Modelling Hardware Avionics Architecture with AADL

Julien Delange – julien.delange@esa.int

1 RATIONALE

This document presents an approach for the modelling of hardware concerns of avionics architecture. 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.

To do so, we propose a dedicated design approach that allows designers to:

1. **Specify mission criteria** in a dedicated formalism
2. **Describe system functional architecture** with its requirements
3. **Refine the functional architecture** by specifying its implementation using generic building blocks
4. **Validate the implementation** by checking mission criteria fulfilment.

This development approach is practical: most of the time, designers reuse existing components to avoid new developments (which is error-prone, costly). In addition, feedback from the industry demonstrates that functional architecture may vary but actual implementations are quite similar and only introduce some variations.

However, components reuse may lead to error: assembly of components from different projects may be incompatible and cause issues that are difficult to detect. In addition, it is especially important to check that planned implementation can meet mission requirements (in terms of processing capacity, power, etc.). Such an assessment is very difficult to trace and find and can significantly increase development costs (in terms of time, money, etc.).

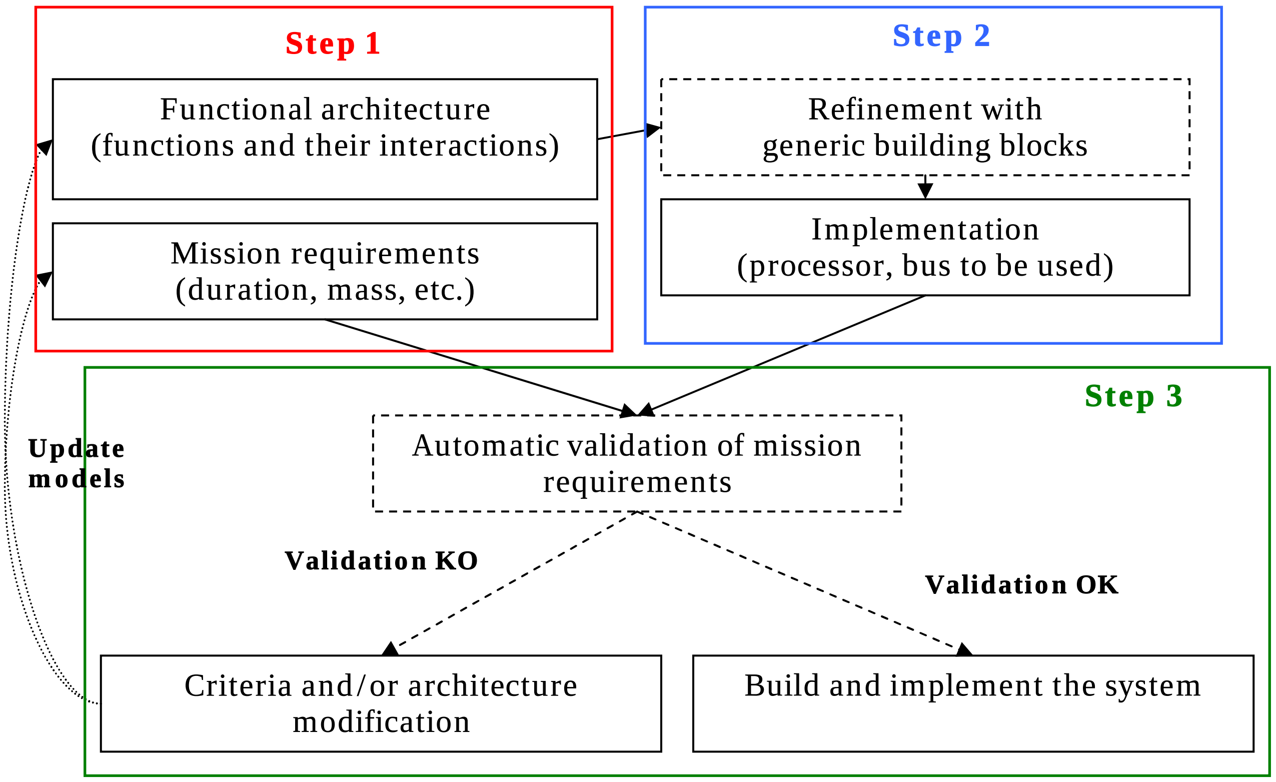
By introducing modelling at the earliest in the development process, we describe all requirements that may be conflicting, even if all requirements are not known. Then, during the refinement of the architecture, the functions are detailed and analysis tools can then detect more potential issues.

2 DESIGN FLOW

We propose the following design approach that consists in three main steps, as illustrated in Figure 1:

1. **System designers specify mission requirements** as well as the functional architecture of the system using models. In that step (illustrated in red in Figure 1), only functional aspects are described and the level of details is still low.

2. **Functions are refined using generic building blocks**, describing the planned implementation of the final system. Each building block specifies the resources they consume (in terms of computation capacity, weight, power, etc.). During that refinement step (illustrated in blue in the Figure 1), all requirements are specified so that the details level is more fine-grained (as illustrated in Figure 2): each component specifies its implementation detail (in terms of memory consumption, etc.).
3. **Automatic validation tools analyze the implementation model** produced during the refinement process and assess the feasibility of the implementation according to mission criteria and system functions. After that step (illustrated in green in Figure 1), designers know if the implementation meets mission criteria and can start to build and implement it or if the architecture and/or mission criteria must be revised.



