

# **Amalthea HW Model – An overview and component description for the new Amalthea HW model v0.9.0**

Bosch Corporate Research  
April 26, 2018  
v0.9



# Contents

<b>1</b>	<b>Structural Modeling of Heterogeneous Platforms</b>	<b>4</b>
<b>2</b>	<b>Recipe and Feature concept: An outlook of upcoming meta-model changes</b>	<b>6</b>
<b>3</b>	<b>General Hardware Model Overview</b>	<b>9</b>
<b>4</b>	<b>Current implementation with features and the connection to the SW Model</b>	<b>12</b>
<b>5</b>	<b>Interpretation of latencies in the model</b>	<b>14</b>
<b>6</b>	<b>Element description</b>	<b>16</b>
6.1	HwModel . . . . .	16
6.2	HwStructure . . . . .	16
6.3	FrequencyDomain . . . . .	18
6.4	PowerDomain . . . . .	19
6.5	ProcessingUnit . . . . .	19
6.6	Memory . . . . .	20
6.7	Cache . . . . .	21
6.8	ConnectionHandler . . . . .	21
6.9	HwAccessElement . . . . .	22
6.10	HwFeatureCategory . . . . .	23
6.11	HwFeature . . . . .	24
6.12	HwPort . . . . .	25
6.13	HwConnection . . . . .	26
6.14	HwAccessPath . . . . .	27
6.15	ProcessingUnitDefinition . . . . .	29
6.16	MemoryDefinition . . . . .	29
6.17	CacheDefinition . . . . .	29
6.18	ConnectionHandlerDefinition . . . . .	30
6.19	LatencyConstant . . . . .	31
6.20	LatencyDeviation . . . . .	31
<b>7</b>	<b>Enums</b>	<b>33</b>

# 1 Structural Modeling of Heterogeneous Platforms

To master the rising demands of performance and power efficiency, hardware becomes more and more diverse with a wide spectrum of different cores and hardware accelerators. On the computation front, there is an emergence of specialized processing units that are designed to boost a specific kind of algorithm, like a cryptographic algorithm, or a specific math operation like "multiply and accumulate". As one result, the benefit of a given function from hardware units specialized in different kinds may lead to nonlinear effects between processing units in terms of execution performance of the algorithm: while one function may be processed twice as fast when changing the processing unit, another function may have no benefit at all from the same change. Furthermore the memory hierarchy in modern embedded microprocessor architectures becomes more complex due to multiple levels of caches, cache coherency support, and the extended use of DRAM. In addition to crossbars, modern SoCs connect different clusters of potentially different hardware components via a Network on Chip. Additionally, power and frequency scaling is supported by state of the art SoCs. All these characteristics of modern and performant SoCs (specialized processing units, complex memory hierarchy, network like interconnects and power and frequency scaling) were only partially supported by the former Amalthea hardware model. Therefore, to create models of modern heterogeneous systems, new concepts of representing hardware components in a flexible and easy way are necessary: Our approach supports modeling of manifold hierarchical structures and also domains for power and frequencies. Furthermore, explicit cache modules are available and the possibilities for modeling the whole memory subsystem are extended, the connection between hardware components can be modeled over different abstraction layers. Only with such an extended modeling approach, a more accurate estimation of the system performance of state of the art SoCs becomes feasible.

Our intention is allowing to create a hardware model once at the beginning of a development process. Ideally, the hardware model will be provided by the vendor. All performance relevant attributes regarding the different features of hardware components like a floating point unit or how hardware components are interconnected should be explicitly represented in the model. The main challenge for a hardware/software performance model is then to determine certain costs, e.g. the net execution time of a software functionality mapped to a processing unit. Costs such as execution time, in contrast to the hardware structure, may change during development time - either because the implementation details evolve from initial guesstimating to real-world measurements, the

implementation is changed, or the tooling is changed. Therefore, the inherent attributes of the hardware, e.g. latency of an access path, should be decoupled from the mapping or implementation dependent costs of executing functions. We know from experience that it is necessary to refine these costs constantly in the development process to increase accuracy of performance estimation. Refinement denotes incorporation of increasing knowledge about the system. Therefore, such a refinement should be possible in an efficient way and also support re-use of the hardware model. The corresponding concepts are detailed in the following section.

## 2 Recipe and Feature concept: An outlook of upcoming meta-model changes

*Disclaimer: Please note that the following describes work in progress - what we call "recipes" later is not yet part of the meta-model, and the concept of "features" is not final.*

The main driver of the concept described here is separation of implementation dependent details from structural or somehow "solid" information about a hardware/software system. This follows the separation of concerns paradigm, mainly to reduce refinement effort, and foster model re-use: As knowledge about a system grows during development, e.g. by implementing or optimizing functionality as software, the system model should be updated and refined efficiently, while inherent details shall be kept constant and not modified depending on the implementation.

An example should clarify this approach: For timing simulation, we require the net execution time of a software function executed on the processing unit it is mapped onto. This cost of the execution depends on the implementation of the algorithm, for instance, as C++ code, and the tool chain producing the binary code that eventually is executed. In that sense, the **execution needs** of the algorithm (for instance, a certain number of "multiply and accumulate" operations for a matrix operation) are naturally fixed, as well as the **features** provided by the processing unit (for instance, a dedicated MAC unit requiring one tick for one operation, and a generic integer unit requiring 0.5 ticks per operation). However is implementation- and tool-chain-dependent how the actual execution needs of the algorithm are served by the execution units. Without changing the algorithm or the hardware, optimization of the implementation may make better use of the hardware, resulting in reduced execution time. The above naturally draws the lines for our modelling approach: Execution needs (on an algorithmic level) are inherent, as well as features of the hardware. Keeping these information constant in the model is the key for re-use; implementation dependent change of costs, such as lower execution time by an optimized implementation in C++ or better compiler options, change during development and are modelled as **recipe**. A "recipe" thus takes execution needs of software and features of the hardware as input and results in costs, such as the net execution time. Consequently, recipes are the main area of model refinement during development. The concept is illustrated in figure 2.1;

