Distributed Real-time Architecture for Mixed Criticality Systems

Meta-models for Application and Platform
D 1.4.1

Project Acronym	DREAMS	**Grant Agreement Number**		FP7-ICT-2013.3.4-610640	
Document Version	1.0	**Date**	31.03.2015	**Deliverable No.**	1.4.1
Contact Person	Simon Barner	**Organisation**		fortiss	
Phone	+49 (0)89360352222	**E-Mail**		barner@fortiss.org	

Contributors

Name	Partner
Simon Barner	FORTISS
Alexander Diewald	FORTISS
Fernando Eizaguirre	IKL
Óscar Saiz	IKL
Lionel Havet	RTAW
Jörn Migge	RTAW
Franck Chauvel	SINTEF
Anatoly Vasilevskiy	SINTEF
Marcello Coppola	ST
George Tsamis	TEI
Gebhard Bouwer	TUV
Donatus Weber	USIEGEN (ES)
Thomas Koller	USIEGEN (DCS)

Table of Contents

Executive Summary

This deliverable defines meta-models for the description of mixed-criticality applications and instances of the DREAMS platform. It is structured into four viewpoints that provide meta-models for the description of different aspects of DREAMS systems. The presentation of each of the viewpoints comprises a specification of the corresponding meta-models, an explanation of their utilization, and a discussion of example model instances.

The *Architecture Viewpoint* clusters meta-models used to describe structural aspects of the system. It comprises a *Logical Viewpoint* that provides a meta-model for the platform-independent description of applications, a *Technical Viewpoint* capturing the structure of instances of the DREAMS platform, and a *Deployment Viewpoint* for the description of mappings between the model elements of the logical and the technical viewpoint. The *Temporal Viewpoint* provides meta-models that can be used to express timing requirements and temporal properties of applications. The *Extra-Functional Viewpoint* groups meta-models for the description of application requirements and platform properties related to safety, security and power consumption. Finally, the *Variability Viewpoint* provides a meta-model that can be used to create separate specifications of variation points of a given (product) model.

In addition to the definition of the meta-models, this document describes editors that can be used to create and manipulate model instances. The toolset is based on both existing tool implementations (that have been extended in the scope of DREAMS Task T1.4) and newly created tool prototypes. All tools are implemented as plugins for the Eclipse platform which allows to integrate the model editors for the different parts of the meta-model.

1 Introduction

This deliverable defines meta-models for the description of mixed-criticality applications and instances of the DREAMS platform. Furthermore, it introduces editors that can be used to create model instances.

1.1 Structure of the Deliverable

The DREAMS application and platform meta-model has been structured into four main viewpoints that provide meta-models for the description of different aspects of DREAMS systems. In the following, these viewpoints that are defined in Chapter 2 will be summarized:

- The group of *Architecture Viewpoints* clusters meta-models used to describe structural aspects of the system. It includes a *Logical Viewpoint* that provides meta-model for the platform-independent description of applications, a *Technical Viewpoint* capturing the structure of instances of the DREAMS platform, and a *Deployment Viewpoint* for the description of mappings between the model elements of the logical and the technical viewpoint.
- The *Temporal Viewpoint* provides meta-models that can be used to express timing requirements and temporal properties of applications.
- The *Extra-Functional Viewpoint* groups meta-models for the description of application requirements and platform properties related to safety, security and power consumption.
- The *Variability Viewpoint* provides a meta-model that can be used to create separate specifications of variation points of a given (product) model.

Section 3 will describe the model editors that can be used to create instances of the meta-models defined in this deliverable. The toolset is based on both existing tool implementations (that have been extended in the scope of DREAMS Task T1.4) and newly created tool prototypes. All tools are implemented as plugins for the Eclipse platform which allowed integrating the model editors for the different parts of the meta-model. Finally, the main part of the document (Chapters 4-9) contains a detailed description of the meta-models defined by viewpoints summarized above.

1.2 Positioning of the Deliverable in the Project

The architectural style document D1.2.1 (which was a direct result of the requirements elicitation in D1.1.1) served as the primary input for the specification of the DREAMS application and platform meta-models. Furthermore, D1.3.1 contains a description of the use of models in the DREAMS development process and also includes an initial overview of some meta-models.

The application and platform meta-models defined in this document will be complemented by platform-specific meta-models (D1.6.1) that define the meta-models related to the configuration of the building blocks of the DREAMS platform. Hence, D1.6.1 will add additional meta-models to the viewpoints defined in this document. Furthermore, deliverables D4.1.2 and D4.1.3 provide information about meta-models for the specification of design goals and constraints.

The meta-models defined in this document D1.4.1 will be used as input specification of the methods and tools developed in the following tasks:

- T4.2 “Offline Adaptation Strategies for Mixed Criticality”: Deliverables D4.1.2 and D4.1.3
- T4.3 “Explicit Variability Configuration”: Deliverables D4.3.2 and D4.3.3
- T5.2: “Simulation, Verification and Fault-injection Framework”: Deliverables D5.2.1 and D5.2.2

The dissemination level of this deliverable is public (PU) i.e. once approved by the European Commission (EC), it will be freely available for download through the DREAMS project website (http://www.dreams-project.eu).

In the scope of this documents, small example model instances of the meta-models defined in this documents will be presented. In course of the intermediate integration, example models based on the current specification of the use cases from the application demonstrators will be created. They will be presented in the integration report D1.5.1 whose dissemination level (confidential / CO) matches the one of the demonstrator use case descriptions.

2 Viewpoints

In this chapter, the criteria for structuring the application and platform meta-models defined in this document will be presented.

2.1 Introduction

The main goal of this document is to present meta-models that are suitable for the

- Platform-independent description of mixed-criticality applications, and the
- Description of instances of the DREAMS HW/SW platform.

Since the focus of both of the above points is on the description of *architecture* of a particular *system-of interest* (here DREAMS systems, i.e. a given mixed-criticality applications deployed to an instance of the DREAMS HW/SW platform), the relevant concepts defined in ISO/IEC/IEEE 42010 [1] will be summarized in the following, and related to the corresponding sections in this document.

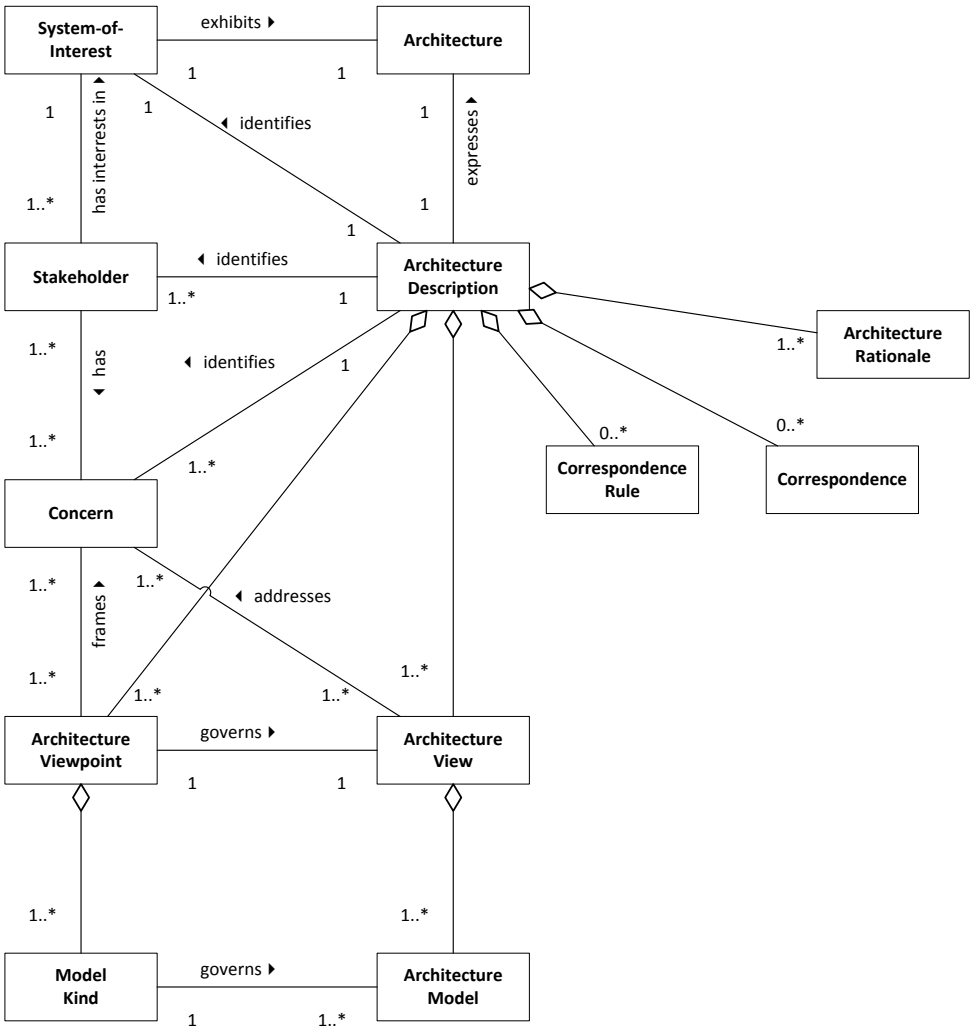


Figure 2.1: Conceptual model of architecture description [1]

An *architecture description* framework is structured into different *architecture viewpoints* that are defined as “way[s] of looking at systems”. Each of the viewpoints covers one or more *concerns* of the description of an architecture.

For each architecture viewpoint, there exist one or more *model kinds* that define the corresponding conventions and (modelling) languages. In this document, *meta-models* are used to define the used model kinds that contain the following information:

- Entities: major sorts of elements
- Attributes: properties of entities
- Relationships: relations between entities
- Constraints

The main part of this document (sections 4 - 9) is concerned with the detailed definition of the meta-models that realize the different viewpoints. For some of the view-points such as the deployment viewpoint (Section 5.3) additional meta-models will be presented in D1.6.1 “Meta-models for platform-specific modelling”.

Finally, Section 3 provides an overview of the different tools and editors that have been developed and / or extended to create *architecture views* (i.e., (*architecture) models* conforming to the meta-models defined in this document) that correspond the selected architecture viewpoints. Additionally, this section will provide a definition of fundamental meta-models that are not specific to the selected viewpoints, but provide modelling constructs shared between different meta-models.

Figure 2.2 on the one hand sketches the use of different viewpoints to provide information about different aspects of the system. On the other, the figure also illustrates the granularity dimension as a second dimension of abstraction used in the DREAMS meta-models: A system is decomposed into sub-systems which are at a lower granularity level and which can themselves be regarded as systems. The process can be applied recursively, until finally basic building blocks are reached.

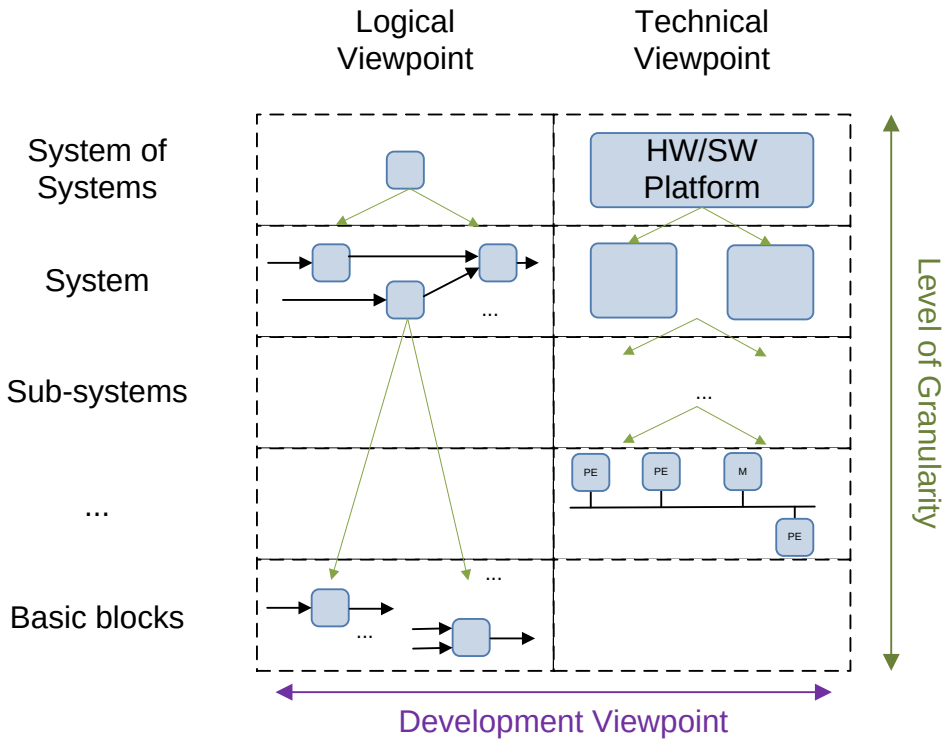


Figure 2.2: Dimensions of abstraction: viewpoints and granularity-levels

In the following, the viewpoints that have been defined to describe DREAMS system will be presented. On the one hand, this comprises a summary of the meta-models constituting the different view-points. On the other hand, the interface between the different meta-models will be sketched.

2.2 Architectural Viewpoints

The *Architecture Viewpoints* clusters meta-models used to describe structural aspects of the system description. They are separated into three sub-viewpoints that will be described in the following. The meta-models provided by the architectural viewpoints are based on the AutoFOCUS3 tool (see Section 3.2).

2.2.1 Logical Viewpoint

The *Logical Viewpoint* (see Section 4) provides meta-models to describe the logical, or functional, aspects of a mixed-criticality application in a platform-independent way. The main meta-model is the *Logical Component Architecture Meta-Model* that provides a meta-model for the description of a component architecture of an application. This meta-model of the application's structure is augmented with additional information provided by further meta-models, e.g.

- For the specification of a component's behaviour (e.g., using state automata or mode automata).
- For the specification of extra-functional properties of a component (e.g., the component's criticality level or the component's security requirements).

Furthermore, the *Logical Component Architecture Meta-Model* is referenced by a number of meta-models from other viewpoints, such as the deployment meta-model (see Section 6), the safety compliance meta-model (see Section 8.1) and the meta-models defined in the temporal viewpoint (see Section7).

2.2.2 Technical Viewpoint

The *Technical Viewpoint* (see Section 5) provides meta-models used to describe the structure of the DREAMS hardware/software platform. It consists of a Platform Architecture Meta-Model that provides a framework for the description of hierarchic platform architectures. Based on this, meta-models for the layers defined in the DREAMS Architectural Style (see D1.2.1) have been defined. These meta-models are augmented with additional information, e.g.

- Specification of extra-functional properties of platform elements (e.g., clock speed of cores, parameters related to power consumption, component reliability annotations, security mechanisms provided by platform elements).
- Linking between different layers of a platform model. In the presented approach, one platform model is used to describe the hardware architecture of a DREAMS platform. Another platform model is used to abstract the (system) software part of the platform that contains links to the elements of the hardware architecture onto which the corresponding system software component has been deployed to.

Lastly, also the meta-models provided by the *Technical Viewpoint* are referenced by meta-models defined in other viewpoints, including the deployment meta-model (see Section 6) and the safety compliance meta-model (see Section 8.1).

2.2.3 Deployment Viewpoint

The *Deployment Viewpoint* collects all deployment related model kinds. For this deliverable D1.4.1, it only comprises meta-models required to describe the mapping of model elements from the logical view to model elements of the technical view. The follow-up document D1.6.1 "Meta-models for platform-specific modelling" will focus on enhancing this viewpoint with description mechanisms for the allocation of platform resources.

2.3 Temporal Viewpoint

The *Temporal Viewpoint* is composed of one meta-model (see Section 7). It is an adapted version of the meta-model defined in the Timmo-2-use project[1] for the AutoFOCUS3 framework (see Section 3.2), allowing making the link between temporal constraint and logical architecture elements (see Section 4).

2.4 Extra-functional Viewpoints

2.4.1 Safety Viewpoint

2.4.1.1 Scope in the DREAMS V-Life-Cycle development process

The Safety Model is associated to the Logical Component Architecture Model and the Platform Architecture Model. It aims at the early detection of some errors during the realization phase. In Figure 2.3, the V-Life-Cycle according to IEC 61508-1 [2] is illustrated for the following phases:

- The architectural specification of a system or family of systems (e.g., Wind Power Turbines).
- The process of choosing the specific architecture of a specific system by resolving variability (e.g., producing a specific wind turbine).
- Also, during the step of defining a deployment for the specific wind turbine.

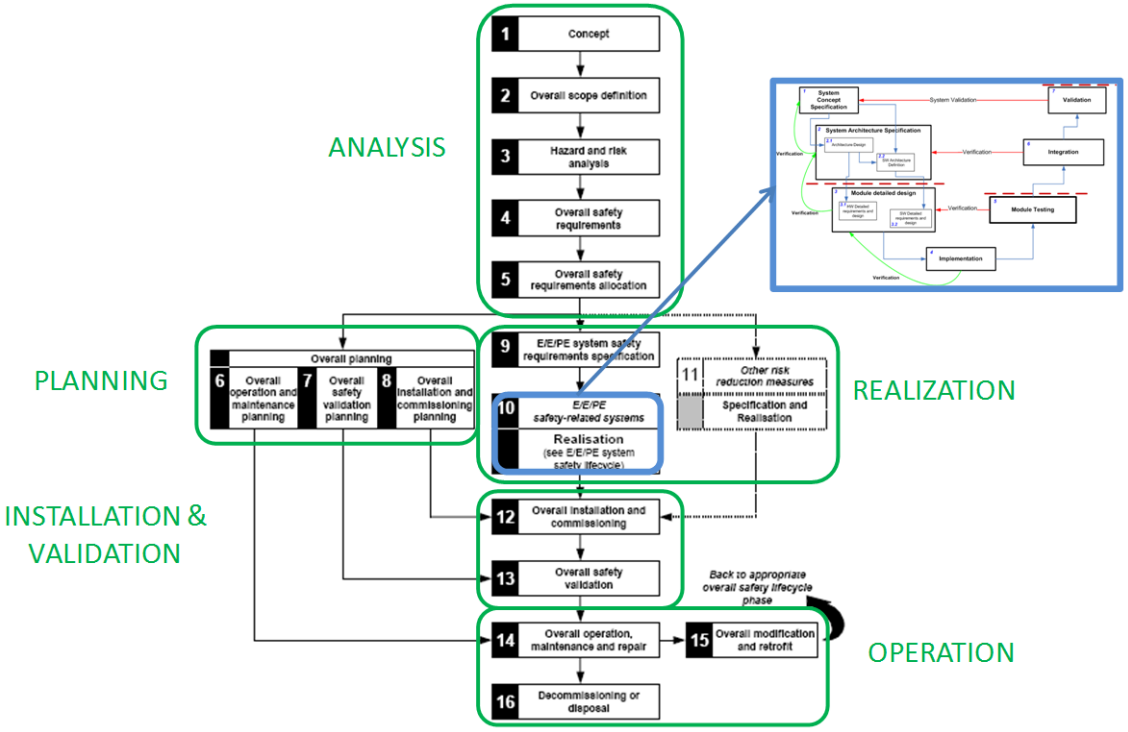


Figure 2.3: IEC 61508-1 (Edition 2.0): General requirements

The safety model defined for a given system (a) is used to **support the checking of *safety consistency rules*** that can help designers in the design of mixed-criticality solutions reducing the risk of late discovery of safety related expensive design pitfalls (that would prevent a certification) and

[1] https://itea3.org/project/timmo-2-use.html

(b) is used **to help in a certification process** by providing some evidences of safety aspects that have taken into account during the realization phase.

The consistency rules to be implemented (as part of the ***SafetyConstraintChecker*** class to be implemented in WP4, see section devoted to this class below for a brief introduction) are not described in deliverable D1.4.1 (devoted to meta-models). However, a list of examples is given below. The set of consistency rules implemented in DREAMS can be easily extended to enrich the capability of catching early errors.

The IEC 61508 distinguishes between hardware integrity (expressed in SILx) and the systematic capability (expressed in SCy). However it is assumed that the expression "SILx" covers the hardware integrity of SILx as well as the systematic capability of SCy.

As mentioned above, the safety model defined for a given system is used to support the checking of *safety consistency rules.* In the following, some examples will be presented:

- *Example Rule* - If a HW Node (DREAMS Node) claims to be SIL3 with hardware fault-tolerance (HFT)=1 and is composed by two processors (i.e., DREAMS Tiles (see D1.2.1)), one claiming SIL3 and the other SIL2, then a warning is given. This way the designer realizes there is an inconsistency.
- *Example Rule* – If a HW element (e.g., DREAMS Node, Tile) claims to be SILx with HFT=1 and is composed of two HW elements (for example a Node composed of two Tiles) that are (two sub examples cases to clarify):
    - SIL(x-1) and SCy (System Capability in IEC 61508), then a warning would be given saying "Warning: Acceptable provided they are part of independent channels"
    - SIL(x-1) and SC(y-1), then an error would be given, because claimed SILx cannot be achieved).
- *Example Rule* - If a HW node is using a "*Program Sequence*" technique (e.g. IEC 61508-2 Table A.10 "Watch-dog with separate time base without time-window), it is checked that the node:
    1. Has an external HW watchdog.
    2. The watchdog is connected to the processor.
    3. They are connected to two independent clocks.
- *Example Rule* - It is checked that if the safety manual declares having a Medium Diagnostic Coverage of (90-99%) and HFT = 0, the maximum allowable SIL level is SIL 2 according to IEC 61508-2 Table 3 (type B components). In case the claimed SIL level was higher a warning is given.
- *Example Rule* - If a Partition (certified or not) claims a SIL3 level, but is deployed to a Hypervisor (certified or not) that claims for example a SIL2 level, and error is given in case the safety function requires SIL3. In case the safety function requires SIL2 this would be acceptable.
- *Example Rule* - Similarly if a Hypervisor (certified or not) claims a SIL3 but is deployed to a processing unit that claims to be SIL2, then an error is given in case the safety function requires SIL3. In case the safety function requires SIL2 this would be acceptable.
- Etc.

To implement these consistency rules, the Safety Manual class defined in the meta-model (see Section 8.1) do not include the full set of attributes defined in Annex D (of both IEC 61508-2 [3] and IEC 61508-3 [4]). Instead, it comprises *only those attributes needed to implement consistency rules*:

- ***FSM*** (Functional Safety Management), IEC 61508,
- ***SIL*** level (Safety Integrity Level, SIL1, SIL2, SIL·, SIL4))

- **SC** (Systematic Capability, SC1, SC2, SC3, SC4)
- ***HFT*** (Hardware Fault Tolerance, HFT1, HFT2, HFT3)
- *Diagnosis and Measure Techniques* and ***Hypothesis*** values and ranges.

The Safety Model and, therefore also the consistency rules, scope mainly the 'architectural specification' phase within the system 'realization' because of the following reasons:

- The analysis, planning, installation and operation of the system do not usually consider the internal implementation details of the system (e.g. multicore partitioning technology); this is only relevant in the 'realization' phase. Therefore, it is interesting to focus the scope in discovering early errors in the realization phase of multicore partitioning scenarios.
- Within the 'realization' phase, the 'system architectural specification' needs to deal with the non-trivial integration of mixed-criticality applications; multiple partitions mapped to multicore platform(s). This is the phase in which the definition of rules for consistency checking can provide higher benefits; improving productivity and reducing the risk of late discovery of safety related design pitfalls.

To summarize the scope of safety meta-models (and therefore also the consistency rule checker provided by WP4) is to support discovering errors during the realization phase of the hardware architecture, and also with a basic support to discover basic errors in the integration of software partitions, hypervisors and deployment of components in a mixed criticality multicore scenario. The focus of the approach is on IEC 61508-2 [3] and also in part on IEC 61508-3 [4] (but not going in depth in IEC 61508-3).

2.4.1.2 Safety Meta-models

The *Safety Viewpoint* is composed of a set of three meta-models. The main meta-model is the *Safety Compliance Model* that gathers the safety specification of the DREAMS Architecture. The complete set of safety meta-models is composed of:

- *IEC 61508 and Diagnostic and Measures Safety Standard Meta-Model:* This meta-model enumerates the IEC 61508 SIL and ASIL safety integrity levels. In addition to this, this meta-model is used to represent the Diagnostic Techniques and Measures recommended in IEC 61508-2 [3], Annex A, to control failures caused by random faults during operation (tables A.2 to A14) and related to the systematic integrity (tables A.15 to A.18) during operation. Safety Manuals declare the techniques used. This way, consistency rules can perform some checks during the realization phase (as shown above) about SIL claimed, etc. Fault avoidance measures for the development of software according to IEC 61508-3, Annex A, are not considered at this point in time since they are applied in a later phase of the assumed overall system development process.
- *Safety Compliance Meta-Model:* This meta-model is used to add safety properties/attributes related to safety and IEC 61508-specific concepts to a hierarchy of Safety Compliant Items (SCI, to be described later). Each SCI item provides a 'Safety Manual'. As described in the previous section, the Safety Manual will contain just the information needed by consistency rules to detect errors early in the realization phase and not all information defined by Annex D of IEC 61508-2 [3] and IEC 61508-3 [4].
- *Safety Compliance Constraint Meta-Model:* This meta-model is used to model the constraints to be met by the deployment of the system in order to achieve a correct deployment/partitioning from the safety point of view.

A more detailed description of the above meta-models can be found in Section 8.1.

2.4.2 Security Viewpoint

The *Security Viewpoint* contains the security meta-model. It allows the modelling of the security services in DREAMS. It uses annotations to extend the logical and the technical viewpoint.

2.4.3 Power Viewpoint

The *Power Viewpoint* in Section 8.3 presents a solution for power modelling for interconnects IPs (ICN) to perform power analysis at system level.

2.5 Variability Viewpoint

The Variability Viewpoint clusters meta-models that can be used to define variability of a given base-meta-model in an orthogonal way, i.e. using a separate variability specification meta-model.

The required variability specification meta-models are provided by the BVR tool from SINTEF (see Section 3.3).

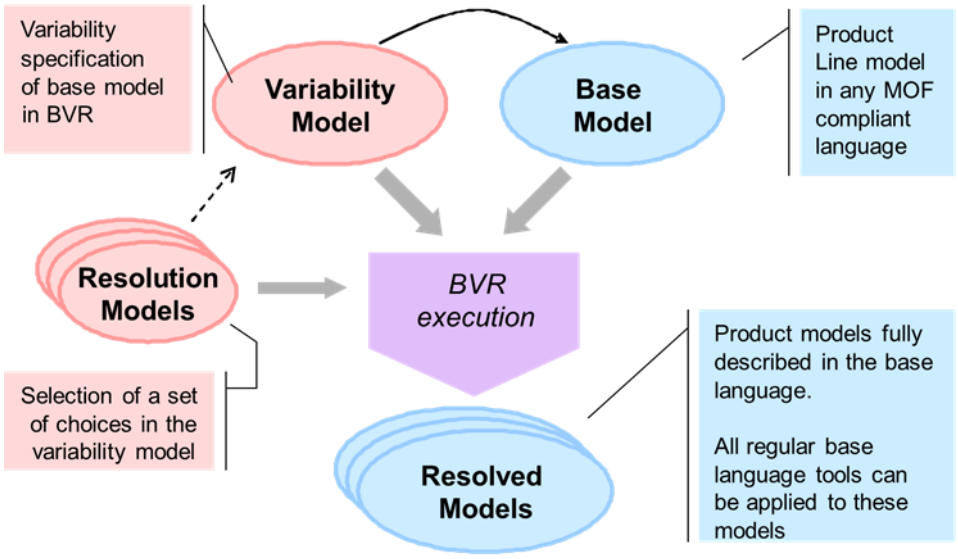


Figure 2.4: BVR conceptual architecture

Figure 2.4 sketches the approach to variability in BVR. The orange ovals as well as violet polygon represent BVR elements whereas the blue ovals depict models in any third-party language. The picture clearly shows that BVR does not amalgamate third-party languages (base models) with variability concepts rather defines variability in a separate model and links the base model by means of references. The BVR execution engine uses specified references to operate on base models to yield resolved models, i.e. products.

In Section 9.3.1, it will be illustrated, how variability models can be used to define variations of a model of a DREAMS system model. After that, in Section 9.3.2, these concepts are applied to the safety meta-model where a *Safety Feature* model is used to express variations of *Safety Compliance* models. Here, a variability model is defined using the BVR tool to express the variability of a *Safety Compliance* model.

3 Model Editors and Toolsets

This section provides an overview of the different model editors and toolsets that have been developed and / or extended to create views that correspond to the architecture viewpoints described in the previous section (i.e., models conforming to the meta-models defined in this document).

In this section, exclusively the modelling functionality of the mentioned toolsets is described, which is the scope of this document. It should be noted that these toolsets also provide additional functionality such as analysis, optimization, model transformation and generation of different artefacts such as program code, configuration and reports. The interfaces between the tools described below, as well as additional tools required for the development of DREAMS-systems have been defined in D1.3.1 “Description of Development Process with Model Transformations”.

Additionally, this section gives an overview of fundamental meta-models that are not specific to the selected viewpoints, but provide modelling constructs shared between different meta-models.

3.1 Overview

To create model instances from the DREAMS meta-model defined in this document, the following model editors are provided which will be described in the remaining sections of this chapter.

- AutoFOCUS3 (see Section 3.2),
- Base Variability Resolution (BVR) Tools (see Section 3.3), and
- Mixed-Criticality Product Line Editor (see Section 3.4).

All of the above model editors are based on the Eclipse Modelling Framework (EMF) [2] [5], and can be installed into the same Eclipse installation. In order to install all model editors, it is recommended to start with an installation of the AutoFOCUS3 Eclipse RCP that has been released for the DREAMS project (see Section 3.2.2), and to install the BVR toolset into this AutoFOCUS3 DREAMS RCP installation (see 3.3). Both the Mixed-Criticality Product Line Editor and the model editor for the DREAMS timing meta-model are already bundled with the AutoFOCUS3 DREAMS RCP.

The result of this full installation is an integrated model editor for the application and platform meta-models defined in this document, which allows the creation of interlinked instances of the models from the different viewpoints defined in Section 2. In other words, the model editors operate on separate models that contain references to models from other tools listed above. Since the tools are installed in the same eclipse instance they share a common workspace and, hence, the inter-model references are consistent, assuming the model files are not moved manually.

As can be seen in Figure 3.1, models created by the BVR Tool (Variability Viewpoint) contain references to AutoFOCUS3 models and to models generated by the Mixed Criticality Product-line editor (Safety Viewpoint). The safety and temporal model instances contain only references to AutoFOCUS3 model instances.

Due to the generality of the variability modelling approach (see Section 9.1), generic `EObject`-references are used to point from the variability meta-model to the respective foreign meta-model. In contrast to that, specific references are used in all other cases to establish links between different meta-models.

The integration of editors (required to specify references to the corresponding foreign meta-model) is as follows: An extension interface of the BVR tooling has been used to support the selection of

[2] http://www.eclipse.org/emf/

model elements in the graphical diagram model editors provided by AutoFOCUS3. In all other cases, the integration of editors relies on the mechanisms provided by the generated EMF tree-editors [5].