Step 1
Functional architecture (functions and their interactions)
Mission requirements (duration, mass, etc.)
Step 2
Refinement with generic building blocks
Implementation (processor, bus to be used)
Step 3
Automatic validation of mission requirements
Validation KO
Validation OK
Criteria and/or architecture modification
Build and implement the system
Update models

Figure 1 - Design approach

To model mission requirements, the functional architecture and planned implementation, we use AADL models. AADL provide a modelling language suitable to software and hardware modelling. It also 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.

The main purpose of this approach is to verify that the final implementation fulfils mission criteria and does not exceed its limits. To do so, an automatic validation tool checks mission requirements against implementation characteristics (in terms of system properties such as bus bandwidth, weight, etc.).

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.

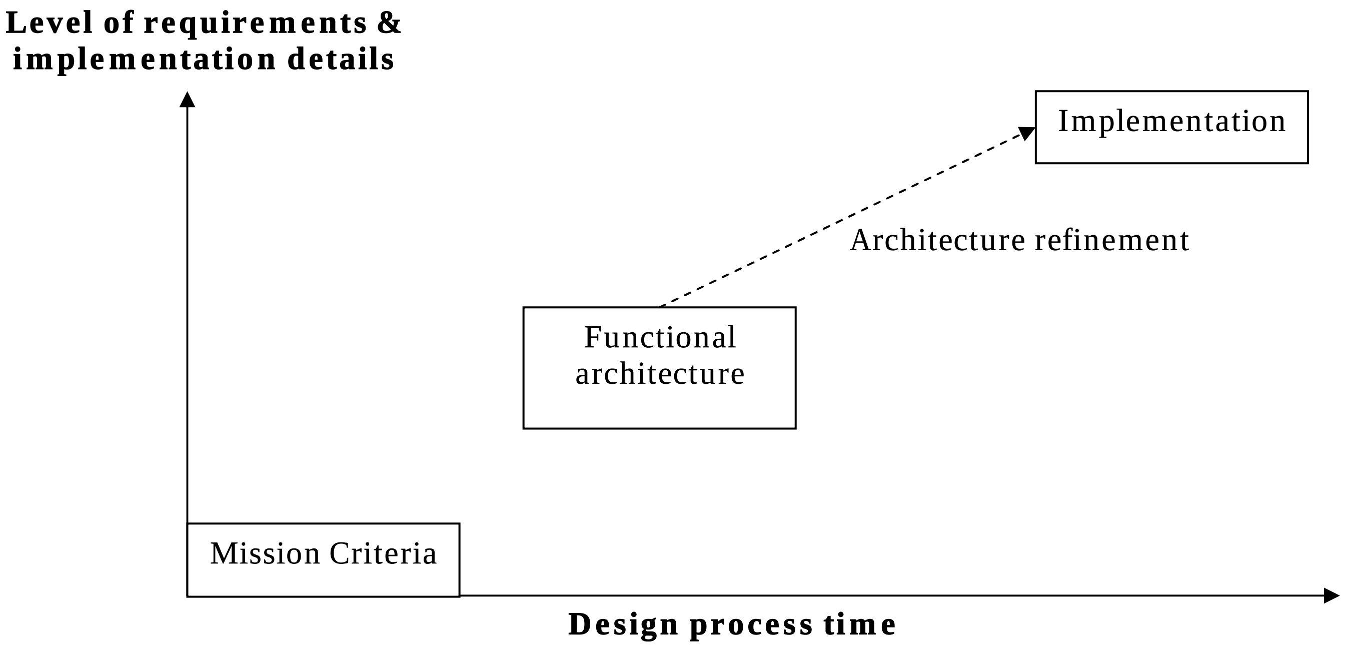


Figure 2 - Architecture details through design process

2.1 Mission criteria

Mission criteria represent requirements to be met by the mission. These requirements must be fulfil by the implementation of the system. There is a partial list of potential mission criteria:

Criteria name	Description	Value/unit
Orbit	Where the mission will take place. Depending on this value, several characteristics must be considered by validation tools (radiation level, etc.).	Ground, LEO, GEO, GTO, DeepSpace, HighRad

Duration	Duration of the mission. This time corresponds to the total mission time and potential deorbitation.	Time
Independence of payload	Specify if payload equipment depends on the central control unit (the OBC) or if each payload is independent.	<Boolean>
Costs	Estimation of costs related to the system.	Low/Medium/High
Interruption time	Maximum interruption time that can be tolerated when the system is operating.	Time
Max units	Maximum of units that can be collocated in the system. This requirements typically represents the maximum physical nodes connected within system implementation.	<integer>
Fault detection	Ability to detect faults and reconfigure the system while in operation.	<Boolean>
Memory need for data	Required size of memory to store system data (incoming values from payload, etc.).	<integer> bytes
Mass	Maximum mass of the system.	<integer> g
Risks	Industrial risks for the design and implementation of the system.	Low/Medium/High
Power	Power consumption of the system.	W

2.2 Functions requirements

Functions requirements represents request or expected performance criteria. There is a potential (and still incomplete) of a list of such requirements:

Requirements	**Description**	**Value/unit**
Mass	Maximum weight of the system.	<integer> g
Power	Power consumption allocated to this function	<integer> W
Memory	Memory request for the function.	<integer> bytes
Costs	Estimation of costs related to the system.	Low/Medium/High
Risks	Industrial risks for the design and implementation of the system.	Low/Medium/High

2.2.1 Buses and buses connections requirements

Even if buses between functions don't model real buses, requirements and constraints could be associated with these architecture elements. We propose the following list of potential criteria to be associated with functional buses:

Requirements	**Description**	**Value/unit**

Buses requirements		
Bandwidth	Requested bandwidth of the bus.	<integer> bytes per s
Latency	Requested latency of this bus	Time
Requirements of buses connections		
Throughput	Amount of data to be sent or received by a bus connection.	<integer> bytes per s
Bus Pattern	Specify the expected bus kind for this connection (point to point, etc ...)	PtP, MultiPoint

2.3 Generic building blocks requirements

Generic building blocks requirements are composed of the functions requirements and other requirements dedicated to the implementation. These requirements are most of the time specific to a given equipment. For example, for a memory, the generic building block will redefine the functional requirement (mass, power consumption, etc.) but also describe the amount of data it provides (number of chip, amount of memory on each chip, etc.).

So, generic building blocks requirements should be seen as:

Requirements of functions + Requirements dedicated to building blocks

By doing so, the requirements of the generic building blocks can also be validated against the one from the function: does the implementation fulfil its associated function requirements? For example, the implementation can be heavier than the expected weight specified at the functional step.

2.4 System requirements and mission criteria modelling

System requirements are described in the AADL root component, a `system` component.

This component defines several AADL properties which specify mission criteria to be met. AADL already provides a set of predefined properties (called the standard properties) that are ready-to-use for system designers. New property sets can also be added by AADL users, extending the definition of the potential requirements of each component.

For that purpose, we introduce our own properties for the definition of mission criteria.

2.5 Functional blocks modelling

Functional blocks are specified using an AADL `system` or `abstract` component. These components are declared as sub-component of the AADL root system that represents the main mission component.

Only these components can be used: no specialized software/hardware component can be introduced at that point. As for mission criteria, these components describe their requirements using standards or user-defined AADL properties. To model required communication between functions, AADL features are added to these components. However, at that point, as no decision is taken on the kind of bus to be used, AADL `bus` or `data` access of these components **must reference generic buses**.

2.5.1 Modelling functional buses

As we pointed out in the previous sections, functions communicate through bus. But at that point of the design process, system designer does not know the kind of bus to be used in its architecture (SpaceWire, 1553, CAN, etc.).
On the one hand, at the functional level, designers want to specify potential interactions between functions without describing all requirements. On the other hand some requirements criteria could be specified at this level, even if all implementation concerns are not known (such as the required bandwidth, etc.).
For this reason, only access to generic components should be added to the AADL components that model system functions. Then, when requirements have to be specified, they must be associated:

1. **On the instance of the generic bus**
2. **On the feature of the AADL** `system/abstract` **component that represents the function.**

2.6 Generic building blocks for the modelling of system implementation

Generic building blocks are represented into an AADL component with a specialized type (such as device, processor, process, memory, etc.). It could also be specified using a system component with several sub-components. In that case, the function is implemented using several other components organized within a hierarchy.
The component (or collection of components if the implementation component is a system) indicates the nature of the implementation (hardware or software) and specifies all its requirements. This specialized component refines its corresponding functional component (abstract or system) and also redefines the buses connections to be used: the implementation component expresses its buses requirements by referencing real bus, not generic ones.

2.7 First example

We illustrate the modelling patterns with a small example that is a simple system with one processor, one memory and one software component that do some computations.

2.7.1 System requirements

The system must consume at most 300W and have 1Gbytes of memory. The corresponding AADL component that models such a system would be:

```
system firstexample
properties
   Memory    => 1Gbytes;
   Mass      => 1 Kg;
   Power     => 30W;
end firstexample;
```

2.7.2 System functions

The system is composed of three functions:

1. **Memory**
2. **Processing unit**
3. Software that perform various computations (such as control algorithms in AOCS)

The memory is then represented with an abstract component and the requirements that must be fulfilled by the implementation.

```
abstract memory
features
   busaccess : requires bus access genericbus;
properties
   Memory    => 1 Gbytes;
   Mass      => 500g;
end memory;
```

The processing unit is also specified using an abstract component with its properties.

```
abstract processing_unit
features
   busaccess : requires bus access genericbus;
properties
   Computation_Capacity  => 100 MIPS;
   Mass                  => 400Kg;
end processing_unit;
```

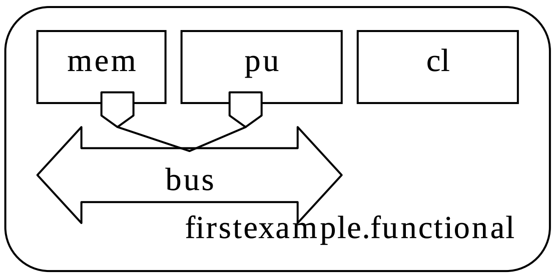
Finally, the control loop component represents the function that does some computations.

```
abstract control_loop
properties
   Computation_Requirement => 60 MIPS;
end control_loop;
```

Readers can also notice that we specify the required connection points for each component: in this example, the processing unit as well as the memory must have an access point to a bus. At this step (functional view), this connection is specified using a “generic bus”, which means it has to be connected but makes no assumption about the implementation of the bus.
Then, the root system is redefined by adding these functional components. We also add a connections section to this AADL component to represent functions inter-connections with the functional bus.

```
system implementation firstexample.functional
subcomponents
  pu :  abstract processing_unit;
  cl :  abstract control_loop;
  mem:  abstract memory;
  thebus:  bus genericbus;
connections
  bus access thebus -> pu.busaccess;
  bus access thebus -> mem.busaccess;
end firstexample.functional;
```

The corresponding AADL graphical notation would be like the following figure:



2.7.3 Generic building blocks

Now, functional components are redefined by replacing them in the architecture with implementation components.