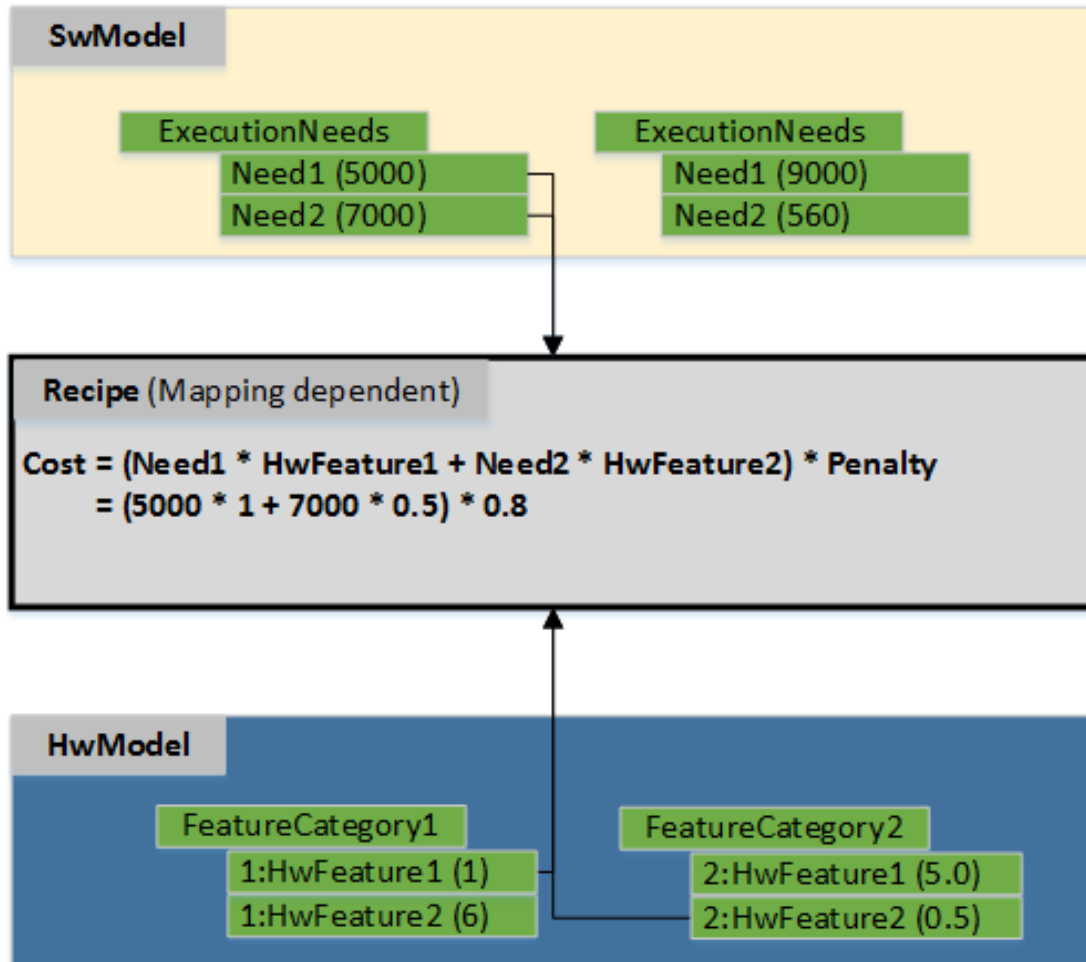


Figure 2.1: Recipe concept

Note that flexibility is part of the design of this approach. Execution needs and features are not limited to a given set, and recipes can be almost arbitrary prescripts of computation. This allows to introduce new execution needs when required to favorably detail an algorithm. For instance, the execution need "convolution-VGG16" can be introduced to model a specific need for a deep learning algorithm. The feature "MAC" of the executing processing unit provides costs in ticks corresponding to perform a MAC operation. The recipe valid for the mapping then uses these two attributes to compute the net execution time of "convolution-VGG16" in ticks, for instance, by multiplying the costs of xyz MAC operations with a penalty factor of 0.8. Note that with this approach execution needs may be translated very differently into costs, using different features.

To further motivate this approach, we give some more benefits and examples of beneficial use of the model:

- Given execution needs of a software function that directly correspond the features of processing units, the optimal execution time may be computed (peak performance).
- While net execution time is the prime example of execution needs, features, and recipes, the concept is not limited to "net execution time recipes", recipes for other performance numbers such as power consumption are possible.
- Recipes can be attached at different "levels" in the model: At a processing unit and at a mapping. If present, the recipe at mapping level has precedence.



### 3 General Hardware Model Overview

The design of the new hardware model is focusing on flexibility and variety to cover different kind of designs to cope with future extensions, and also to support different levels of abstraction. To reduce the complexity of the meta model for representing modern hardware architectures, as less elements as possible are introduced. For example, dependent of the abstraction level, a component called *ConnectionHandler* can express different kind of connection elements, e.g. a crossbar within a SoC or a CAN bus within an E/E-architecture. A simplified overview of the meta model to specify hardware as a model is shown below. The components *ConnectionHandler*, *ProcessingUnit*, *Memory* and *Cache* are referred in the following as basic components.

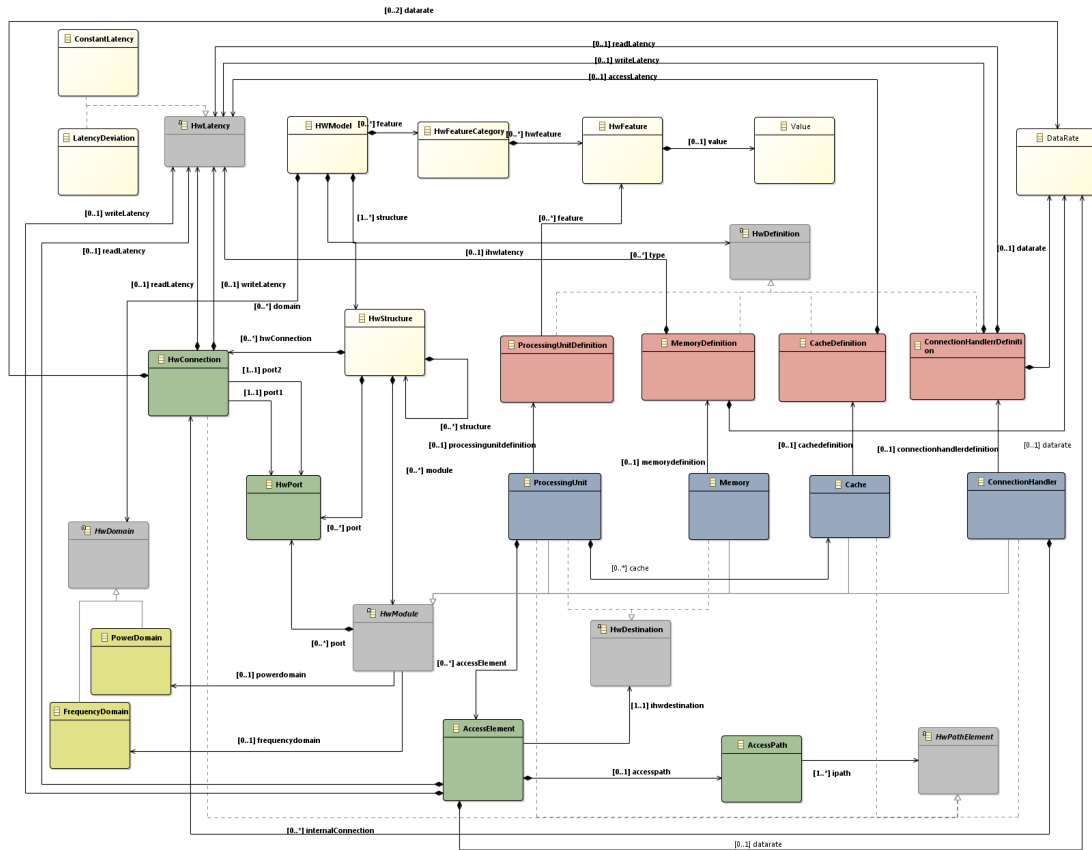


Figure 3.1: Class diagram of the hardware model

The root element of a hardware model is always the *HwModel* class that contains all domains (power and frequency), definitions, and hardware features of the different component definitions. The hierarchy within the model is represented by the *HwStructure* class, with the ability to contain further *HwStructure* elements. Therewith arbitrary levels of hierarchy could be expressed <sup>1</sup>. Red and blue classes in the figure are the definitions and the main components of a system like a memory or a core.

Figure 3.3 shows the modeling of a processor. The *ProcessingUnitDefinition*, which is created once, specifies a processing unit with general information (which can be a CPU, GPU, DSP or any kind of hardware accelerator). Using a definition that may be re-used supports quick modeling for multiple homogeneous components within a heterogeneous architecture. *ProcessingUnits* then represent the physical instances in the hardware model, referencing the *ProcessingUnitDefinition* for generic information, supplemented only with instance specific information like the *FrequencyDomain*.