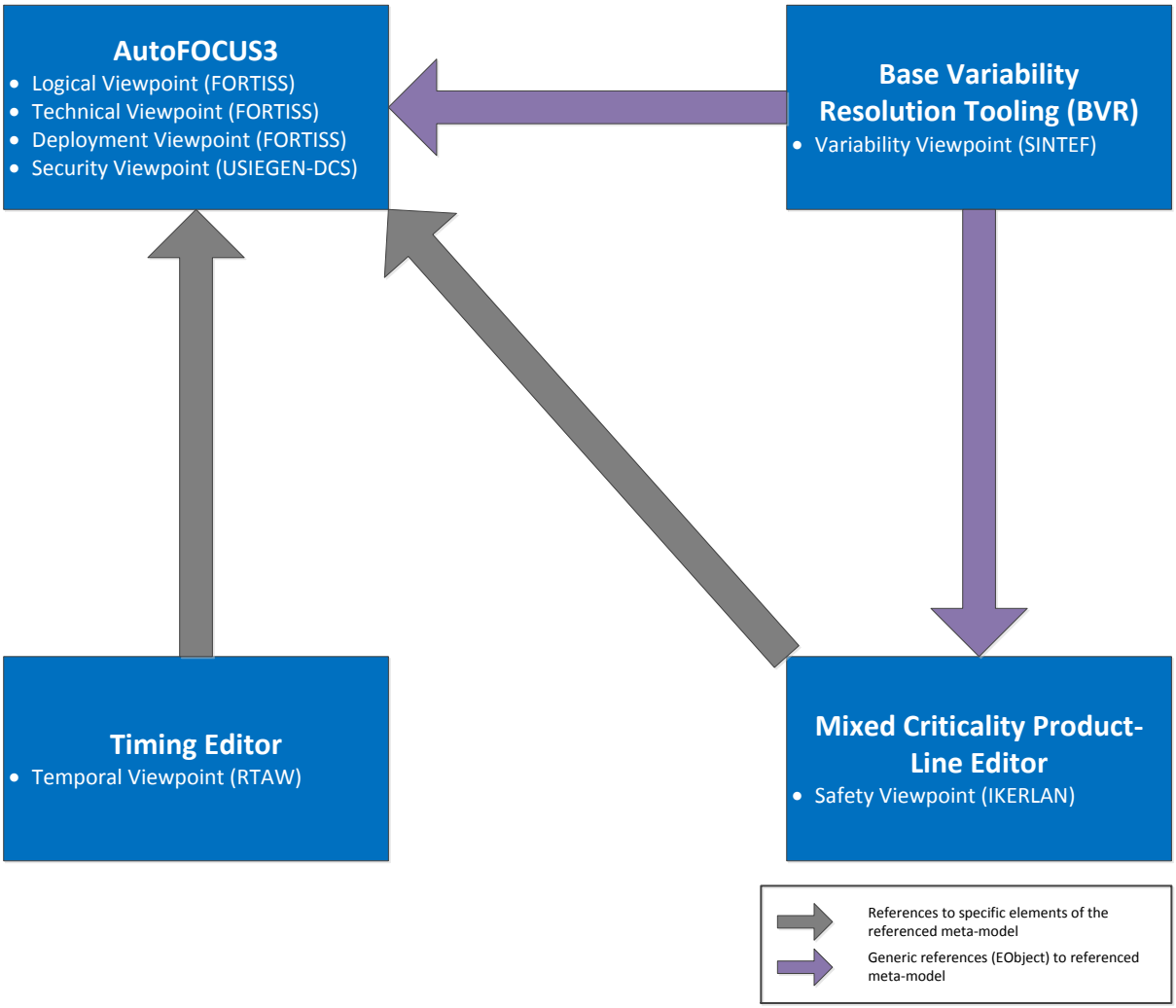


Figure 3.1: Model Editors and References between Models from different Viewpoints.

Figure 3.2 shows a screenshot of the full model editor installation discussed above. In the Navigator on the left side of the figure, an AutoFOCUS3 model (*DREAMS_Example_ControlUnit.af3_23*), a safety compliance model (Safety-*DREAMS_Example_ControlUnit.drm_safetycompliance*), and a variability model (*Var-DREAMS_Example_ControlUnit.bvr*) can be seen. These models are opened in their corresponding model editors shown on the right side of the figure. An example for inter-model references can be seen in the lower right part of the figure where the safety compliance model references an AutoFOCUS3 model.

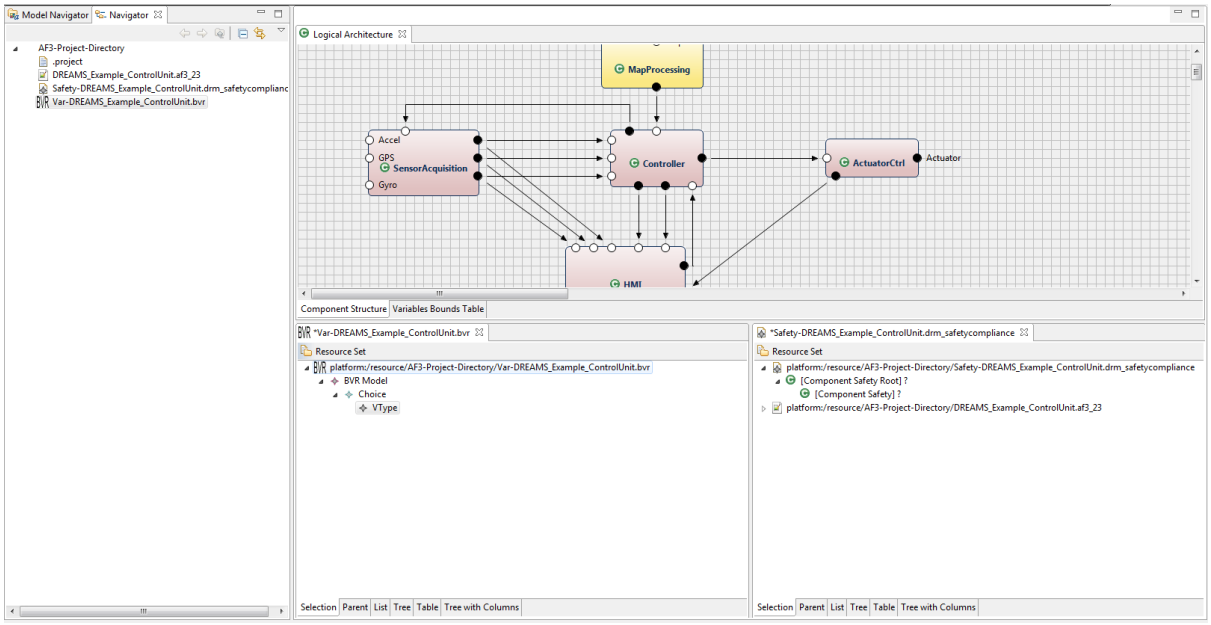


Figure 3.2: AutoFOCUS3 DREAM RCP, with BVR and Mixed Criticality Product Line Editor installed.

The descriptions of each the underlying meta-models will provide an overview of its interface to other meta-models, i.e. a list of the referenced entities. As pointed out in the introduction, small example instances of the meta-models will be presented in this document, whereas models of the DREAMS application demonstrators will be presented in deliverable D1.5.1.

3.2 AutoFOCUS3

3.2.1 Tool Summary

AutoFOCUS3[3] (AF3) is a tool based on the Eclipse Modelling Framework (EMF) [5] that supports the development of embedded systems based on the Focus modelling theory [6]. AF3 uses models in all development phases including requirements analysis, design of the logical architecture, platform architecture, implementation and deployment. Furthermore, AF3 features formal analyses and synthesis methods.

It should be noted that in the scope of DREAMS, the model of execution defined in the DREAMS architectural style (see D1.2.1) is used rather than the one defined by the Focus modelling theory. Hence, in the following, a number of fundamental AutoFOCUS3 meta-models will be described that provide the basis for the implementation of meta-models appropriate in a DREAMS context. In this document, the following meta-models provided by AF3 are used and extended to support the following viewpoints defined in Chapter 2.

- Logical viewpoint: In AutoFOCUS3, systems are described using component models of the software architectures which are enriched with specifications of the executable behaviour (see Section 3).
- Technical viewpoint: The execution platform is described using a topology model containing the corresponding hardware and software elements such as execution units, communication transmission units and endpoints (see Section 4).
- Deployment viewpoint: A mapping model is used to specify how an application (described in the logical viewpoint) is a mapped to the platform (see Section 5).

[3] http://af3.fortiss.org/

In the scope of DREAMS, AutoFOCUS3 provides the following:

- (Graphical) model editors for meta-models defined by the viewpoints listed above.
- Multi-objective design-space exploration (DSE) that supports the system architect in finding Pareto-optimal mappings of models of mixed-criticality applications to models of the DREAMS platform. A description of the implementation and the use of the DSE can be found in Chapter 3 of deliverable D4.1.2.

3.2.2 Installation

In order to allow for an easy installation of an AutoFOCUS3 for use in the DREAMS project, a dedicated Eclipse RCP application (based on Eclipse Kepler SR2) can be obtained by members of the DREAMS consortium[4] as follows:

3.2.2.1 System requirements

- An x86-based computer running a 32- or 64-bit version of Windows, Linux or MacOSX.
- Java Runtime Environment (JRE) version 8. This requirement actually stems from the BVR tool-set (see Section 3.2.6), AutoFOCUS3 runs on JRE version 6 and above.

3.2.2.2 Obtaining and Installing AutoFOCUS3 DREAMS Edition

- Go to https://download.fortiss.org/projects/dreams/af3/rcp/.
- Download and extract the archive matching your platform. This will create the `AF3-DREAMS` directory.
- Launch the platform-specific executable (e.g., `AF3-DREAMS/autofocus3-phoenix.exe` on Windows).
- Close the welcome screen.
- To install additional components (such as the Base Variability Resolution Tool, see Section 3.2.6), use the *Help → Install New Software* menu.

3.2.3 Overview of Tool Architecture

As pointed out in Section 3.2.1, AutoFOCUS3 is based on Eclipse. In order provide an overview on the AutoFOCUS3 modules that are relevant for this deliverable, the tool architecture will be briefly presented in the following. Each AutoFOCUS3 module consists of a plugin containing the functionality (e.g., a meta-model, an analysis, etc.) and user-interface (UI) plugin required to interact with the modules' functionality. Depending on the modules functionality, its UI contribution can consist of providing new model elements to the model element library, graphical model editors, or user interfaces to algorithms. Figure 3.3 shows the dependency graph of the plugin `eu.dreamsproject.platform.ui`. This plugin has dependencies to all AutoFOCUS3 plugins described in this deliverable.

[4] Since the AutoFOCUS3 distribution for use in the DREAMS project also contains contributions provided by confidential deliverables (e.g., from T4.2), and confidential example models from the application demonstrators, the download site is restricted to members of the DREAMS consortium. The login credentials may be obtained from the project internal web-based collaboration tool:

https://dreams.teams.uni-siegen.de/work_packages/wp01/Shared%20Documents/Working%20Document/D1.4.1/AutoFOCUS3%20(DREAMS%20Edition)%20Download%20Information

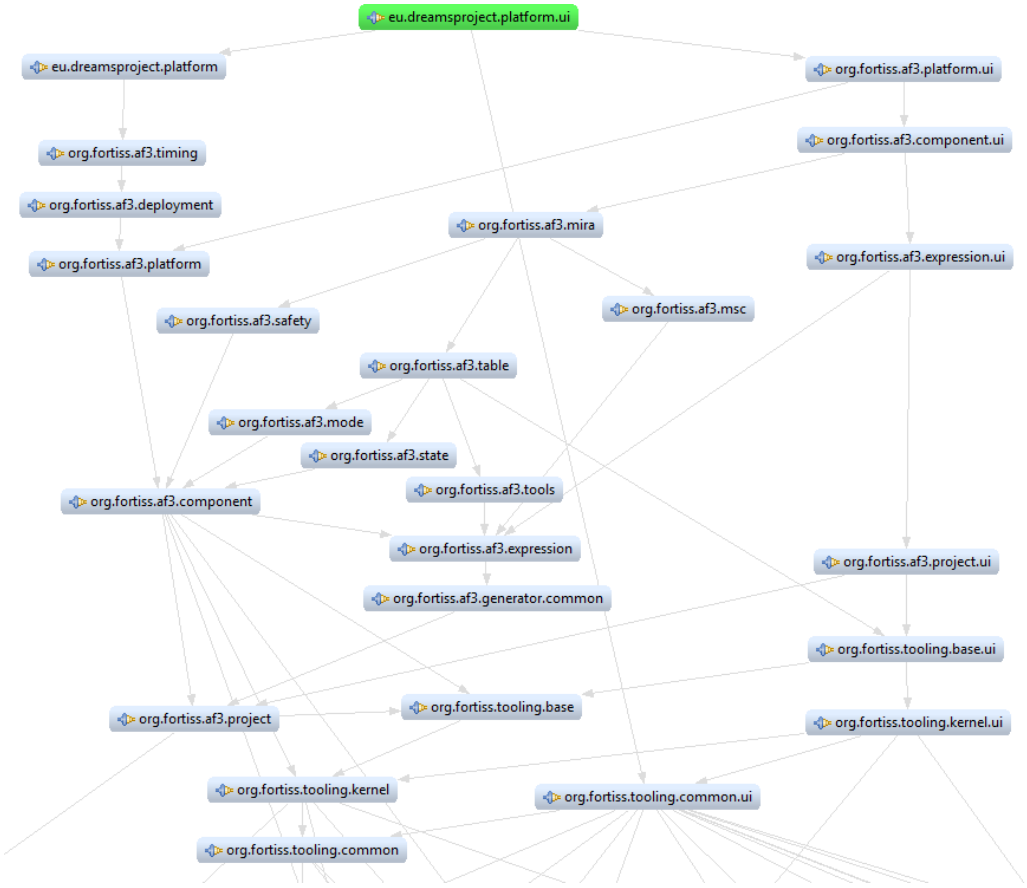


Figure 3.3: Dependency graph of the plugin eu.dreamsproject.platform.ui.

3.2.4 Getting Started With AutoFOCUS3

This section provides a brief "getting started" guide to the development of models using the graphical editors provided by AutoFOCUS3. More details on model editing and all further features can be found in the AutoFOCUS3 user guide[5].

Most graphical model editors use the hierarchical meta-model that will be introduced in Section 3.2.6.2 as backend. In the following, the steps required to create AutoFOCUS3 models will be illustrated. In AutoFOCUS3, new models are instantiated by creating a new AutoFOCUS3 project, as shown in Figure 3.4.

Figure 3.4: Creating a new AutoFOCUS3 project.

From the context menu of an AutoFOCUS3 project, new architectures, e.g. component architectures (see Section 4), can be created (shown in Figure 3.5).

[5] http://af3.fortiss.org/docs/ →User Documentation or the *Help → AF3 Help menu* in the AutoFOCUS3 RCP application.

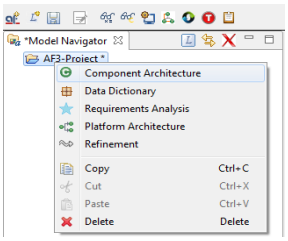


Figure 3.5: Creating a new Component Architecture

Hierarchical element models are usually edited using the diagram editor (see centre of Figure 3.6). On the left side of the user interface, the model navigator displays the AutoFOCUS3 workspace with all available projects and a tree view of the contained (hierarchical element) models.

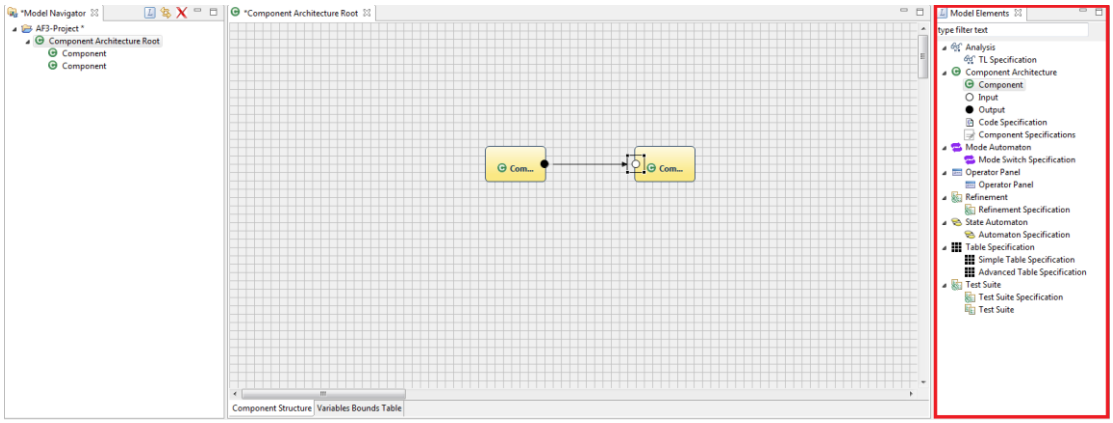


Figure 3.6: Screenshot of an AutoFOCUS3 Component Architecture.

New model elements can be added to the architecture by dragging them from the model element library (highlighted by the red rectangular in Figure 3.6) into the diagram editor. The model element dynamically updates the set of offered model elements (i.e., it only offers element that are syntactically compatible with the current model).
Likewise, the diagram editor enforces the syntactical correct composition of model elements. Connections (e.g., between `Ports` of logical `Components`) can be created by dragging from the source connector to the target connector with the *ALT*-key pressed. The editors of many meta-models also support to drag the connection directly between `Components` (the `Ports` are created automatically in this case).
All architecture models are edited using these diagram editors, which includes the DREAMS platform meta-model (see Figure 3.7).

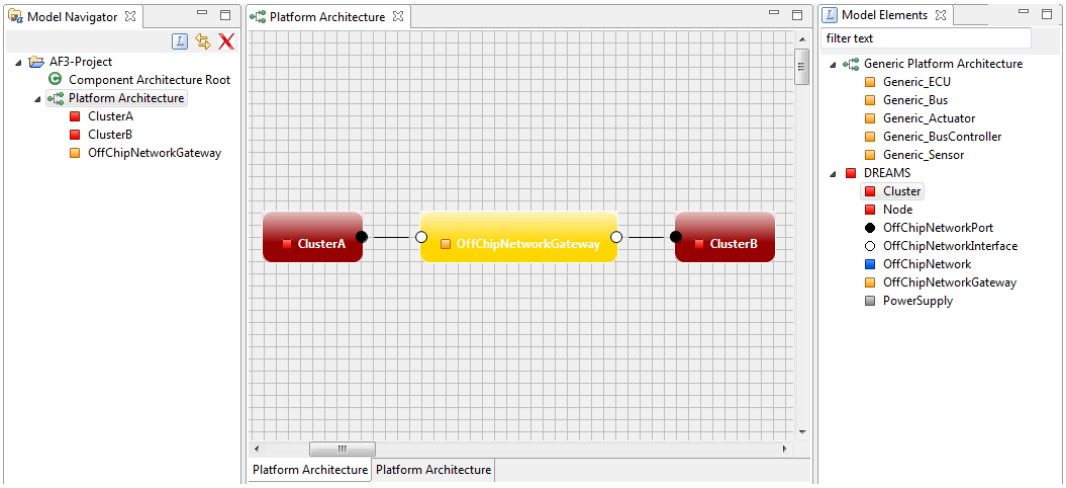


Figure 3.7: Screenshot of an AutoFOCUS3 Platform Architecture containing a DREAMS platform model.

3.2.5 Model Element Attributes

AutoFOCUS3 provides three different ways to model attributes of model elements. The purpose of this section is to give a general overview on these attribute specification mechanisms, and to briefly introduce the corresponding graphical views. A detailed description of the individual attributes can be found in sections that describe the corresponding part of DREAMS meta-model.

- *Properties:* intrinsic model element attributes (i.e., that always exist if the corresponding model element is instantiated). The *Properties View* provides a local view on all properties provided by the respective model element.
- *Specifications:* attributes that are explicitly added by the user to the respective model element. Specifications are indicated in the *Model Navigator* and usually provide dedicated editors.
- *Annotations:* model element attributes that always exist if the corresponding model element is instantiated. The *Annotation View* provides a global view on the annotations of all model elements of a project root element (e.g., a component architecture, see Section 4).

In the following, the different ways and the associated views will be described based on the simple component architecture shown below.

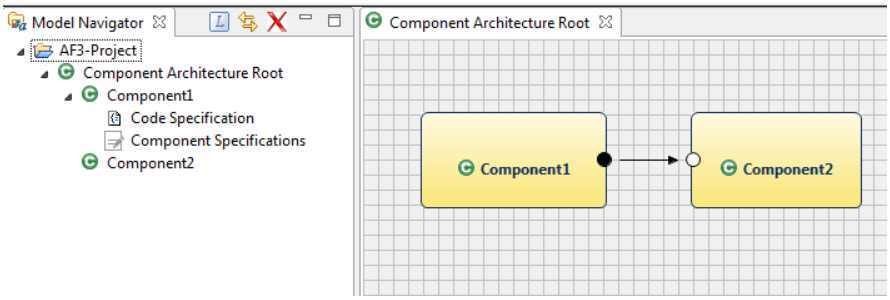


Figure 3.8: Example for different was of specifying model element attributes

The example is restricted to the attributes contributed by the component architecture plugin, i.e. Eclipse plugin implementing the logical component architecture. It should be noted that other plugins will contribute further attributes for *Component*s.

3.2.5.1 Properties

The *Properties View* provides a local view on all properties of the currently selected component. The following screenshot illustrates the properties of *Component1*.

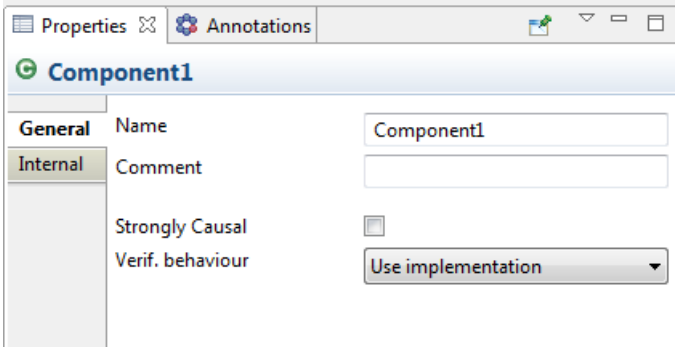


Figure 3.9: Model Element Properties

The following properties are always available (in the *General* tab) for any model element type (i.e., not restricted to *Components*:

- Name
- Comment

3.2.5.2 Specifications

Specifications are attributes that are derived from `IModelElementSpecification`. They can optionally be added to any `IModelElement` model element that is compatible with the respective specification type.

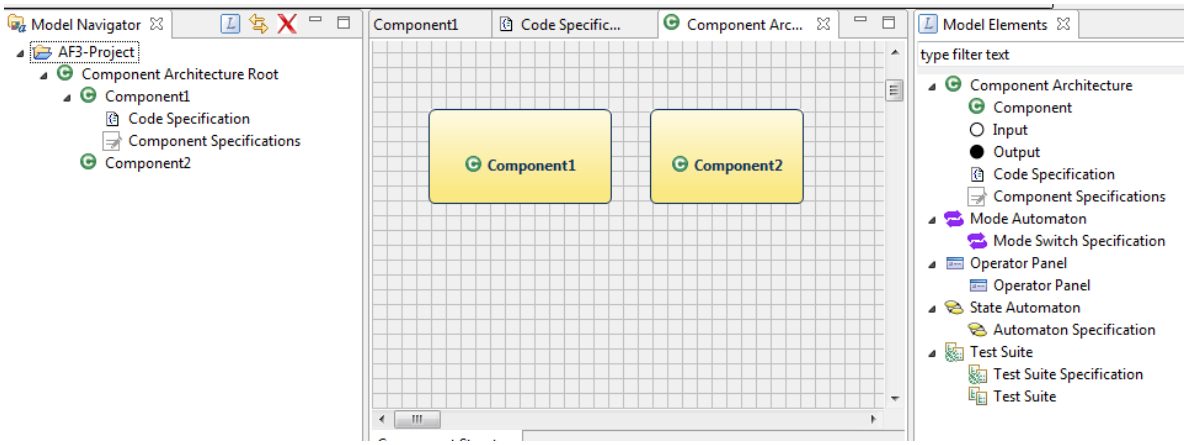


Figure 3.10: Model Element Specifications

The instantiation of `IModelElementSpecifications` can be performed in the following two ways:

- Programmatically, i.e. specifications may be attached to model elements during their construction or as the result of a computation that processes the corresponding model.
- Interactively, i.e. specifications can also be selectively added by the user. For this, the corresponding specification must be made available in the *Model Elements* library (see right-hand side of Figure 3.10). The specifications that have been added to model element are indicated in the *Model Navigator* (see left-hand side of Figure 3.10). A double-click on these specifications opens an editor dedicated to the respective specification type.

3.2.5.3 Annotations

Although the concept of specifications offers a lot of flexibility, it has the following drawbacks: On the one hand, the effort to create new specifications into the tool is relatively large. On the other hand, `IModelElementSpecifications` are not suitable for mandatory attributes since the user would have to explicitly add them to every model element. Also, mandatory attributes could be implemented by directly defining attributes in the corresponding classes. However, this approach would not scale well when sharing attributes between classes that are not in a direct inheritance relationship and would require a change of the meta-model in case attributes or their assignment to meta-model elements is changed.

In AutoFOCUS3, the concept of annotations provides an approach that (from the developer's perspective) simplifies and - as far as possible - automates the integration of additional (mandatory) attributes into the tool.

In the meta-model, annotations are derived from `IAnnotatedSpecification` and define one or more `EAttribute` or `EReference` to be annotated to `IModelElements`. For the

integration of the annotation (i.e., the instantiation for the matching model elements and the integration into the GUI) the following steps are required:

- Implementation of an annotation specific `IAnnotationValueProvider` (see below).
- Binding of the model element to be annotated with the corresponding `IAnnotationValueProvider` using the `annotation` Eclipse extension point. This registration ensures instantiation of the annotation and the integration into a tabular view (see Figure 3.12).

Annotations can be defined within any plugin that (also indirectly) imports `org.fortiss.af3.tooling.base`.

- To create simple annotations that provide a parameter of a primitive type as, a concrete class inheriting from both `IAnnotatedSpecification` and `IHiddenSpecification` must be created that contains the desired attribute(s). In this case, the value provider should inherit from `EStructuralFeatureValueProviderBase`.
- Annotations that do not contain attributes entered by the user, but values that are result of a calculation should inherit from the `DerivedAnnotation` base class and employ the `DerivedAnnotationValueProvider`.

In example given in Figure 3.11, the `MemorySize` annotation inherits from `IAnnotatedSpecification` (and `IHiddenSpecification`) and uses a value provider based on `EStructuralFeatureValueProviderBase`. The code example in the lower right corner of the figure illustrates the registration `MemorySizeValueProvider` for all elements of type `eu.dreamsproject.platform.model.RAM` with the `annotation` extension point (typically in the `plugin.xml` file of the plugin where the corresponding `IAnnotatedSpecification` is declared).

From the users' perspective, the *Annotation View* (see Figure 3.12) provides a global view on all model element annotations within the current project root element (e.g., within a *Component Architecture* or a *Platform Architecture*). In the view, each model element is represented as a row. The row for the model element that is currently selected in the associated model diagram editor is highlighted with a green background.

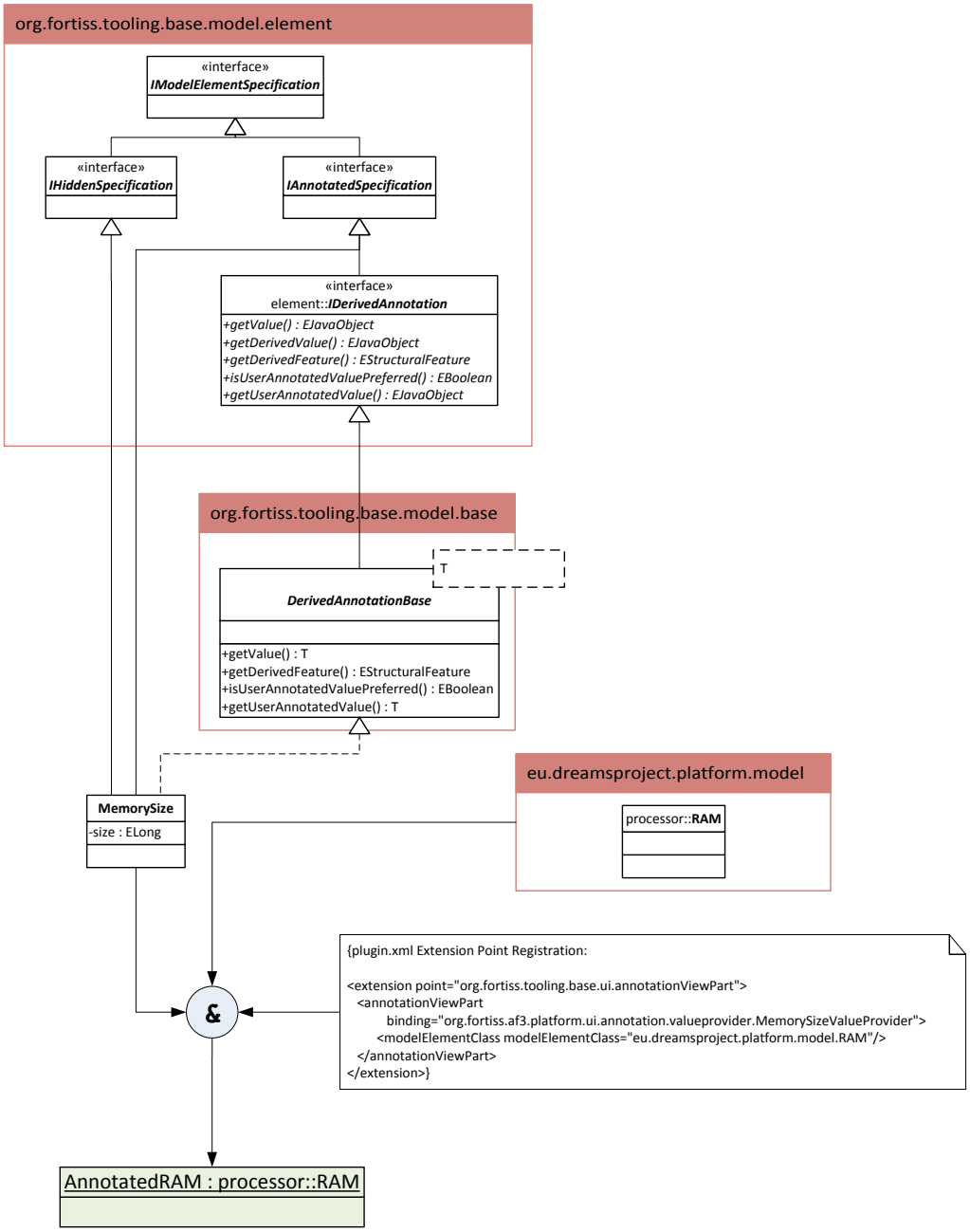


Figure 3.11: Integrating a new annotation type (example: `MemorySize` annotation)

The view's columns represent the annotations that are associated with the respective model element. Here, the following three cases can be distinguished:

- White cell - the model element contains the respective annotation that is *editable by the user*.
- Grey cell - the model element *does not contain* the respective annotation.