The memory function is then specified using a memory component.

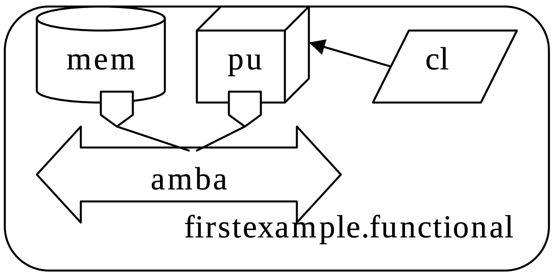
```
memory  RAM
properties
   Word_Size  => 32 bytes;
   Word_Count => 35 000 000;
   Weight     => 200 g;
end memory;
```

```
processor leon
properties
   Computation_Capacity => 100 MIPS;
   Weight               => 500 g;
end leon;
```

```
process attitude_control
properties
   Computation_Requirements => 65 MIPS;
end attitude_control;
```

```
system implementation firstexample.impl extends firstexample.functional
subcomponents
   mem : refined to memory RAM;
   pu  : refined to processor leon;
   cl  : refined to attitude_control;
   bus : refined to amba;
properties
   Actual_Processor_Binding => (reference (pu)) applies to cl;
end firstexample.impl;
```

The corresponding graphical AADL notation with specialized components is specified below:



3 EXAMPLE

In the following, we illustrate the proposed design method using a practical example. This example is a small satellite that takes pictures of a planet, processes the data and sends the result to the ground.

This simplified system shows the overall approach of the method and how mission requirements can be automatically validated using AADL models.

3.1 Mission requirements

- Duration: 1 year
- Payload storage needs: 1 Gbytes
- Max weight: 70Kg

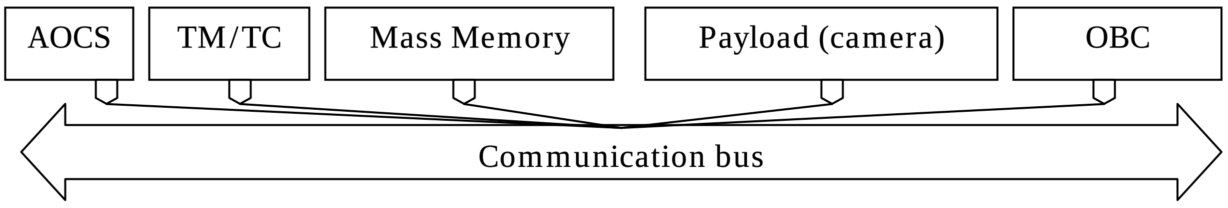
```
system fakemission
properties
   ARAM_Properties::Max_Mass              => 70 Kg;
   ARAM_Properties::Mission_Duration      => 365 day;
   ARAM_Properties                        => 1Gbyte;
end fakemission;
```

3.2 System functions

The system is composed of the following functions:

- A **Mass Memory** subsystem to store data from the payload
- A **TM/TC** subsystem which receives telecomand from earth and sends telemetry (data acquired by the payload)
- **AOCS** to control the spacecraft using sensors and actuators.
- **Payload** equipment that acquired data from digital sensors. In this mission, there is only one payload equipment: a camera that takes pictures of a planet.
- An **On-Board Computer** (OBC) that processes the data from the Payload and stores its data to the Mass Memory.

All these functions are then connected through a bus to communicate together (OBC to send its data to the TM/TC or the Mass Memory subsystems). The overall architecture is depicted in the following figure.



The following AADL textual model provides a description of the architecture with its function. Here, the word **implementation** means that the fakemission component is extended with its sub-components: it does not indicate any implementation guidance. So, this model adds system functions with abstract components.
Required connections to a bus are also illustrated using the connections section of the component: sub-components are connected to a generic bus. It means that these components will communicate together using a bus. At that point, the properties and requirements of the bus are not specified: it will be done during architecture refinement with the specification of the bus implementation.

```
system implementation fakemission.functional
subcomponents
   aocs       : abstract faocs;
   tmtc       : abstract ftmtc;
   mm         : abstract fmm
      {ARAM_Properties::Minimum_Size => 100 GByte;};
   payload    : abstract fpayload
      {ARAM_Properties::Required_Bandwidth => 20 KBytesps;};
   obc        : abstract fobc;
   thebus     : bus genericbus
      {ARAM_Properties::Bandwidth => 1000 KBytesps;
       ARAM_Properties::Bus_Type  => mtp};
connections
   bus access thebus -> aocs.busacccess;
   bus access thebus -> tmtc.busacccess;
   bus access thebus -> mm.busacccess;
   bus access thebus -> payload.busacccess;
   bus access thebus -> obc.busacccess;
end fakemission.functional;
```

We don't include the specification of the different AADL abstract components here because they are globally identical. The differences would be in the definitions of their associated AADL properties. In addition, as all AADL components are connected to the same functional bus, the generic AADL abstract component would look like this:

```
abstract functionname
features
  busaccess : requires bus access genericbus;
properties
  --  Properties of the function
end functionname;
```

3.3 Refinement to a concrete implementation

Now, we define several AADL components using specialized types. Each of them represents the concrete implementation of one system function, refining its properties, providing its implementation kind (hardware/software) and indicating the type of bus to be used (SpaceWire, 1553, etc.).