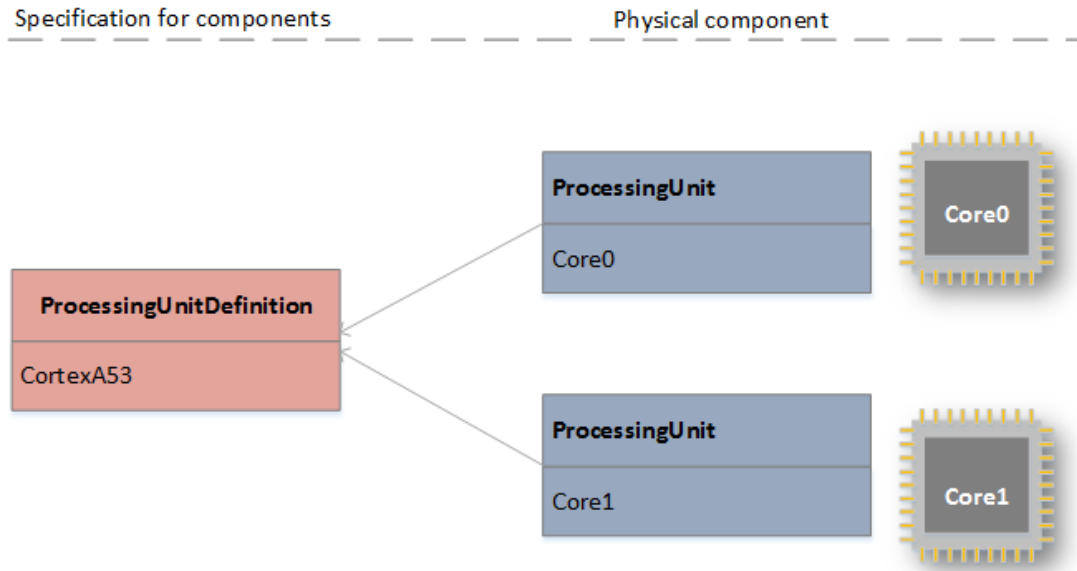


Figure 3.2: Link between definitions and module instances (physical components)

Yellow represents the power and frequency domains that are always created at the top level of the hardware model. It is possible to model different frequency or voltage values, e.g., when it is possible to set a systems into a power safe mode. All components that reference the domain are then supplied with the corresponding value of the domain.

All the green elements in the figure are related to communication (together with the blue base component *ConnectionHandler*). Green modeling elements represent ports,

<sup>1</sup> includes the "classical" hierarchy from the original Amalthea model, System → ECU → Microcontroller → Core, but also allows more flexible clustering.

static connections, and the access elements for the *ProcessingUnits*. These *ProcessingUnits* are the master modules in the hardware model. The following example shows two *ProcessingUnits* that are connected via a *ConnectionHandler* to a *Memory*. There are two different possibilities to specify the access paths for *ProcessingUnits* like it is shown for *ProcessingUnit\_2* in figure 3.3. Every time a *HwAccessElement* is necessary to assign the destination e.g. a *Memory* component. This *HwAccessElement* can contain a latency or a data rate dependent on the use case. The second possibility is to create a *HwAccessPath* within the *HwAccessElement* which describes the detailed path to the destination by referencing all the *HwConnections* and *ConnectionHandlers*. It is even possible to reference a cache component within the *HwAccessPath* to express if the access is cached or non-cached. Furthermore it is possible to set addresses for these *HwAccessPath* to represent the whole address space of a *ProcessingUnit*. A typical approach would be starting with just latency or data rates for the communication between components and enhance the model over time to by switching to the *HwAccessPaths*.

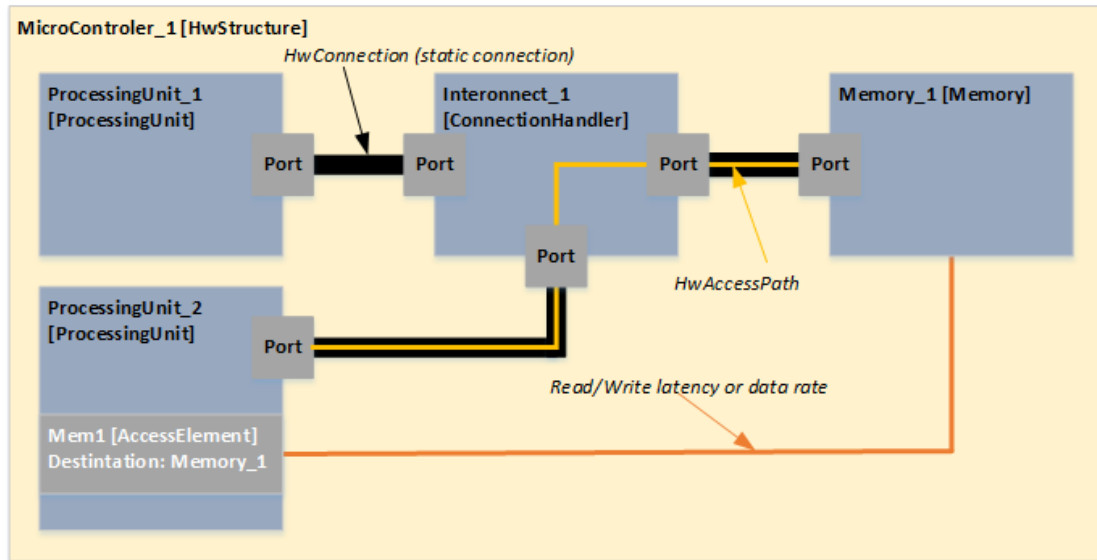


Figure 3.3: Access elements in the hardware model

## 4 Current implementation with features and the connection to the SW Model

In chapter 2 the long time goal of the feature and recipe concept is explained. As an intermediate step before introducing the recipes we decided to connect the HwModel and SwModel by referencing to the name of the hardware *FeatureCategories* from the *ExecutionNeed* element in a Runnable. The following figure 4.1 shows this connection between the grey Runnable item block and the white Features block. Due to the mapping (Task or Runnable to ProcessingUnit) the corresponding feature value can be extracted out of the ProcessingUnitDefinition.

