Amalthea HW Model – An overview and component description for the new Amalthea HW model proposal

Bosch Corporate Research

February 6, 2018
v0.8

This report describes a new proposal for the Amalthea hardware model.

The implementation and the report are work in progress.

To collect feedback, the report and the model are shared with the community. Please consider that changes to the specification and the model itself are reserved. Comments and suggestions are welcome.

The Eclipse APP4MC project provides a preview product which includes the new hardware model. It is based on Eclipse Neon.3 and has a size of approximately 330 MB.

Available downloads for different operating systems:

- `org.eclipse.app4mc.platform-0.8.3-SNAPSHOT-...-linux.gtk.x86_64.zip`
- `org.eclipse.app4mc.platform-0.8.3-SNAPSHOT-...-macosx.cocoa.x86_64.zip`
- `org.eclipse.app4mc.platform-0.8.3-SNAPSHOT-...-win32.win32.x86_64.zip`
- `org.eclipse.app4mc.platform-0.8.3-SNAPSHOT-...-win32.win32.x86.zip`

The product is provided in a separate git feature branch of Eclipse APP4MC and can be downloaded from the eclipse build infrastructure:

https://ci.eclipse.org/app4mc/job/build-src-branches-pipes/job/hmackamul%252Ffeature%252Fnew-hw-model/lastStableBuild/artifact/build/org.eclipse.app4mc.platform.product/target/products/

Contents

1 Structural Modeling of Heterogeneous Platforms **4**
1.1 General Hardware Model Overview 5
1.2 Interpretation of latencies in the model 7
1.3 Element description 9
1.3.1 HwModel 9
1.3.2 HwStructure 9
1.3.3 FrequencyDomain 11
1.3.4 PowerDomain 12
1.3.5 ProcessingUnit 12
1.3.6 Memory 13
1.3.7 Cache 13
1.3.8 ConnectionHandler 14
1.3.9 HwAccessElement 15
1.3.10 HwFeature 16
1.3.11 HwPort 17
1.3.12 HwConnection 18
1.3.13 HwAccessPath 19
1.3.14 ProcessingUnitDefinition 20
1.3.15 MemoryDefinition 21
1.3.16 CacheDefinition 21
1.3.17 ConnectionHandlerDefinition 22
1.3.18 LatencyConstant 23
1.3.19 LatencyDeviation 23
1.4 Enums 24

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 or set of math operations like “multiply and accumulate”. The benefit from specialization is different and leads to nonlinear effects between processing units in terms of performance for algorithms. 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 the different clusters including different hardware components via a Network on Chip. These characteristics of modern and performant hardware specialized processing units, complex memory hierarchy, network like Interconnects are only partially supported by the former Amalthea hardware model and tools for performance simulation. Therefore, to create models of modern heterogeneous systems, new concepts of representing hardware components in a flexible and easy way are necessary: Beside of modeling a manifold hierarchical structures, domains for power and frequencies are the state of the art. Furthermore cache and memory subsystem modeling is mandatory and the connection between hardware components has to be modeled over different abstraction layers. Only with such an extended modeling approach, a more accurate estimation of the system performance becomes feasible.

Our intention is 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 information 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., execution time of a software functionality that is mapped to a processing unit. These costs are sometimes pretty hard to obtain and, in contrast to the hardware structure, may change during development time. Therefore, the inherent costs 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 multiple times in the development process to increase accuracy of performance estimation. Further this refinement should be possible in an efficient way and support model re-use.

1.1 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.