In terms of buses, our system implementation will interconnect its functions using a 1553 bus.

3.3.1 *Mass Memory*

The implementation of the Mass Memory is represented using an AADL memory component that requires an access to a 1553 bus. It means that this function is implemented with a hardware component directly connected to the bus. AADL properties of the component describe its constraints (such as the provided memory size).

```
memory MM
features
   busaccess : requires bus access bus1553;
properties
    ARAM_Properties::Mass => 2Kg;
    Word_Size             => 32 bytes;
    Word_Count            => 35 000 000;
end MM;
```

3.3.2 *TM/TC subsystem*

The TM/TC (Telemetry and Telecommand communication) function is implemented using a dedicated hardware device connected to the 1553 bus. This piece of hardware communicates directly to the ground and exchange with the On-Board Computer using the 1553 bus.

```
device tmtc
features
   busaccess : requires bus access bus1553
      {ARAM_Properties::Required_Bandwidth => 500 Bytesps;
       ARAM_Properties::Expected_Latency   => 100 ms .. 200 ms;};
properties
    ARAM_Properties::Mass => 400 g;
end tmtc;
```

3.3.3 *AOCS*

AOCS function gathers sensors and actuators. It collects received values from sensors and sends command to actuators to control the spacecraft. We divide this function into two components, each of them represented using AADL device components. Then, a global AADL system component assembles these devices and connects them to the 1553 bus.

```
device aocs_sensors
features
   busaccess : requires bus access bus1553
      {ARAM_Properties::Required_Bandwidth => 4000 Bytesps;};
properties
   ARAM_Properties::Mass => 2 Kg;
end aocs_sensors;
```

```
device aocs_actuators
features
   busaccess : requires bus access bus1553
      {ARAM_Properties::Required_Bandwidth => 1000 Bytesps;};
properties
    ARAM_Properties::Mass => 6 Kg;
end aocs_actuators;


system aocs
features
   busaccess : requires bus access bus1553
      {ARAM_Properties::Required_Bandwidth => 5000 Bytesps;};
properties
    ARAM_Properties::Mass => 10 Kg;
end aocs;

system implementation aocs.i
subcomponents
   actuators : device aocs_actuators;
   sensors   : device aocs_sensors;
connections
   busaccess -> actuators.busaccess;
   busaccess -> sensors.busaccess;
end aocs.i;
```

3.3.4 Payload subsystem

The payload function can host various devices which collect data to be processed by the on-board computer. In our case, the payload is composed of one low-resolution camera that collects one image and stores it internally. Later, the On-Board Computer (OBC) retrieves and processes it (and potentially sends it to the memory or directly to the ground).

We model the camera using an AADL `device` component. The expected bus throughput is associated with the `bus access` feature, indicating that the device will output 100 Kbits per second. The maximum amount of mass memory used during the mission is evaluated to 10 Mbytes.

Finally, the device is added to a global AADL `system` component that contains all payload equipments. This intermediate system component is used to ease the integration of other components: if designers want to add a new payload equipment, he has to add a new sub-component to this system component without modifying the other parts of the AADL model.

```
system payload
features
   busaccess : requires bus access bus1553
      {ARAM_Properties::Required_Bandwidth => 20 KBytesps;
       ARAM_Properties::Expected_Latency   => 100 ms .. 200 ms;};
properties
    Weight => 2Kg;
end payload;
```

```
device camera
features
   busaccess : requires bus access bus1553
      {ARAM_Properties::Required_Bandwidth => 20 KBytesps;};
properties
    Weight       => 900 g;
    Memory_Needs => 10 Mbyte;
end camera;

system implementation payload.i
subcomponents
   equipment : device camera;
connections
   bus access busaccess -> equipment.busaccess;
end payload.i;
```

3.3.5 *On-Board Computer (OBC)*

The OBC function is represented using two AADL components:

1. An AADL `processor` that models the physical hardware part as well as the underlying operating system
2. An AADL `process` component that represents the software to be run on the target processor. The property `Actual_Processor_Binding` specifies the association between software process and its associated processor.