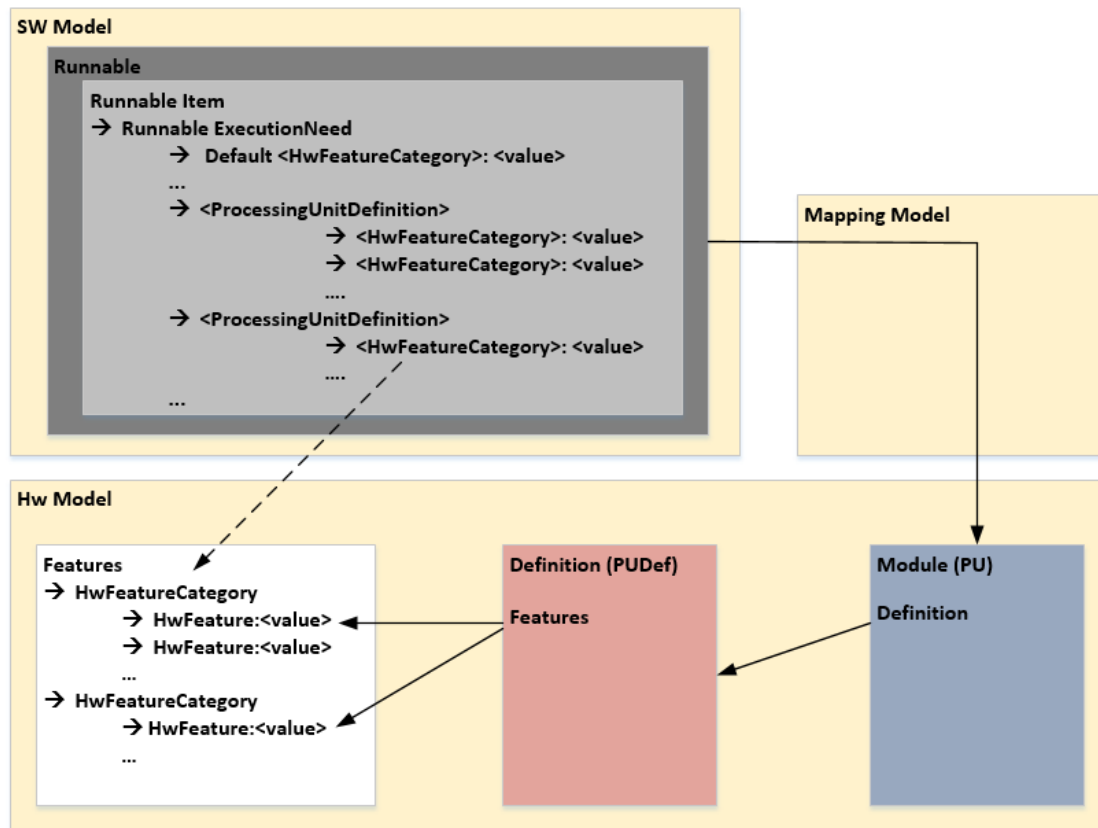


Figure 4.1: Connection between HW Features and Runnables

A concrete example based on the old hardware model is the "instruction per cycle" value (IPC). To model an IPC with the new approach a *HwFeatureCategory* is created with the name *Instructions*. Inside this category multiple IPC values can be created for different *ProcessingUnitDefinitions*.

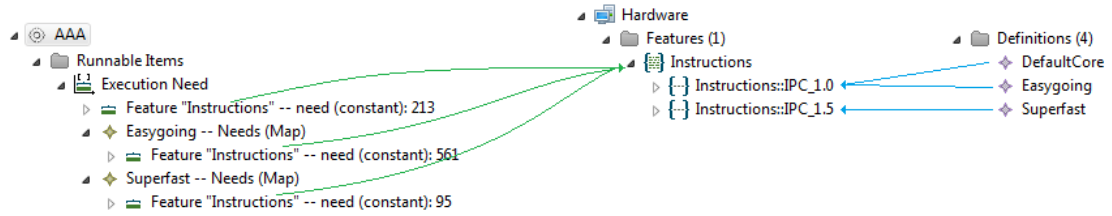


Figure 4.2: Execution needs example

## 5 Interpretation of latencies in the model

In the model read and write access latencies are used. An alternative which is usually used in specifications or by measurements are request and response latencies. Figure 5.1 shows a typical communication between two components. The interpretation of a read and write latency for example at *ConnectionHandlers* is the following:

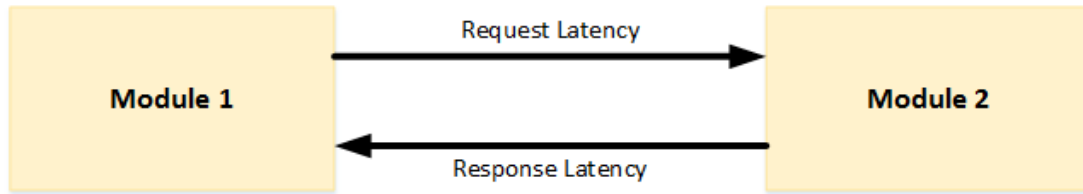


Figure 5.1: Request and response latency

$$readLatency = requestLatency + responseLatency \quad (5.1)$$

$$writeLatency = requestLatency \quad (5.2)$$

The access latency of a *Memory* component is always added to the read or write latency from the communication elements, independent if its one latency from an *HwAccessElement* or multiple latencies from a *HwAccessPath*.

As a concrete example for figure 3.3 in case using only read and write latencies:

$$TotalReadLatency = readLatency(HwAccessElement) + accessLatency(Memory) \quad (5.3)$$

$$TotalWriteLatency = writeLatency(HwAccessElement) + accessLatency(Memory) \quad (5.4)$$

As a concrete example for figure 3.3 in case using an access element with an hw access path:

$n = \text{Number of path elements}$

$$TotalReadLatency = \left( \sum_{p=1}^n readLatency(p) \right) + accessLatency(Memory) \quad (5.5)$$

$$TotalWriteLatency = \left( \sum_{p=1}^n writeLatency(p) \right) + accessLatency(Memory) \quad (5.6)$$

PathElements could be *Caches*, *ConnectionHandlers* and *HwConnections*. In very special cases also a *ProcessingUnit* can be a *PathElement* the *ProcessingUnit* has no direct effect on the latency. In case the user want to express a latency it has to be annotated as *HwFeature*.

## 6 Element description

The following tables describe the different model elements and their attributes in detail. For different elements short examples are attached.

### 6.1 HwModel

The *HwModel* class is the root element of the hardware model. It always contains one or multiple *HwStructures*, *Power-* and *FrequencyDomains* and optionally different *FeaturesCategories* for the *HwModule* definitions.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the hardware model
Structures	Containment	HwStructure	*	Hierarchical structure of the hardware model
Feature Categories	Containment	HwFeatureCategory	*	FeatureCategory for the HwModel
Domains	Containment	HwDomain	*	Frequency- and PowerDomains
Definitions	Containment	HwDefinition	*	Definitions of ProcessingUnits, Memories, Caches and ConnectionHandlers

Table 6.1: HwModel

### 6.2 HwStructure

A *HwStructure* is a hierarchical element which can contain all kind of *HwModules*, *HwConnections* and other *HwStructures*. Different *HwStructures* can be connected via one or more *HwPorts* with other structures or modules of a top level *HwStructures*. By combining different *HwStructures* any kind of hierarchal systems can be expressed. By setting the structure type attribute (e.g. Cluster, ECU) the structural level in the hardware is directly expressible in the model.



