------
  -- FPA_Control --
  -----------------

  device FPA_control
    --  The FPA_control device corresponds to the device that controls
    --  the camera itself (take picture, etc..)

  features
    ctrlin      : in data port Data_Types::FPA_ctrl;
    ctrlout     : out data port Data_Types::FPA_ctrl;
  properties
    ARAM_Properties::Realizes => (classifier
(GAIA::Functions::FPA_control));
  end FPA_control;

  device implementation FPA_control.i
  end FPA_control.i;

  ---------------
  -- FPA_block --
  ---------------

  system FPA_block
  --  The FPA_block component assembles both main FPA
  --  functions: image acquisition and FPA device control.
  features
    bus_access : requires bus access Library::genericbus;
  end FPA_block;

  -------------------------------
  -- FPA_block_without_runtime --
  -------------------------------

  system FPA_block_without_runtime extends FPA_block
  -- FPA_block_without_runtime contains the FPA device for image
  -- acquisition. It sends RAW (uncompressed) data to the obc that
  -- compresses the pictures.  This component is used for deployment 1.
  features
    dataout : out data port Data_Types::FPA_data;
  end FPA_block_without_runtime;

  system implementation FPA_block_without_runtime.i
  subcomponents
    datapart    : device FPA.i;
```

```
  ctrlpart    : device FPA_control.i;

connections
  port datapart.dataout -> dataout;
  port datapart.ctrlout -> ctrlpart.ctrlin
  {Bus_Properties::Required_Bandwidth => 20_000_000 bitsps;};

properties
  Physical_Properties::Mass => 2 Kg;
  Physical_Properties::Power_Consume => 20 W;
end FPA_block_without_runtime.i;

----------------------------
-- FPA_block_with_runtime --
----------------------------

system FPA_block_with_runtime extends FPA_block
-- FPA_block_with_runtime contains the FPA device for image
-- acquisition, the FPA_control to control the FPA device as well as
-- a runtime that compress the raw data from the FPA device. Then,
-- the runtime sends directly the compressed data to the mass memory
-- subsystem.  This component is used for deployment 2.

features
  output : out data port Data_Types::compressed_data;
end FPA_block_with_runtime;

system implementation FPA_block_with_runtime.i
subcomponents
  datapart    : device FPA.i;
  ctrlpart    : device FPA_control.i;
  processing  : processor compress_runtime_lc.i;

connections
  port datapart.dataout -> processing.raw;
  port datapart.ctrlout -> ctrlpart.ctrlin
  {Bus_Properties::Required_Bandwidth => 20_000_000 bitsps;};

  bus access bus_access -> processing.bus_access;

properties
  Physical_Properties::Mass => 3 Kg;
  Physical_Properties::Power_Consume => 35 W;
end FPA_block_with_runtime.i;

----------------------
-- Compress_Runtime --
----------------------

processor compress_runtime
  --  The compress_runtime component is the processor used to
  --  compress the incoming raw data from the FPA device.

features
  bus_access : requires bus access Library::genericbus;
```

```
  properties
    ARAM_Properties::Realizes =>
    (classifier (GAIA::Functions::compress_data),
     classifier (GAIA::Functions::process_data));
  end compress_runtime;

  processor compress_runtime_hc extends compress_runtime
    -- compress_runtime_hc means "High Capacity": it has a high
    -- computation capacity to compress incoming raw data from all FPA
    -- devices.  It is used in deployment 1 as a processor in the obc
    -- subsystem.
  features
    FPAdata1       : in data port Data_Types::FPA_data;
    FPAdata2       : in data port Data_Types::FPA_data;
    FPAdata3       : in data port Data_Types::FPA_data;
    FPAdata4       : in data port Data_Types::FPA_data;
    FPAdata5       : in data port Data_Types::FPA_data;
    FPAdata6       : in data port Data_Types::FPA_data;
    FPAdata7       : in data port Data_Types::FPA_data;
    compressed     : out data port Data_Types::compressed_data;
  end compress_runtime_hc;

  processor implementation compress_runtime_hc.i
  end compress_runtime_hc.i;

  processor compress_runtime_lc extends compress_runtime
    -- compress_runtime_lc means "Low Capacity": it has a low
    -- computation capacity and is embedded in the
    -- FPA_block_with_runtime subsystem. It has a lower computation
    -- capacity because it compresses only one picture at the same
    -- time whereas the High Capacity version must compress 8 pictures
    -- with the same deadline.  This version is used in deployment 2.