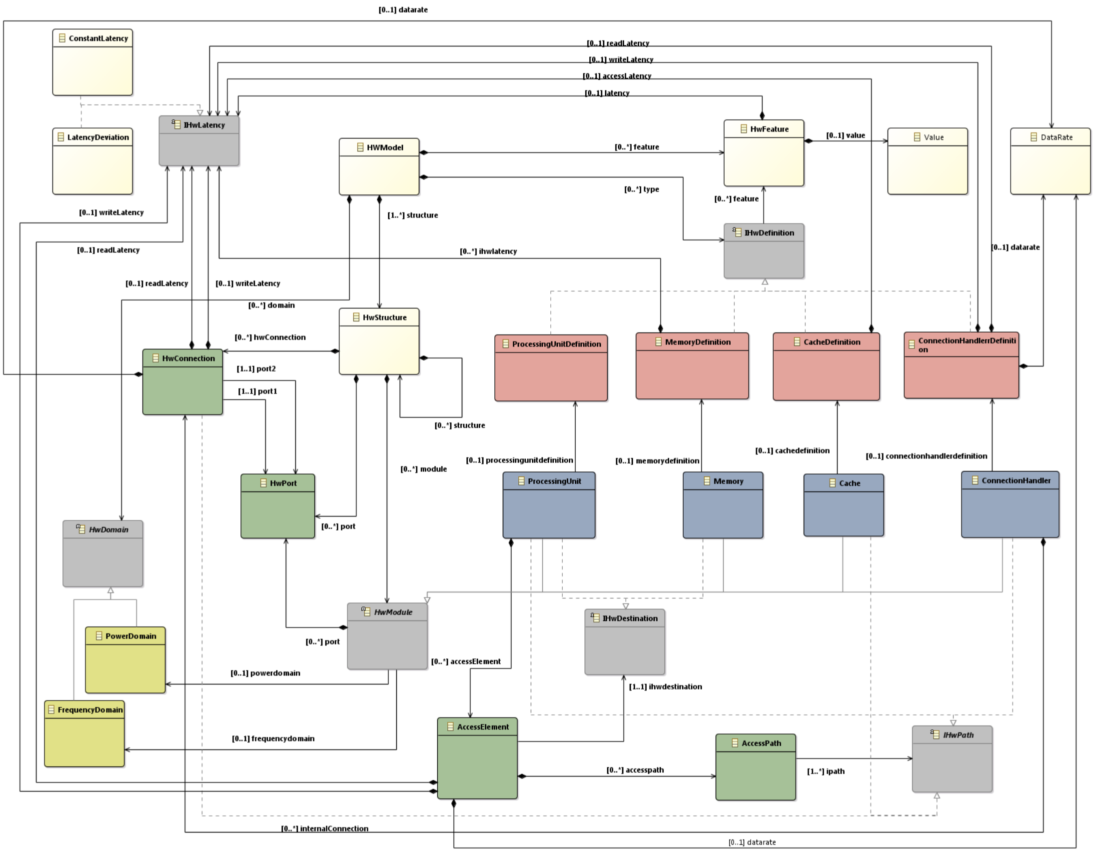

Figure 1.1: Class diagram of the hardware model

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 [1]. Red and blue classes in the figure are the definitions and the main components of a system like a memory or a core.

Figure 1.3 shows the modeling of a processor. The *ProcessingUnitDefiniton*, 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 *ProcessingUnitDefiniton* for generic information, supplemented only with instance specific information like the *FrequencyDomain*.

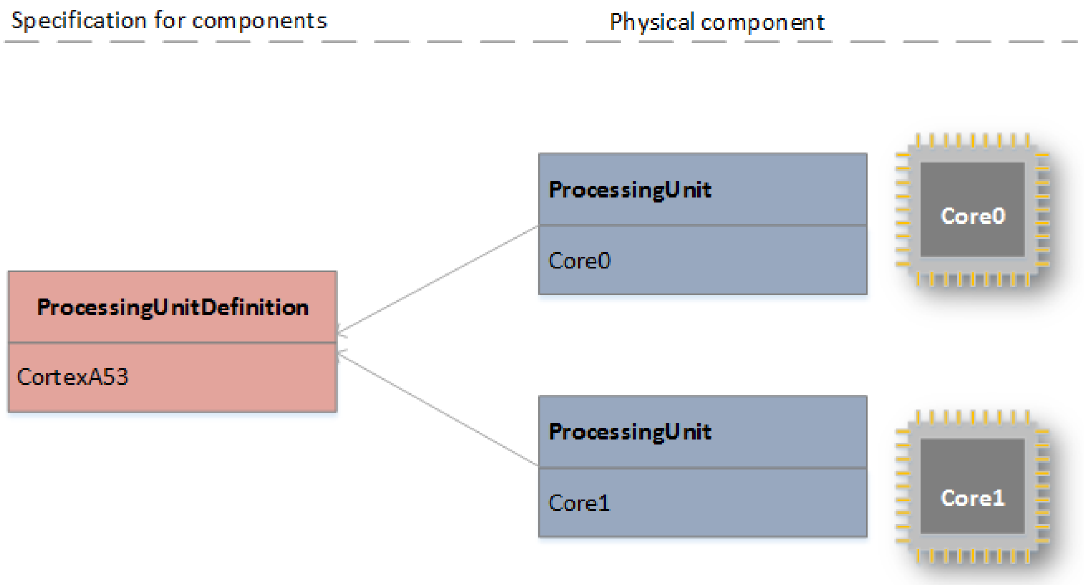

Figure 1.2: Link between definitions and module instances (physical components)

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, 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

[1] includes the "classical" hierarchy from the original Amalthea model, System –> ECU –> Microcontroller –> Core, but also allows more flexible clustering.