Blue cell - the model element contains the respective annotation. However, its value is the result of a calculation (and hence the cell is read-only).

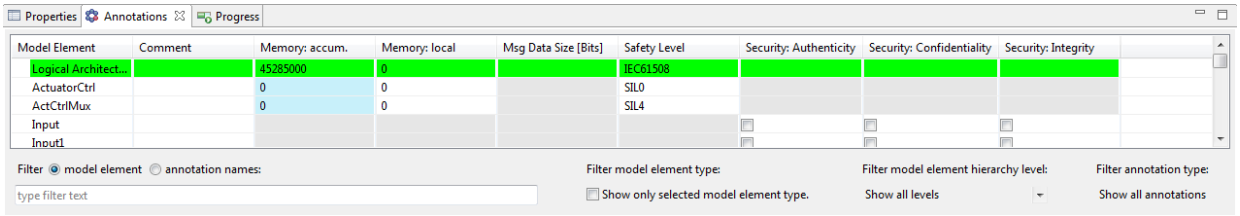


Figure 3.12: Model Element Annotation View

Like the *Properties View*, the *Annotation View* provides the following two annotations for any model element.

- Name
- Comment

In fact, the *Name* and *Comment* annotations provide an alternative way to access the corresponding properties.

Again, different plugins may contribute annotations to different model element types. In the example, the component architecture plugin contributes the following two annotations to *Component*s:

- *Memory: local*: Memory need of a component (annotated by user - white cell).
- *Memory: accumulated*: Memory of a component and all children (calculated based on *Memory: local* annotation - blue cell).

It can be seen that the memory annotations are not contributed to the *Output port* (grey cell).

At the bottom, the *Annotation View* provides a number of row and column filters.

Here, the following *row filters* can be used to restrict the set of model elements that is shown in the view:

- *Filter model element name*: only model elements are shown whose name matches the filter string.
- *Filter model element type*: if checked, only model elements are shown that have the same type as the model element that is selected in the associated model diagram editor.
- *Filter model element hierarchy level*: Filters the set of model elements based on the model structure. The following options are available:
    - *Show all levels*: Any model element beneath the currently selected project root element is shown.
    - *Show current level*: Only model elements are shown that have the same hierarchy level as the currently selected model element.
    - *Show selected sub-model*: The currently selected sub-model and its entire offspring is shown.

The following *column filters* can be used to restrict the set of annotations that is shown in the view:

- *Filter annotations name*: only annotations are shown whose name matches the filter string.
- *Filter annotation type*: Either all annotation types or only annotations of the selected type are shown.

3.2.6 Fundamental Meta-Models

In the following, a number of meta-models will be presented that provide the basis for all AutoFOCUS3 based meta-models.

The following description (and also the description of all other AutoFOCUS3-based meta-models) is structured as follows:

- A table lists general information on the meta-model
  - Brief description
  - Name of EMF Ecore-File in which the meta-model is defined
  - Name of Eclipse-plugin which hosts the meta-model in an AutoFOCUS3 installation. As pointed out in Section 3.2.3 in more detail, for each meta-model the corresponding graphical editor is provided by a companion plugin `<name of model plugin>.ui`.
  - Java base package of the classes defined by the meta-model.
- A UML class diagram visualizing the meta-model
- A description of the meta-models classes (and their attributes)

While users of the AutoFOCUS3 tool cannot directly create instances of these meta-models, for them mainly the attributes defined in the base classes described in this section are relevant.

For developers who contribute new AutoFOCUS3-based meta-models, or who develop algorithms based on the DREAMS meta-model, these fundamental meta-models provide a generic and abstract interface to process DREAMS models, which – for most meta-models – includes a number of classes with utility methods to process the given meta-model (in Java package `<name of model plugin>.utils`.)

3.2.6.1 AutoFOCUS3 Kernel Meta-Model

The AutoFOCUS3 kernel meta-model provides fundamental modelling entities that are shared between meta-models of all AutoFOCUS3 meta-models.

Table 3.1 provides an overview:

Name	**AutoFOCUS3 Kernel Meta-Model**	
Description	The goal of the kernel meta-model is to provide fundamental modelling entities shared between meta-models of arbitrary viewpoints	
Ecore file	`kernel.ecore`	
Plugin	`org.fortiss.af3.kernel`	
Packages	`org.fortiss.af3.kernel.model`	Kernel meta-model
Dependencies	N/A	

Table 3.1: AutoFOCUS3 Kernel Meta-Model (overview)

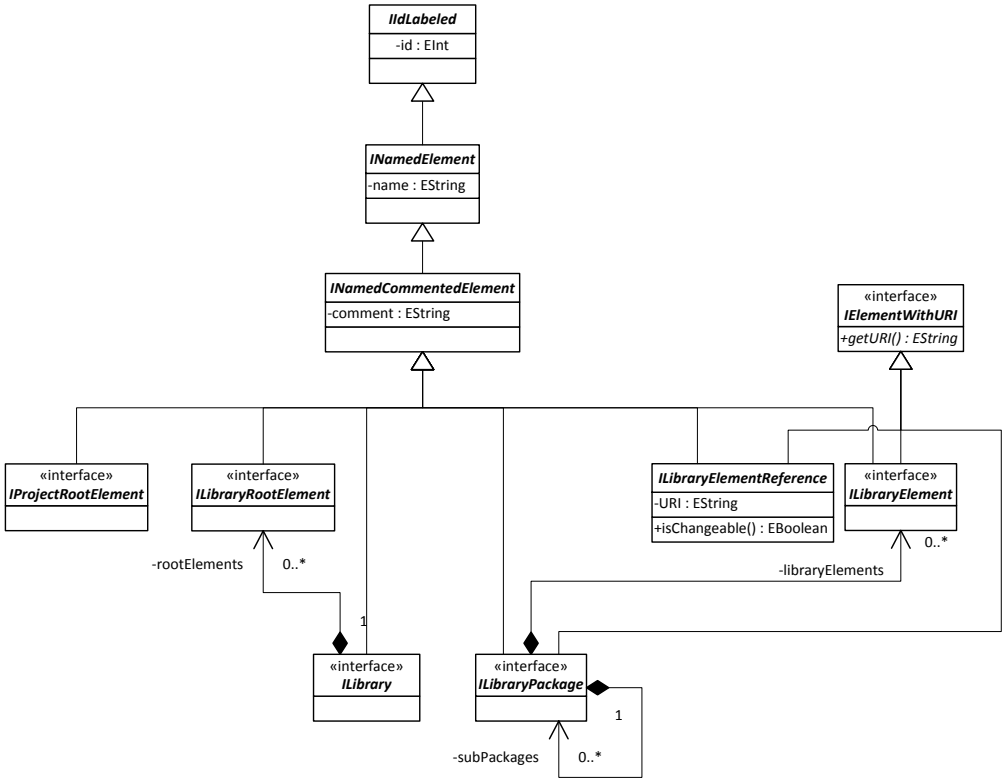


Figure 3.13: AutoFOCUS3 Kernel Meta-Model (UML Diagram of package `org.fortiss.tooling.kernel.model`)

The meta-model consist of a single package `org.fortiss.af3.kernel.model` that contains following classes (see Figure 3):

- `IIdLabeled`
  - Model elements implementing this interface have a unique identifier.
  - **Attributes:**
    - `id`: The element's ID
- `INamedElement`
  - Model elements implementing this interface have a unique id and a name.
  - **Attributes:**
    - `name`: The elements name
- `INamedCommentedElement`
  - Model elements implementing this interface have a unique id, a name and a comment.
  - **Attributes:**
    - `comment`: A comment describing the model element.
- `IElementWithURI`
  - Elements that can be referenced using a URI.
- `IProjectRootElement`
  - Super class of all root elements contained in the project.
- `ILibraryRootElement`
  - Super class of all root elements contained in the libraries.
- `ILibrary`
  - Super class of all libraries.
  - **Attributes:**
    - `rootElements`: Root elements of the library

- `ILibraryPackage`
  - Super class of all library package (that can contain library elements and/or sub-packages)
  - **Attributes:**
    - `libraryElements`: Elements contained in this package
    - `subPackages`: Sub-packages of this package
- `ILibraryElementReference`
  - Super class of all references to elements contained in a library.
- `ILibraryElement`
  - Super class of all elements contained in a library.
  - **Attributes:**
    - `wrappedElement`: Element referenced by this library entry

3.2.6.2 AutoFOCUS3 Hierarchic Element Meta-Model

The AutoFOCUS3 hierarchic element meta-model provides the basis for hierarchical meta-models. It is based on the on the AutoFOCUS3 kernel meta-model (see Section 3.2.6.1).

In this document, the meta-models for the following view-points are based on this meta-model.

- Logical viewpoint (see Section 3).
- Technical viewpoint (see Section 4).
- Mapping meta-model of deployment viewpoint (see Section 5).
- Temporal viewpoint (see Section 7)

Table 3.2 provides an overview of the AutoFOCUS3 Kernel Meta-Model.

Name	**AutoFOCUS3 Hierarchic Element Meta-Model**	
Description	The goal of the hierarchic element meta-model is to provide the basis for hierarchical models (e.g., component models)	
Ecore file	`base.ecore`	
Plugin	`org.fortiss.af3.tooling.base`	

Packages	`org.fortiss.af3.tooling.base.model.element`
	`org.fortiss.af3.tooling.base.model.base`
	`org.fortiss.af3.tooling.base.model.layout`
		Hierarchic Element MM interface
		Base classes for concrete MMs
		GUI layout information store

Dependencies	`org.fortiss.tooling.kernel`	

Table 3.2: Hierarchic Element Meta-Model (overview)

The meta-model consists of three packages that will be described in the following.

3.2.6.2.1 Hierarchic Element Interface

The package `org.fortiss.af3.tooling.base.model.element` defines the interface of the hierarchic element meta-model (see Figure 3.14) that is shared with derived concrete meta-models. It can be used by (plugin) developers to interface with AutoFOCUS3-based meta-model in a generic way.

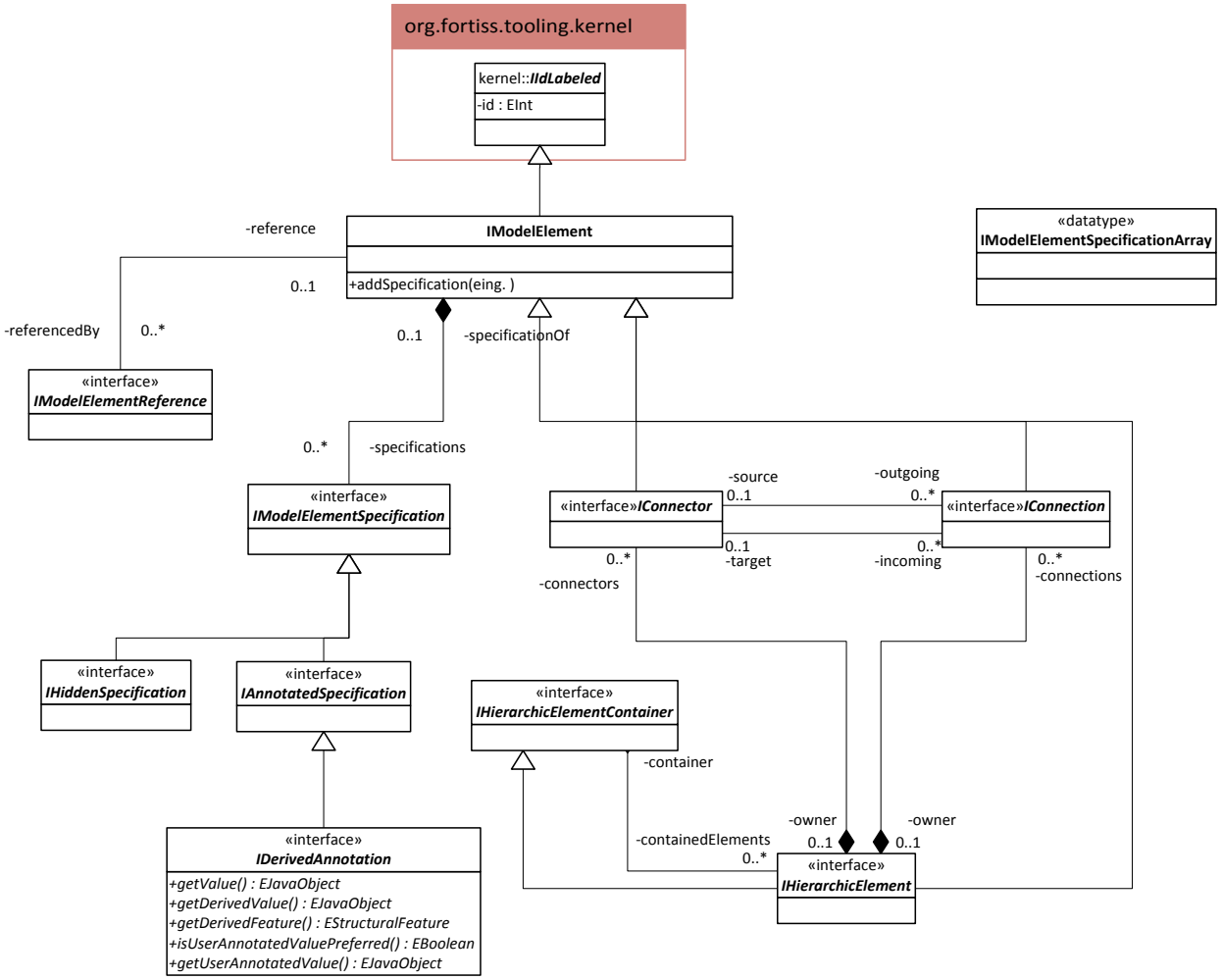


Figure 3.14: Hierarchic Element Meta-Model (UML Diagram of Interface package `org.fortiss.tooling.base.model.element`)

It contains the following classes:

- `IModelElement`
  - Super class of first class model elements.
  - **Attributes:**
    - `specifications`: List of model element specifications providing additional model element properties (see Section 3.2.5).
    - `referencedBy`: List of model element references.
  - **Operations:**
    - `addSpecification(IModelElementSpecification)`: Adds an `IModelElementSpecification` to the given `IModelElement`
- `IModelElementReference`
  - Super class of `EObject`s referencing model elements.
  - **Attributes:**
    - `reference`: The referenced model element.

- `IModelElementSpecification`
  - Super class of model element specifications. Such specifications provide additional pluggable properties.
  - **Attributes:**
    - `specificationOf`: The `IModelElement` which owns this specification
- `IHiddenSpecification`
  - Super class of model element specifications, which should be excluded from the navigator view.
- `IAnnotatedSpecification`:
  - Super class of model element specifications that represent annotations (i.e., specifications that are guaranteed to exist exactly once for the model elements for which the annotation has been registered).
- `IDerivedAnnotation`
  - Interface for `IAnnotationSpecifications` that are derived from the state of other annotations and/or model elements.
    - Concrete specifications must provide a specialized `getValue()` `EOperation`s that perform the required calculation.
    - Concrete specifications may provide additional `EOperation`s that provide an advanced query interface to the annotation.
    - The corresponding `IAnnotationValueProvider` should be based on `DerivedAnnotationValueProviderBase`.
  - **Operations:**
    - `getValue()`:Wrapper method for returning derived (calculated) values. It may return values annotated by the user if the calculation fails, or the user input is preferred, based on the configuration of the concrete annotation.
    - `getDerivedValue()`: Returns the actual derived (calculated) values. It may return values annotated by the user if the calculation fails, or the user input is preferred, based on the configuration of the concrete annotation.
    - `getDerivedFeature()`: Returns the `EStructuralFeature` that stores the annotation of the model element associated with this `IDerivedAnnotation`. Returns `null` if no element specific behaviour is desired.
    - `isUserAnnotatedValuePreferred()`:Default implementation of a method indicating whether the user annotated value, if available, shall be preferred over the derived one. The default is `true`, i.e. user annotated values are preferred. Shall be overridden, if a different behaviour is desired.
    - `getUserAnnotatedValue()`:Returns the `EAttribute` which is used to store the user annotated values, if the concrete annotation is designed therefore. By default, this function returns `null`, indicating no user annotated values are supported. This method must be re-implemented by concrete annotations if any other behaviour is desired.
- `IModelElementSpecificationArray`
  - An array of model element specifications.
- `IConnector`
  - Super class of connectors. Connectors reference incoming and outgoing connection model elements.
  - **Attributes:**
    - `incoming`: The incoming connections.
    - `outgoing`: The outgoing connections.
    - `owner`: The `IModelElement` which owns this connector

- `IConnection`
  - Super class of connections. Connections are aggregated in a hierarchic model element and reference two connectors from that element or any direct sub-element.
  - **Attributes:**
    - `source`: The connection's source connector.
    - `target`: The connection's target connector.
    - `owner`: The `IModelElement` which owns this connection
- `IHierarchicElementContainer`
  - Super class of hierarchic model elements.
  - **Attributes:**
    - `containedElements`: The contained hierarchic model elements.
- `IHierarchicElement`
  - Super class of hierarchic model elements.
  - **Attributes:**
    - `connections`: List of aggregated connection model elements. Usually a hierarchic element aggregates all connections of its direct sub-structure.
    - `connectors`: List of aggregated connectors.
    - `container`: The container which this element belongs to.

3.2.6.2.2 Hierarchic Element Base Classes

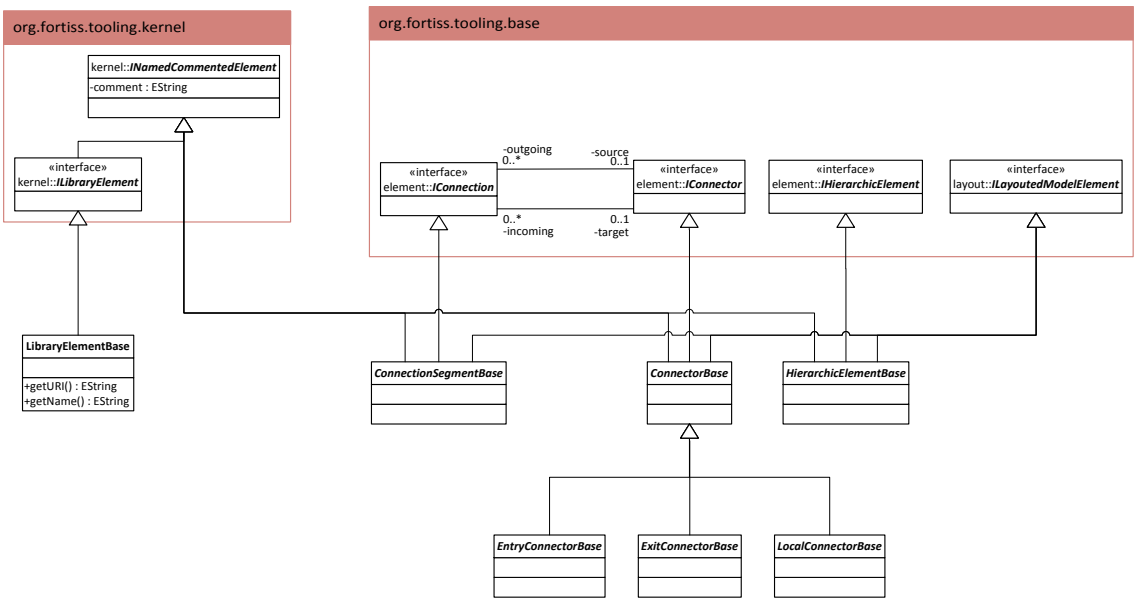


Figure 3.15: Hierarchic Element Meta-Model (UML Diagram of base class package `org.fortiss.tooling.base.model.base`)

The package `org.fortiss.af3.tooling.base.model.base` (see Figure 3.15) provides base classes that implement the interface defined in `org.fortiss.af3.tooling.base.model.element`. They are used as a basis for concrete hierarchical element meta-models such as the logical (see Section 3), technical (see Section 4) and the mapping meta-model provided by the deployment viewpoint (see Section 5).

The package defines the following classes:

- `LibraryElementBase`
  - Base class for members of the model element library

- `HierarchicElementBase`
  - Base class for hierarchic model elements
- `ConnectorBase`
  - Base class for connectors
- `EntryConnectorBase`
  - Base class for connectors with incoming connections
- `ExitConnectorBase`
  - Base class for connectors with outgoing connections
- `LocalConnectorBase`
  - Base class for connectors with local connections
- `ConnectionSegmentBase`
  - Base class for connections

3.2.6.2.3 Hierarchic Element GUI Layout Data Store

The package `org.fortiss.af3.tooling.base.model.layout` (see Figure 3.16) provides a data store for graphical representations (see Section 0) of hierarchic element models that are based on `org.fortiss.af3.tooling.base.model.element`.

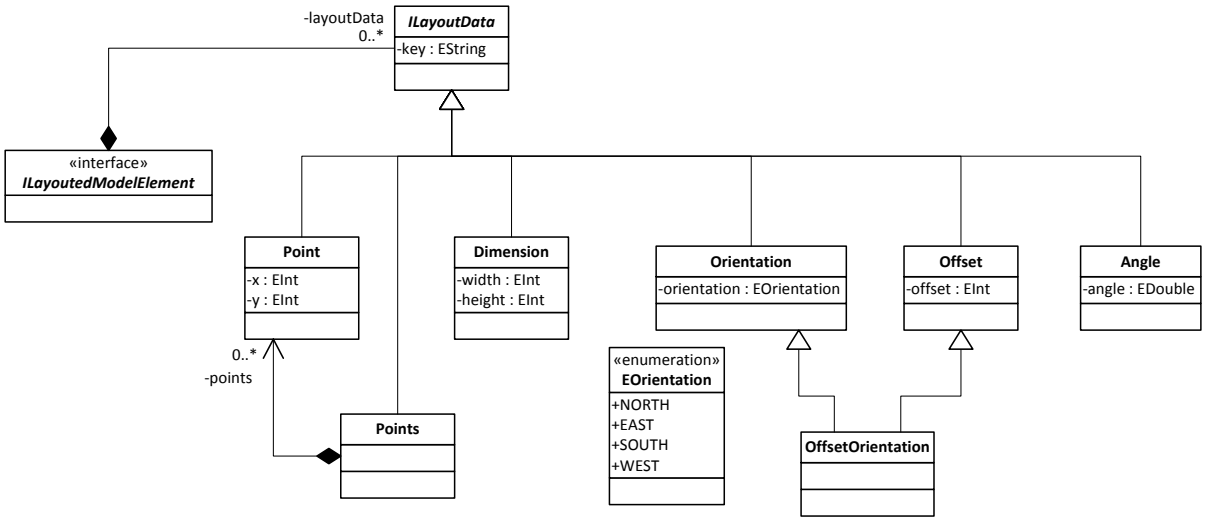


Figure 3.16: Hierarchic Element Meta-Model (UML Diagram of Layout Data Store package `org.fortiss.tooling.base.model.layout`)

- `ILayoutedModelElement`
  - Super class of model elements with layout data.
  - **Attributes:**
    - `layoutData`: Stores the aggregated layout data.
- `ILayoutData`
  - Super class of all layout data objects.
  - **Attributes:**
    - `key`: The layout key indicating how the layout data is to be interpreted.
- `Point`
  - Layout data for 2D locations.
  - **Attributes:**
    - `x`: The horizontal X coordinate.
    - `y`: The vertical Y coordinate.

- `Points`
  - Layout data for a sequence of 2D locations.
  - **Attributes:**
    - `points`: The aggregated locations.
- `Dimension`
  - Layout data for 2D dimensions.
  - **Attributes:**
    - `width`: Object width
    - `height`: Object height
- `EOrientation`
  - Enumeration of 2D directions and orientations.
- `Orientation`
  - Layout data for 2D orientations.
  - **Attributes:**
    - `orientation`: The orientation
- `Offset`
  - Layout data of a single dimensional offset.
  - **Attributes:**
    - `offset`: The offset value.
- `OffsetOrientation`
  - Combines an offset with an orientation.
- `Angle`
  - Layout data for an angle (the interpretation of the double value is application dependent).
  - **Attributes:**
    - `angle`: The double value of the angle.

3.3 Base Variability Resolution (BVR) Tool

The *Base Variability Resolution* (BVR) [7] helps manage the variability that emerges when using domain specific modelling languages (DSML). Expressing variability explicitly at the model level enables the construction of product lines while preserving the compatibility with existing domain specific tooling including model editors, model analyses, model transformations, and eventually code generators. By fostering reuse, product lines reduce waste, improve quality, and shorten time-to-market.

We summarize below the key features of the BVR tool chain from the user perspective. We further detail the underlying data model, which captures variability, and show how one can define and resolve variation points. We refer the interested reader to the official documentation [8] for a comprehensive treatment.

3.3.1 Tool Summary

BVR is developed on the top of the Eclipse platform. Although Eclipse is primarily an integrated development environment (IDE) targeting Java and web development, it provides a very general platform to build and integrate various textual and graphical editors in a single IDE.

BVR takes advantage of the *Eclipse modelling framework* (EMF)[6] [5], which is the *de facto* standard for MOF-based tooling. By providing a standardized implementation of the MOF specification [9],

[6] See http://eclipse.org/modelling/emf/

EMF makes BVR compatible with a large community of model-based tools. EMF also provides convenient facilities such as, standard tree-based editors or XMI and XML serialization to name a few.

The BVR Tool Bundle consists of the numerous plugins which support a variability engineer in defining variability model. The following plugins are the most essential to enable specification of variability:

- *BVR Model* – BVR meta-model
- *BVR VSpec MVC Editor* – provide editor to define *VSpec* tree also known as feature tree or variability abstraction
- *BVR Resolution MVC Editor* – resolution editor to define an instance of your *VSpec* tree, i.e. define a product configuration by selecting desirable features of the future product
- *BVR MVC Realization Editor* – realization editor allows an engineer to define how abstract feature from *VSpec* are actually realized in the model. Further, one may define fragment substitution operations to specify model modification to yield a product
- *SPLCATool* – set of tools to generate an optimal set of products to perform efficient testing in software product lines
- *BVR :: Thirdparty* – contains interfaces and default implantation to enable cooperation BVR with third-party tooling

3.3.2 Installation

3.3.2.1 Prerequisites

- The BVR tool bundle is tested against Eclipse Kepler, which is available at: http://eclipse.org/downloads/packages/eclipse-modeling-tools/keplerr
- The tool bundle is also tested against AutoFOCUS3 DREAMS edition (see Section 3.2.2).
- Java 8 is required to run the bundle (the plugins should work on Linux as well as Windows).
- The BVR Papyrus related plugins do not support the recent version of Papyrus, thus one should ensure to run Papyrus 0.9.

3.3.2.2 BVR Tool Bundle Update Site

- The BVR Tool Bundle update site can be found at: http://bvr.modelbased.net/update/site.xml
- The detailed footage of the installation process is available from here: http://bvr.modelbased.net/installation.swf

3.3.3 Getting Started with BVR Tool Bundle

- The following demo explains and exhibits BVR concepts, editors and variability process as well as integration with third-party tools: http://bvr.modelbased.net/demo/
- This short demo shows briefly VSpec and Resolution editors: http://bvr.modelbased.net/demo2/demo.swf
- An example of the variability definition on UML based models and Papyrus integration is available at: http://bvr.modelbased.net/demo3/demo.htm
- Integration between BVR and AutoFOCUS3 DREAMS edition are shown at: http://bvr.modelbased.net/af3bvr/af3bvrdemo.htm

3.3.4 Visualization of Variability Models

BVR provides three main visualisations for the variability models: namely the Ecore tree editor, the table view and the feature tree view. We illustrate below these three views and provide simple guidelines for when to use each of these.

- The ***Ecore Tree-based View***, is not provided by BVR *per se*, but is available to browse (and modify) any well formed Ecore model. It presents the objects that compose a given model, layout as a tree following the containment relationship. As BVR models are themselves Ecore models, it is possible to visualize and edit them using this tree-based view. This can be a convenient way to carry on surgical updates in corrupted models for instance, but will not provide any support for variability-related tasks, such as derivation of validation (see Section 9.2.2).

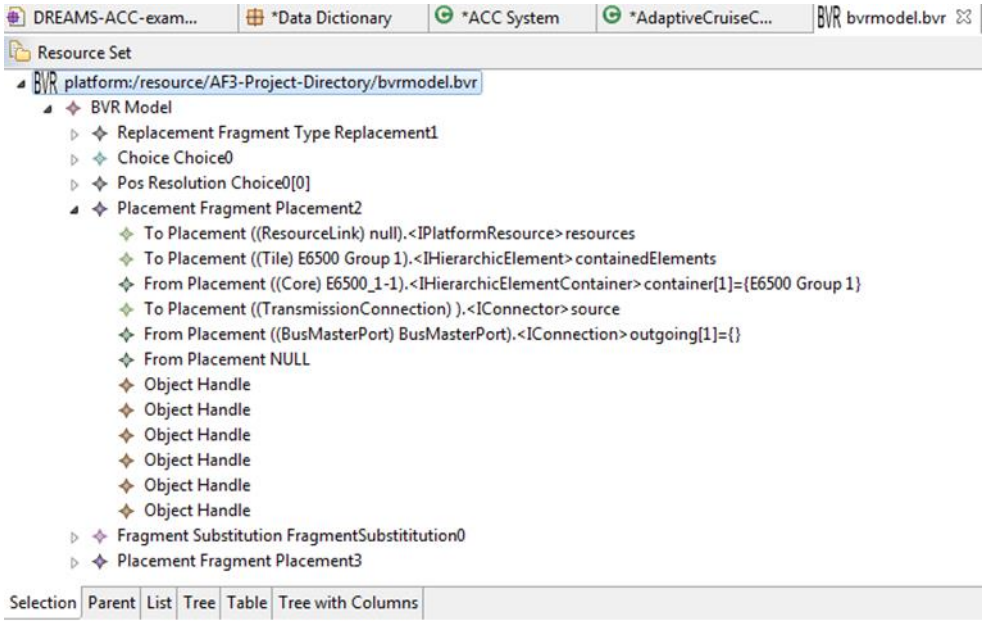


Figure 3.17: BVR Tree View

- The ***Feature Tree View*** is the default editor recommended to build, edit and visualize BVR models. Variation points are layout as a tree, which reflects the primary decomposition chosen to capture variability. Figure 3.18 below shows a simplified variability models for the DREAMS platform depicted as a feature tree. By contrast with the Ecore tree-based view where every object described in the BVR meta-model is shown, only user level concepts are represented here.