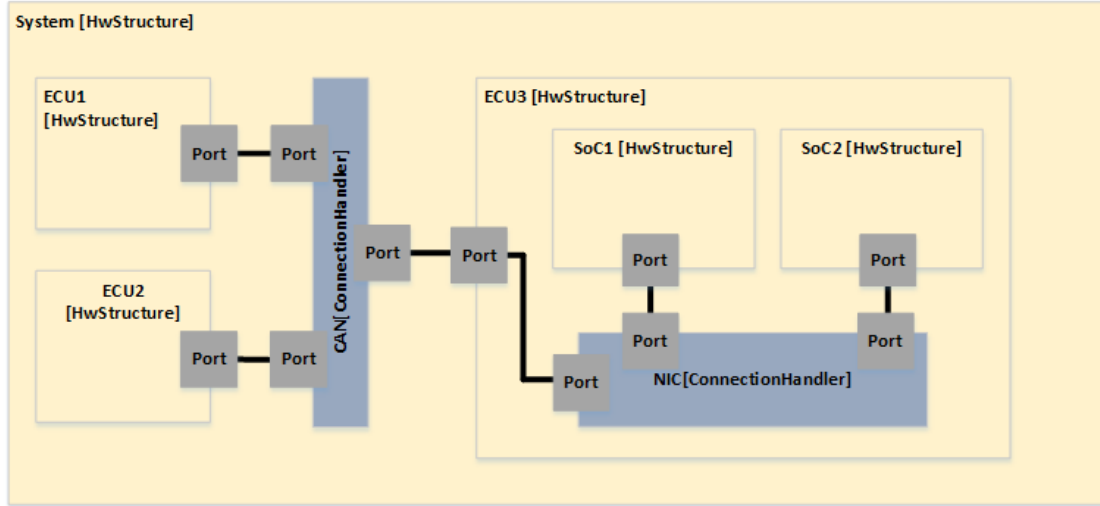


Figure 6.1: HwStructure example

Figure 6.1 shows an example for creating a hierarchy within an E/E-architecture. The *HwStructure System* (which is called "System") is created as top level structure within the HwModel. It contains three other structures which represents different ECUs. The structures are connected via *HwPorts*, *HwConnections* and a *ConnectionHandler*. Usually structures in the model can be viewed as black boxes on the top level. *ECU3* allows a look inside, where additional structures for two SoCs are visible.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the hardware structure
Structure Type	Enum	Structure-Type	1	Defines the type of the structure (e.g. ECU)
Modules	Containment	HwModule	*	Modules of the structure (e.g. Memory)
Ports	Containment	HwPort	*	Ports to connect the structure
Structures	Containment	HwStructure	*	Hardware structure to build hierarchical designs
Connections	Containment	HwConnection	*	Connections within a structure

Table 6.2: HwStructure

## 6.3 FrequencyDomain

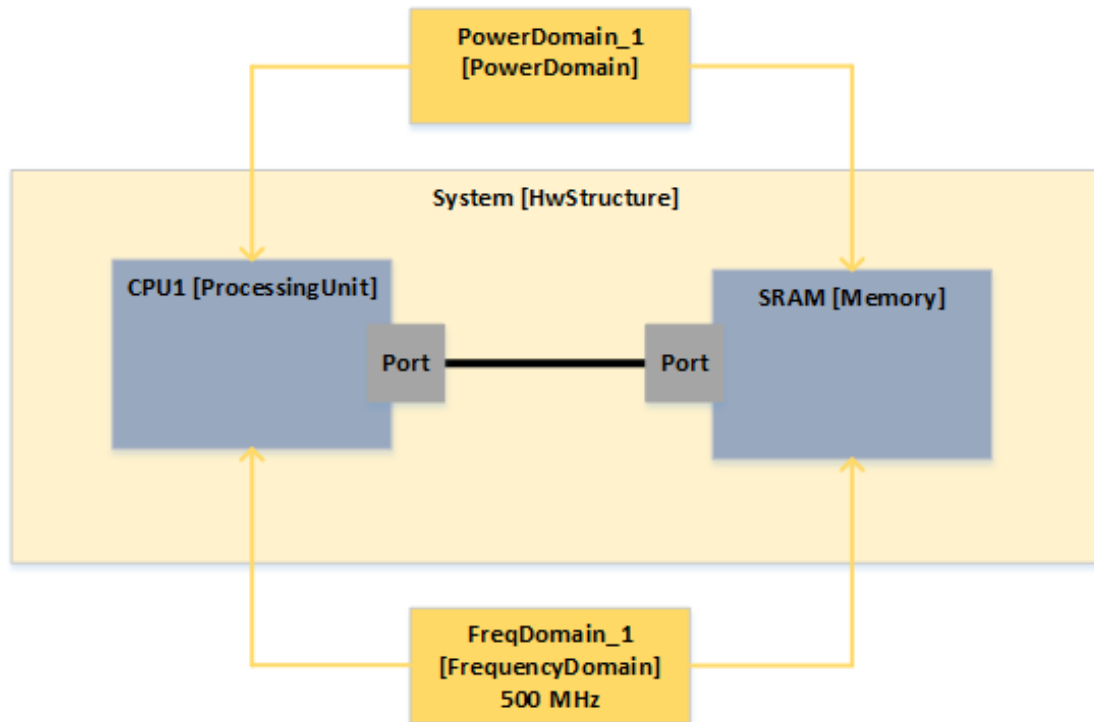


Figure 6.2: Frequency- and PowerDomain example

A *FrequencyDomain* is inherited from *HwDomain*. This element describes a frequency domain which can be referenced by all elements of the type *HwModule* to define the default frequency value for operation. In future the *FrequencyDomain* should also contain possibleValues which should specify the different frequencies for different operation modes. For this extension a general Amalthea mode concept is necessary.

Figure 6.2 shows an example for a *FrequencyDomain* and a *PowerDomain*. They are always created at the top level in the root element *HwModel*. Every basic component is able to reference a *FrequencyDomain* and a *PowerDomain*. (Note: The links between domains and modules are only references, there are no visible connections inside the model)

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the frequency domain
DefaultValue	Containment	Frequency	1	Default frequency value
Clock Gating	Boolean	Boolean	1	Possibility to power down the domain (default = false)

Table 6.3: FrequencyDomain

## 6.4 PowerDomain

A *PowerDomain* is inherited from *HwDomain*. This element describes a power domain which can be referenced by all elements of the type *HwModule*, to define the default voltage value for operation. In future the *PowerDomain* should also contain "possibleValues" which should specify the different voltages for different operation modes. For this extension a general Amalthea mode concept is necessary.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the power domain
DefaultValue	Containment	Voltage	1	Default voltage value
PowerGating	Boolean	Boolean	1	Possibility to power down the domain (default = false)

Table 6.4: PowerDomain

## 6.5 ProcessingUnit

A *ProcessingUnit* is a *HwModule* that can be used to model a wide set of different hardware components like a GPU, hardware accelerator, CPU, etc. The capability and the functionality of a *ProcessingUnit* are represented by different *HwFeatures* within the *ProcessingUnitDefinition*. The *ProcessingUnit* can be referenced by *AccessPaths* and *HwAccessElements*. The *ProcessingUnits* are the master modules in the model and every *ProcessingUnit* can has their own access space.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the processing unit instance
Ports	Containment	HwPort	*	Ports of the component
Caches	Containment	Cache	*	Included caches by the Processing Unit e.g. L1 Cache
Access Elements	Containment	Access- Element	*	Access element for a specific memory or processing unit
Frequency Domain	Reference	Frequency Domain	1	Frequency domain which supplies the module with a frequency
PowerDomain	Reference	PowerDomain	1	Power domain which supplies the mod- ule with a voltage
Definition	Reference	ProcessingUnit Definition	1	Definition with all features for the pro- cessing unit instance

Table 6.5: ProcessingUnit

## 6.6 Memory

A *Memory* is a component of type *HwModule* to express any kind memory like SRAM, DRAM, Flash, etc. in the model, caches are modeled separately. The *Memory* element can be referenced as destination by a *HwAccessElement*.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the memory instance
Ports	Containment	HwPort	*	Ports of the component
Frequency Domain	Reference	Frequency Domain	1	Frequency domain which supplies the module with a frequency
PowerDomain	Reference	PowerDomain	1	Power domain which supplies the mod- ule with a voltage
Definition	Reference	Memory- Definition	1	Definition with all features for the memory instance