ProcessingUnit_2 in figure 1.3. Every time an *HwAccessElement* is necessary to assign the destination e.g. a *Memory* component. This *HwAccessElement* can contain a latency or a bandwidth 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 its 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*.

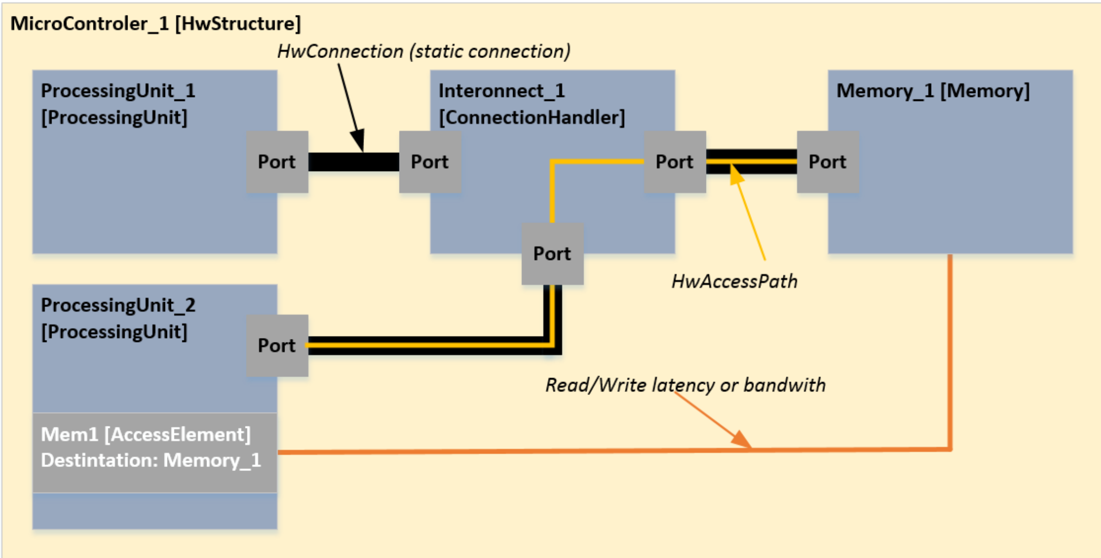

Figure 1.3: Access elements in the hardware model

1.2 Interpretation of latencies in the model

In the model are read, write and access latencies are used. In hardware specifications or measurements often request and response latencies are used. Figure 1.4shows a typical communication between two components. The interpretation of a read and write latency for example at *ConnectionHandlers* is the following:

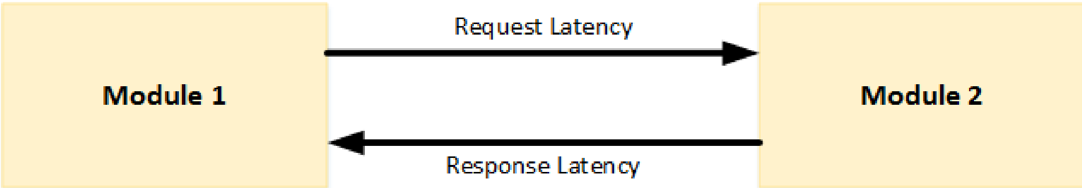

Figure 1.4: Request and response latency

$$readLatency = requestLatency + responseLatency \tag{1.1}$$

$$writeLatency = requestLatency \tag{1.2}$$

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 1.3 in case using only an access element:

$$TotalReadLatency = readLatency(HwAccessElement) + accessLatency(Memory) \tag{1.3}$$

$$TotalWriteLatency = writeLatency(HwAccessElement) + accessLatency(Memory) \tag{1.4}$$

As a concrete example for 1.3 in case using only an access element with access path:

n = Number of path elements

$$TotalReadLatency = (\sum_{p=0}^{n} readLatency(p)) + accessLatency(Memory) \tag{1.5}$$

$$TotalWriteLatency = (\sum_{p=0}^{n} writeLatency(p)) + accessLatency(Memory) \tag{1.6}$$

PathElements could be *Caches*, *ConnectionHandlers* and *HwConnections*. In very special cases also a *ProcessingUnit* can be a PathElement in this case the latency has to be annotated as *HwFeature*.