  features
    raw        : in data port Data_Types::FPA_data;
    compressed : out data port Data_Types::compressed_data;
  end compress_runtime_lc;

  processor implementation compress_runtime_lc.i
  end compress_runtime_lc.i;

  ---------
  -- OBC --
  ---------

  system obc
  --  The obc subsystem corresponds to the computer that compresses
  --  the incoming raw data from the 8 FPA devices. Its implementation
  --  contains a high capacity processing resources, able to compress
  --  all pictures with a tight timing constraint (1ms).  Two
  --  processor are used to introduce redundancy.
  --
  --  This subsystem is used in deployment 1.
  features
    bus_access  : requires bus access Library::genericbus;
    rawdata1    : in data port Data_Types::FPA_data;
```

```
    rawdata2    : in data port Data_Types::FPA_data;
    rawdata3    : in data port Data_Types::FPA_data;
    rawdata4    : in data port Data_Types::FPA_data;
    rawdata5    : in data port Data_Types::FPA_data;
    rawdata6    : in data port Data_Types::FPA_data;
    rawdata7    : in data port Data_Types::FPA_data;
    rawdata8    : in data port Data_Types::FPA_data;
    compressed  : out data port Data_Types::compressed_data ;
  end obc;

  system implementation obc.i
  subcomponents
    computer1 : processor compress_runtime_hc.i;
    computer2 : processor compress_runtime_hc.i;

  connections
    bus access bus_access -> computer1.bus_access;
    bus access bus_access -> computer2.bus_access;

  properties
    Physical_Properties::Mass => 9 Kg;
    Physical_Properties::Power_Consume => 40 W;
  end obc.i;

  -----------------
  -- Mass_Memory --
  -----------------

  memory mass_memory
  --  The mass_memory component corresponds to the memory device used
  --  to store and retrieve pictures. It is the memory device that
  --  contains all the compressed data produced by either the on board
  --  computer or the FPA subsystem block.

  features
    bus_access : requires bus access Library::genericbus;
  end mass_memory;

  memory implementation mass_memory.i
  properties
    ARAM_Properties::Realizes => (classifier
(GAIA::Functions::manage_memory));
  end mass_memory.i;

  --------------------
  -- Memory_Runtime --
  --------------------

  processor memory_runtime
  --  The memory_runtime processor is used on the memory subsystem to
  --  allocate memory and organize the memory organization of the
  --  hardware part of the memory.

  features
    bus_access  : requires bus access Library::genericbus;
    input       : in data port Data_Types::compressed_data;
```

```
  properties
    ARAM_Properties::Realizes => (classifier
(GAIA::Functions::store_data));
  end memory_runtime;

  processor implementation memory_runtime.i
  end memory_runtime.i;

  -----------------------
  -- Memory_Management --
  -----------------------

  system memory_management
  --  The memory_management component gathers all components required
  --  for the implementation of the memory subsystem.  It contains the
  --  hardware part of the memory itself the mass_memory memory
  --  component as well as the runtime that controls it - the
  --  memory_runtime component).
  --
  --  In deployment 1, it is connected to the obc that sends processed
  --  and compressed data.
  --
  --  In deployment 2, it is connected to FPA subsystems that sends
  --  directly compressed data. For communication purposes, this
  --  subsystem requires a connection to a bus.

  features
    bus_access  : requires bus access Library::genericbus;
    to_cdmu     : out data port Data_Types::compressed_data;

  properties
    ARAM_Properties::Required_Memory => 800 GByte;
  end memory_management;

  system memory_management_two_links extends memory_management
  features
    link1 : in data port Data_Types::compressed_data;
    link2 : in data port Data_Types::compressed_data;
  end memory_management_two_links;

  system memory_management_eight_links extends memory_management
  features
    link1 : in data port Data_Types::compressed_data;
    link2 : in data port Data_Types::compressed_data;
    link3 : in data port Data_Types::compressed_data;
    link4 : in data port Data_Types::compressed_data;
    link5 : in data port Data_Types::compressed_data;
    link6 : in data port Data_Types::compressed_data;
    link7 : in data port Data_Types::compressed_data;
    link8 : in data port Data_Types::compressed_data;
  end memory_management_eight_links;

  system implementation memory_management_two_links.i
  subcomponents
    mm      : memory mass_memory.i;
    runtime : processor memory_runtime.i;
```