Then, these two components are gathered within a single system component that defines the actual implementation of the OBC function.

```
processor leon
features
   busaccess : requires bus access bus1553;
properties
   ARAM_Properties::MIPS            => 100;
   ARAM_Properties::Mass            => 500 g;
   ARAM_Properties::Required_Memory => 19 Mbyte;
end leon;

process control_program
properties
   Source_Code_Size => 300 Kbyte;
   Source_Data_Size => 3 Mbyte;
end control_program;

system obc
features
   busaccess : requires bus access bus1553
   {ARAM_Properties::Required_Bandwidth => 20 KBytesps;
    ARAM_Properties::Expected_Latency   => 100 ms .. 500 ms;};
```

```
properties
    ARAM_Properties::Mass => 1000 g;
end obc;

system implementation obc.i
subcomponents
   cpu : processor leon;
   prs : process control_program;
connections
   bus access cpu.busaccess -> busaccess;
properties
   ARAM_Properties::Required_Memory => 30Mbytes;
   Actual_Processor_Binding => (reference (cpu)) applies to prs;
end obc.i;
```

3.3.6 Bus implementation (1553)

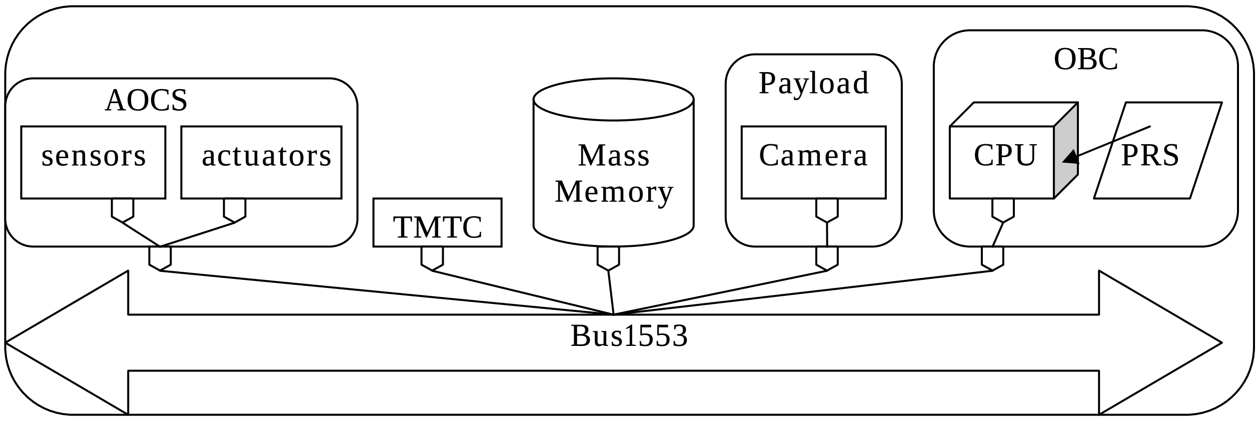
We also model the bus to be used in the system implementation. This would be a 1553 bus, which allows connection of several nodes. For this example, we also specify that this implementation of the 1553 bus has a speed of 1 Mbits per second and a worst latency time of 100 ms.

```
bus bus1553
properties
   ARAM_Properties::Bus_Type     => mtp;
   ARAM_Properties::Bandwidth    => 1 MBytesps;
   ARAM_Properties::Max_Latency  => 100 ms;
end bus1553;
```

3.4 Final system implementation

The following AADL model (graphical and textual) represents the final implementation. It consists of AADL `system` component that extends the functional one with specialized AADL sub-components. In the following, each sub-component is refined with the specialized components defined in previous sections.

```
system implementation fakemission.implementation extends fakemission.functional
subcomponents
   aocs        : refined to system aocs;
   tmtc        : refined to device tmtc;
   mm          : refined to memory mm;
   payload     : refined to system payload;
   obc         : refined to system obc;
   thebus      : refined to bus1553;
end fakemission.implementation;
```


AOCS
sensors
actuators
TMTC
Mass Memory
Payload
Camera
OBC
CPU
PRS
Bus1553

4 VALIDATION OF THE ARCHITECTURE

Once our architecture is refined, it includes all components specificities and requirements so that we can use a validation tool that will process the model and check for requirements enforcement. This section provides some ideas about model validation and what the tool would check in the component hierarchy. In particular, we detail the three different validation steps that involve every aspects of the design process (mission criteria definition, functional architecture and system implementation). In addition, we also perform an additional validation step of the functional and implementation architectures, without considering the other representations of the architecture: this is more a consistency validation of the model.

4.1 Mission criteria vs. Implementation validation

First, we can validate mission criteria fulfilment against system implementation. In our example, we can validate two aspects:

1. **Mass of the implementation is less than the expected Mass**. This can be done by validating than the sum of the mass of all AADL sub-components of the root component of the implementation (`fakemission.impl`) is less than the property Max_Mass of the mission criteria system (`fakemission`). This requirement is enforced, mission criteria requires that the mass is less than 70Kg while the sum of the mass of all sub-components is equal to 15.4 Kg.
2. **Size of the Mass Memories is greater or equal to the size of the size required by the system**. This is done by validating that the sum of all memory sub-components of the root system implementation (`fakemission.impl`) is greater than the property `Required_Memory` defined on the mission criteria system (`fakemission`). Note that the size of a memory components is calculated using the `Word_Size` and `Word_Count` properties. In our case, the system requires 1 Gbytes and the Mass Memory has a `Word_Size` property of 32 bytes and a `Word_Count` property of 35 000 000. So, the requirement if enforced (32 bytes * 35000000 = 1120 000 000 bytes).