1.3 Element description

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

1.3.1 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 *HwFeatures* for the *HwModule* definitions. An example is shown in Section **??**.

Attribute	Type	Value	Mul	Description
name	String	String	1	Name of the hardware model
structure	Containment	HwStructure	*	Hierarchic structure of the hardware model
features	Containment	HwFeature	*	Features of the HwModel
domains	Containment	HwDomain	*	Frequency- and PowerDomains
definitions	Containment	HwDefinition	*	Definitions of ProcessingUnits, Memories, Caches and ConnectionHandlers

Table 1.1: HwModel

1.3.2 HwStructure

A *HwStructure* is a hierarchical element which can contain all kind of *HwModules* and *HwConnections*. 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 if hierarchal systems can be expressed. By setting the type attribute (e.g. Cluster, ECU) the structural level in the hardware is directly expressible in the model.

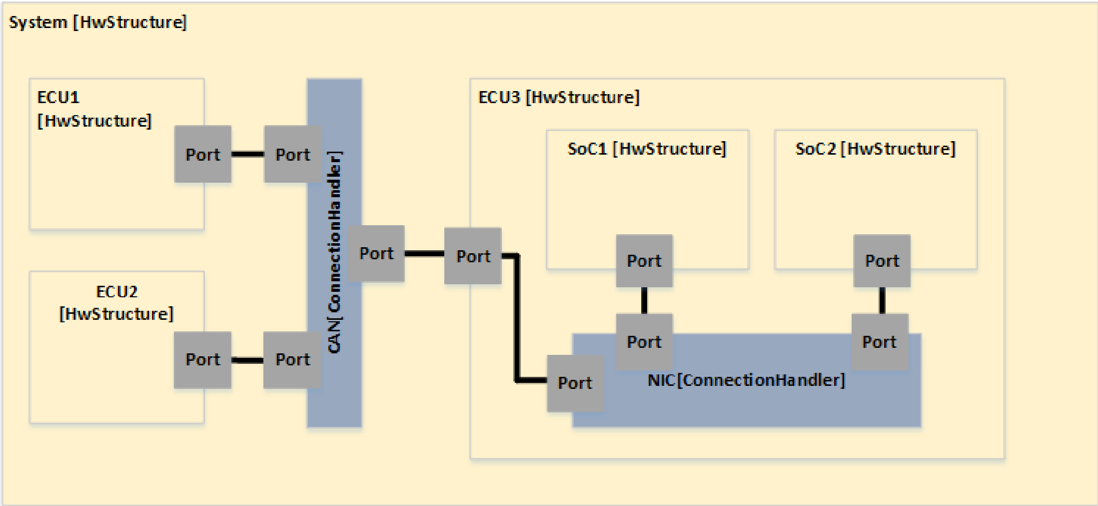

Figure 1.5: HwStructure example

Figure 1.5 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
type	Enum	Structure-Type	1	Defines the type of the structure (e.g. ECU)
module	Containment	HwModule	*	Modules of the structure (e.g. Memory)
port	Containment	HwPort	*	Ports to connect the structure
connections	Containment	HwConnection	*	Connections within a structure