```
  connections
    bus access bus_access -> runtime.bus_access;

  properties
    Physical_Properties::Mass => 15 Kg;
    Physical_Properties::Power_Consume => 45 W;
  end memory_management_two_links.i;

  system implementation memory_management_eight_links.i
  subcomponents
    mm      : memory mass_memory.i;
    runtime : processor memory_runtime.i;

  connections
    bus access bus_access -> runtime.bus_access;
  properties
    Physical_Properties::Mass => 15 Kg;
    Physical_Properties::Power_Consume => 45 W;
  end memory_management_eight_links.i;

end Blocks;
```

8.4 System implementations

```
-- This package models alternatives for the Gaia implementation
-- design.

package GAIA::Implementations
public

  with ARAM_Properties;
  with Bus_Properties;

  with Blocks;
  with Library;
  with GAIA;
  with GAIA::Functions;

  -- Gaia implementation design
  --
  -- In this model, we propose a model that supports the
  -- implementation view of the Gaia mission.
  --
  -- The Gaia system component type extends Gaia::Gaia, and thus
  -- inherits its requirements, and also the validation rules to be
  -- checked.

  system Gaia extends GAIA::Gaia
  end Gaia;
```

```
  ------------------------------------------------------------------------
-----
  -- Gaia candidate design #1
  --  This design corresponds to the first iteration, with the
  --  following functions:
  --  * 7 couples of (Get_Data + Ctrl FPA) units, made redundant using
  --  a N + 1/N scheme
  --    => hence 8 functions, named U1_1 to U1_8
  --  * 1 couple of (Process Data + Compress Data), duplicated
  --    => hence 2 functions, named U2_1 and U2_2
  --  * 1 mass memory unit, named U3_1

  system implementation gaia.first_architecture
  subcomponents

    -- From the design document, one can infere that

    -- * U1 units are pure hardware, using one board,
    --   consumes 20W, requires 600MIPs, weight 2Kg

    U1_1 : system blocks::fpa_block_without_runtime.i;
    U1_2 : system blocks::fpa_block_without_runtime.i;
    U1_3 : system blocks::fpa_block_without_runtime.i;
    U1_4 : system blocks::fpa_block_without_runtime.i;
    U1_5 : system blocks::fpa_block_without_runtime.i;
    U1_6 : system blocks::fpa_block_without_runtime.i;
    U1_7 : system blocks::fpa_block_without_runtime.i;

    --    U1_8 : system blocks::fpa_block_without_runtime.i;
    -- Note: this component is added to follow the N + 1/N redundancy
    -- pattern. It is disabled for now.

    -- * U2 units are OBCs, 4200 MIPs, cycle time range of 1ms,
    --   consumes 40W, weights 9Kg.

    U2_1 : system blocks::obc.i;
    U2_2 : system blocks::obc.i;

    -- * U3 unit is a mass memory. We select the two links variant, as
    --   this memory is to be connected to U2 units. It requires
    --   1MIPS, consumes 45W, weights 15kg.

    U3_1 : system blocks::memory_management_two_links.i;

    transportlayer : bus Library::genericbus;

    -- Evaluation results:
    --  total weight: 7 * 2 + 2 * 9 + 15 = 47 kg
    --  total power:  7 * 20 + 2 * 40 + 45 = 265 W

  connections
    -- Each U1 unit is connected to the two U2 units

    port U1_1.dataout -> U2_1.rawdata1
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
```

```
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_2.dataout -> U2_1.rawdata2
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_3.dataout -> U2_1.rawdata3
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_4.dataout -> U2_1.rawdata4
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_5.dataout -> U2_1.rawdata5
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_6.dataout -> U2_1.rawdata6
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_7.dataout -> U2_1.rawdata7
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    -- port U1_8.dataout -> U2_1.rawdata8
    -- {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    -- Actual_Connection_Binding => (reference (transportlayer));};

    port U1_1.dataout -> U2_2.rawdata1
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_2.dataout -> U2_2.rawdata2
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_3.dataout -> U2_2.rawdata3
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_4.dataout -> U2_2.rawdata4
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_5.dataout -> U2_2.rawdata5
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_6.dataout -> U2_2.rawdata6
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U1_7.dataout -> U2_2.rawdata7
    {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    -- port U1_8.dataout -> U2_2.rawdata8
    -- {Bus_Properties::Required_Bandwidth => 50_000_000 bitsps;
    -- Actual_Connection_Binding => (reference (transportlayer));};

    port U2_1.compressed -> U3_1.link1
    {Bus_Properties::Required_Bandwidth  => 70_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};
    port U2_2.compressed -> U3_1.link2
    {Bus_Properties::Required_Bandwidth  => 70_000_000 bitsps;
    Actual_Connection_Binding => (reference (transportlayer));};