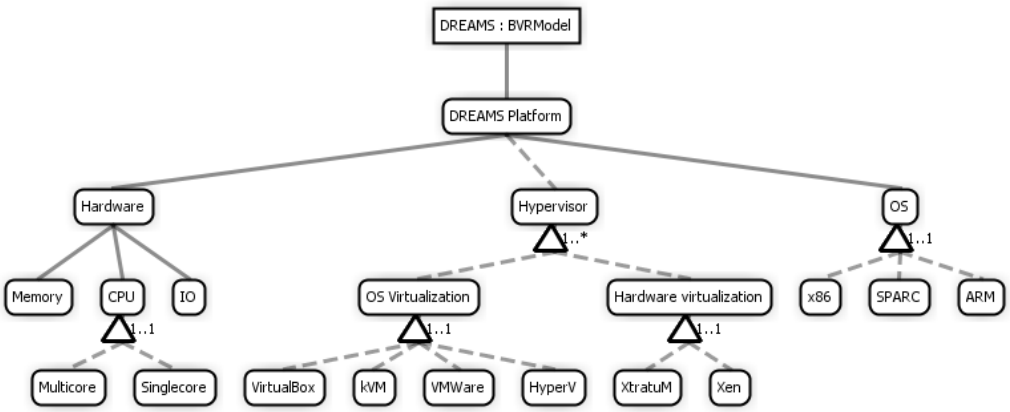


Figure 3.18 Feature tree of the variability inherent to the DREAMS platform,

3.4 Mixed-Criticality Product Line Editor

3.4.1 Tool Summary

The Mixed-Criticality Product Line Editor is tool that helps to define the safety related concepts of the system. The tool is composed by the following editors:

- The **IEC61508 and Diagnostic and Measures Safety Standard editor**: this editor allows the representation of a usable model of the IEC61508 standard and a subset of its Diagnostics and Measures Techniques (tables A.2 to a A.14 and A.15 to A.17), allowing to add safety attributes to Component, Platform and System Software entities that have safety requirements.
- The **Safety Compliance editor**: this editor allows defining Safety Compliance models based in IEC61508-2 and IEC61508-3 standard, allowing adding safety attributes to Component, Platform and System Software entities that have safety requirements.
- The **Safety Compliance Variability editor** (based on BVR) of the Safety Compliance editor: this editor allows defining variability in the Safety Compliance models defined with the previous editor.

Safety Compliance models will be used to check correctness of the system from the safety point of view.

3.4.2 Installation

The Mixed-Criticality Product Line Editor is bundled with the AutoFOCUS3 DREAMS Eclipse RCP application (see Section 3.2.2 for installation notes).

3.4.3 IEC61508 and Diagnostic and Measures Safety Standard editor

This editor allows defining IEC61508 based standards with SIL or ASIL (ISO26262) integrity levels for each one. For each standard, its *SafetyIntegrityLevels (SILx)* are defined, and for IEC 61508 its SCy levels are also defined, as shown in Figure 3.19:

In addition to this, the editor allows defining a model of the diagnosis techniques and measures for IEC 61508 safety standard.

The root of the hierarchy starts with the Starting from the *IEC 61508 Based Safety Standard node*. Apart from the SIL and SC levels described above, t, the user can create one of these entities:

- Technique Folder
- Technique Table
- Technique Item Description
- Random Failure Technique
- Systematic Failure Technique

For each *Technique Item Description,* the following properties can be defined:

- Name: Description of the item
- Notes: Additional notes about the item
- See IEC61508-7: Reference to the IEC61508-7 item, which describes in deeper detail the technique or measure.

Figure 3.20 shows that Techniques and Measures have been represented (Techniques & Measures node) and tables of Annex A of the IEC 61508-2 tables A.2 to A.17 have been defined. IEC 61508-3 part would also be interesting but goes beyond the scope of current definition of safety meta-model, although deployment of SW components into Partitions, and Partitions into Hypervisors, and Hypervisors into Tiles is checked (all these elements are SCItems with a Safety Manual defining at least claimed SILx and SCy). Anyway, in depth extension of safety meta-model to IEC 61508-3 in the future would be an interesting work.

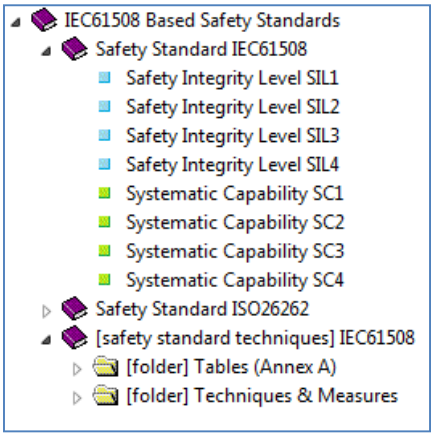


Figure 3.19: Safety Integrity Levels and System Capability Levels

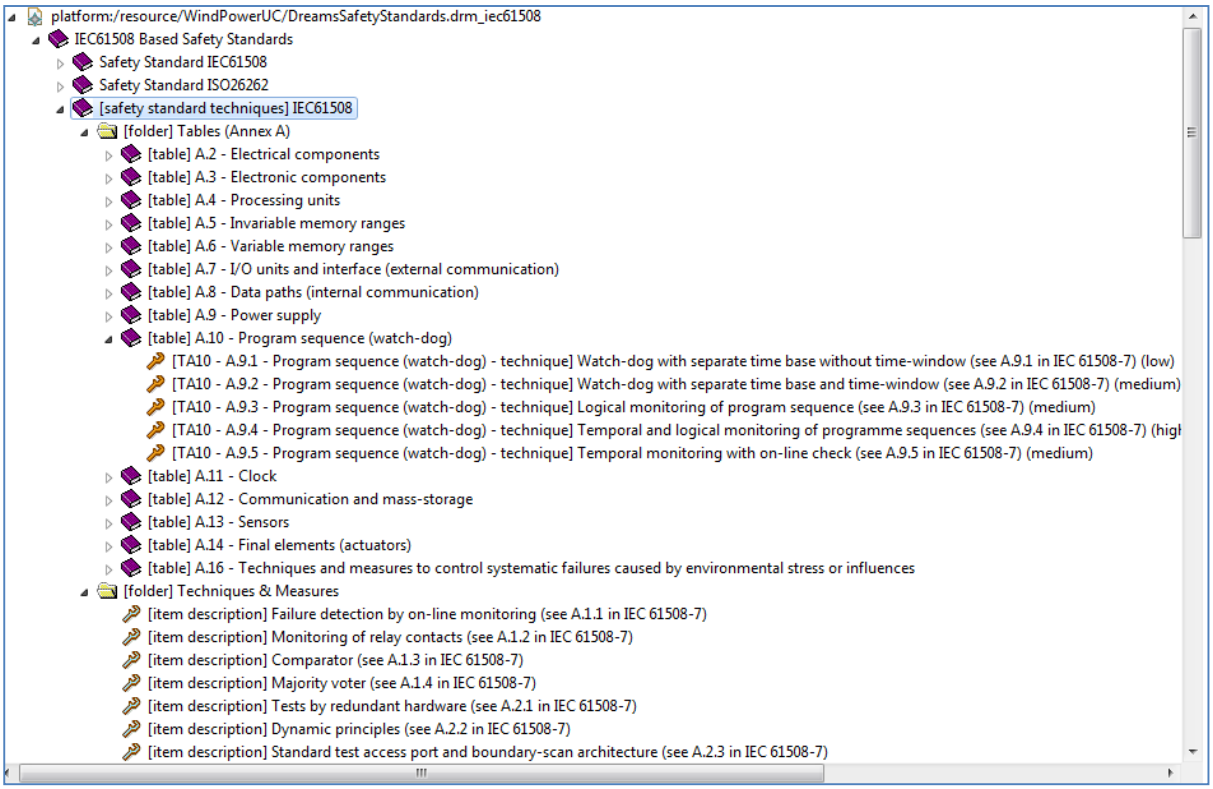


Figure 3.20: Safety Standard Techniques of IEC 61508-2 Annex A

For each technique, its main properties are also defined. Figure below show the properties of a given technique (A9.1) for Table A.10 Program Sequence of IEC61508-2. For example, each *Random Failure Technique* has the following properties (see Figure 3.21):

- The Table it belongs to.
- The maximum *diagnostic coverage* that can be achieved with this technique.

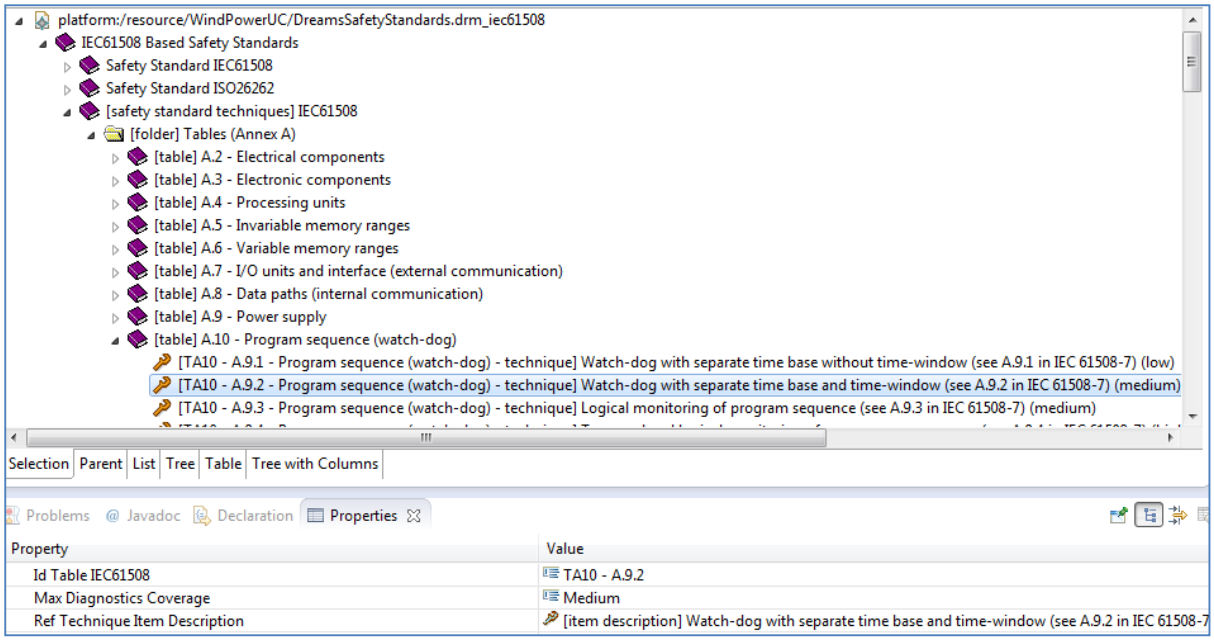


Figure 3.21: Example of Random Failure Technique

And also, for each Systematic Failure Technique (as shown in Figure 3.22), the following properties:

- Group: The user can define if this technique is mandatory or is part of a group. Two groups are available:
  - AtLeastOneBlackShaded.
  - AtLeastOneGreyShaded.

At least one technique of each group must be selected.

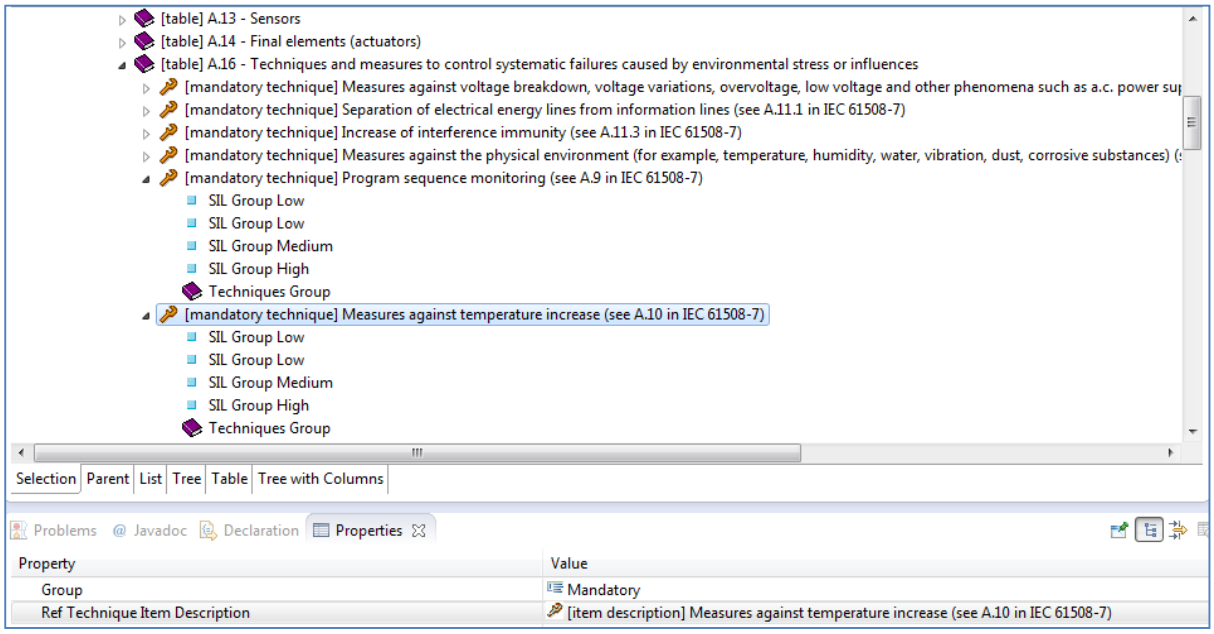


Figure 3.22: Example of Systematic Failure Technique

And also the effectiveness and importance in function of the SIL level of the entity, as shown in Figure 3.23.

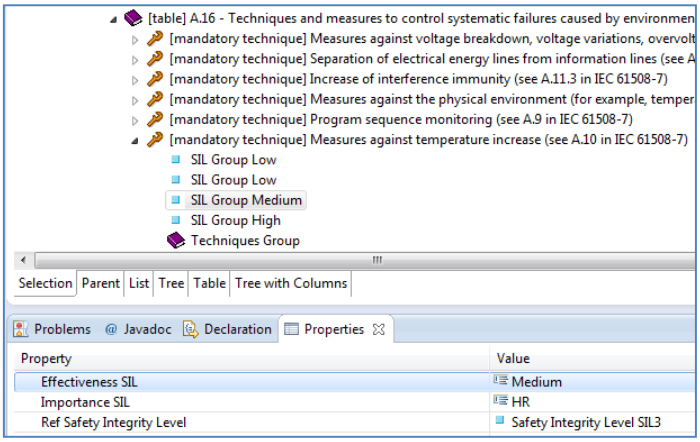


Figure 3.23: Effectiveness for a SIL group

To summarise, this model allow representing IEC61508 standard in a usable way so that safety concepts can be associated to Component, Platform and System Software entities, allowing to check safety consistency of the model being developed.

3.4.4 Safety Compliance Model Editor

3.4.4.1 Toolset Summary and Functionality

This editor allows defining a hierarchy Safety Compliant Items (SCI), and a 'Safety Manual' of each SCI (SCItem). A SCItem can represent one of the following DREAMS entities:

- The Root representing the whole system (HW/SW)
- (HW) DREAMS Node
- (HW) DREAMS Cluster
- (HW) DREAMS Tile
- (SW) DREAMS Hypervisor
- (SW) DREAMS Partition
- (SW) DREAMS Component

Each of these SCItems may a Safety Manual (to be described later). The hierarchy partially mimics the structure of a DREAMS system model (see Section 2.2). Figure 3.24 shows the very root of a Safety Compliance Model.

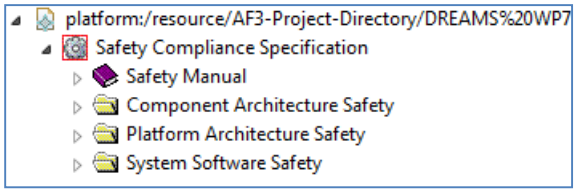


Figure 3.24: Safety Compliance model main nodes

The root consists of a *Safety Compliance Specification* entity, the very root of the model. The specification is composed by:

- A *Safety Manual* (to be described later)
- A *Component Architecture Safety* Folder
- A *Platform Architecture Safety* Folder
- A *System Software Safety* Folder

A description of the above folders follows in the next sections.

3.4.4.2 Component Architecture Safety Folder

Under this folder, safety specification of components (DREAMS software Component entities) is given. It is important to note that this model contains only safety relevant software components (i.e., software components which are part of safety functions and have defined safety requirements components that have safety requirements). Non safety software components do not appear here.

In the same way that a DREAMS system model can have multiple Component Architecture models, this folder can have multiple roots which correspond to different logical component models (see Figure 3.25).

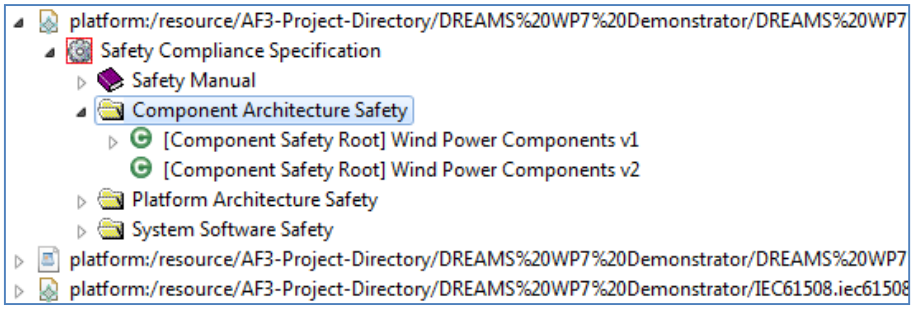


Figure 3.25: Component Architecture Safety Components Folder

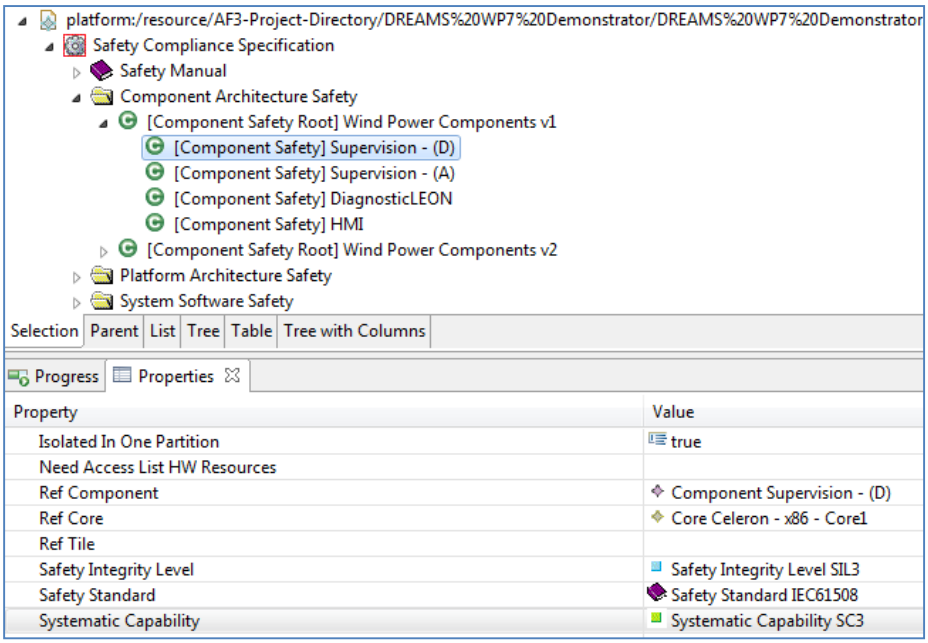


Figure 3.26: Safety Properties of a Safety Component

For each component, the following safety relevant properties (see Figure 3.26) may be defined:

- *RefComponent*: Reference to the Component of the project.
- *Safety Standard and Safety Integrity Level*: SIL level claimed for the Component.
- *RefCore*: if the safety engineer wants to make sure that any deployments involving this component deploys the component into a given core, this field contains a reference to the core. For example: The safety engineer in Wind Turbine demonstrator (WP7) wants to ensure (for whatever reason) that for any *Deployment* (determined automatically using the design-space exploration (provided by WP4) or even manually defined) the *Supervision(D)* software Component in Figure 3.26 always is deployed into a Partition that is finally

deployed into a given Core of a given Tile. The consistency rules will check that for any deployment the Partition is finally deployed into that core.

- *RefTile*: if the safety engineer wants to make sure that any deployments involving this component deploys the component in a given tile, this field contains a reference to the tile. A similar example as above can be given here. However, in this case, the engineer wants to ensure that Component is deployed into a Partition that is deployed to the given Tile (without specifying a particular Core).
- *Isolated in One Partition (Boolean)*: True if the safety engineer wants to make sure (for whatever reason) that any deployments involving this component deploys the component "alone" in one partition (i.e., not shared with any other component). In other words, the Partition will contain only this Component.
- *NeedAccessListHWResources*: List of hardware resources (watchdogs, clocks, tiles, etc.) to which the component need access rights. This is for example needed for a software Component that resets a Watchdog and is deployed into a Partition. In this case, the Partition has to be configured in the hypervisor as having access to those hardware resources. Then, when Hypervisors configuration files are generated (WP4), the hypervisor and Partition will have access rights to the HW resources (particularly, the watchdog).

3.4.4.3 Platform Architecture Safety Folder

Under this folder, safety specification of HW elements (DREAMS Platform Architecture elements) is given. It is important to note that this models only is composed by safety relevant hardware Cluster, Node and Tile elements which are part of safety functions and have defined safety requirements. Non-safety hardware elements do not appear here.

In the same way that a DREAMS system model can have multiple Platform Architecture models, this folder can have multiple roots corresponding to different platform architecture models (see Figure 3.27).

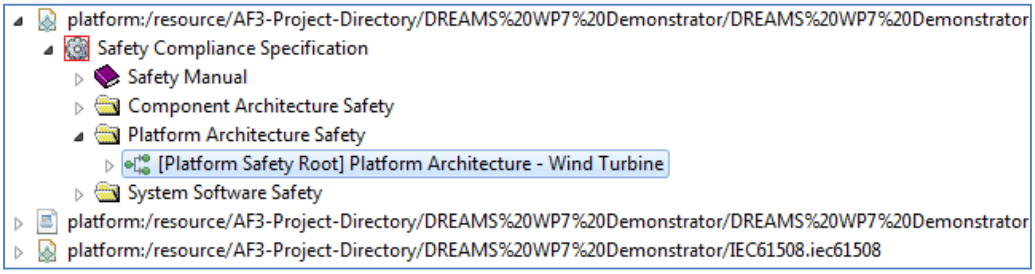


Figure 3.27: Platform Architecture Safety roots

Each Platform Architecture partially mirrors the Platform Architecture of a DREAMS system model (with links to the corresponding entities), and adds Safety Manuals to some of the items (see Figure 3.28).

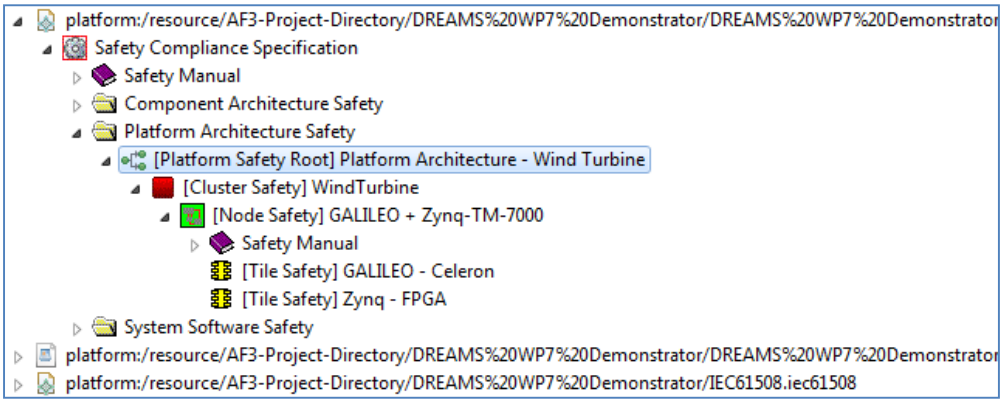


Figure 3.28: Platform Architecture Safety elements hierarchy with Safety Manual

The hierarchy, relevant from the safety point of view, consist of:

- *HW Root*
  - *Clusters*
    - *Nodes*
      - *Tiles*

3.4.4.4 System Software Safety Folder

In this folder, the safety specification of Hypervisors and Partitions (defined via the Virtualization Layer of the technical architecture model, see Section 5.2.6) is given. Figure 3.29 shows an example for the Wind Turbine use case.

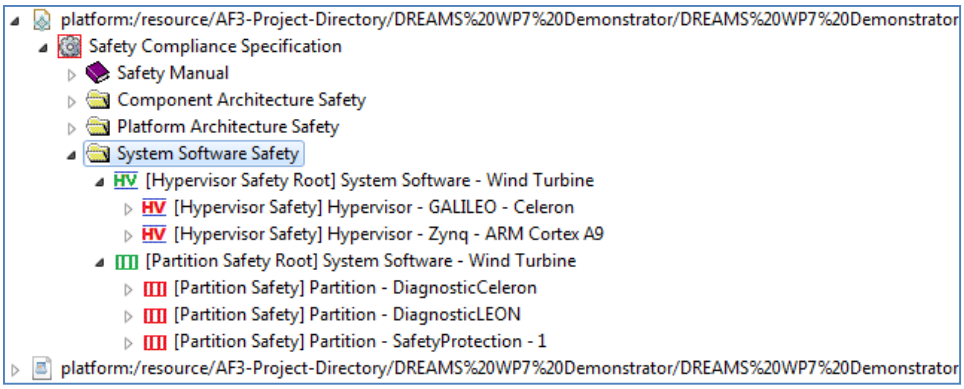


Figure 3.29: System Software Safety roots: Hypervisors and Partitions

Apart from the Safety Manual that can be attached to Hypervisors and Partitions, the safety engineer may want to specify (for whatever reason) that a given Hypervisor be deployed into a given Tile, or that a given Partition be deployed to a given core.

As SCItems, each Hypervisor and Partition will have its own safety manual (figure below). Again, non-safety-relevant hypervisors or partitions (elements which are not part of safety functions) do not appear in this model.

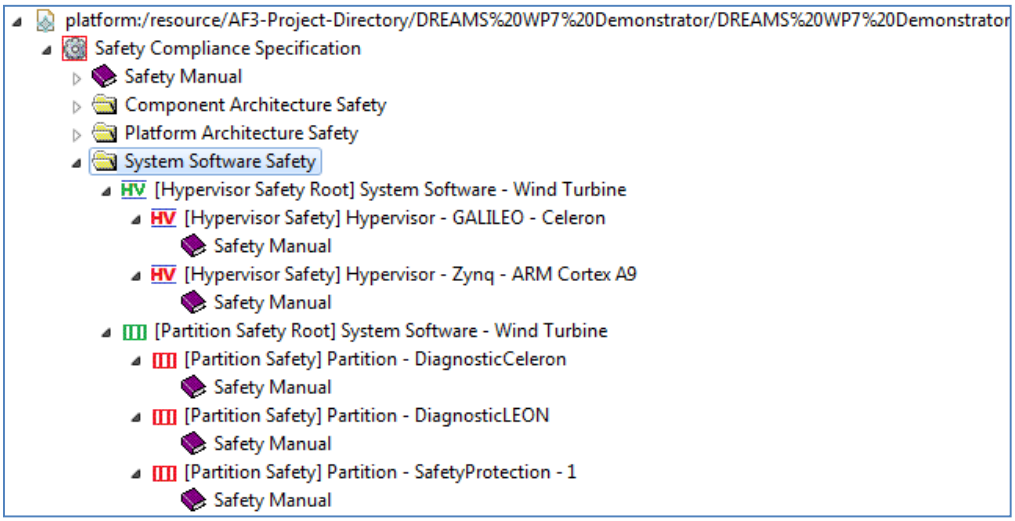


Figure 3.30: Safety Manuals for Hypervisors and Partitions

3.4.4.5 Safety Manual

A Safety Manual defines the way in which the entity containing the manual manages safety, so that a given SIL level is achievable. The safety manual (see Figure 3.31) can be understood as a "declaration" (made by the designer or provider of the item) about how the SCItem manages safety.

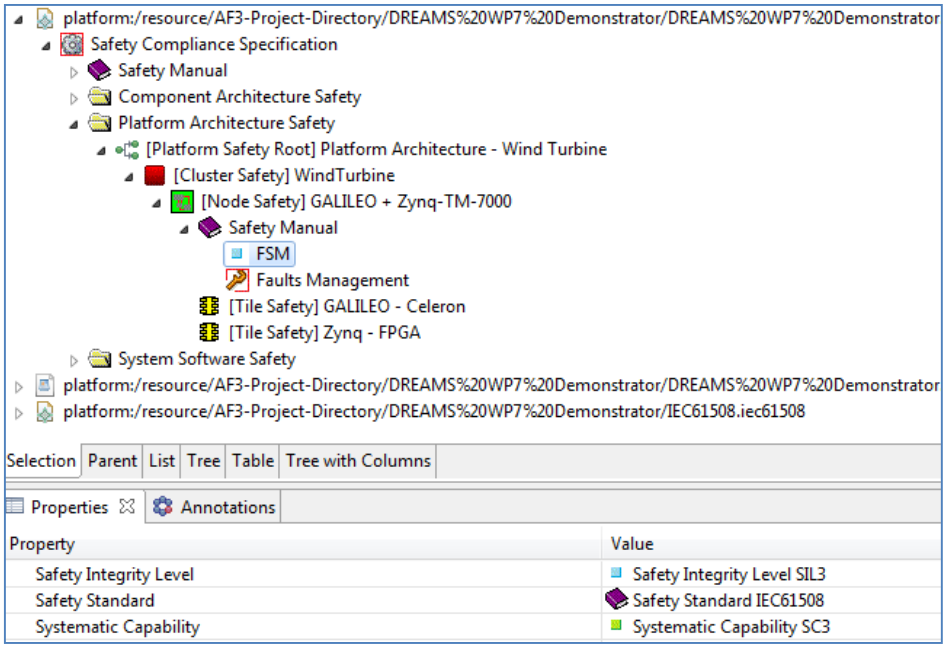


Figure 3.31: Detailed Safety Manual Example

A **Safety Manual** contains:

- Hardware Fault Tolerance level (HFT) only for hardware elements
- Safety Standard (i.e. IEC 61508)
- Safety Integrity Level (i.e. IEC61508-SIL2) claimed
- Systematic Capability Integrity Level (i.e. IEC61508-SIL2) claimed
- *Faults Management*, with the following properties:
  - List of *Random Failure techniques* used by the entity
  - List of *Systematic Failure Techniques* used by the entity
- *Hypothesis values*
  - Values of parameters under which is claimed the safety level
- *Hypothesis ranges* (Category - Min value - Max value)
  - Ranges of parameters under which is claimed the safety level

In short, information consigned in the Safety Manuals will be used to asses if claimed levels of SIL are consistent.

3.4.4.6 Hierarchy of Safety Manuals and Variability due to different Deployments

Hierarchy of Safety Manuals is also a key aspect. For example, if one node, down in the hierarchy, claims a SIL1 level but, higher in the hierarchy, an ancestor claims SIL3, then at least a warning should be raised. Depending on the architecture components with a lower SIL may be combined to result in a higher SIL (e.g., see IEC 61508-2, Chapter 7.4.3)

Variability may complicate things further because perhaps, a given node can have one tile or multiple tiles depending on a variation point or, for example, one component may or may not exists depending on the configuration of the final system.