Table 1.2: HwStructure

1.3.3 FrequencyDomain

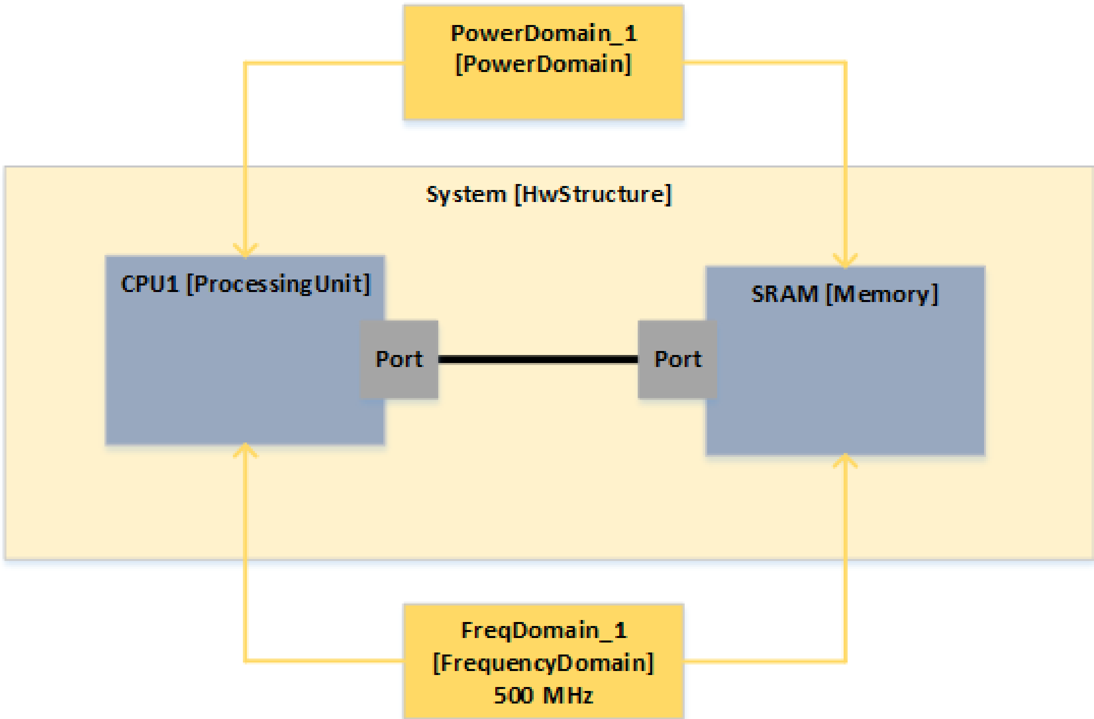

Figure 1.6: Frequency- and PowerDomain example

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 possible frequency values for operation.

Figure 1.6 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 link between domains and modules are only a 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
possibleValues	Containment	Frequency[]	1	Different possible value levels for a domain
clockGating	Boolean	Boolean	1	Possibility to power down the domain

Table 1.3: FrequencyDomain

1.3.4 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 possible voltage values for operation. For an example see figure 1.6

Attribute	Type	Value	Mul	Description
name	String	String	1	Name of the power domain
defaultValue	Containment	Voltage	1	Default voltage value
possibleValues	Containment	Voltage[]	1	Different possible value levels for a domain
powerGating	Boolean	Boolean	1	Possibility to power down the domain

Table 1.4: PowerDomain

1.3.5 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
access-Elements	Containment	Access-Element	*	Access element for a specific memory or processing unit
definition	Reference	ProcessingUnit Definition	1	Definition with all features for the processing unit instance

Table 1.5: ProcessingUnit

1.3.6 Memory

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

Attribute	Type	Value	Mul	Description
name	String	String	1	Name of the memory instance
ports	Containment	HwPort	*	Ports of the component
definition	Reference	Memory-Definition	1	Definition with all features for the memory instance

Table 1.6: Memory

1.3.7 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.

Attribute	Type	Value	Mul	Description
name	String	String	1	Name of the cache instance
ports	Containment	HwPort	*	Ports of the component
definition	Reference	CacheDefinition	1	Definition with all features for the cache instance