4.2 Mission criteria vs. Functional architecture validation

Mission criteria fulfilment is also checked against the functional architecture. **We validate that the memory size required by the mission is provided by the functional architecture**. This can be done by checking that the value of the `Required_Memory` property of the mission criteria system (`fakemission`) is lesser or equal to the sum of the `Minimum_Size` of all AADL sub-components of the functional system (`fakemission.functional`). In our case, the value is equal : mission criteria requires 1 Gbyte while the only AADL functional sub-component that specifies a memory defines a minimum size of 1 Gbyte.

4.3 Functional architecture vs. Implementation validation

Finally, we can check system implementation and its refinement into an implementation. The following aspects are validated:

1. **Size of memory components**: the tool can check that implementation of memory components provide a memory size that is greater or equal to its functional definition. In our case, this is done be verifying that the `Minimum_Size` property of the `mm` component (1 Gbyte) is less or equal than the value `Word_Size` * `Word_Count` (32 bytes * 35000000 = 1120 000 000 bytes) of the `mass_memory` component.
2. **Bus latency**: model validation tool checks that bus latency is less than the latency expected by each component connected to it. This is done by inspecting every component that has an access to the bus and checking that the `Latency` property of the bus is less or equal to the lower value of the `Expected_Latency` of each component that access to the bus. For example, the `tmtc` component expects latency between 100ms and 200ms. As the bus (`bus1553`) provides a latency of 100 ms, this requirement is enforced (but this validation has to be done on every component that accesses the instance of the `bus1553` component).
3. **Bus bandwidth**: validation tool checks that bus capacity is greater or equal to the bandwidth required by the components that access it. As for the latency, this is done by inspecting every component that has an access to the bus and checking that the `Required_Bandwidth` property of the accessing component is less or equal to the Bandwidth property associated to the bus. For example, the `aocs` sub-system specifies a `Required_Bandwidth` property of 5000 Bytesps. As the bus provides a bandwidth of 1 Mbyteps, this requirement is enforced (but is has to be done for every component that accesses the bus).

4.4 Functional architecture consistency validation

Some aspects of the functional architecture can be validated without considering mission criteria or implementation concerns. Validation tools can verified the following aspects on our example:

1. **Bus bandwidth**: the Bandwidth of the bus is sufficient regarding the requirement of the components that access it. This is done by verifying that the `Bandwidth` property of the bus (component `genericbus`) is greater or equal to the expected bandwidth bus access connected to the bus. For example, in our functional view, the `payload` subsystem requires 100 KBytesps and the bus provides a bandwidth of 1000 KBytesps so that the requirement is enforced.
2. **The bus type**: if the bus is a Multipoint one (value of the property `Bus_Type set` to `mtp`), several components can share an access to it. On the other hand, if the value of the `Bus_Type` property is `ptp` (this is the case for SpaceWire buses), only two components can share the bus. In our example, the bus is a multipoint one and several components have an access to it.

4.5 Implementation architecture consistency validation

Finally, validation tools can also verified some consistency on the implementation model without considering other system representations. On our example, the following aspects can be checked:

1. **Bus type**. The bus that connects components is adequate. If the bus connects more than two components, the `Bus_Type` property must have the value `mtp`. If the bus connects only two components, it can have a `Bus_Type` property set to `ptp` (Point to Point) or mtp (Multi-Point).
2. **Bus latency**
   a. Latency of the bus (the `Max_Latency` property associated to the bus component) must be less or equal than the lower value of the `Expected_Latency` property associated to every component that accesses it. In our example, the bus component has a property `Max_Latency` with a value of 100 ms while the components that access it requires a latency that it at least 100 ms. So, this requirement is enforced (and has to be processed on every component that accesses to the bus).
   b. In a system that contains several sub-systems, the expected latency of the sub-system is less or equal than the latency expected by each of its sub-components.
3. **Bus bandwidth**
   a. Bandwidth of the bus (the `Bandwidth` property associated to the bus component) must be less than or equal to the `Required_Bandwidth` of every component that accesses it. In our example, the bus component (`bus1553`) has a property `Bandwidth` with a value of `1 MBytesps` while the components that access it require at least `200 KBytesps`. So, this requirement is enforced (and has to be processed on every component that accesses to the bus).
   b. In a system that contains several sub-components, the required bandwidth of the sub-system is greater or equal than the sum of the bandwidth required by all sub-components. For example, in our example, the `aocs` sub-system requires a bandwidth of `200 Bytesps` and each of each sub-component requires a bandwidth of `100 Bytesps`. So, this requirement is enforced but has to be processed on every sub-component with the system `type` contained in the root component.

5 MODELLING WITH AADL: CURRENT ISSUES

This section presents some issues users can encounter when using AADL to model their architecture, especially during the refinement process. It also provides a general view of potential issues and ongoing work to perform for the establishment of a modelling platform.

5.1 Level of refinement

When refining an AADL `abstract` component into a specialized one, the standard is not really clear about the `feature` refinement and how `features` are refined. In particular, in our context, we want to change the bus type associated to a `bus access feature`. For example, we have an abstract component with a bus access to a `genericbus` component.

```
abstract mycomponent
features
  ba : requires bus access genericbus;
end mycomponent;
```

This component is then used in a `system implementation`, for example:

```
system implementation example.i
subcomponents
  s : abstract mycomponent;
end example.i;
```

Then, we define a new component with a concrete type

```
processor cpu
features
  ba : requires bus access implementationbus;
end cpu;
```

And finally, we define a new system implementation that extends the first one and refines its sub-component.

```
system implementation example.i2 extends example.i
subcomponents
   S : refined to processor cpu;
end example.i;
```

The following example leads to an error, because the concrete component `cpu` redefines feature `ba` of the `mycomponent` component.