The same project can also have multiple deployments, depending on the System Architect's requirements. A given component, depending on the deployment, can be running on a different tile or a different core, and therefore, the claimed SIL must be checked against the SIL of the platform deployed.

3.4.5 Safety Constraint Checker – F(Deployment, SafetyComplianceSpecification)

3.4.5.1 Linking everything together and checking

The specific mapping linking software Components, software Partitions, software Hypervisors, hardware Tiles and hardware Cores is given by a Deployment.

Safety Consistency Rules checker, checks consistency for a given tuple composed by five elements for a given project:

- (1) Component Model
- (2) Platform Model
- (3) System Software Model
- (4) Deployment (as the glue of 1,2 and 3)
- (5) Safety Compliance Specification Model

This way, Safety Model and Safety Consistency Rules checking will help the DSE tool to produce valid Deployments from the safety point of view (for the given safety information provided in Safety Manuals and the set of rules implemented). The next section explains the fundamentals about how the safety consistency checking works.

3.4.5.2 Safety Consistency checking

Above sections show that safety consistency checking must be a function with the following parameters:

- *(4) Deployment contains*
  - (1) With a *Component Architecture* with variability resolved (see Section 3.4.6 for an overview how variability is considered for safety compliance models).
  - (2) With a *Platform Architecture* with the variability resolved
  - (3) With a *System Software Architecture* with the variability resolved
  - *(5) SafetyComplianceSpecification (w*ith variability resolved)

The class *SafetyConstraintChecker* contains a function (*evaluateSafetyCompliance*) (see Section 8.1.3) that receives a (a) *deployment, (b)* a collection of safety constraints (explained in Sections 8.1.2 and 8.1.3) and (c) a *safetyComplianceSpecification,* and returns a list of constraints violated by the deployment. The constraint model is described later in Section 8.1).

Notice that there is no specific editor for *Safety Constraints Model*. The reason is that it is not the designer who creates this model, but it is generated automatically by the *SafetyConstraintChecker* class.

3.4.6 Safety Compliance Variability Model Editor

The goal of this editor is to create models to capture the safety compliance related variability.

3.4.6.1 Toolset Summary

The variability will be modelled using BVR tool provided by SINTEF (see Section 3.2.6). Following this approach the variability is modelled in a separate model. Then a mapping between the variability model and the safety consistency models must be done. After that the system is able to create concrete instances of the models with variability using replacement mechanisms.

3.4.6.2 Core Meta-models

The used meta-model is the one of BVR (see Section 9.1). The models based on these meta-models will describe the variability of the safety related models.

4 Logical Viewpoint

4.1 Logical Component Architecture Meta-Model

The (logical) component architecture meta-model is used to describe the logical, or functional, aspects of an application with AutoFOCUS3. The component architecture meta-model is based on the AutoFOCUS3 hierarchic element meta-model (see Section 3.2.6.2), i.e. a component may contain (hierarchical) subcomponents. Hence, the meta-model allows for the definition of the architecture and functional aspects the desired abstraction level.

The components of a `ComponentArchitecture` have `Input-` and `OutputPorts` that describe the (logical) messages which a `Component` receives or emits. `Channels` link these `Ports` to describe the functional dependencies between the components. Note that in the logical view, ports may also remain unconnected to model "external" inputs or outputs, e.g. from sensors or to actuators. `Components` also have attached specifications and annotations contain additional attributes.

Table 5.1 provides an overview of the AutoFOCUS3 component architecture meta-model.

Name	**Logical Component Architecture Meta-Model**	
Description	The goal of the hierarchic element meta-model is to provide modelling support for describing the logical, or functional, aspects of an application.	
Ecore file	`component.ecore`	
Plugin	`org.fortiss.af3.component`	

Packages	`org.fortiss.af3.component`
	`org.fortiss.af3.component.annotation`
	`org.fortiss.af3.component.behavior`
	`org.fortiss.af3.component.generator`
		AutoFOCUS3 component meta-model
		Component-related annotations
		Behaviour specification of Components
		Code generation for Components

Dependencies	`org.fortiss.af3.expression` (not covered in this document) `org.fortiss.tooling.base` (see Section 3.2.6.2) `org.fortiss.tooling.kernel` (see Section 3.2.6.1)	

Table 4.1: Component Architecture Meta-Model (overview)

The meta-model of the Component Architecture consists of the package `org.fortiss.af3.component.model` and the subpackage `org.fortiss.af3.component.model.annotation` that provides property annotations of logical elements. They will be described in the following.

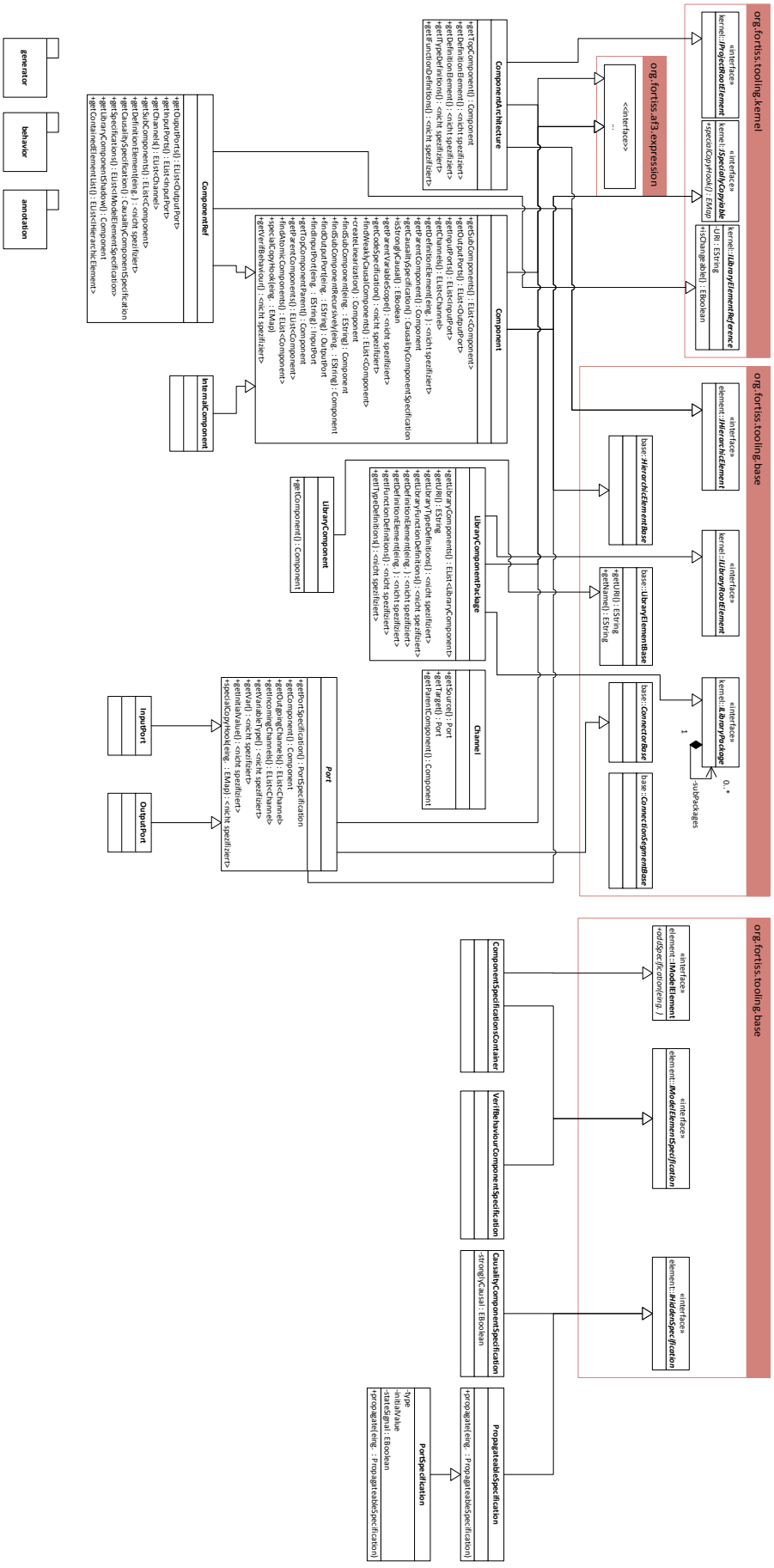


Figure 4.1: UML diagram of the AutoFOCUS3 Component Architecture

For modelling the logical architecture of an application, Components, Ports, and Channels are most relevant. Components are used to define functional blocks and logical groups, Ports define the input and and output interfaces. of Components. Finally, Channels define the data flow (or data dependencies) between Components.

A description of the relevant elements of the meta-model is given in the following:

- ComponentArchitecture
  - Root element of logical architectures, contains components
  - **Operations:**
    - getTopComponent(): Returns the top-level component that is associated with this architecture.
- Component
  - Describes a logical/functional block
  - (May) describe a logical container for subComponents
  - **Operations:**
    - getSubComponents(): Returns the list of direct subComponents
    - getOutputPorts():
    - getInputPorts(): Returns the Output-/InputPorts attached to this Component
    - getChannels(): Returns the Channels that are contained *within* this Component
    - getParentComponent(): Returns the direct parent of this Component
    - findSubComponent(EString):
    - findSubComponentRecursively(EString):
      Returns the (first) subComponent whose name matches the given EString. Returns null if no component with a matching name is found.
    - findOutputPort(EString):
    - findInputPort(EString):
      Returns the (first) Output-/InputPort whose name matches the given EString. Returns null if no matching Output-/InputPort is found.
    - findTopComponentParent(): Returns the topmost Component. This is typically the Component associated with the ComponentArchitecture.
    - getParentComponents(): Returns all parent Components of this Component.
    - findAtomicComponents(): Returns all atomic subComponents of this Component. An atomic Component is a Component that does not contain any subComponents.
    - specialCopyHook(EMap): Used to copy Channels between Componnets.
    - getVerifBehaviour(): Returns the Specification that defines the appearance of this Component within a verification of this ComponentArchitecture.
      See VerifBehaviourComponentSpecification.
- Port:
  - Defines the input or output of the Component to which the Port is attached.
  - Allows Components to interact with their “environment”
  - Ports are transparent to the subelements of the Component to which they are attached.

    - **Operations:**
        - `getPortSpecification()`: Returns the `PortSpecification` of this `Port` (see below).
        - `getComponent()`: Returns the `Component` to which this `Port` is attached.
        - `getOutgoingChannels()` / `getIncomingChannels()`: Returns the outgoing / the incoming `Channel`s of this `Port`. Note that both `OutputPort`s and `InputPort`s may have incoming and outgoing `Channel`s, e.g. the `OutputPort` of a non-atomic `Component` can have incoming channels from sub-`Component`s.
        - `specialCopyHook(EMap)`: Used to copy `Channel`s between `Componnet`s.
- `InputPort`:
    - Represents a `Port` that receives input data of a `Component`.
- `OutputPort`:
    - Represents a `Port` that emits data from a `Component`.
- `Channel`:
    - Connects `Port`s.
    - Used to define the data flow of a modeled application, as `Channel`s set `Component`s into relation.
    - `Channel`s have a direction; Thereby, inter-`Component` dependencies can be defined.
    - **Operations:**
        - `getSource()`: Returns the source `Port` of this `Channel`. The source `Port` is the emitter of a message.
        - `getTarget()`: Returns the target `Port` of this `Channel`. The target `Port` is thus a receiver of the message emitted by the source `Port`.
        - `getParentComponent()`: Returns the parent `Component` of this `Channel`. For example, if a `Component` *A* has two child `Component`s *A_1* and *A_2*, and a `Channel` *C_12* connects *A_1* and *A_2*, then `C_12.getParentComponent()` will return *A*.

In additon to these basic elements, libraries may be defined withing AutoFOCUS3 which allow the definition of reuseable logical blocks. These blocks can the be used in a `ComponentArchitecture` while the actual detailed description resides at one location within the `ComponentArchitecture`. The elements of a library only contain references to the original instance, but since `ComponentRef` implements the `Component` interface, their use is transparent (i.e., the behave like `Component`s that have directly been defined).

- `LibraryComponentPackage`
    - Subelement of an AutoFOCUS3 library or another `LibraryComponentPackage`.
    - Contains references to `Component`s, and/or relevant “data dictionaries”
    - **Operations:**
        - `getLibraryComponents()`: Returns the `LibraryComponents` that are part of this package.
- `LibraryComponent`
    - Wrapped element for representing a `Component` within a `LibraryComponentPackage`
    - **Operations:**
        - `getComponent()`: Returns the `Component` associated with this `LibraryComponent`

- `ComponentRef`:
  - Class that references a `Component`.
  - **Operations:**
    - `getOutputPorts()`:
    - `getInputPorts()`: Returns the Output-/InputPorts attached to the referenced `Component`.
    - `getChannels()`: Returns the Channels that are contained *within* the referenced `Component`.
    - `getSubComponents()`: Returns the list of direct sub`Component`s of the referenced `Component`.
    - `getSpecifications()`: Returns the list of specifications that are attached to the referenced `Component`.
    - `getContainedElementList()`: Returns a list of the directly contained `IHierarchicElement`s of the referenced `Component`.

4.2 Logical Component Architecture Specifications

As pointed out in Section 3.2.3, additional attributes can be defined for model elements using fields in the class definition of the meta-model, specifications and annotations. The attributes implemented using field have been pointed out above (or respectively the operations used to access these fields). In addition to that, several specifications (see Section 3.2.5.2) are defined within the Component Architecture meta-model that will be explained in the following.

The most relevant specifaction is the `PortSpecification` that allows to define the data that is exchanged via `Port`s.

- `PortSpecification`:
  - Defines the type of data that is emitted or received by the `Port` to which this specification is attached. Also defines the initial value of the associated `Port`.
  - **Attributes:**
    - `type`: Defines the type of data that is emitted/receiver by the associated `Port`.
    - `initialValue`: Defines the initial value of the attached `Port`.
  - **Operations:**
    - `propagate(PropagateableSpecification)`:
      see `PropagatableSpecification`
- `PropagatableSpecification`:
  - Defines a general interface that allows subclasses to propagate their properties (of this specification) to a target specification.
  - **Operations:**
    - `propagate(PropagateableSpecification)`: Interface method to propagate properties of this `PropagateableSpecification` to the target `PropagateableSpecification` which must be given as a parameter.
      Empty method that must be implemented by subclasses.

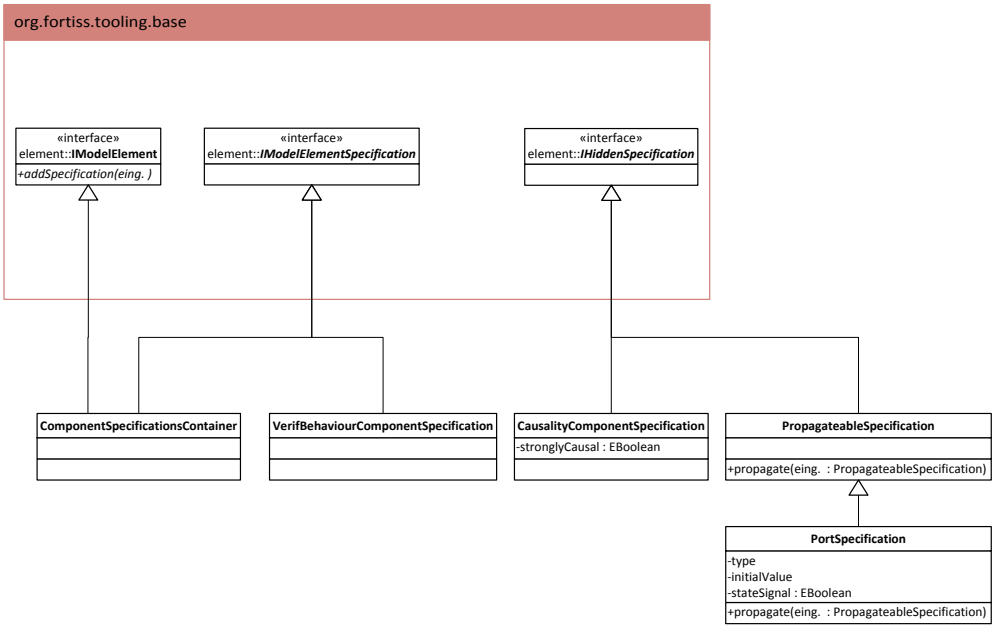


Figure 4.2: AutoFOCUS3 Component Architecture Meta-Model (UML Diagram 2/2 of package org.fortiss.af3.component.model)

In the following, different possibilities to specifiy a `Component`'s behaviour using the dedicated specifications will be briefly summarized. In the scope of DREAMS, a formal specification of an application's behavior is not required, since models are mainly intended as input for offline design optimization and platform configuration generation tools. For completeness, the specifcations provided by AutoFOCUS3 are summarized in the following. Additionally, the `ModeAutomaton` specification could be used to define the different modes of the application demonstators, allowing to define different application sets (with different characteristics such as safety integrity levels and component WCETs for different modes).

- `CodeSpecification`
  - A `CodeSpecification` can be used to specify the behavior of a `Component` at code level using a deditated domain-specific language.
  - AutoFOCUS provides a dedicated editor with syntax-colouring and online syntax checking.
- `StateAutomaton`
  - A `StateAutomaton` can be used to define a `Component`'s behaviour using a finite-state machines.
  - Each `StateAutomaton` must specify an initial state, in which the `Component` is started.
  - An `State` of a `StateAutomaton` can contain another `StateAutomaton`, resulting into a so-called hierarchical state automaton.
  - A `StateAutomaton` can contain Data State Variables (DSV), that can be accessed in all states of the `Component`'s state automaton.
  - A `StateAutomaton` is defined using the following mode elements:
    - `States`: Define the states of a `Component`.
    - `Transitions`: Define a change from one `State` to another `State` defined in the `StateAutomaton`.
    - `Guards`: Define a condition or a set of conditions for triggering / firing an `Action`.

        - Action: Defines the actions to accomplish when conditions defined in Guard are met. An Action should normally contain some Transition between one State and another State.
    - The values of DSV can trigger an Action, when a condition is met. These conditions are implemented using Guards.
- ModeAutomaton
    - A ModeAutomaton controls the switching of running modes of a Component during the lifecycle.
    - For each ModeAutomaton, an initial Mode must be defined (i.e., the mode in which the mode automaton starts).
    - A ModeAutomaton is defined using the following mode elements:
        - Mode: Describes the current configuration of computational data flow a component is in.
        - Switch: Controls the change of the mode to execute. A Switch element contains Guards which specify the mode switch conditions.
        - Guards: Define a condition or a set of conditions for triggering a mode switch.
        - ModeComponentStructureSpecification: This specification encapuslates a Component that defines the behaviour of the given mode.

4.3 Logical Component Architecture Annotations

Lastly, a number of properties are contributed to model elements of the logical view using annotations (see Section 3.2.5.3).

In order to provide a better overview, all annotations which are registered for model elements from the logical viewpoint (i.e., also if they are contributed by other viewpoints) are summarized in the following tables.

4.3.1 Annotations Registered for Components

Table 4.2 lists the annotations that are registered for logical Components.

Annotation name	plugins	Description
MemoryRequirement [DerivedAnnotation]	org.fortiss.af3.component	Defines the required memory of a component. The local value is subject to user input, while the accumulated value considers the memory requirements of the subComponents and itself.
SafetyIntegrityLevel [DerivedAnnotation]	org.fortiss.af3.safety	This annotation allows defining the required safety level for a Component using the levels defined in different safety standards (e.g., SIL2 from IEC 61508). The annotation for the top level Component (which is associated with the ComponentArchitecture, is used to select the safety standard that defines the available levels (considered standards DO178C, IEC61508, and ISO26262). The information provided by the SafetyIntegrityLevel annotation is mainly intended to support the architecture design and deployment phase of the development process. The Safety View (Section 7.1) provides additional concepts that are used support verification and validation activities in the development process. Therefore, the Safety View provides

		the Safety Standard Meta-Model (see Section 7.1.4) which is used internally by the corresponding tool-support (e.g., for safety consistency checks and report generation) and into which `SafetyIntegrityLevel` annotation form the logical view can easily be transformed.
EventTriggerAnnotation	eu.dreamsproject.rtaw.timing	This annotation allows defining the trigger of a `Component` based on the `EventTrigger` defined in the DREAMS timing viewpoint (see Section 7).

Table 4.2: Annotations for Components

4.3.2 Annotations registered for Ports

Likewise, Table 4.3 lists annotations that have been registered for logical `Port`s.

Annotation Name	Corresponding plugins	Description
MessageSize [DerivedAnnotation]	eu.dreamsproject.application	Returns the size of the raw data that is sent via the annotated `OutputPort`. It is given in bits and calculated via the data type (see `PortSpecification` in Section 4.2) that is defined for the annotated `OutputPort`.
InputEventAnnotation	eu.dreamsproject.rtaw.timing	This annotation allows defining the event triggering at an `InputPort` based on the `InputEvent` defined in the dreams timing viewpoint (see Section 7).
OutputEventAnnotation	eu.dreamsproject.rtaw.timing	This annotation allows defining the event triggering at an `OutputPort` based on the `OutputEvent` defined in the dreams timing viewpoint (see Section 7).

Table 4.3: Annotations for Ports

4.4 Interfaces to other Meta-Models

The logical component architecture meta-model does not contain references to meta-models from other viewpoints described in this document. However, as pointed out in 2.2.1, it is referred to by a number of meta-models defined in other viewpoints.

4.5 Logical Component Architecture Model Example Instances

4.5.1 Component Architecture with Annotations

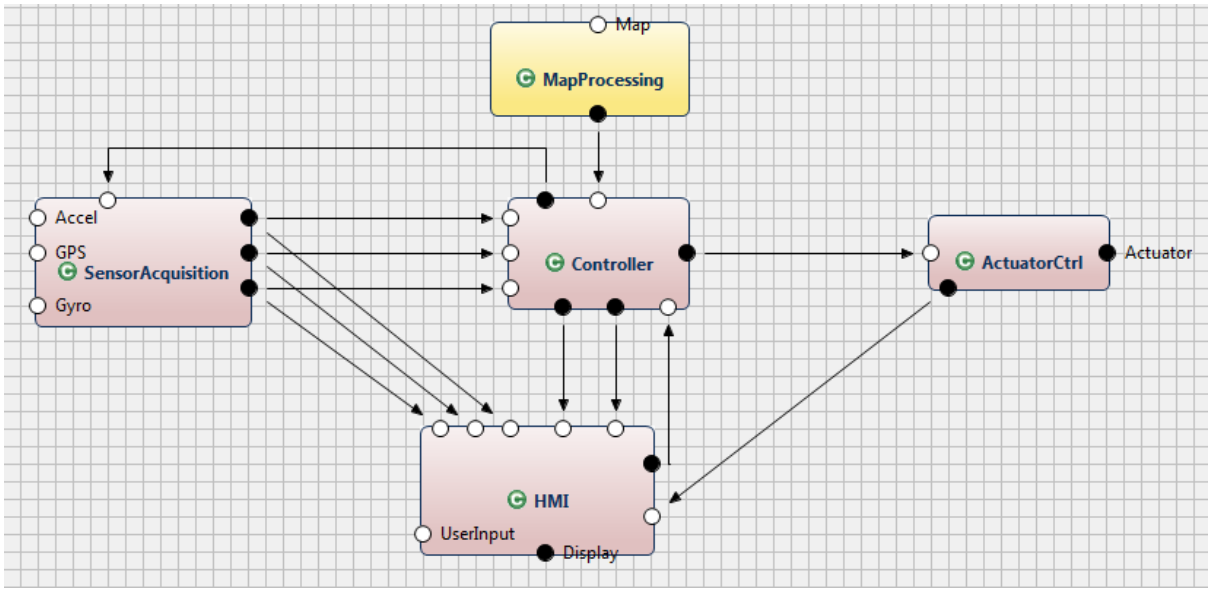


Figure 4.3: Exemplary model of a navigation application

As an example, a model of the logical architecture of a navigation application is illustrated in Figure 4.3. It consists of the `Component`s *SensorAcquisition* (reads and pre-processes sensor data), *Controller* (algorithmic performing the navigation), *MapProcessing* (provides access to a stored map), *HMI* (user input / display), and *ActuatorControl* (controls a motor or similar). The exemplary application model is centered on the *Controller* `Component` that receives refined sensor data from the `Component` *SensorAcquisition* and performs the actual navigation using additional information from a map. The results from the *Controller* are output to the *ActuatorControl* `Component` to transform these results into physical actions and to the HMI `Component` that displays the results and forwards commands issued by the user to the *Controller*.

Each of these `Component`s has attached `InputPort`s (white circles) and `OutputPort`s (black circles) that may be used to connect `Component`s via `Channel`s (black arrows). Disconnected `Port`s are used to model in- and outputs from or to the environment of the logical architecture, like data from sensors (e.g. the *GPS* port at the `Component` *SensorAcquisition*) or sending commands to actuators (e.g. via the *Actuator* `Port` or the *ActuatorCtrl* `Component`).

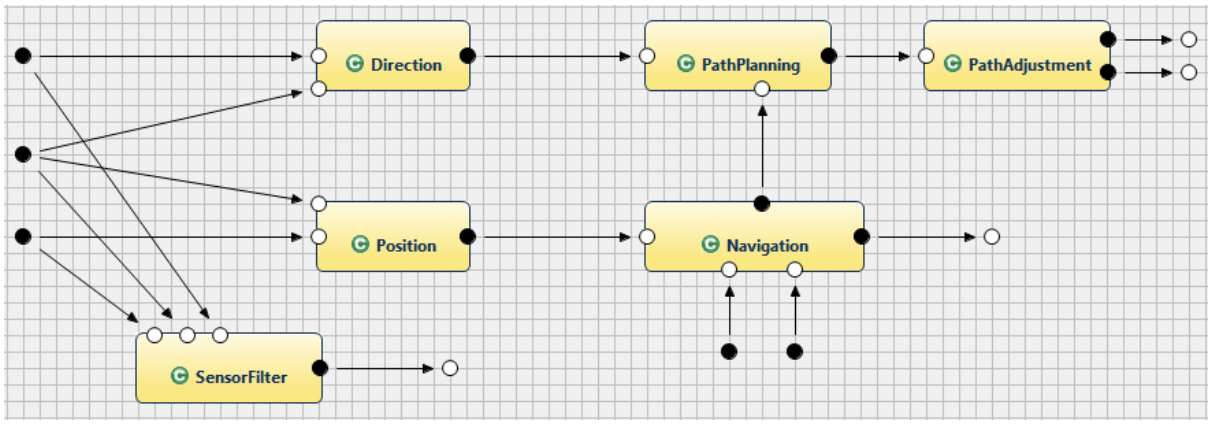


Figure 4.4: Component model of the Controller

The internal structure of the *Controller* Component is illustrated in Figure 4.4 and the annotated properties of the contained Components are given in Figure 4.5. Since the Components related to the path planning are safety-relevant due to their direct impact on the maneuvers of resulting vehicle, their annotated SIL value is high (SIL4), whereas the navigation function is uncritical (SIL0). Furthermore, the figure illustrates that further on-functional properties can be annotated to logical components (here: memory consumption). As it will be pointed out in Chapter 6, parameters that depend on the mapping of a logical Component to an ExecutionUnit provided by the platform (see Chapter 5) are described by the Deployment meta-model.

Model Element	Comment	Memory: accum.	Memory: local	Safety Level
ActuatorCtrl		0	0	SIL0
Controller		15535000	0	SIL0
HMI		23450000	0	SIL0
MapProcessing		250000	250000	SIL0
SensorAcquisition		6050000	0	SIL0

Figure 4.5: Annotations of the subcomponents of the Controller Component

4.5.2 Mode Automaton Specification

As pointed out in Section 4.2, a ModeAutomaton can be used to specify mode switches inside logical Components. The Mode Automaton controls the mode changes of a component in distinct periods of the time. A component may contain different running modes in lifecycle.

In Figure 4.6, a Mode Automaton example associated to a Component is displayed.

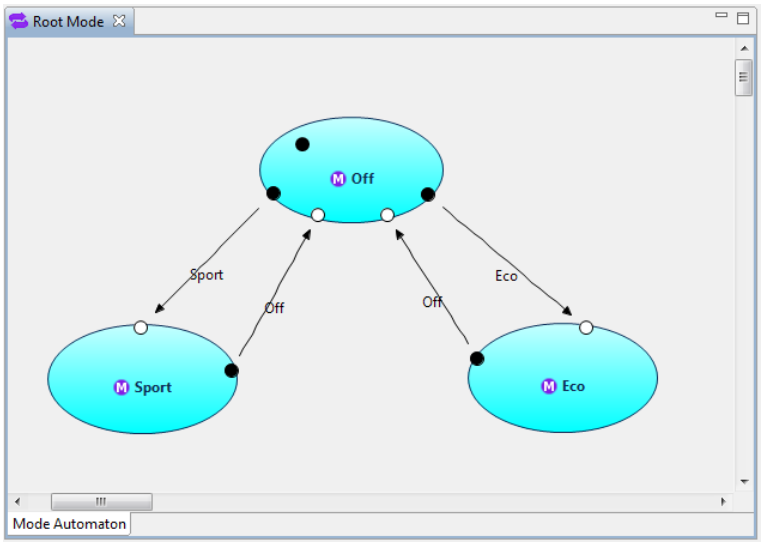


Figure 4.6. Example of Mode Automation applied to a component controlling automotive driving mode.

As defined in Section 4.2, a ModeAutomaton is composed of Modes (represented as blue ellipses) and Switches (represented as arrows). The linking of Modes and Switches is performed using input and output connectors (black and white circles, respectively). An initial mode, which the mode automaton starts with, is required in mode automaton. This initial mode is contains a black dot, which highlights that the mode node is the initial mode.

Mode changes are defined using switches. The conditions for switching between modes are provided by specifying Guards of a Switch. If no Guard of an outgoing Switch is specified, the current Mode remains active. If a Switch can be fired (the condition specified in the Guard is met), then new Mode will be active, and the Component specifying the behaviour of this Mode will take over the computation job.

A mode generally contains a component structure. A mode component structure is a computation of the outputs when the corresponding mode active is. Every mode must contain one sub-component structure. A mode component structure must have the same input and output `Port`s as the parent component, which contains this mode automaton (denoted “mode node” the following). Every time a switch is triggered, the corresponding mode component structure is executed. The mode component structure uses the mode node’s `InputPort`s, and delivers the result of its computation to the mode node’s `OutputPort`s.

5 Technical Viewpoint

5.1 Platform Architecture Meta-Model

The platform architecture meta-model provides the basis for the description of platform architectures in AutoFOCUS3. It consists of a platform architecture meta-model (described in this section) that is based on the AutoFOCUS3 hierarchic element meta-model (see Section 3.2.6.2) and additional attributes contributed by a number of annotations (see Section 5.3).

In order to describe a concrete platform architecture, a specialized meta-model needs to be derived from the AutoFOCUS3 platform meta-model described in this section. Hence, all classes in this meta-model are abstract types. The meta-models for the DREAMS architecture are described in next sections 4.2-4.6.