Table 1.7: Cache

1.3.8 ConnectionHandler

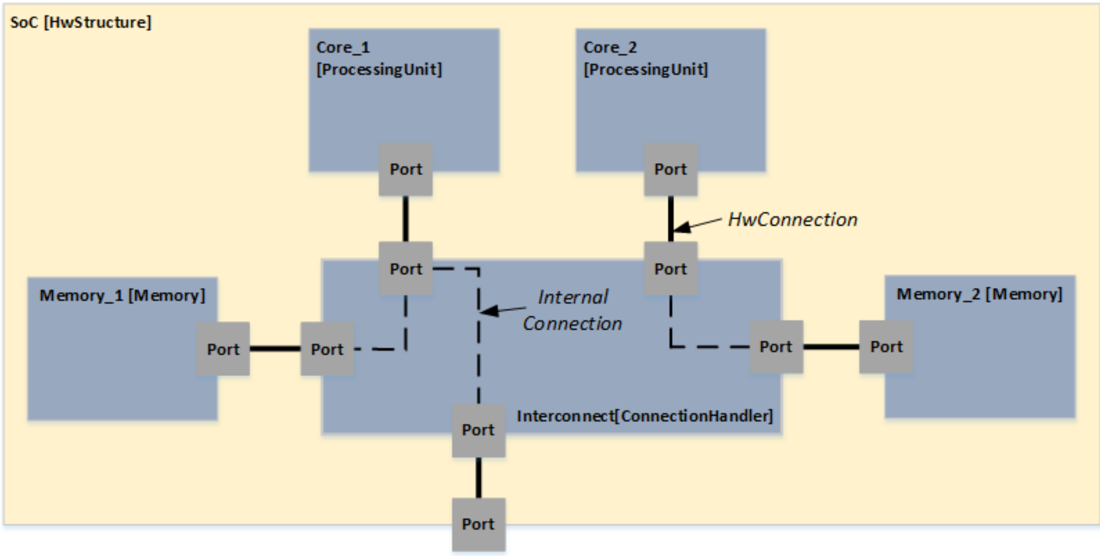

Figure 1.7: ConnectionHandler example

A *ConnectionHandler* is a component of type HwModule which can be used whenever multiple *HwConnections* 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 1.7 shows an example where a *ConnectionHandler* is used as an interconnect within a SoC. Optional it is also possible to model *InternalConnections* inside the *ConnectionHandler* to model explicit the different connections. However it is also possible to use read and write latencies of the *ConnectionHandlerDefinition* for the complete *ConnectionHandler* without using *InternalConnections*. A short example where a *ConnectionHandler* is used as a CAN bus is illustrated in figure 1.5. 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[2]. 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
definition	Reference	Connection-Handler-Definition	1	Definition with all features for the connection handler instance

Table 1.8: ConnectionHandler

1.3.9 HwAccessElement

An *HwAccessElement* can be used to specify the access relationship between two *ProcessingUnits* or a *ProcessingUnit* and a *Memory*. With multiple *HwAccessElements* the whole access or even address space of a *ProcessingUnit* can be represented. An *HwAccessElement* represents always the view from a specific *ProcessingUnit*. For the *HwAccessElement* exists two different approaches to express latency or an data: 1. directly using latencies for read and write accesses 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 1.3 can be used.

[2] Under the circumstance the validation rule that every HwPort has only one HwConnection is kept

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	*	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. date rate to the destination

Table 1.9: HwAccessElement

1.3.10 HwFeature

A *HwFeature* is an abstract element to represent any kind of special functionality. The cost function (*Recipes*) of an algorithm will be placed in an intermediate layer outside of the hardware model. However the *HwFeatures* will be directly referenced by the *Recipes*. All *HwFeatures* are placed inside the *HwModel*. Specific definitions for the basic components are able to reference such *HwFeatures* to express their functionality. HwFeatures could be reused several times by different definitions. A *HwFeature* can contain a latency to express static costs in a model. Figure 1.8 shows an example how recipes are used in a model. *NOTE: The Recipes and the HwFeatures concept is still work in progress. Changes to the HwFeatures are probable.*

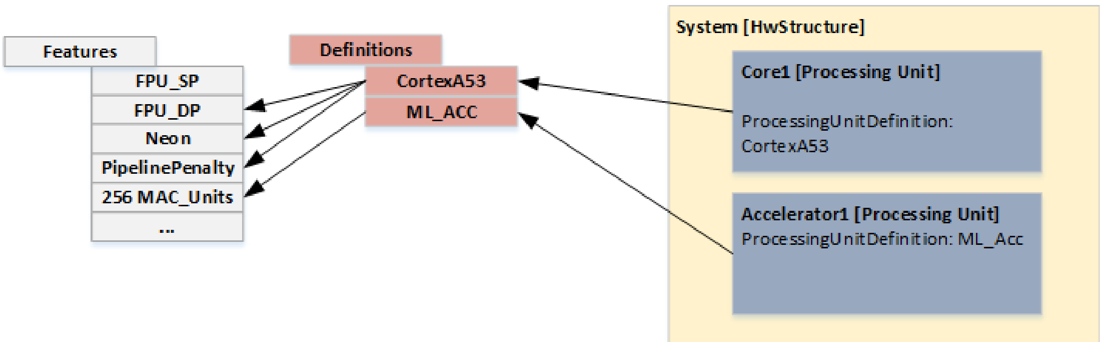

Figure 1.8: HwFeature example

Attribute	Type	Value	Mul	Description
name	String	String	1	Name of the hardware feature
latency	Containment	HwLatency	1	latency of a hardware feature
value	Containment	Value	1	factor to express the influence of a hardware feature
type	Enum	HwFeatureType	1	Type to express the purpose of the feature (performance, power, both , information)
description	String	String	1	Textual description of the hardware feature

Table 1.10: HwFeature

1.3.11 HwPort

HwPorts are elements to which can be connected via *HwConnections*. Every module can contain multiple *HwPorts*. Every communication, input or output is handled via the *HwPorts* of a component. It is only allowed to have one *HwConnection* per *HwPort*, expect 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. Figure 1.9 shows an example with delegated *HwPorts* and *InternalConnections*.