    bus access transportlayer -> U3_1.bus_access;
```

```
    bus access transportlayer -> U2_1.bus_access;
    bus access transportlayer -> U2_2.bus_access;

    bus access transportlayer -> U1_1.bus_access;
    bus access transportlayer -> U1_2.bus_access;
    bus access transportlayer -> U1_3.bus_access;
    bus access transportlayer -> U1_4.bus_access;
    bus access transportlayer -> U1_5.bus_access;
    bus access transportlayer -> U1_6.bus_access;
    bus access transportlayer -> U1_7.bus_access;
    -- bus access transportlayer -> U1_8.bus_access;
  end gaia.first_architecture;

  ------------------------------------------------------------------------
-----
  -- Gaia candidate design #2
  --  This design corresponds to the second iteration, with the
  --  following functions:
  --  * 7 couples of (Get_Data + Ctrl FPA + compress + process) units,
  --  made redundant using a N + 1/N scheme
  --  * 1 mass memory unit, named U3_1

  system implementation gaia.second_architecture
  subcomponents

    -- From the design document, one can infere that

    -- * U1 units are pure hardware, using one board,
    --   consumes 35W, requires 600MIPs, weight 3Kg

    U1_1 : system blocks::fpa_block_with_runtime.i;
    U1_2 : system blocks::fpa_block_with_runtime.i;
    U1_3 : system blocks::fpa_block_with_runtime.i;
    U1_4 : system blocks::fpa_block_with_runtime.i;
    U1_5 : system blocks::fpa_block_with_runtime.i;
    U1_6 : system blocks::fpa_block_with_runtime.i;
    U1_7 : system blocks::fpa_block_with_runtime.i;
    -- U1_8 : system blocks::fpa_block_with_runtime.i;

    -- * U2 units requires 1 MIPs consumes 45W, weights 15Kg.

    U2_1   : system blocks::memory_management_eight_links.i;

    transportlayer : bus Library::genericbus;

    -- Evaluation results:
    --  total weight: 7 * 3 + 1 * 15 = 36 kg
    --  total power:  7 * 35 + 45 = 290 W

  connections
    port U1_1.output -> U2_1.link1
    {Bus_Properties::Required_Bandwidth  => 10_000_000 bitsps;
     Actual_Connection_Binding => (reference (transportlayer));};
    port U1_2.output -> U2_1.link2
    {Bus_Properties::Required_Bandwidth  => 10_000_000 bitsps;
```

```
     Actual_Connection_Binding => (reference (transportlayer));};
    port U1_3.output -> U2_1.link3
    {Bus_Properties::Required_Bandwidth  => 10_000_000 bitsps;
     Actual_Connection_Binding => (reference (transportlayer));};
    port U1_4.output -> U2_1.link4
    {Bus_Properties::Required_Bandwidth  => 10_000_000 bitsps;
     Actual_Connection_Binding => (reference (transportlayer));};
    port U1_5.output -> U2_1.link5
    {Bus_Properties::Required_Bandwidth  => 10_000_000 bitsps;
     Actual_Connection_Binding => (reference (transportlayer));};
    port U1_6.output -> U2_1.link6
    {Bus_Properties::Required_Bandwidth  => 10_000_000 bitsps;
     Actual_Connection_Binding => (reference (transportlayer));};
    port U1_7.output -> U2_1.link7
    {Bus_Properties::Required_Bandwidth  => 10_000_000 bitsps;
     Actual_Connection_Binding => (reference (transportlayer));};
    -- port U1_8.output -> U2_1.link8
    -- {Bus_Properties::Required_Bandwidth  => 10_000_000 bitsps;
    --  Actual_Connection_Binding => (reference (transportlayer));};


    bus access transportlayer -> U2_1.bus_access;

    bus access transportlayer -> U1_1.bus_access;
    bus access transportlayer -> U1_2.bus_access;
    bus access transportlayer -> U1_3.bus_access;
    bus access transportlayer -> U1_4.bus_access;
    bus access transportlayer -> U1_5.bus_access;
    bus access transportlayer -> U1_6.bus_access;
    bus access transportlayer -> U1_7.bus_access;
    -- bus access transportlayer -> U1_8.bus_access;
  end gaia.second_architecture;

end GAIA::Implementations;
```

9 DEFINITION OF NEW AADL PROPERTIES

9.1 Mission Properties

```
-- This package defines properties for defining space mission
-- objectives and requirements.