Table 5.1 provides an overview of the AutoFOCUS3 platform architecture meta-model.

Name	**Platform Architecture Meta-Model**	
Description	The goal of the hierarchic element meta-model is to provide the basis for the description of platform architectures.	
Ecore file	`platform.ecore`	
Plugin	`org.fortiss.af3.platform`	
Packages	`org.fortiss.af3.platform` `org.fortiss.af3.platform.annotation`	AutoFOCUS3 platform meta-model Platform-related annotations
Dependencies	`org.fortiss.af3.component` (see Section 4.1) `org.fortiss.tooling.base` (see Section 3.2.6.2) `org.fortiss.tooling.kernel` (see Section 3.2.6.1)	

Table 5.1: Platform Architecture Meta-Model (overview)

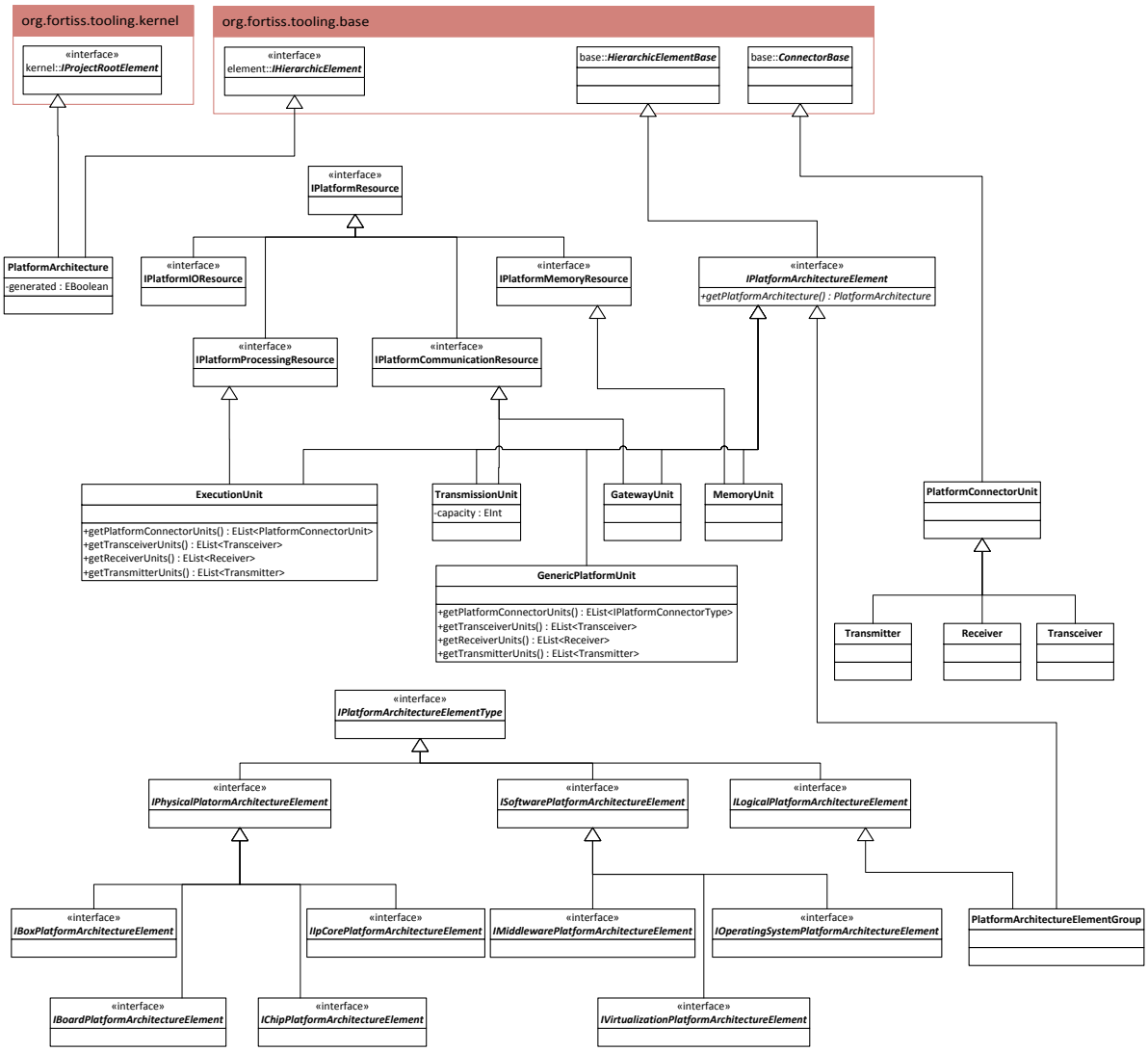


Figure 5.1: AutoFOCUS3 Platform Architecture Meta-Model (UML Diagram 1/2 of package `org.fortiss.af3.platform.model`)

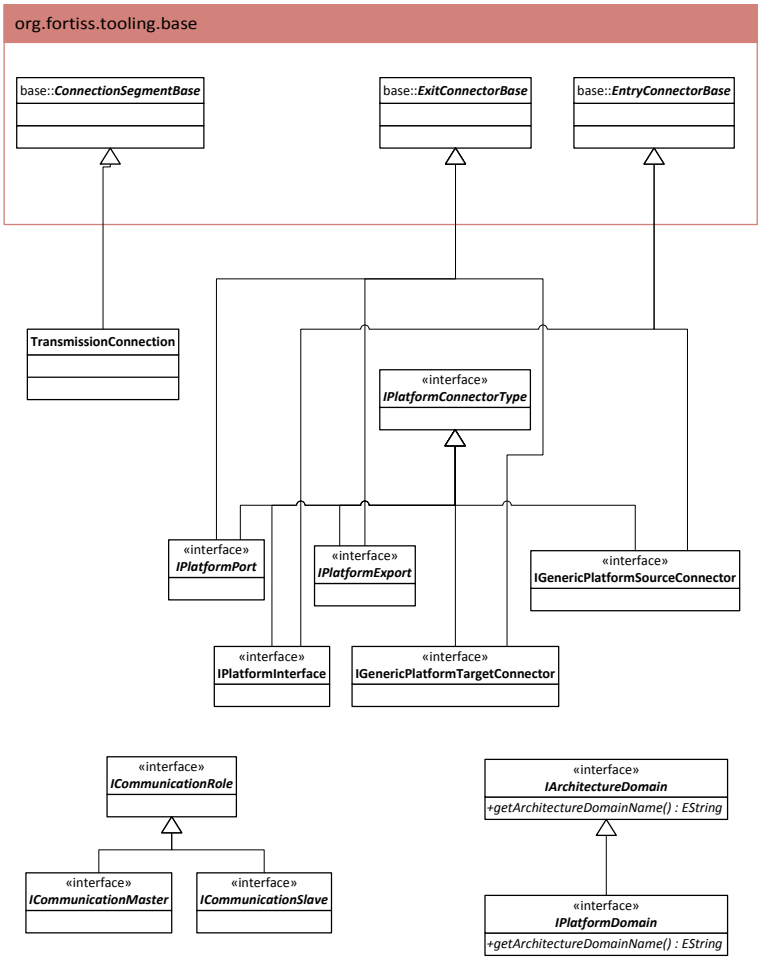


Figure 5.2: AutoFOCUS3 Platform Architecture Meta-Model (UML Diagram 2/2 of package `org.fortiss.af3.platform.model`)

The meta-model consist of the package `org.fortiss.af3.platform.model` that contains the core definition of the platform architecture meta-model, as well as the package `org.fortiss.af3.platform.model.annotation` that provides a number of annotations for platform architecture model elements.

The `org.fortiss.af3.platform.model` package contributes several groups of classes. It is shown in Figure 5.1 and Figure 5.2.

For the description of the platform architecture root (container element for all platform architectures) and the structural platform elements, the following classes are available:

- `PlatformArchitecture`
  - Root element for platform architecture meta-model.
  - **Attributes:**
    - `generated`: Flag if the platform architecture has been generated.
- `IPlatformArchitectureElement`
  - Base class for all platform architecture elements
  - **Operations:**
    - `getPlatformArchitecture()`: Returns the `PlatformArchitecture` for this `IPlatformArchitectureElement`.
- `IPlatformResource`
  - Base marker interface for platform elements that classify the different platform resources.

- `IPlatformCommunicationResource`
  - Interface to mark communication resources (i.e., resources that move data in system).
- `IPlatformProcessingResource`
  - Interface to mark processing resources (i.e., resources that support the execution of software)
- `IPlatformMemoryResource`
  - Interface to mark memory resources (i.e., resources that support the storage of data).
- `IPlatformIOResource`
  - Interface to mark I/O resources (i.e., resources that interface the platform to its environment).
- `ExecutionUnit`
  - Base class for execution units, i.e., platform elements which allow the execution of software
  - **Operations:**
    - `getReceiverUnits()`
    - `getTransmitterUnits()`
    - `getTransceiverUnits()`
    - `getPlatformConnectorUnits()`
- `TransmissionUnit`
  - Base class for transmission units, i.e., communication platform elements that allow the transmission of data (e.g., busses, networks, etc.).
- `GatewayUnit`
  - Base class for gateways units, i.e. dedicated communication platform elements that move traffic between transmission units residing at different levels of the platform architecture.
- `MemoryUnit`
  - Base class for memory units (e.g., RAM, ROM resources)
- `GenericPlatformUnit`
  - Placeholder for generic platform elements (e.g., custom IP blocks) that are not described by any of more specific base classes.
  - **Operations:**
    - `getReceiverUnits()`
    - `getTransmitterUnits()`
    - `getTransceiverUnits()`
    - `getPlatformConnectorUnits()`

For a more fine-grained specification how a platform architecture element is implemented, the meta-model provides the following marker interfaces that can be inherited additionally. For example, a concrete processor would inherit both from `ExecutionUnit` (see above) and `IChipPlatformArchitectureElement`.

- `IPlatformArchitectureElementType`
  - Marker interface to specify the type of platform architecture elements.
- `ILogicalPlatformArchitectureElement`
  - Model element is a logical grouping.
- `IPhysicalPlatformArchitectureElement`
  - Base marker interface for platform elements implemented in hardware.

- `IBoxPlatformArchitectureElement`
  - Marker interfaces for “boxes”, i.e. electronic devices hosting one or more computer systems (~ ECU).
- `IBoardPlatformArchitectureElement`
  - Marker interface for electronic circuit boards (hosting multiple chips).
- `IChipPlatformArchitectureElement`
  - Marker interface for electronic chips that can host multiple hardware IP components in a single package.
- `IIpCorePlatformArchitectureElement`
  - Marker interface for hardware IP component (may contain chip IP components).
- `ISoftwarePlatformArchitectureElement`
  - Marker interface for platform architecture elements implemented in software.
- `IVirtualizationPlatformArchitectureElement`
  - Marker interface for software platform architecture elements that provide a virtualization layer of the underlying hardware.
- `IOperatingSystemPlatformArchitectureElement`
  - Marker interface for operating systems and their sub-components.
- `IMiddlewarePlatformArchitectureElement`
  - Marker interface for middleware components (i.e., platform architecture elements implemented in software that belong neither into the virtualization nor the operating system layer).
- `PlatformArchitectureElementGroup`
  - Logical group of platform architecture elements.

In order to allow for a modular definition of hierarchical platform architectures, and to foster the reuse of sub meta-models, a concept is required to define the compatibility of platform architecture elements. An encoding of the composition rules into the type-system provided by the meta-model is not flexible enough since it does not support the re-use of sub-meta-models in different contexts (e.g., two different concrete platform architectures might allow the use of memory elements at different levels). Therefore, the AutoFOCUS3 platform meta-model provides the `IArchitectureDomain` base marker interface, from which derived marker interfaces should be defined by meta-models providing concrete element types. For each platform meta-model that might combine multiple existing meta-models providing a number of platform architecture element domains, an implementation of the `IPlatformHierarchicalCompositionRules` (see Section 5.2.6) interface must be provided that defines the compatibility of the different architecture domains.

- `IArchitectureDomain`
  - Marker interface to specify platform architecture domain of hierarchical platforms. Platforms / platform element libraries must provide concrete domains (and derive its platform elements from these domains), as well as an implementation of `IPlatformHierarchicalCompositionRules` where the composition rules are encoded (i.e., composability of the different domains).
  - **Operations:**
    - `getArchitectureDomainName()`: Returns the architecture domain's name.
- `IPlatformDomain`
  - `IArchitectureDomain` depicting the `PlatformArchitecture` itself.

Finally, the last group of classes is used to specify the interfaces of platform architecture elements:

- `PlatformConnectorUnit`
  - Base class for connectors of platform architecture elements.
- `Transmitter`
  - Platform connector that supports outbound traffic, only.
- `Receiver`
  - Platform connector that supports inbound traffic, only.
- `Transceiver`
  - Platform connector that supports both inbound and outbound traffic.
- `TransmissionConnection`
  - Connection between platform connector units of two platform architecture elements. It should be noted that the `TransmissionConnection` is a purely logical link that is used to model the connection of any platform architecture elements. All required attributes are described in the corresponding platform architecture elements and platform connector units. If not noted otherwise, `TransmissionConnection`s are undirected (despite the fact that they inherit the `source` and `target` attributes from the `IConnection` interface).
- `ICommunicationRole`
  - Marker interface to specify which role a platform element takes in the communication.
- `ICommunicationMaster`
  - Marker interface to specify that platform element is a communication master that actively initiates the communication.
- `ICommunicationSlave`
  - Marker interface to specify that platform element is a communication slave that can accept communication requests from communication masters.
- `IPlatformConnectorType`
  - Marker interface to further classify the type of platform connector units
- `IPlatformPort`
  - Platform connector unit is a port that be connected / that can implement a given platform interface.
- `IPlatformInterface`
  - Platform connector unit is an interface that can be implemented by platform ports.
- `IPlatformExport`
  - Platform connector unit exports services for use at the parent level.
- `IGenericPlatformSourceConnector`
  - Generic platform (source) connector used to connect platform elements where interconnect has no special role, like transmitting communication information.
- `IGenericPlatformTargetConnector`
  - Generic platform (target) connector used to connect platform elements where interconnect has no special role, like transmitting communication information.

5.2 DREAMS Platform Meta-Model

The DREAMS platform meta-model provides the types required to describe instances of the DREAMS architecture (see D1.2.1 “Architectural Style of DREAMS”). It is based on the AutoFOCUS3 platform architecture model (see Section 5.1), and provides a number of architecture domains (each of which is declared in a separate sub-package of the meta-model).

Table 5.2: DREAMS Platform Meta-Model provides an overview of the AutoFOCUS3 platform architecture meta-model.

Name	**DREAMS Platform Meta-Model**	
Description	Meta-Model for the description of instances of the DREAMS architecture.	
Ecore file	`dreams.ecore`	
Plugin	`eu.dreamsproject.platform`	
Packages	`eu.dreamsproject.platform.model.cluster`	Cluster: physically distributed computer system
	`eu.dreamsproject.platform.model.node`	Node: multi-core chip containing tiles connected by a network-on-chip
	`eu.dreamsproject.platform.model.tile`	Tile: processor cluster / single processor core / IP core connected to the NoC
	`eu.dreamsproject.platform.model.noc`	NoC: Internal structure of network-on-chip
	`eu.dreamsproject.platform.model.hypervisor`	Hypervisor: virtualization of physical resources into partitions
	`eu.dreamsproject.platform.model.processor`	Processor: Internal structure of processors
	`eu.dreamsproject.platform.model.processor.annotation`	Processor domain elements annotations
Dependencies	`org.fortiss.af3.platform` (see Section 5.1) `org.fortiss.af3.component` (see Section 4.1) `org.fortiss.tooling.base` (see Section 3.2.6.2) `org.fortiss.tooling.kernel` (see Section 3.2.6.1)	

Table 5.2: DREAMS Platform Meta-Model

In the next sections, each of the architecture domains will be described in more detail. Finally, Section 5.2.6 describes the composition rules that specify how elements from the different architecture domains can be combined.

5.2.1 Cluster Domain

5.2.1.1 Cluster Meta-Model

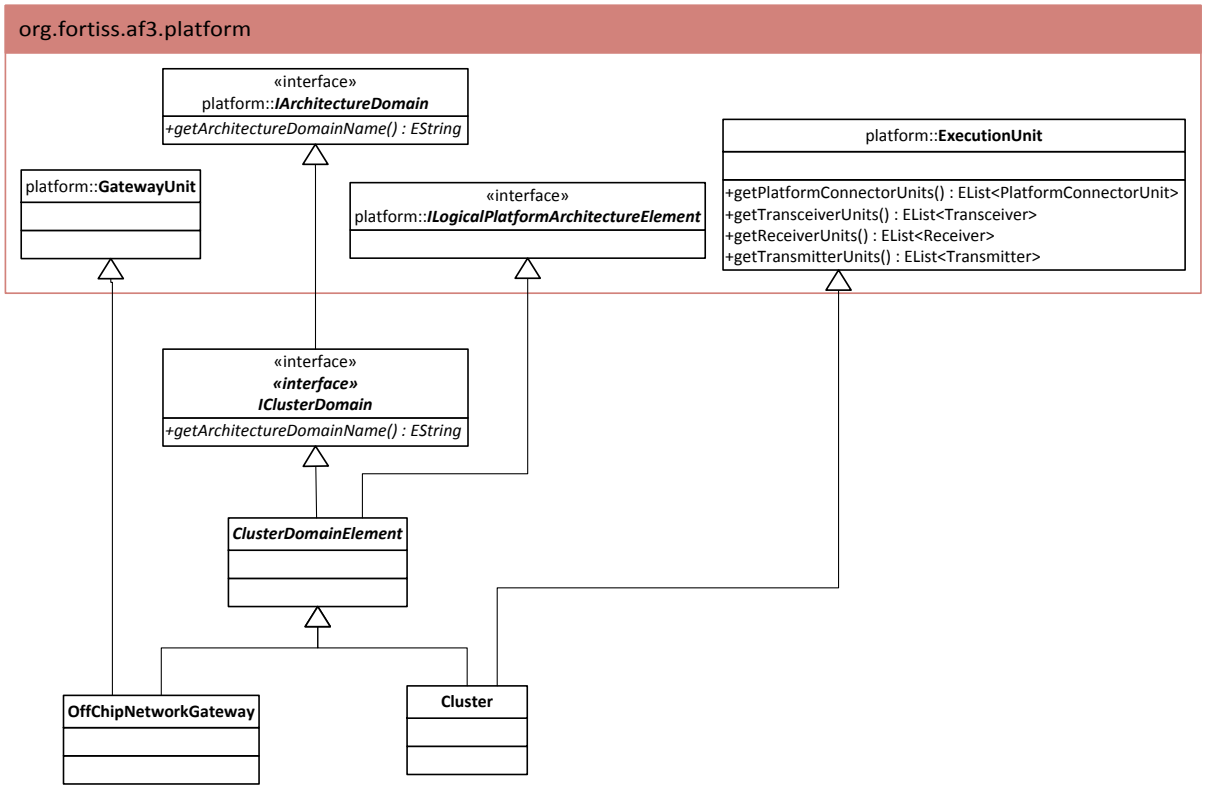


Figure 5.3: DREAMS Platform Meta-Model / Cluster (UML Diagram of `package eu.dreamsproject.platform.model.cluster`)

The `eu.dreamsproject.platform.model.cluster` package is used to model the DREAMS cluster level, i.e. to logically group the interconnection of entire physically distributed computer systems. The meta-model contains the following classes (see Figure 5.3):

- `IClusterDomain`:
  - The `IArchitectureDomain` identifying model elements of the cluster domain.
  - **Operations:**
    - `getArchitectureDomainName()`: Returns the architecture domain's name.
- `ClusterDomainElement`: Base class for structural elements of the cluster domain
- `Cluster`: A DREAMS cluster, i.e. a (logical) group of nodes that are connected via an off-chip network (see Section 5.2.2).
- `OffChipNetworkGateway`: `GatewayUnit` providing a bridge between the `OffChipNetwork`s of connected `Cluster`s.

As it can be seen from Figure 5.3, the model elements in this package are based on the concepts provided by the AutoFOCUS3 platform Meta-Model (see Section 5.1):

- The purpose of the elements at the cluster domain is provide a logical grouping of physically distributed computer systems (which are modelled at the node domain, see Section 5.2.2). Hence, the cluster domain elements are modelled as logical elements (base marker interface `ILogicalPlatformArchitectureElement` of `IClusterDomainElement`).
- The structural elements `Cluster` and `OffChipNetworkGateway` are hierarchic model elements (`HierarchicElementBase` via inheritance hierarchy).

- Clusters are modelled as ExecutionUnits, and hence they (or, model elements in their offspring, respectively) are deployment targets (see Section 6.1) for software which is described using logical components (see Section 4.1).
- The communication between the Clusters and the OffChipNetworkGateways is realized by OffChipNetworkInterfaces and OffChipNetworkPorts (see Section 5.2.2). The Ports from the node meta-model are reused in the cluster meta-model since it provides only a logical grouping.
- The mode of communication is modelled as bidirectional (base class Transceiver of OffChipNetworkPort and OffChipNetworkInterface) with masters actively initiating the communication (marker interface ICommunciationMaster).

5.2.1.2 Cluster Model Example Instance

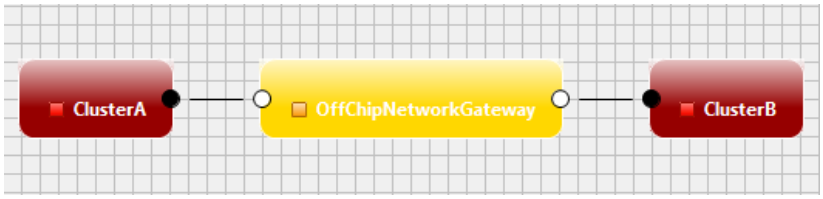


Figure 5.4: Cluster domain example model instance

In Figure 5.4, a simple model consisting of two Clusters can be seen. Both Clusters are connected via an OffChipNetworkGateway. The Clusters have attached OffChipNetworkPorts (connectors represented by black circles) which are each connected to an OffChipNetworkInterface of the contained OffChipNetwork and to the OffChipNetworkInterface of the OffChipNetworkGateway element (see Section 5.2.2). Thus, a connection between the internal OffChipNetworks of *Cluster_A* and *Cluster_B* is modelled.

The eu.dreamsproject.platform.model.nodes package is used to model the DREAMS node level, i.e. to model the internals of a single DREAMS cluster. Hence, a model at the node level describes the structural elements and the topology of a physically distributed computer system. The meta-model contains the following classes (see Figure 5.5).

- INodeDomain:
  - The IArchitectureDomain identifying model elements of the node domain.
  - **Operations:**
    - getArchitectureDomainName():Returns the architecture domain's name.
- NodeDomainElement: Base class for structural elements of the node domain
- NodeDomainConnector: Base class for IPlatformConnectorUnits of the node domain
- Node: A DREAMS node, i.e. electronic control unit (or computer) hosting a multi-core chip containing tiles connected by a network-on-chip
- OffChipNetwork: An off-chip network to interconnect multiple nodes
- OffChipNetworkPort: Off-chip communication port of structural elements at the node level (Nodes, OffChipClusterGateways).
- OffChipNetworkInterface: Communication interface of an OffChipNetwork.
- PowerSupply: Model element of an individual (independent) power supply.
- PowerOut: NodeDomainConnector attached to PowerSupply for connecting Nodes.
- PowerIn: NodeDomainConnector allowing to connect power supplies to Nodes.

5.2.2 Node Domain

5.2.2.1 Node Meta-Model

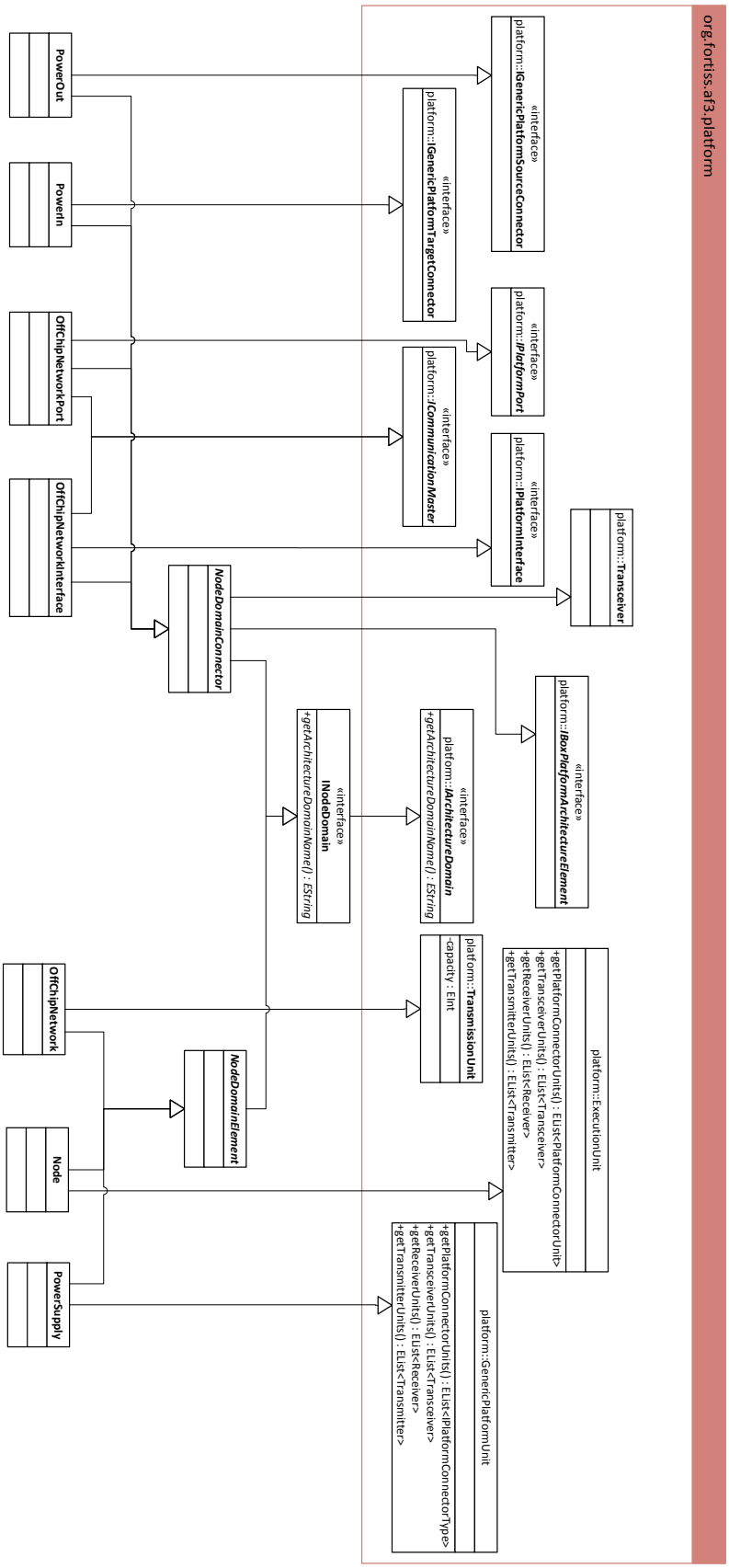


Figure 5.5: DREAMS Platform Meta-Model / Node (UML Diagram of `package` `eu.dreamsproject.platform.model.node`)

Application of concepts from AutoFOCUS3 platform Meta-Model (see Section 5.1):

- The base marker interface `IBoxPlatformArchitectureElement` of `NodeDomainElement` and `NodeDomainConnector` indicates that the system entities modelled by the node domain are electronic devices which provide a dedicated housing.
- The structural elements `Node`, `OffChipNetwork` and `OffChipClusterGateway` are hierarchic model elements (`HierarchicElementBase` via inheritance hierarchy).
- `Node`s are modelled as `ExecutionUnit`s, and hence they (or, model elements in their offspring, respectively) are deployment targets (see Section 6.1) for software which is described using logical components (see Section 4.1).
- Likewise, `OffChipNetworks` being modelled as `TransmissionUnits`, and `OffChipClusterGateway`s being modelled as `GatewayUnits`, are part of the communication facilities of a DREAMS system.

The mode of communication is modelled as bidirectional (base class `Transceiver` of `OffChipNetworkPort` and `OffChipNetworkInterface`) with masters actively initiating the communication (marker interface `ICommunciationMaster`). Here, `OffChipNetworkPort`s constitute the interface of `Node`s and `OffChipClusterGateway`s to the `OffChipNetwork` (whose is interface is modelled by `OffChipNetworkInterface`).

5.2.2.2 Node Model Example Instance

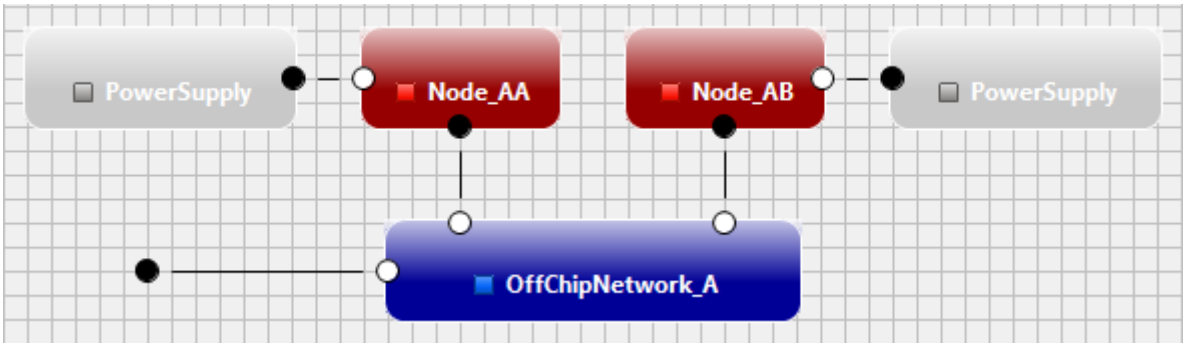


Figure 5.6: Node domain example model instance

An exemplary model from the `Node` domain is shown in Figure 5.6, i.e. the internal structure of a `Cluster`. The example consists of two `Node`s, one `OffChipNetwork`, and two `PowerSupply`s. The `OffChipNetwork`, which represents e.g. a TTEthernet network or an EtherCAT network, has three attached `OffChipNetworkInterface`s. Three `OffChipNetworkPort`s (represented by black connectors) at the `Node`s and at the right hand side of the `OffChipClusterGateway` are connected to these `OffChipNetworkInterface`s. The `NetworkInterface` located at the left side of the `OffChipNetwork` in the example is connected to an `OffChipNetworkPort` of the containing `Cluster`.

As pointed out in Section 5.2.1.2, the `Cluster`'s `OffChipNetworkPort` can be connected to the `OffChipNetworkInterface` of an `OffChipNetworkGateway`. Since Clusters only represent a logical grouping of platform elements, the `OffChipNetworkGateway` (that is used to describe the connection of the off-chip networks of two different clusters) resides at the cluster-domain. In contrast to that, `OnChipOffChipGateway`s (see Section 5.2.3.2 for an example) and `NetworkInterface`s (see Section 5.2.5.2 for an example) are used to route communication from different levels of the architecture. Hence, `OnChipOffChipGateway`s and `NetworkInterface`s reside at the lower of the two architecture levels that are connected by

them (tile-domain, and processor-domain, respectively) and their interface to the containing architecture level is expressed using specializations of `IPlatformExport` (`OnChipOffChipExport` and `OnChipNetworkExport`, respectively).