Table 6.6: Memory

## 6.7 Cache

A *Cache* is a component of type *HwModule* to express the special behavior of a *Cache*. It is used to create cache topologies within a system. The *Cache* can be referenced by *AccessPaths* to express if it is a cached or non-cached access. It is also the only *HwModule* which can be directly contained by a *ProcessingUnit*.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the cache instance
Ports	Containment	HwPort	*	Ports of the component
Frequency Domain	Reference	Frequency Domain	1	Frequency domain which supplies the module with a frequency
PowerDomain	Reference	PowerDomain	1	Power domain which supplies the module with a voltage
Definition	Reference	CacheDefinition	1	Definition with all features for the cache instance

Table 6.7: Cache

## 6.8 ConnectionHandler

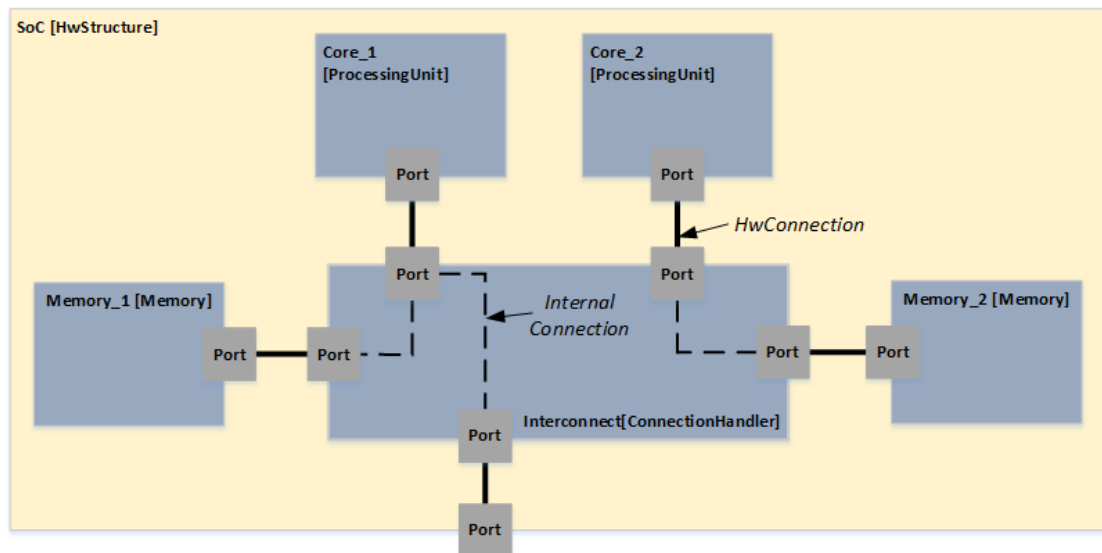


Figure 6.3: ConnectionHandler example

A *ConnectionHandler* is a component of type *HwModule* which can be used whenever multiple *HwConnections*, (*HwPorts*) have to be combined. It is possible to represent whole bus systems or interconnects with a single *ConnectionHandler*, or elements like small routers within a NoC.

Figure 6.3 shows an example where a *ConnectionHandler* is used as an interconnect within a SoC. Optional it is possible to model *InternalConnections* inside a *ConnectionHandler* to model explicit or restrict different connections. However it is also possible to use default read and write latencies of the whole *ConnectionHandlerDefinition*, individual latencies can be attached to *InternalConnections*. A short example where a *ConnectionHandler* is used as a CAN bus is illustrated in figure 6.1. For detailed models where all modules connected via *HwConnections* and different *ConnectionHandlers*, the *ConnectionHandlers* should be the only module where contentions in the hardware model can occur<sup>2</sup>. A *ConnectionHandler* can be referenced by *HwAccessPaths*.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the connection handler instance
Ports	Containment	HwPort	*	Ports of the component
Internal Connections	Containment	HwConnection	*	Internal connection between the ports
Frequency Domain	Reference	Frequency Domain	1	Frequency domain which supplies the module with a frequency
PowerDomain	Reference	PowerDomain	1	Power domain which supplies the module with a voltage
Definition	Reference	Connection-Handler-Definition	1	Definition with all features for the connection handler instance

Table 6.8: ConnectionHandler

## 6.9 HwAccessElement

A *HwAccessElement* can be used to specify the access relationship between two *ProcessingUnits* or a *ProcessingUnit* and a *Memory*. With multiple *HwAccessElements* the

<sup>2</sup>Under the circumstance that the validation rule that every *HwPort* is referenced by only one *HwConnection* is kept

whole access or even address space of a *ProcessingUnit* can be represented. A *HwAccessElement* represents always the view from a specific *ProcessingUnit*. There exist two different approaches to express latency or an data rate for a *HwAccessElement*: 1. directly using latencies or data rates or 2. modeling the exact path to the destination by attaching a *HwAccessPath* which references the specific connection elements like *ConnectionHandlers*, *HwConnection*, etc. For the second approach it is also possible to work directly with addresses. As a small example for the *HwAccessElement* figure 3.3 can be used.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the address element
Destination	Reference	HwDestination	1	Destination for the processing unit
AccessPaths	Containment	HwAccessPath	1	Access path to the destination
Read Latency	Containment	HwLatency	1	Read latency to the destination
Write Latency	Containment	HwLatency	1	Write latency to the destination
Data Rate	Containment	DataRate	1	Max.data rate to the destination

Table 6.9: HwAccessElement

## 6.10 HwFeatureCategory

*NOTE: The description for FeatureCategory and HwFeature are only valid als long the concept which is explained in chapter 2 is not integrated in Amalthea. Afterwards the links between the SW Model and the HW model might change and so the description and meaning of FeatureCategory and HwFeature.*

A *FeatureCategory* can be used to group different *HwFeatures* which belong to the same category. In the current state the *HwFeatureCategory* can be directly referenced inside the *RunnableItem ExecutionNeed* within the software model. An example for the connection between *FeatureCategory*, *HwFeatures*, the software mode and the mapping model is shown in 4.1 and as a concrete use case in figure 4.2.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the hardware feature
Type	Enum	HwFeatureType	1	Type to express the purpose of the feature (performance, power, performance_and_power)
Description	String	String	1	Textual description of the hardware feature
HwFeature	Containment	HwFeature	*	Hardware feature with a factor

Table 6.10: HwFeatureCategory

## 6.11 HwFeature

*NOTE: The description for FeatureCategory and HwFeature are only valid as long the concept which is explained in chapter 2 is not integrated in Amalthea. Afterwards the links between the SW Model and the HW model might change and so the description and meaning of FeatureCategory and HwFeature.*

All *HwFeatures* are created inside a *FeatureCategory* and referenced by *ProcessingUnitDefinitions*. In future we will extend the Feature concept for *MemoryDefinitions*, *CacheDefinitions* and *ConnectionHandlerDefinitions*. *HwFeatures* could be reused several times by different definitions. A *HwFeature* can contain a value to express the capability of a *ProcessingUnit*. Currently this value is used as a factor to calculate the cost for a *Runnable* executed on a *ProcessingUnit*. Figure 6.4 shows an example how *HwFeatures* and *FeatureCategories* are used in a model.