property set Mission_Properties is

  --------------------
  -- Mission Timing --
  --------------------

  Mission_Time_Units: type units
  (ps,
  ns   => ps  * 1000,
  us   => ns  * 1000,
  ms   => us  * 1000,
```

```
  sec  => ms  * 1000,
  min  => sec * 60,
  hr   => min * 60,
  day  => hr  * 24,
  week => day * 7);

  Mission_Max_Time: constant Mission_Duration_Type => 1000 week;

  Mission_Duration_Type: type aadlinteger 0 ps .. Mission_Max_Time
  units Mission_Time_Units;

  Mission_Duration : inherit Mission_Duration_Type applies to (system);

  --------------------------------
  -- Communication requirements --
  --------------------------------

  Bandwidth_To_Earth : Data_Volume applies to (abstract, system, device,
bus);

end Mission_Properties;
```

9.2 Physical Properties

```
-- This package defines some units related to physics of systems.

property set Physical_Properties is

  -----------------
  -- Power units --
  -----------------

  Power_Units: type units
  (W,
   KW  => W * 1000,
   MW  => KW * 1000,
   GW  => MW * 1000,
   TW  => GW * 1000);

  Max_Power: constant Power_Units => 2#1#e32 W;
  Power : type aadlreal 0 W .. Max_Power units Power_Units;
  Power_Consume : Power applies to (processor, device, memory, system);
  Power_Provide : Power applies to (device);
  Total_Power   : Power applies to (system);

  ---------
  -- EMC --
  ---------

  Ionizing_Dose_Units : type units (
  rad,
  Krad  => rad * 1000,
  Mrad  => Krad * 1000,
  Grad  => Mrad * 1000,
```

```
  Trad  => Grad * 1000);

  Max_Ionizing_Dose: constant Ionizing_Dose_Units => 2#1#e32 rad;

  Ionizing_Type : type aadlinteger
     0 rad .. Max_Ionizing_Dose units Ionizing_Dose_Units;

  TID : Ionizing_Type applies to (processor, device, memory);

  ----------------
  -- Mass units --
  ----------------

  Mass_Units: type units (
  g,
  Kg => g  * 1000,
  T  => Kg * 1000);

  Maximum_Mass: constant Mass_Units => 2#1#e32 g;

  Mass_Type : type aadlreal 0 g .. Maximum_Mass units Mass_Units;

  Mass : Mass_Type applies to (processor, device, memory, system);
  -- Define the mass of an element. Note: if both a system and its
  -- subcomponents define a mass, then it is assumed that the mass of
  -- the system should be more or equal to the sum of the mass of its
  -- subcomponents.

  Max_Mass : Mass_Type applies to (processor, device, memory, system);

end Physical_Properties;
```

9.3 Processor Properties

```
property set Processor_Properties is

  Frequency : type aadlinteger 0 Hz .. Max_Aadlinteger units
  (Hz,
  KHz => Hz  * 1000,
  MHz => KHz * 1000,
  GHz => MHz * 1000);

  CPU_Speed : Frequency applies to (processor);
  MIPS : aadlinteger 0 .. Max_Aadlinteger applies to (processor,
abstract);
end Processor_Properties;
```

9.4 Bus Properties

```
property set Bus_Properties is

  Required_Bandwidth : Data_Volume applies to
```

```
    (abstract, system, device, bus access, connection);

  Bandwidth : Data_Volume applies to (abstract, system, device, bus);

  Bus_Type : enumeration (mtp, ptp) applies to (bus, system, abstract);
  --  mtp: multi-point bus
  --  ptp: point-to-point bus

  Expected_Latency : Time_Range applies to (bus, bus access, abstract);

  Max_Latency : Time applies to (bus, bus access, abstract);

end Bus_Properties;
```

9.5 Dedicated ARAM Properties

```
property set ARAM_Properties is

  Required_Memory : inherit Size applies to
    (system, abstract, device, processor);

  Minimum_Size : inherit Size applies to (memory, abstract);

  Realizes : inherit list of classifier (abstract, system) applies to
    (processor, bus, device, abstract, memory, process);
  --  In our modeling patterns, we use AADLv2 abstract components to
  --  model Java-like interfaces, i.e. a set of services to be
  --  implemented. The Realizes functions indicates which interface is
  --  implemented by a concrete component type

  Actual_Function_Binding : inherit list of reference (abstract, system)
    applies to (processor, bus, device, abstract, memory, process);
  --  Define the binding of abstract function to actual implementations.

end ARAM_Properties;
```