Each of the two `Node`s present in the example is connected to an independent `PowerSupply`. The connection is established via `PowerOut`s at the `PowerSupply`s and `PowerIn`s attached to the `Node`s. The information about the power supply of `Node`s can be considered during safety analysis (e.g., shared vs. separated power supply)

5.2.3 Tile Domain

5.2.3.1 Tile Meta-Model

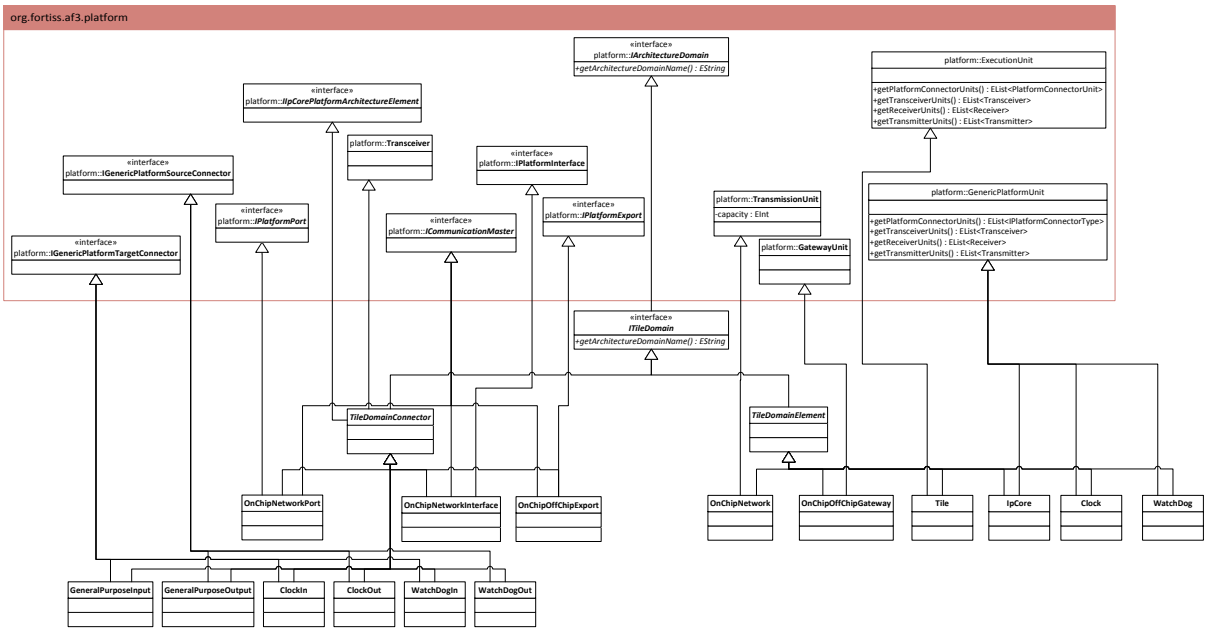


Figure 5.7: DREAMS Platform Meta-Model / Tile (UML Diagram of package `eu.dreamsproject.platform.model.tile`)

The `eu.dreamsproject.platform.model.tile` package is used to model the DREAMS tile level, i.e. to model the internals of a single DREAMS node. Hence, a model at the tile level describes the structural elements and of a multi-processor system on-chip whose elements are interconnected by an on-chip network. The meta-model contains the following classes (see Figure 5.7).

- `ITileDomain`:
  - The `IArchitectureDomain` identifying model elements of the tile domain.
  - **Operations:**
    - `getArchitectureDomainName()`:Returns the architecture domain's name.
- `TileDomainElement`: Base class for structural elements of the tile domain
- `TileDomainConnector`: Base class for `IPlatformConnectorUnits` of the tile domain
- `Tile`: A DREAMS tile, i.e. a multi-core or single-core processing unit that is connected to the `OnChipNetwork` via its `OnChipNetworkPort`.
- `OnChipNetwork`: An on-chip network to connect multiple tiles

- OnChipOffChipGateway: A gateway from the on-chip to the off-chip level.
- IpCore: A placeholder for a generic IP core that is connected to the OnChipNetwork via its OnChipNetworkPort.
- OnChipNetworkPort: On-chip communication port of structural elements at the tile level (Tiles, OnChipOffChipGateways).
- OnChipNetworkInterface: Communication interface of on-chip communication network
- OnChipOffChipExport: It is required to model the communication routes to other nodes.
- WatchDog: Model element representing a watchdog timer which can trigger a reset of connected elements that fail the reset the watchdog timer in time and hence are considered to be in a "failed" state.
- WatchDogIn: Connector to be attached to elements which shall be monitored by a WatchDog.
- WatchDogOut: Connector at the WatchDog to which monitored elements can be connected.
- Clock: Model element that represents clock sources.
- ClockIn: Connector of the model element to which a clock signal shall be provided.
- ClockOut: Connector at the clock source from which a clock signal to connected elements is emitted.
- GeneralPurposeInput: Connector representing a digital input port of the respective model element.
- GeneralPurposeOutput: Connector representing a digital output port of the respective model element.

Application of concepts from AutoFOCUS3 platform Meta-Model (see Section 5.1):

- The base marker interface IIpCorePlatformArchitectureElement of TileDomainElement and TileDomainConnector indicates that the system entities modelled by the tile domain are IP cores that possibly are contained in the same package.
- The structural elements Tile, IpCore, OnChipNetwork and OnChipOffChipGateway are hierarchic model elements (HierarchicElementBase via inheritance hierarchy).
- Tiles are modelled as ExecutionUnits, and hence they (or, model elements in their offspring, respectively) are deployment targets (see Section 6.1) for software which is described using logical components (see Section 4.1).
- Likewise, OnChipNetworks being modelled as TransmissionUnits, and OnChipOffChipGateways being modelled as GatewayUnits, are part of the communication facilities of a DREAMS system.
- The mode of communication is modelled as bidirectional (base class Transceiver of OnChipNetworkPort and OnChipNetworkInterface) with masters actively initiating the communication (marker interface ICommunciationMaster). Here, OnChipNetworkPorts constitute the interface of Tiles, IpCores and OnChipOffChipGateways to the OnChipNetwork (interface modelled by OnChipNetworkInterface). As mentioned above, in addition to OnChipNetworkPorts, also OnChipOffChipExports can be attached to OnChipOffChipGateway. Then, the route to the off-chip communication can be described using a link from the OnChipOffChipExport to the OffChipNetworkPort owned by the Node that contains the respective OnChipOffChipGateway.

- WatchDogs and Clocks can be connected to Tiles level to model different Clock Domains and the monitoring of Tiles (which represent multi-core processors). This is especially relevant for safety analysis. Note that each WatchDog must be connected to a Clock source since its nature as a timer requires a clock signal.

5.2.3.2 Tile Model Example Instance

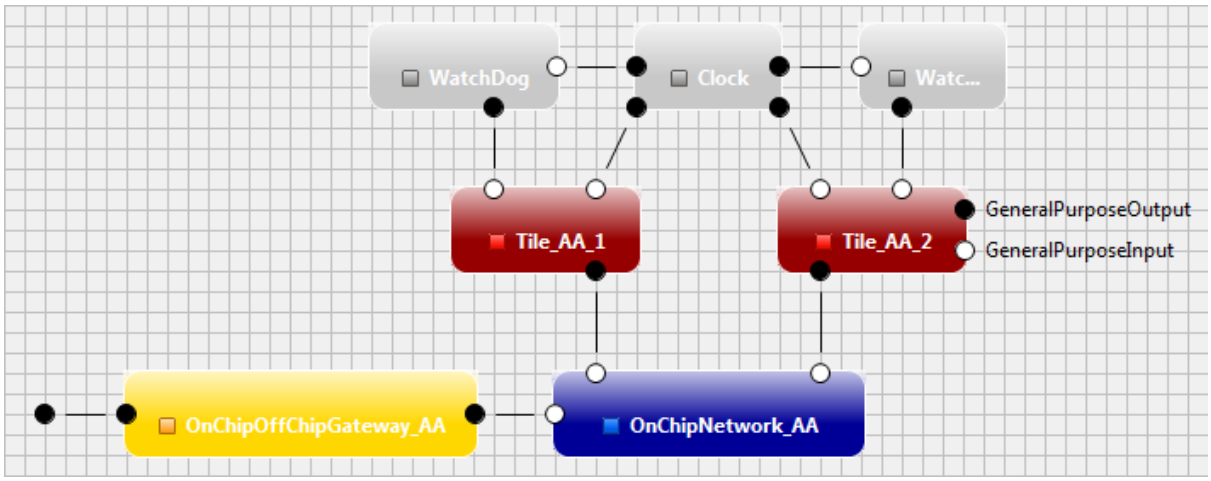


Figure 5.8: Tile domain example model

Figure 5.8 shows an exemplary model at the Tile-level, i.e. the internal structure of a Node. There are two Tiles, one OnChipNetwork, and an OnChipOffChipGateway. The white connectors attached to the OnChipNetwork represent the OnChipNetworkInterfaces. Likewise, the connectors attached to the Tiles and at the right hand side of the OnChipOffChipGateway represent OnChipNetworkPorts that are connected to the corresponding OnChipNetworkInterfaces of the OnChipNetwork.

The OnChipOffChipGateway depicts the gateway of the OnChipNetwork shown in this example to the network at the containing layer (i.e., an OffChipNetwork at the Node layer). The connector in the very bottom left of the figure represents the OffChipNetworkPort of the Node that contains the model shown in Figure 5.8. It is connected to an OnChipOffChipExport (left connector of OnChipOffChipNetworkGateway) that is used to model the connection to the containing Node's OffChipNetworkPort.

In the example, a common Clock source provides a clock signal via ClockOuts to the connected Tiles which receive the signal via attached ClockIns. Furthermore, WatchDogs are connected to the two present Tiles via WatchDogOuts (at the WatchDogs) and WatchDogIns (at the Tiles). Since WatchDogs are essentially timers, they require a clock signal and, hence, they are connected to the Clock that provides the signal to the Tiles. The Tile *Tile_AA_2* additionally has an attached GeneralPurposeOutput Port and a GeneralPurposeInput Port modelling the generic GPIOs of processors or boards.

5.2.4 NoC Domain

5.2.4.1 NoC Meta-Model

The `eu.dreamsproject.platform.model.noc` package is used to model the internals of `OnChipNetwork`s (see Section 5.2.3). It contains the following classes (see Figure 5.9).

- `INocDomain`:
  - The `IArchitectureDomain` identifying model elements of the NoC domain.
  - **Operations:**
    - `getArchitectureDomainName()`:Returns the architecture domain's name.
- `NocDomainElement`: Base class for structural elements of the tile domain
- `NocDomainConnector`: Base class for `IPlatformConnectorUnit`s of the NoC domain
- `NocRouter`: A router of the `OnChipNetwork`.
- `NocInputUnit`: An input unit of a `NocRouter` of the `OnChipNetwork`.
- `NocOutputUnit`: An output unit of a `NocRouter` of the `OnChipNetwork`.

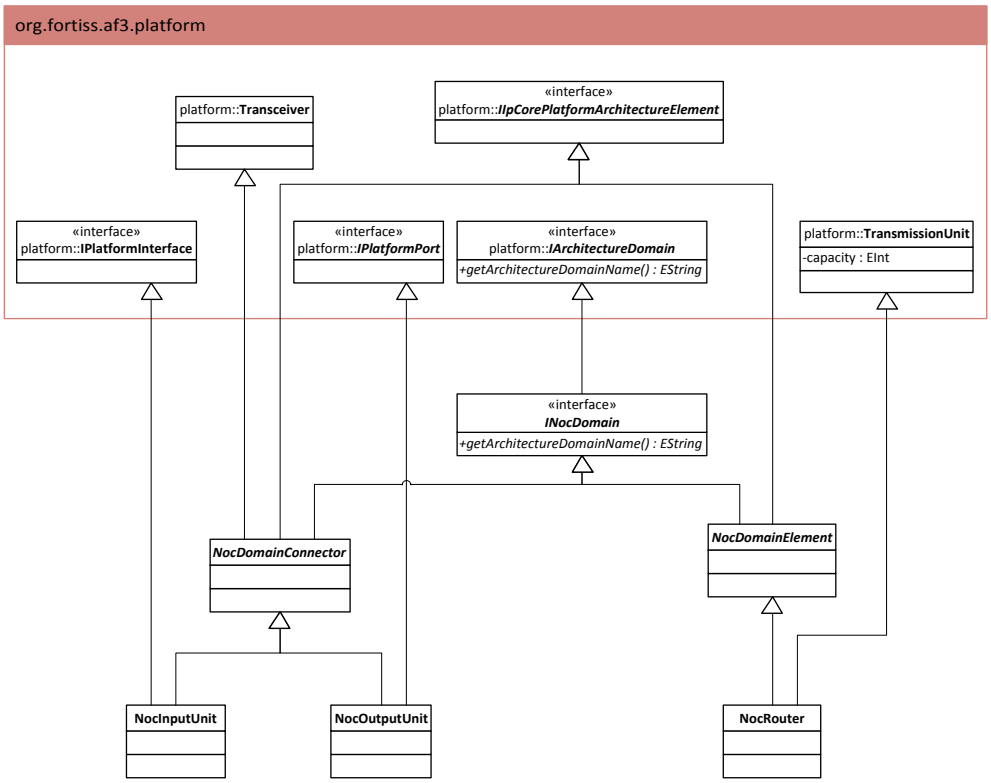


Figure 5.9: DREAMS Platform Meta-Model / NoC (UML Diagram of package `eu.dreamsproject.platform.model.noc`)

Application of concepts from AutoFOCUS3 platform Meta-Model (see Section 5.1):

- The base marker interface `IIpCorePlatformArchitectureElement` of `NodeDomainElement` and `NodeDomainConnector` indicates that the system entities modelled by the node domain are IP cores that possibly are contained in the same package.
- `NocRouter`s are modelled as `TransmissionUnit`s and constitute the most fine-grained level of in the model of the DREAMS communication facilities.

- The internal structure of an on-chip network is modelled using directed `TransmissionConnections` between the `OnChipNetworkInterfaces` of the `OnChipNetwork` and the `InputUnits` / the `OutputUnits` of the `NocRouters` contained by the `OnChipNetwork`.

5.2.4.2 NoC Model Example Instance

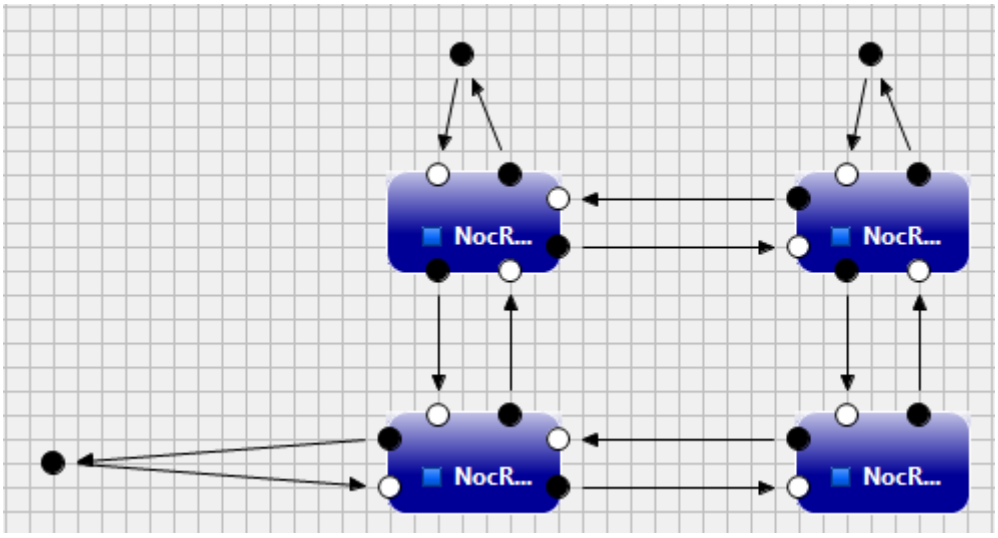


Figure 5.10: NoC domain example model

Figure 5.10 shows an exemplary model at the NoC-Domain, i.e. the internal structure of an `OnChipNetwork`. The black connectors at the left and at the upper side of the figure represent the `OnChipNetworkInterfaces` of the containing `OnChipNetwork`. The example contains four `NoCRouters` whose communication interfaces are represented by `Input-` and `OutputUnits`. The `InputUnits` are represented by the white connectors, while the black connectors attached to `NoCRouters` are `OutputUnits`, respectively. Note the internal structure of the `OnChipNetwork`, i.e. the interconnection between the different `NoCRouters` is modelled using directed connections (arcs) which allows to model complex communication topologies for `OnChipNetworks`. For instance, this can be used to segregate the communication of the platform components (e.g., `Tiles`) connected to the corresponding `OnChipNetworkInterfaces` into different classes. In the topology depicted in the simple example in Figure 5.10 does not impose any restrictions onto the communication flow between `Tiles` connected to the corresponding `OnChipNetworkInterfaces`, but provides redundant communication routes.

It should be noted, that on all other levels of the platform meta-model, communication links are modelled as undirected connections (edges). Hence – unlike `InputUnits` and `OutputUnits` – `OnChipNetworkInterfaces` are modelled as bidirectional communication elements.

5.2.5 Processor Domain

5.2.5.1 Processor Meta-Model

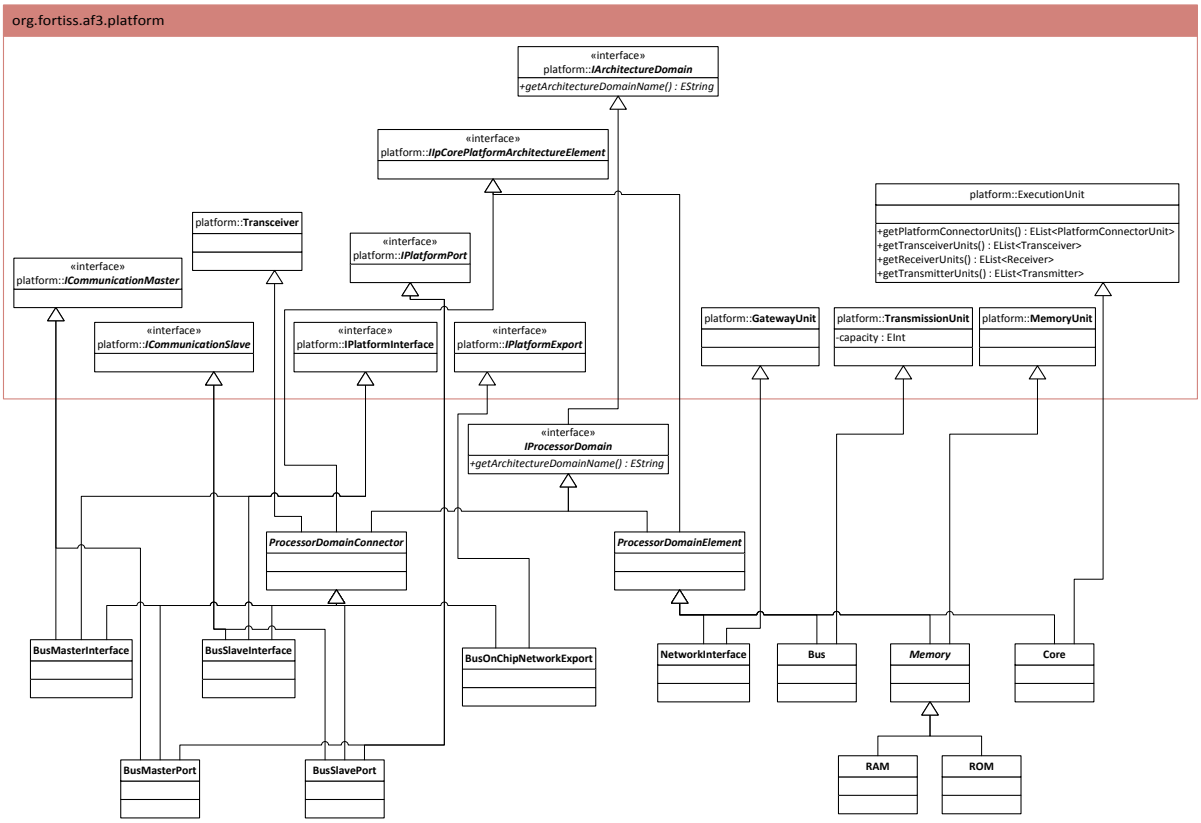


Figure 5.11: DREAMS Platform Meta-Model / Processor (UML Diagram of package eu.dreamsproject.platform.model.processor)

The package `eu.dreamsproject.platform.model.processor` is used to model the DREAMS processor level, i.e. the internals of a DREAMS tile. Thus, the system elements of this package include busses, cores, memories and network interfaces. Hence, it is possible to describe multicore processors whose cores are connected via a bus and are able to access the `OnChipNetwork` using `NetworkInterface` that is connected to the bus. The meta-model contains the following classes (see Figure 5.11):

- `IProcessorDomain`:
  - The `IArchitectureDomain` used to identify model elements that belong to the domain of processors.
  - **Operations:**
    - `getArchitectureDomainName()`:Returns the architecture domain's name.
- `ProcessorDomainElement`: Base class of the structural elements that describe a `Tile`.
- `ProcessorDomainConnector`: Base class of the structural elements used to describe the communication of `ProcessorDomainElement`s.
- `Core`: Structural model element used to describe a single `Core` of a processor.
- `Memory`: Base class used to describe memory (storage) elements of a processor. Accessible via the `Bus` of the same parent `Tile`.
- `RAM`: Model element used to describe `Memory` which that is volatile.
- `ROM`: Model element used to describe read-only, non-volatile `Memory`.

- Bus: Model element describing the (main) communication resource on processor level which connects Cores, Memory, and NetworkInterfaces.
- NetworkInterface: Model element that connects the processor elements to the OnChipNetwork via a BusOnChipNetworkExport to which this Tile is connected.
- BusMasterInterface: Model element to describe interfaces of a processor Bus which is capable of handling bus master arbiters.
- BusSlaveInterface: Model element to describe interfaces of a processor Bus which is only capable of serving slave devices.
- BusMasterPort: The Port of a ProcessorDomainElement which is connected to a BusMasterInterface of a processor Bus. The ProcessorDomainElement must be capable of fulfilling the role of a Bus master.
- BusSlavePort: The Port of a ProcessorDomainElement which is connected to a BusSlaveInterface of a processor Bus. The ProcessorDomainElement cannot take over the master role at this Bus.

Application of concepts from AutoFOCUS3 platform Meta-Model (see Section 5.1):

- The base marker interface IIpCorePlatformArchitectureElement of ProcessorDomainElement and ProcessorDomainConnector indicates that the system entities modelled by the tile domain are IP cores that possibly are contained in the same package.
- The structural elements Core, Memory, Bus and NetworkInterface are hierarchic model elements (HierarchicElementBase via inheritance hierarchy).
- Cores are modelled as ExecutionUnits, and hence they are possible deployment targets (see Section 6.1) for software which is described using logical components (see Section 4.1).
  Nevertheless, the typical lowest deployment granularity within a DREAMS architecture will consider Partitions as deployment targets. Those will be executed on top of Cores and within Hypervisors providing the middleware between both model elements.
- Likewise, Buses are modelled as TransmissionUnits, and NetworkInterfaces are modelled as GatewayUnits which both are part of the communication resources of a DREAMS system.
- Furthermore, Memory, which appears at this level in the form of RAM and ROM, is modelled as a MemoryUnit that describes any kind of memory or storage. Thus, RAM and ROM are also hierarchical elements.
- The mode of communication is modelled as bidirectional (base class Transceiver of ProcessorDomainConnector) with masters actively initiating the communication (marker interface ICommunciationMaster). Here, BusMaster- and BusSlavePorts constitute the interface of Cores, Memories and NetworkInterfaces to the Bus whose interfaces are modelled as BusMaster- and BusSlaveInterfaces. As mentioned above, BusOnChipNetworkExports can be attached to NetworkInterfaces in addition to BusMaster- and BusSlavePorts. Then, the route to the off-chip communication can be described using a link from the OnChipOffChipExport to the OffChipNetworkPort owned by the Node that contains the respective OnChipOffChipGateway.
- In contrast to the DREAMS meta-models described in the previous sections, the communication role (master or slave) is especially important considering the Bus architecture where one device must have absolute control over the communication. Otherwise, interfering access would render any information on the Bus unusable. Hence, the explicit separation into master and slave ports and interfaces here.

5.2.5.2 Processor Model Example Instance

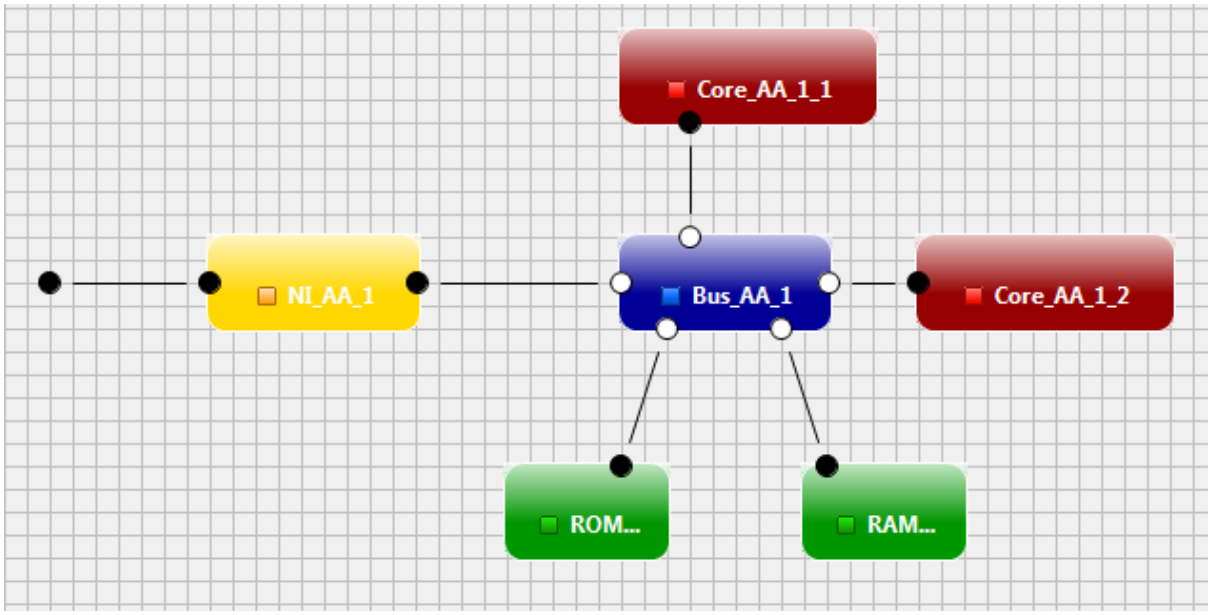


Figure 5.12: Processor domain example model

Figure 5.12 contains an exemplary model at the processor domain, i.e. the internals of a `Tile`. The model contains two `Core`s, one `RAM` and one `ROM` `Memory`, a `Bus`, and a `NetworkInterface` (NI). All mentioned elements are connected via the `Bus`. The `Port`s (black connectors) attached to the `Core`s and to the `NetworkInterface` are `MasterPort`s since they need to be able to initiate communication via the `Bus`. These `MasterPort`s are connected to `BusMasterInterface`s, and thus, the model elements mentioned above are able to communicate. In contrast, the `Memory` elements are connected via `BusSlavePort`s to the `BusSlaveInterface`s of the `Bus`, as these elements do not initiate any communication (passive elements).

The left hand side of the `NetworkInterface` is a model of the gateway to the `OnChipNetwork` at containing layer (i.e., to the `Node` layer). The left-most black connector is the `OnChipNetworkPort` of the `Tile` that contains the discussed example model. This `Port` is connected to the `NetworkInterface`'s `OnChipNetworkExport` (left connector of `NetworkInterface` component) that depicts the interface of the processor domain to the on-chip-network.

As a result (and also considering the meta-models of the other levels of the DREAMS architecture discussed in the previous sections) the model describes that there is a possible communication route from the two `Core`s shown in Figure 5.12 to resources in located in other `Tile`s (via the `OnChipNetwork`) or `Node`s (via `OnChipNetwork`s and `OffChipNetwork`s) . Likewise, the model contains the relevant information to determine routes to the `Core`s and the `Memory`s from Figure 5.12 from remote resources.