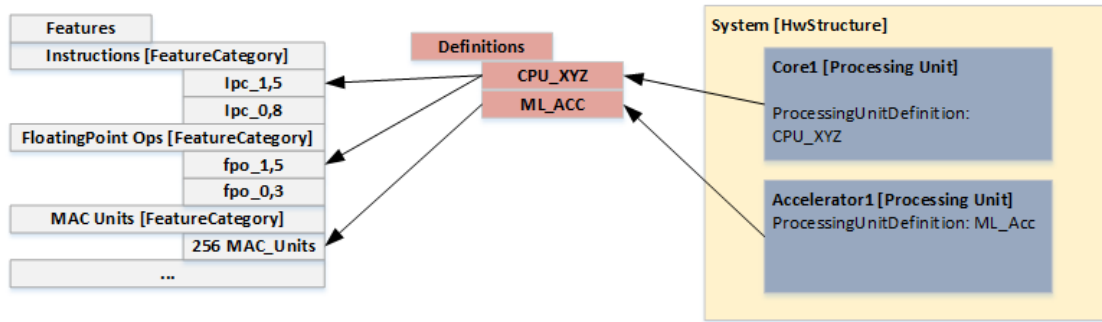


Figure 6.4: HwFeature example



Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the hardware feature
Value	Containment	Value	1	assigned factor to the corresponding feature

Table 6.11: HwFeature

## 6.12 HwPort

*HwPorts* are elements which can be connected via *HwConnections*. Every module can contain multiple *HwPorts*. Every communication, input or output is handled via a *HwPort* of a component. It is only allowed to have one *HwConnection* per *HwPort*, except the *HwPort* is categorized as delegated port which means it is just a hierarchical connection between *HwStructures*. In this case the ports can have two *HwConnections*. The second exception is if inside a *ConnectionHandler*, *InternalConnections* are used. In this case a *HwPort* can be directed with a *HwConnections* and an *InternalConnection*. Figure 6.5 shows an example with delegated *HwPorts* and *InternalConnections*.

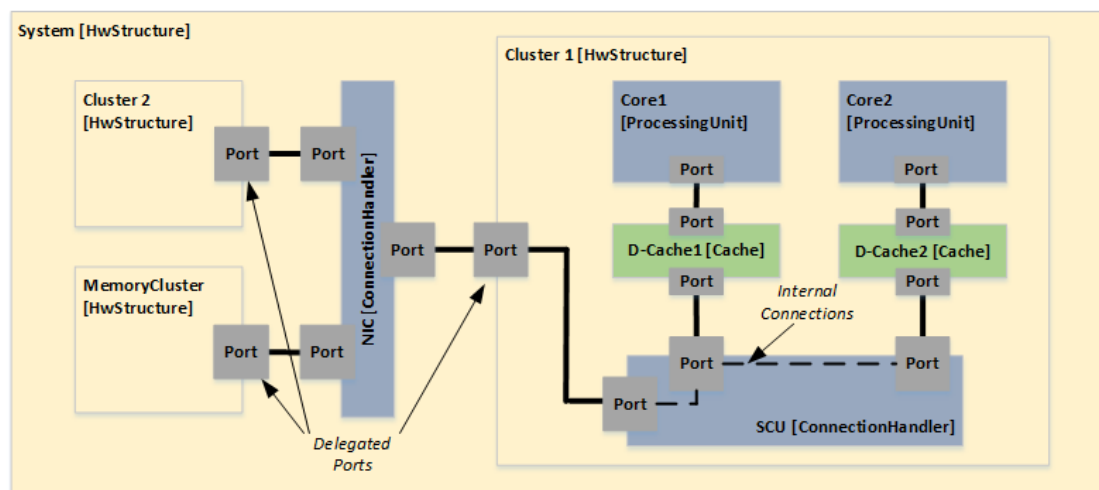


Figure 6.5: HwPorts example

For *HwPorts* it is always possible to select if the port is an *initiator* or a *responder* port. The following example shows that an initiator port is always connected to a responder port (comparable to TLM modeling).

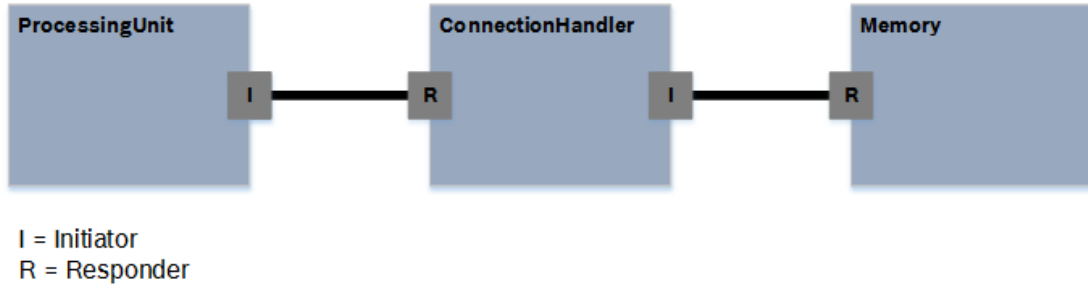


Figure 6.6: Initiator and responder ports

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the hardware port
BitWidth	Int	Int	1	Bit width e.g. 32 bit (default = 0)
Priority	Int	Int	1	Priority of the hardware port (default = 0)
Type	Enum	PortType	1	Port type (initiator, responder)
Delegated	Bool	Bool	1	Delegated ports are hierarchical structure ports
Port Interface	Enum	PortInterface	1	Type to express special interfaces for validation

Table 6.12: HwPort

## 6.13 HwConnection

A *HwConnection* is an element to model structural connections between two *HwPorts*. *HwConnections* are always placed within *HwStructures*. It is possible to directly annotate a read and write latency at a *HwConnection*. *HwConnections* can be referenced by *HwAccessPaths*. The *HwConnection* does not have a reference to a *FrequencyDomain*, the frequency is always provided by the element which is in front of the *HwConnection* in the *HwAccessPath*.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the hardware connection
Port1	Reference	HwPort	1	Port1 for the connection
Port2	Reference	HwPort	1	Port2 for the connection
Read latency	Containment	HwLatency	1	Constant or distribution in cycles for a read access
Write latency	Containment	HwLatency	1	Constant or distribution in cycles for a write access
Data rate	Containment	DataRate	1	Max. Data rate of the connection (value and unit)

Table 6.13: HwConnection

## 6.14 HwAccessPath

A *HwAccessPath* is an element to describe the connection route of a *ProcessingUnit* to its destination (*Memory* or *ProcessingUnit*). The *HwAccessPath* is defined through an ordered list of interface elements (*HwConnections*, *Caches* and *ConnectionHandlers*) and is a containment of an *HwAccessElement*. Figure 6.7 shows an example of an *HwAccessPath*, how a *ProcessingUnit* is connected via two *HwConnections* and a *ConnectionHandler* with a *Memory*.