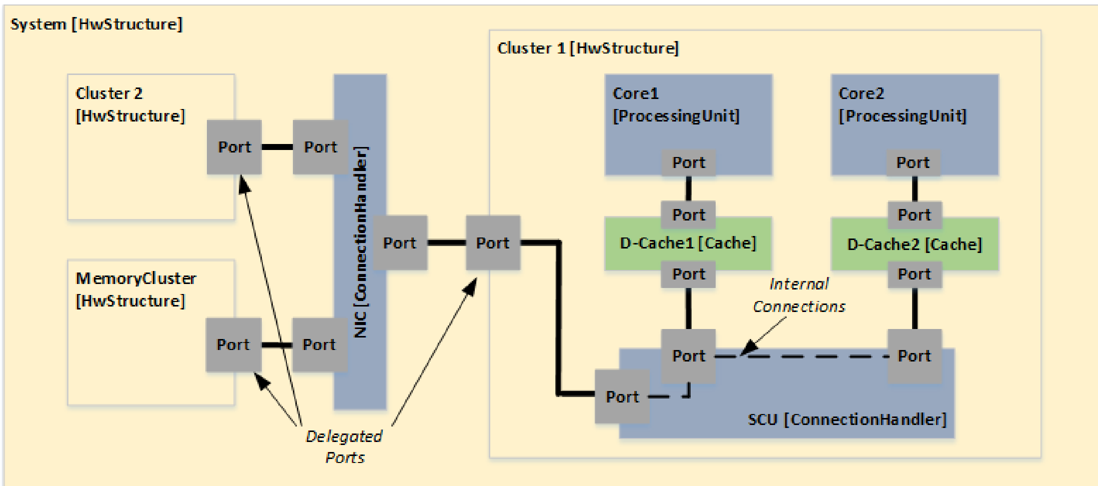

Figure 1.9: HwPorts example

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

Table 1.11: HwPort

1.3.12 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*.

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
dataRate	Containment	DataRate	1	Data rate of the connection (value and unit)

Table 1.12: HwConnection

1.3.13 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 IPaths interface elements (*HWConnections, Caches* and *ConnectionHandler*) and is a containment of an *HwAccessElement*. Figure 1.10 shows an example of an *HwAccessPath*, how a *ProcessingUnit* is connected via two *HwConnections* and a *ConnectionHandler* with a *Memory*.

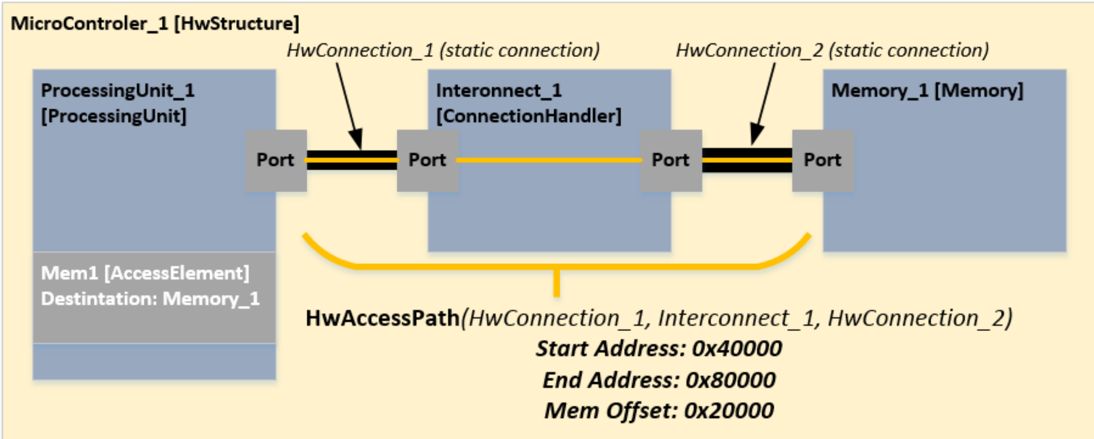

Figure 1.10: HwAccessPath example

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 smaller then *endAddress* minus the *startAddress*.

$$memory\_size \geq endAddress - startAddress \tag{1.7}$$

In the case the the *ProcessingUnit* should not start at address 0 (from the memories point of view) the *memOffset* attribute can be used.