5.1.1 *First proposed workaround: concrete component extending the abstract one*

One solution would be to model the cpu component as an extension of mycomponent and refine the ba feature. Then, the cpu component would be declared like this:

```
processor cpu extends mycomponent
features
  ba : refined to requires bus access implementationbus;
end cpu;
```

When using such a construction, current AADLv2 parser (such as Ocarina) reports an error.

5.1.2 *Second proposed workaround: features refinement in instances*

Another workaround would be to allow the redefinition of feature in the instance tree. In that case, we could refine the feature definition of the cpu component in the declaration of the example.i2 component. The declaration would also be the following:

```
system implementation example.i2 extends example.i
subcomponents
   S : refined to processor cpu
                {ba refined to requires bus access implementationbus;};
end example.i;
```

Such a workaround would introduce a modelling flexibility and such, would ease components reuse.

5.2 Prototypes

The prototype concept introduced by the version 2 of AADL seems quite promising for our approach. In particular, when the differences between two components is not significant, we could model a prototype and instantiate them later to add their specificities.

For example, one can consider a generic execution platform, called cpu. When designing the component for the first time, the designer does not know the kind of bus connected to this execution platform. Then, he just adds a prototype to specify that the component requires a bus access, without specifying it. Later, he can specify new components that extend it and defines the type of bus actually used.

For example, we specify the following cpu component:

```
processor cpu
prototypes
   Bus_type : requires bus access;
features
```

```
   ba : requires bus access bus_type;
end cpu;
```

Later, the designer can specify an execution platform that requires an access to a SpaceWire bus (component `cpu.i`):

```
processor cpu.spw extends cpu.i
         (bus_type => requires bus access spacewire)
end cpu.spw;
```

However, the definition and the use of components prototypes are not clear in the standard. Especially, no example illustrates the use of prototype on bus access, even if the standard indicates that it is legal from a syntax and semantic point of view.

5.3 Tool support for validation

5.3.1 Requirements Enforcement and Analysis Language (REAL)

The Requirements Enforcement and Analysis Language (REAL) provides a formalism to validate AADL models by analyzing the components hierarchy. It processes the AADL components tree, inspects their content (properties, sub-components, features, etc.) and checks them against so-called *theorems*. A REAL theorem is a piece of code associated to a component that makes more explicit the requirements to be validated on a given component.

For example, a REAL theorem can specify that the amount of memory provided by an AADL `memory` component must be greater than the memory required by its associated process components. Another example: a REAL theorem can specify that a SpaceWire bus (that is a point-to-point bus) cannot connect more than two. For that, it will inspect SpaceWire buses and check if more than two components are connected to this bus.

REAL provides an efficient way to analyze and validate AADL models. Especially in our context, the validation of the system could be achieved using predefined REAL theorems that checks for power consumption, memory capacity, etc.

At the end, the tool has one limitation: in case of the use of typed properties (such as memory, computing capacity, etc.), the tool does not consider units and only analyzes property values. This can be a major issue, especially where unit concerns can lead to real bugs. However, this can be solved by improving the tool and describing how to validate each property with respect to its units.

5.3.2 Dedicated annex for requirements enforcement

To standardize the way models are validated and cope with issues identified in the REAL tool, the AADL committee is currently investigating the potential of an annex for requirements enforcement.

This new annex to the language would describe how to validate AADL models. Such a language would be processed by appropriate tools to verify that a model enforces a given set of rules.

5.4 Tool for AADL specification (edition and analysis of models)

5.4.1 OSATE2

OSATE is the reference tool for manipulating and editing AADL models. OSATE2 is the latest version with the ability to process AADLv2 models. However, at this time, no stable version of this tool was released. This tool could be interesting for the edition of AADL models. However, it does not provide any graphical edition of AADL models, which is a major requirement for a modelling framework.

5.4.2 Ocarina

Ocarina is an AADL tool that provides much functionality such as model analysis, code generation, and model validation. This is a command-line based tool and only processes textual AADL models. However, for this moment, this is the first tool with the ability to process AADLv2 models.

5.4.3 AADLv2 support

The current support of AADLv2 in commercial or open-source tool is still in progress. In the two reference tools (OSATE and Ocarina), only Ocarina is able to process AADLv2 models. However, some legality rules are still not implemented which makes the support of AADLv2 still incomplete. As a matter of fact, AADLv2 support needs to be improved but incoming activities should address these issues.

5.4.4 Graphical Editor

At that time, there is no graphical editor for AADL, either for version 1 or version 2. Some projects have been started but didn’t reach the quality level to be used in projects. Consequently, the design and the implementation of a graphical editor for AADL are still required and would be a planned activity of the ARAM project.

6 AADL DEFINITION OF ARAM DEDICATED PROPERTIES

```
property set ARAM_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);

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);

Power_Provide: Power applies to (device);

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: 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);
```

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

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

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

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

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);

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

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

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

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

end ARAM_Properties;
```

7 REFERENCES

1. *"A Practive 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