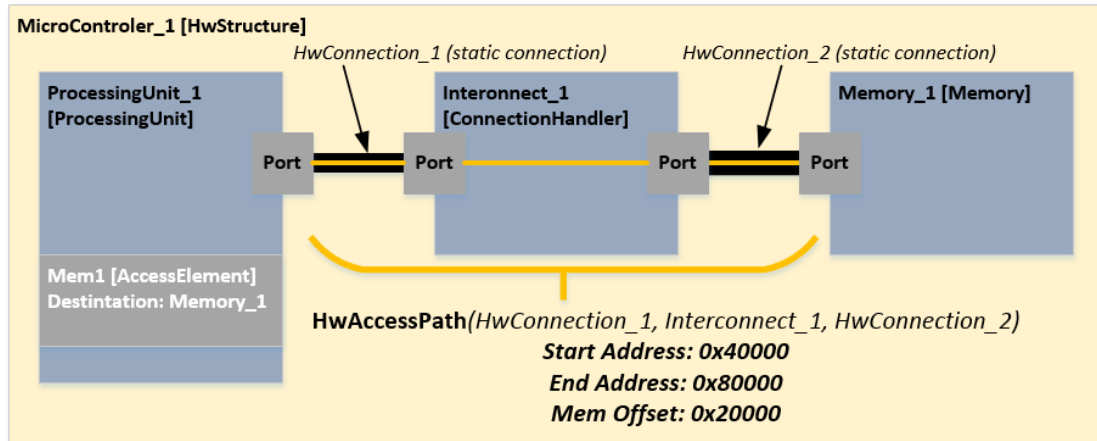


Figure 6.7: HwAccessPath example

In the following example the possible `memOffset` attribute is explained. Every *ProcessingUnit* can access a *Memory* or other *ProcessingUnit* over a different address. The size of the *Memory* has to be equal or greater than `endAddress` minus the `startAddress`.

$$memory\_size = endAddress - startAddress \quad (6.1)$$

In the case the the *ProcessingUnit* should not start at address 0 (from the memory point of view) the `memOffset` attribute can be used. With help of this attribute the access area for the memory can be changed, figure 6.8 shows an example.

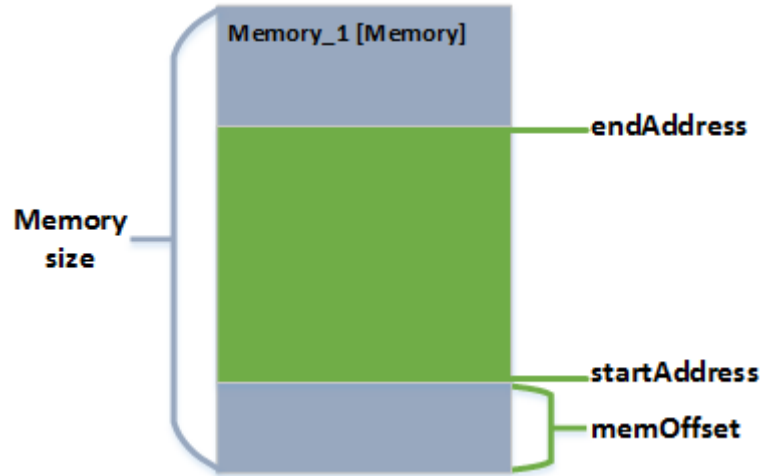


Figure 6.8: Memory address example

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the hardware access path
Path Elements	Reference	HwPath	*	Path elements for the access path
Start Address	Long	Long	1	Start address for the memory
End Address	Long	Long	1	End address for the memory
Mem Offset	Long	Long	1	Offset for accessing only a partition of a memory

Table 6.14: HwAccessPath

## 6.15 ProcessingUnitDefinition

The example in figure 3.2 is representative for any kind of definition in the model. This means for specifying a compute resource a *ProcessingUnitDefinition* is created once which is then referenced by the number of *ProcessingUnit* instances of this kind. For a *ProcessingUnitDefinition* the references to *HwFeatures* is one of the key concepts of the hardware model. The concept is explained in figure 4.1.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the processing unit definition
Pu Type	Enum	PuType	1	Type of the processing unit e.g. (Core, GPU, etc.)
Features	Reference	HwFeature	*	Hardware features of the definition

Table 6.15: ProcessingUnitDefinition

## 6.16 MemoryDefinition

The example in figure 3.2 is representative for any kind of definition in the model. This means for specifying a memory, a *MemoryDefinition* is created once which is then referenced by the number of *Memory* instances of this kind.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the memory definition
Access Latency	Containment	HwLatency	1	Constant or distribution of access latency in cycles
Data Rate	Containment	DataRate	1	Max. data rate for the memory
Size	Containment	Size	1	Size of the memory

Table 6.16: MemoryDefinition

## 6.17 CacheDefinition

The example in figure 3.2 is representative for any kind of definition in the model. This means for specifying a cache, a *CacheDefinition* is created once which is then referenced

by the number of *Cache* instances of this kind.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the memory definition
Access Latency	Containment	HwLatency	1	Constant or distribution of access latency in cycles
Size	Containment	Size	1	Size of the memory
Cache Type	Enum	CacheType	1	Cache type (e.g. data, instruction)
Write Strategy	Enum	WriteStrategy	1	Cache write strategy (e.g. write-back)
Coherency	Bool	Bool	1	Cache coherency (default = false)
Exclusive	Bool	Bool	1	Exclusive cache (default = false)
Line Size	Int	Int	1	Line size in bits
Hit Rate	Double	Double	1	Percentag hit rate of the cache (default = 0.0)
NWays	Int	Int	1	N ways associative (default = 0)

Table 6.17: CacheDefinition

## 6.18 ConnectionHandlerDefinition

The example in figure 3.2 is representative for any kind of definition in the model. This means for specifying a bus or Interconnect etc., a *ConnectionHandlerDefinition* is created once which is then referenced by the number of *ConnectionHandler* instances of this kind.

Attribute	Type	Value	Mul	Description
Name	String	String	1	Name of the memory definition
Sched Policy	Enum	SchedPolicy	1	Enumeration of different scheduling policies
Read Latency	Containment	HwLatency	1	Constant or distribution in cycles for a read access
Write Latency	Containment	HwLatency	1	Constant or distribution in cycles for a write access
Data Rate	Containment	DataRate	1	Max. Data rate of the connection (value and unit)

Table 6.18: ConnectionHandlerDefinition

## 6.19 LatencyConstant

A *LatencyConstant* is used to determine a constant number of clock cycles and can be attached to various other elements e.g. access latencies for a *MemoryDefinition* or a read latency for a *ConnectionHandlerDefinition* .

Attribute	Type	Value	Mul	Description
Cycles	Long	Long	1	Constant number of clock cycles (default = 0)

Table 6.19: LatencyConstant

## 6.20 LatencyDeviation

A *LatencyDeviation* is an object which allows to create a distribution out of different possibilities e.g. Weibull, Gaussian etc. The *LatencyDeviation* can be attached to various elements e.g. access latencies for a *MemoryDefinition* or a read latency for a *ConnectionHandlerDefinition* .

Attribute	Type	Value	Mul	Description
Cycles	Deviation	Deviation	1	Deviation for a specific element in clock cycles

Table 6.20: LatencyDeviation



## 7 Enums

In the following all enums are listed. In the case an enum the default value is always `__undefined__` (omitted in the following lists). That means that in case of an enum there are no default values for interfaces or other kind of types.

In future there will be an option to extend the predefined enums with further port interfaces, hardware structure types etc. by selecting the option `__other__`. Then a second attribute field will appear to specify a custom entry. Moreover only new enums are explicitly mentioned in this report. Enums and classes which are already part of the existing Amalthea meta model are not described.

### **StructureType:**

*{System, ECU, Microcontroller, SoC, Cluster, Group, Array, Area, Region}*

### **CacheType:**

*{instruction, data, unified}*

### **VoltageUnit:**

*{V, mV, uV}*

### **PortType:**

*{initiator, responder}*

### **SchedPolicy:**

*{RoundRobin, FCFS, PriorityBased}*

### **WriteStrategy:**

*{none, writeback, writethrough}*

### **PuType:**

*{GPU, CPU, Accelerator}*

### **PortInterfaces:**

*{custom, can, flexray, lin, most, ethernet, spi, i2c, axi, ahb, apb, swr}*

### **HwFeatureType:**

*{performance, power, performance\_and\_power}*