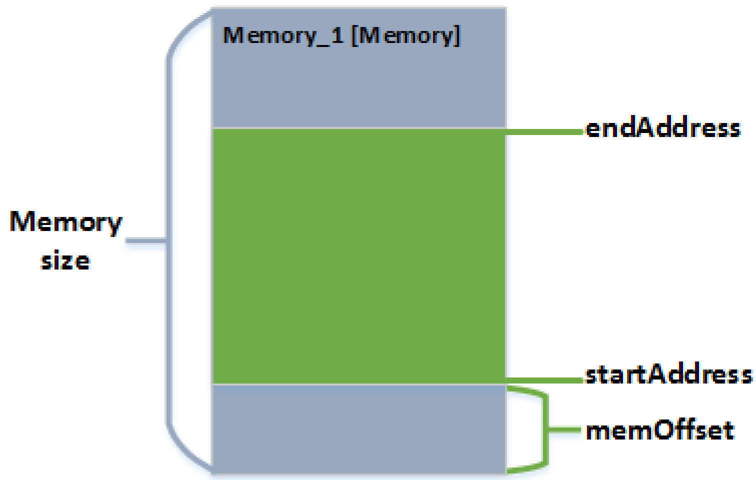

Figure 1.11: Memory address example

Attribute	Type	Value	Mul	Description
name	String	String	1	Name of the hardware access path
pathElements	Reference	HwPath	*	Path elements for the access path
startAddress	Long	Long	1	Start address for the memory
endAddress	Long	Long	1	End address for the memory
memOffset	Long	Long	1	Offset for accessing only a partition of a memory

Table 1.13: HwAccessPath

1.3.14 ProcessingUnitDefinition

The example in figure 1.2 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.

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

Table 1.14: ProcessingUnitDefinition

1.3.15 MemoryDefinition

The example in figure 1.2 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
accessLatency	Containment	HwLatency	1	Constant or distribution of access latency in cycles
memory bandwidth	Containment	DataRate	1	Max. memory bandwidth
size	Containment	Size	1	Size of the memory
features	Reference	HwFeature	*	Hardware features

Table 1.15: MemoryDefinition

1.3.16 CacheDefinition

The example in figure 1.2 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
accessLatency	Containment	HwLatency	1	Constant or distribution of access latency in cycles
size	Containment	Size	1	Size of the memory
features	Reference	HwFeature	*	Hardware features
cacheType	Enum	CacheType	1	Cache type (e.g. data, instruction)
writeStrategy	Enum	WriteStrategy	1	Cache write strategy (e.g. write-back)
coherency	Bool	Bool	1	Cache coherency
exclusive	Bool	Bool	1	Exclusive cache
line Size	Int	Int	1	line size in bits
nWays	Int	Int	1	N ways associative

Table 1.16: CacheDefinition

1.3.17 ConnectionHandlerDefinition

The example in figure 1.2 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
schedPolicy	Enum	SchedPolicy	1	Enumeration of different scheduling policies
features	Reference	HwFeature	*	Hardware features
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
dataRate	Containment	DataRate	1	Data rate of the connection (value and unit)

Table 1.17: ConnectionHandlerDefinition

1.3.18 LatencyConstant

A *LatencyConstant* is used to determine a constant number of clock cycles and can be attached to various other elements e.g. *HwConnection* or *HwFeature*.

Attribute	Type	Value	Mul	Description
constantCycles	Long	Long	1	Constant number of clock cycles

Table 1.18: LatencyConstant

1.3.19 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. *HwConnection* or *HwFeature*.

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

Table 1.19: LatencyDeviation

1.4 Enums

In the following all enums are listed. In the case an enum is used by any class the default value of that enum is always _undefined_. 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:

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

CacheType:

{_undefined_, instruction, data, unified}

VoltageUnit:

{_undefined_, V, mV, uV}

PortType:

{_undefined_, initiator, responder}

SchedPolicy:

{_undefined_, RoundRobin, FCFS, PriorityBased, _other_}

WriteStrategy:

{_undefined_, none, writeback, writethrough, _other_}

PuType:

{_undefined_, GPU, CPU, Accelerator, _other_}

PortInterfaces:

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

HwFeatureType:

{_undefined_, performance, power, both, information}