5.2.6 Hypervisor Domain

5.2.6.1 Hypervisor Meta-Model

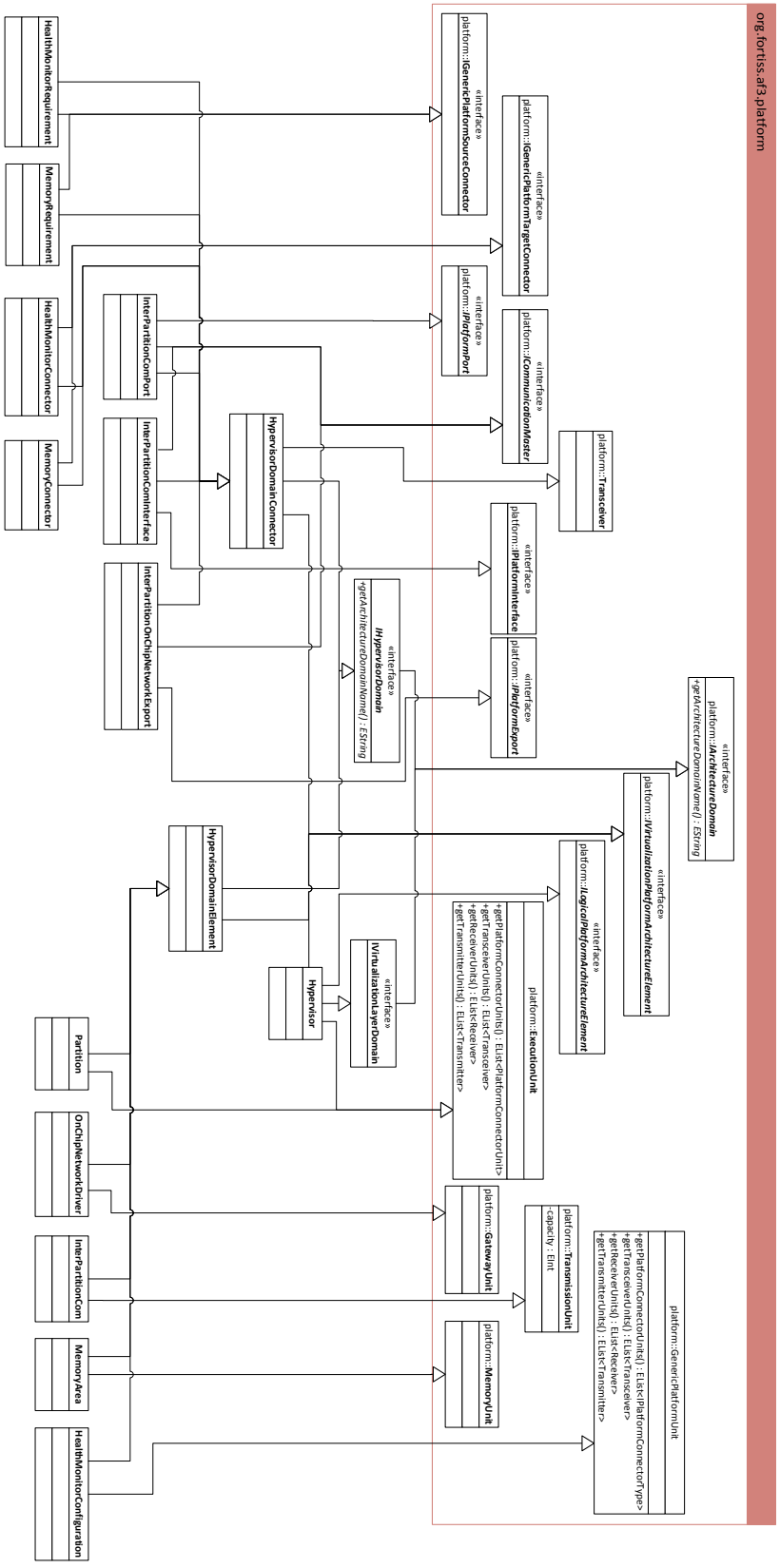


Figure 5.13: DREAMS Platform Meta-Model / Hypervisor (UML Diagram of package `eu.dreamsproject.platform.model.hypervisor`)

The package eu.dreamsproject.platform.model.hypervisor is used to model hypervisors within system software layer of a DREAMS application. The model of the system software layer is instantiated in a separate PlatformArchitecture that is linked to the model of the physical platform layer using the ResourceLink annotation (see Section 5.3.7). The meta-model contains the following classes (see also Figure 5.13):

- IHypervisorDomain:
  - The IArchitectureDomain to identify model elements belonging to the domain of hypervisors.
  - **Operations:**
    - getArchitectureDomainName():Returns the architecture domain's name.
- IVirtualizationLayerDomain: The IArchitectureDomain to identify model elements providing virtualization services.
- Hypervisor: Class representing a hypervisor, i.e. a system software layer module that virtualizes ExecutionUnits of the physical platform (e.g., a processor (Tile)). The virtualized physical resources are designated by the ResourceLink annotation (see Section 5.3.7).
- HypervisorDomainElement: Base class for structural elements that are attached to Hypervisors or that are sub-elements of Hypervisors.
- HypervisorDomainConnector: Base class for describing communication structure of HypervisorDomainElements.
- Partition: Isolated and virtualized execution environment for software components provided by a Hypervisor. Using the ResourceLink annotation, it is linked to ExecutionUnits of the physical platform resource to which its containing Hypervisor is linked (e.g., Cores of the corresponding Tile).
- OnChipNetworkDriver: Model element representing a system partition of Hypervisor that has access to the OnChipNetwork resource of the physical platform layer (referenced using the ResourceLink annotation).
- InterPartitionCom: Class to express communication facility provided by the Hypervisor that provides message exchange between Partitions.
- InterPartitionComPort: Communication port of virtual structural elements, i.e. Partitions.
- InterPartitionComInterface: Communication interface located at InterPartitionCom that provides the inter-partition communication service.
- MemoryArea: Model element used to represent memory areas assigned to Partitions or to Hypervisors. A partition can have one or more assigned MemoryAreas, and a MemoryArea can be shared by multiple partitions to model shared memory. MemoryAreas assigned to Hypervisors have a 1:1 relation. Each MemoryArea is linked to a MemoryUnit of the underlying physical platform using the ResourceLink annotation.
- MemoryRequirement: HypervisorDomainConnector that is attached to Partitions or Hypervisors to model their need of and the connection to an allocated MemoryArea.
- MemoryConnector: HypervisorDomainConnector that provides access to MemoryAreas.
- HealthMonitorConfiguration: Model element of the health status self-monitoring capabilities of Hypervisors. It can be connected to Hypervisors and parametrized by annotations to model the configuration of a health monitor.

- HealthMonitorRequirement: HypervisorDomainConnector attached to Hypervisors to model their need of a HealthMonitorConfiguration.
- HealthMonitorConnector: HypervisorDomainConnector attached to HealthMonitorConfiguration to establish a connection to an associated Hypervisor.

Application of concepts from AutoFOCUS3 platform Meta-Model (see Section 5.1):

- The base marker interface IVirtualizationPlatformArchitectureElement of Hypervisor, HypervisorDomainElement, and HypervisorDomainConnector indicates that these system elements are part of the virtualization layer of the DREAMS system, i.e. no hardware platform elements.
- Additionally, Hypervisor inherits from the base marker interface ILogicalPlatformArchitectureElement which indicates that this system element is a logical entity, i.e. it has no concrete physical realization (in the hardware sense).
- The structural elements Partition, OnChipNetworkDriver, and InterPartitionCom are hierarchic model elements (HierarchicElementBase via inheritance hierarchy).
- Partitions and Hypervisors are modelled as ExecutionUnits, and hence they are (possible) deployment targets (see Section 6.1) for software which is described using logical components (see Section 4.1).
- Likewise, OnChipNetworkDrivers are modelled as GatewayUnits, and InterPartitionComs are modelled as TransmissionUnits, both being part of the communication facilities of a DREAMS system.
- The communication within the IHypervisorDomain is modelled being bidirectional (base class Transceiver of HypervisorDomainConnector) with masters actively initiating the communication (marker interface ICommunciationMaster). Here, InterPartitionComPorts constitute the interface of Partitions and OnChipNetworkDrivers to the InterPartitionCom that is provided by the Hypervisors. As mentioned above, InterPartitionOnChipNetworkExports can be attached to OnChipNetworkDrivers in addition to InterPartitionComPorts. Then, the route to the off-chip communication can be described using a link from the OnChipOffChipExport to the OffChipNetworkPort owned by the Node that contains the respective OnChipOffChipGateway.

5.2.6.2 Hypervisor Model Example Instance

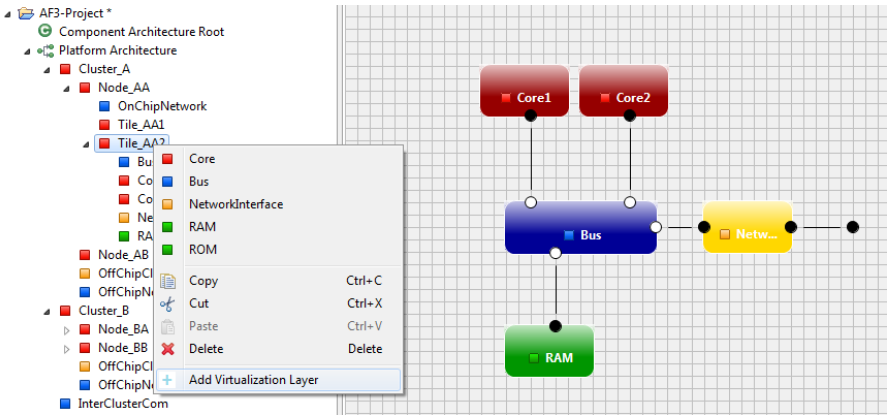


Figure 5.14: Instantiation of system software layer

As pointed out in Section 5.2.6.1, Hypervisors are model elements to describe a virtualization of processor Tiles. In the meta-model provided by the Technical View, it is represented by an additional PlatformArchitecture that hosts the Hypervisor model elements that are linked to the corresponding Tiles of the physical platform architecture. Figure 5.14 shows, how Hypervisors in the system software layer are added using the "Add virtualization layer" command from the context menu of Tile model elements.

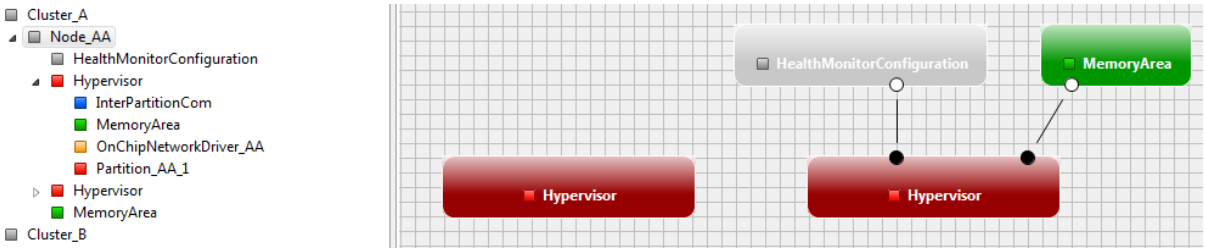


Figure 5.15: Example of the hypervisor layer of a virtual platform model

The resulting system software PlatformArchitecture that includes an example instance of the Hypervisor meta-model is shown in Figure 5.16. In the example, two Hypervisors are defined where a MemoryArea and a HealthMonitorConfiguration are attached to one Hypervisor. The mapping of a Hypervisor to the corresponding Tile is represented by a ResourceLink annotation that is bound to the Hypervisor instance (see Section 5.3). The structure above the Hypervisors reflects the structure of the referenced physical platform architecture, i.e. the node and cluster level is mirrored by corresponding logical PlatfomArchitectureElementGroup model elements.

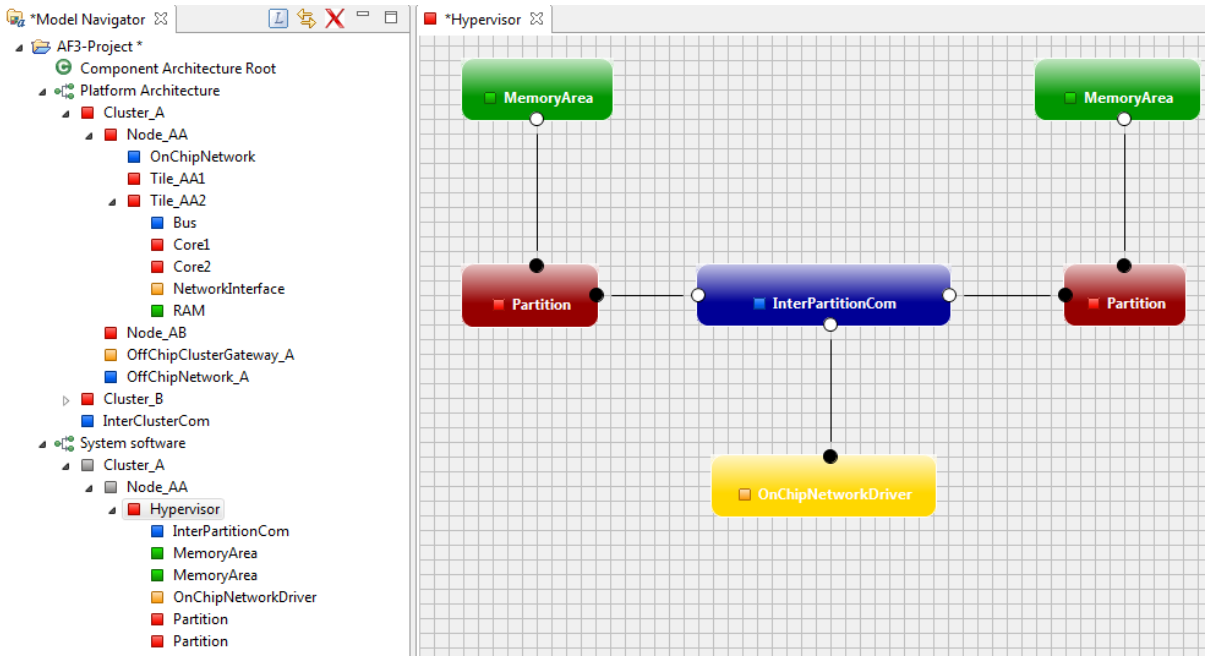


Figure 5.16: Example of a Hypervisor/Partition model.

The internal structure of a Hypervisor is illustrated in Figure 5.16. The model includes Partitions, MemoryAreas, InterPartitionComs and an OnChipNetworkDrivers. As pointed out above, Partitions are modelled as ExecutionUnits of the system software layer that are linked to Cores that are contained in the Tile to which the Hypervisor is linked.

The Hypervisor's partition-to-partition communication facility that enables the exchange of messages between the partitions hosted by the same hypervisor instance is represented by the InterPartitionCom model element. The connection is established by connecting InterPartitionComPorts (black connector) of Partitions with InterPartitionComInterfaces (white connectors) attached to InterPartitionCom model elements. Furthermore, system partitions such as the OnChipNetworkDrivers can be connected to InterPartitionComs by which the access to the OnChipNetwork of the Hypervisor is modelled. The resource mapping of these system partitions is again described using ResourceLink annotations (i.e., the OnChipNetwork hosted by the Tile to which the given Hypervisor is linked).

Finally, the access of partitions to physical memory resources is described using MemoryAreas that can be assigned to one or more Partitions. The MemoryAreas from the example are linked to the RAM resource hosted by the *Tile_AA2* using ResourceLink annotations.

ResourceLink annotations (i.e., the OnChipNetwork hosted by the Tile to which the given Hypervisor is linked).

Finally, the access of partitions to physical memory resources is described using MemoryAreas that can be assigned to one or more Partitions. The MemoryAreas from the example are linked to the RAM resource hosted by the Tile_AA2 using ResourceLink annotations.

5.3 Platform Architecture Annotations

The following tables provide an overview of annotations registered for meta-model entities from the Platform Architecture Meta-Model. Annotations that are registered for super classes (like ExecutionUnits) are naturally attached to all inheriting classes (like Tiles).

5.3.1 Annotations registered for all Platform Elements

Annotation Name	Corresponding plugins	Description
ArchitectureDomainLabel [DerivedAnnotation]	org.fortiss.af3.platform	Returns a label that denotes the IArchitectureDomain of the annotated model element.
PlatformArchitectureElementTypeLabel [DerivedAnnotation]	org.fortiss.af3.platform	Returns a label indicating the "physical" type of the annotated model element, like a logical element or part of an IP Core.

5.3.2 Annotations registered for ExecutionUnits

Annotation Name	Corresponding plugins	Description
ExecutionUnitPower	eu.dreamsproject.platform	The average power consumption of the annotated hardware element when executing a software Component for a given time.
DeploymentGranularity [DerivedAnnotation]	eu.dreamsproject.platform	Boolean flag that allows to specify the ExecutionUnits onto which Components shall be mapped. If the flag is set to *true* for a given ExecutionUnit *ex*, child ExecutionUnits of *ex* are not considered as deployment targets. Hence, this annotation can be used to define the deployment granularity of a hierarchical PlatformArchitecture.
FailureRate	eu.dreamsproject.platform	The failure rate of the annotated hardware element given as its failure probability.

SafeFailureFraction	org.fortiss.af3.safety	The Safe Failure Fraction of the annotated hardware element as defined in IEC 61508.

5.3.3 Annotations registered for Cores

Annotation Name	Corresponding plugins	Description
ProcessorSpeed	org.fortiss.af3.platform	Maximum CPU frequency that can be achieved by the annotated `Core`.

5.3.4 Annotations registered for TransmissionUnits

Annotation Name	Corresponding plugins	Description
FailureRate	eu.dreamsproject.platform	The failure rate of the annotated hardware element given as its failure probability. In case the `TransmissionUnit` represents a so-called black channel, this parameter is the "residual error rate" according to IEC 61784-3 with the assumption of a bit error rate of 10^{-2}.
SafeFailureFraction	org.fortiss.af3.safety	The Safe Failure Fraction of the annotated hardware element as defined in IEC 61508.
TransmissionUnitBand-width	org.fortiss.af3.timing	Bandwidth of the annotated `TransmissionUnit` given in Mbyte per second. Describes the raw throughput.
TransmissionUnitPower	eu.dreamsproject.platform	Power consumption of the annotated `TransmissionUnit` for transmitting a single byte.

5.3.5 Annotations registered for MemoryUnits

Annotation Name	Corresponding plugins	Description
MemoryAddress	org.fortiss.af3.platform	The start address of the annotated `MemoryUnit`. Used in hardware platforms for global address spaces and for segregation of virtual memory allocations.
MemorySize	org.fortiss.af3.platform	Capacity of a `MemoryUnit` in Bytes.

5.3.6 Annotations registered for RAM

Annotation Name	Corresponding plugins	Description
RamType	org.fortiss.af3.platform	Allows a fine-grained specification of the RAM type that is used to implement the annotated RAM element.

5.3.7 Annotations registered for Tiles , Partitions and MemoryAreas

Annotation Name	Corresponding plugins	Description
ResourceLink	org.fortiss.af3.platform	Resource requirements (1:n relationship) between platform elements in different layers of the platform, e.g. from elements of the system software layer to elements of the physical platform.

5.3.8 Annotations registered for Partitions

Annotation Name	Corresponding plugins	Description
PartitionFlags	eu.dreamsproject.platform	Flags to be set when configuring the annotated `Partitions`. - None - System - Boot - ICache disabled - DCache disabled - Floating point support

5.3.9 Annotations registered for HealthMonitorConfigurations

Annotation Name	Corresponding plugins	Description
HealthMonitorConfiguration	eu.dreamsproject.platform	Actions that shall be triggered by the health monitor of the connected hypervisor if the defined (faulty) behaviour is detected. Furthermore, it can be defined whether monitored misbehaviour shall be logged. The following events are defined: - Internal error - Unexpected trap - Partition error - Partition integrity - Mem protection - Overrun - Scheduler error - Watchdog timer - Incompatible interface - Undefined instruction - Prefetch abort - Data abort - Data alignment fault - Data background fault - Data permission fault - Instruction alignment fault - Instruction background fault - Instruction permission fault The following action can be triggered: - Ignore (= do nothing) - Shutdown - Partition cold reset - Partition warm reset - Hypervisor cold reset - Hypervisor warm reset - Suspend - Halt - Propagate - Switch to maintenance

5.4 Interfaces to other Meta-Models

The technical architecture meta-model does not contain references to meta-models from other viewpoints described in this document. However, as pointed out in Section 2.2.2, the technical architecture meta-model is referred to by a number of meta-models defined in other viewpoints.

6 Deployment Viewpoint

This viewpoint collects all deployment related model kinds. For this deliverable, it only comprises meta-models required to describe the mapping of model elements from the logical view to model elements of the technical view. The follow-up document D1.6.1 “Meta-models for platform-specific modelling” will focus on enhancing this viewpoint with description mechanisms for the allocation of platform resources.

6.1 Deployment Meta-Model

The mapping meta-model is used to describe the mapping of a model element from the logical view to model elements of the technical view, e.g. of `Component`s to `Core`s or `Partition`s, or of (logical) `Port`s to `Transceiver`s provided by the platform.

A `Deployment` of the mapping model can be instantiated as follows:

- Manually by the designer, using the deployment model editor provided by AutoFOCUS3 (see Section 6.4).
- As the result of a Design Space Exploration (see Chapter 3 of deliverable D4.1.2). Here, the exploration evaluates the Events defined in temporal viewpoint and the `DeploymentGranularity` annotation of `ExecutionUnit`s to derive the relevant model elements considered by the exploration. Thereby, `Event`s are used to select the components to be deployed from the hierarchical logical architecture. `Component`s (including their attached `Port`s) that are not referenced by any `Event` will not be considered in the exploration. If a `Component` or at least one of its associated `Port`s is referenced by an `Event`, but none of its contained elements, only the containing `Component` is deployed by the DSE (since it contains the most fine-grained activation specification in the corresponding sub-model). Hence, for a given `ComponentArchitecture`, the temporal viewpoints allows to specify the granularity at which the deployment of logical `Component`s to the `PlatformArchitecture` should be performed. The deployment targets (`Partion`s in DREAMS) of the DSE are defined by the `DeploymentGranularity` annotation that allows specifying the set of `Partition`s considered by the DSE.

Name	**Deployment Meta-Model**	
Description	The goal of the deployment meta-model is creating a link between the logical architecture and the platform which realizes the logical architecture.	
Ecore file	`deployment.ecore`	
Plugin	`org.fortiss.af3.deployment`	
Packages	`org.fortiss.af3.deployment` `org.fortiss.af3.deployment.generator`	AutoFOCUS3 deployment meta-model Package for deployment-dependent code generation (currently empty)
Dependencies	`org.fortiss.af3.component` (see Section 4.1) `org.fortiss.af3.platform` (see Section 5.1) `org.fortiss.tooling.base` (see Section 3.2.6.2) `org.fortiss.tooling.kernel` (see Section 3.2.6.1)	

Table 6.1: Deployment Meta-Model

In Figure 6.1, the UML class diagram of the mapping meta-model is shown.

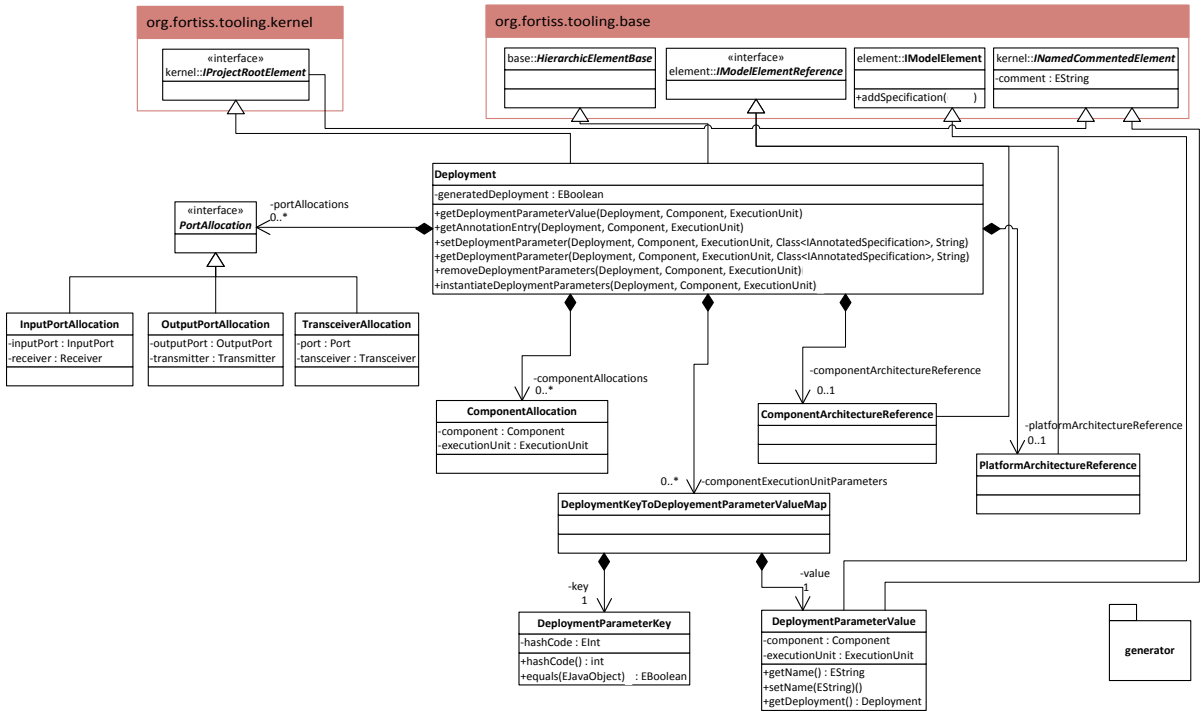


Figure 6.1: Mapping Meta-Model (UML Diagram of package org.fortiss.af3.deployment.model)

The following classes are defined to describe a mapping:

- `Deployment`:
  - "Root" class that contains the logical architecture ↔ hardware platform mapping.
  - **Attributes:**
    - `generatedDeployment`: indicates whether this deployment is the result of a Design Space Exploration.
  - **Operations:**
    - `hasDeploymentParameters()`: Returns whether this `Deployment` has deployment specific parameters (via annotations).
    - `clearDeploymentParameters()`: Removes all parameters defined in the `DeploymentKeyToDeploymentParameterValueMap` of this `Deployment`.
    - `instantiateDeploymentParameter(Component, ExecutionUnit)`: Instantiates the `DeploymentKeyToDeploymentParameterValueMap` considering each possible `Component-ExecutionUnit` combination.
    - `removeDeploymentParameter(Component, ExecutionUnit)`: removes the deployment specific parameters of s given `Component-ExecutionUnit` pair from the `DeploymentKeyToDeploymentParameterValueMap`.
    - `getDeploymentParameter(Component, ExecutionUnit, EJavaClass<IAnnotatedSpecification>, EString, EString)`: Returns the deployment specific parameter of a given `Component-ExecutionUnit` pair and the `IAnnotatedSpecification` defining the parameter.
    - `setDeploymentParameter(Component, ExecutionUnit, EJavaClass<IAnnotatedSpecification>, EString)`: Sets the

deployment specific parameter of a given Component–ExecutionUnit pair and the IAnnotatedSpecification defining the parameter.

- ComponentArchitectureReference:
  - References the ComponentArchitecture whose subelements are deployed onto a PlatformArchitecture.
- PlatformArchitectureReference:
  - References the PlatformArchitecture onto which a defined ComponentArchitecture is deployed.
- ComponentAllocation:
  - Connects a Component with an ExecutionUnit that executes the Component's realization.
  - **Attributes:**
    - component: References the Component of this Component↔ExecutionUnit allocation, i.e. the deployment source.
    - executionUnit: References the ExecutionUnit of this Component↔ExecutionUnit allocation, i.e. the deployment target.
- PortAllocation:
  - Marker interface for allocations of Ports and Transceivers.
- TranceiverAllocation:
  - Allocates the Port of a Component to a Transceiver of the given hardware platform. The Transceiver naturally needs to be located at the same ExecutionUnit onto which the respective Component has been allocated.
  - The PortAllocation is needed for ExecutionUnits that are connected via bidirectional interfaces to communication resources like busses.
  - **Attributes:**
    - port: References a Port(input or output) that is allocated to a Transceiver.
    - transceiver: References a Transceiver to which a Port(input or output) is allocated.
- InputPortAllocation:
  - Allocates the InputPort of a Component to a Receiver of the given hardware platform. The Receiver naturally needs to be located at the same ExecutionUnit onto which the respective Component has been allocated.
  - The InputPortAllocation is required instead of the PortAllocation if the connected communication resource of the ExecutionUnit differentiates between incoming and outgoing messages.
  - **Attributes:**
    - inputPort: References an InputPort that is allocated to a Receiver.
    - receiver: References a Receiver to which an InputPort is allocated.
- OutputPortAllocation:
  - Allocates the OutputPort of a Component to a Transmitter of the given hardware platform. The Transmitter naturally needs to be located at the same ExecutionUnit onto which the respective Component has been allocated.
  - The OutputPortAllocation is required instead of the PortAllocation if the connected communication resource of the ExecutionUnit differentiates between incoming and outgoing messages.

  - **Attributes:**
    - `outputPort`: References an `OutputPort` that is allocated to a `Transmitter`.
    - `transmitter`: References a `Transmitter` to which an `OutputPort` is allocated.
- `DeploymentKeyToDeploymentParameterValueMap`:
  - Map that relates a pair consisting of a `Component` and an `ExecutionUnit` to a set of parameters describing the properties of an `ComponentAllocation` of the defined pair.
  - **Attributes:**
    - `key`: DeploymentParameterKey that defines a pair of a `Component` and an `ExecutionUnit` that is used to identify their deployment specific parameters.
    - `value`: `DeploymentParameterValue` to which the parameters for a pair of a `Component` and an `ExecutionUnit` are bound (as annotations).
- `DeploymentParameterKey`:
  - Used to identify the deployment specific parameters of a pair of a `Component` and an `ExecutionUnit`.
  - **Attributes:**
    - `hashCode`: Contains the hash code of the `DeploymentParameterKey` that is derived from the hash codes of the corresponding `Component` and `ExecutionUnit`.
  - **Operations:**
    - `hashCode()`: Returns the `hashCode` that identifies a `DeploymentParameterKey` object, i.e. a specific `Component-ExecutionUnit` pair.
    - `equals(EJavaObject)`: Evaluates whether the object on which the method is called equals the object given as a parameter.
- `DeploymentParameterValue`:
  - Contains the deployment specific parameters of a pair of a `Component` and an `ExecutionUnit`.
  - **Attributes:**
    - `component`: References the `Component` of the associated `Component-ExecutionUnit` pair.
    - `executionUnit`: References the `ExecutionUnit` of the associated `Component-ExecutionUnit` pair.
  - **Operations:**
    - `getName()`: Returns the name of the referenced `component` and `executionUnit` in the form of a tuple of their names.
    - `setName(EString)`: Overriden in order to make the name of `DeploymentParameterValue` read-only.
    - `getDeployment()`: Returns the `Deployment` which contains this `DeploymentParameterValue`.

`Port`s model all inbound and outbound interfaces of `Component`s. This includes communication between components as well the connection of the modelled system with its environment (reception of sensor values and control commands for actuators).

Hence, the mapping of ports from the logical component architecture is divided into `TranceiverAllocation`s, `InputPortAllocation`s, and `OutputPortAllocation`s.

- In a deployed application, communication between Components corresponds to the exchange of messages over TransmissionUnits connecting the corresponding ExecutionUnits. In case the Transceivers of ExecutionUnits is capable of performing bidirectional communication, the mapping of a Port (i.e., an OutputPort in case of a sending Component, and an InputPort, in case of a receiving Component) used as an interface for inter-Component communication is captured by the attributes of a TranceiverAllocation.
- If the platform element to which a logical Port should be mapped to allows only unidirectional communication, the mapping is described using Input- and OutputPortAllocations. On the one hand, this is the case in if the InputPort (OutputPort) of a Component is mapped to the Receiver (Transmitter) provided by sensor (actuator) of the platform. Aside from interfacing sensors and actuators, the separation of inbound and outbound communication in the mapping can also be relevant for TransmissionUnits that provide (separated) unidirectional communication channels.

6.2 Deployment Annotations

In the deployment model, each deployment-specific parameter can be specified for every possible mapping of Components to ExecutionUnits. Those parameters are realized as annotations that are bound to DeploymentParameterValues which are contained in the Deployment's DeploymentKeyToDeploymentParameterValueMap. The map identifies the parameters of a specific Component-ExecutionUnit pair using DeploymentParameterKeys. In the following table, the annotated parameters are summarized.

Annotation Name	Corresponding plugins	Description
EnergyConsumption	org.fortiss.af3.platform	Contains the average energy consumption (in Joule) when executing the Component of the annotated Component-ExecutionUnit pair on the corresponding ExecutionUnit.
Wcet	org.fortiss.af3.timing	Allows the user to define the WCET (in seconds) when executing the Component of the annotated Component-ExecutionUnit pair on the corresponding ExecutionUnit.

6.3 Interfaces to other Meta-Models

As described in Section 6.1, the Deployment Meta-Model describes the mapping of model elements of a logical component architecture (see Section 4) to the model elements of a platform architecture (see Section 5). As side from that, the Deployment Meta-Model does not relate to any other meta-model defined in this document.

6.4 Deployment Model Example Instance

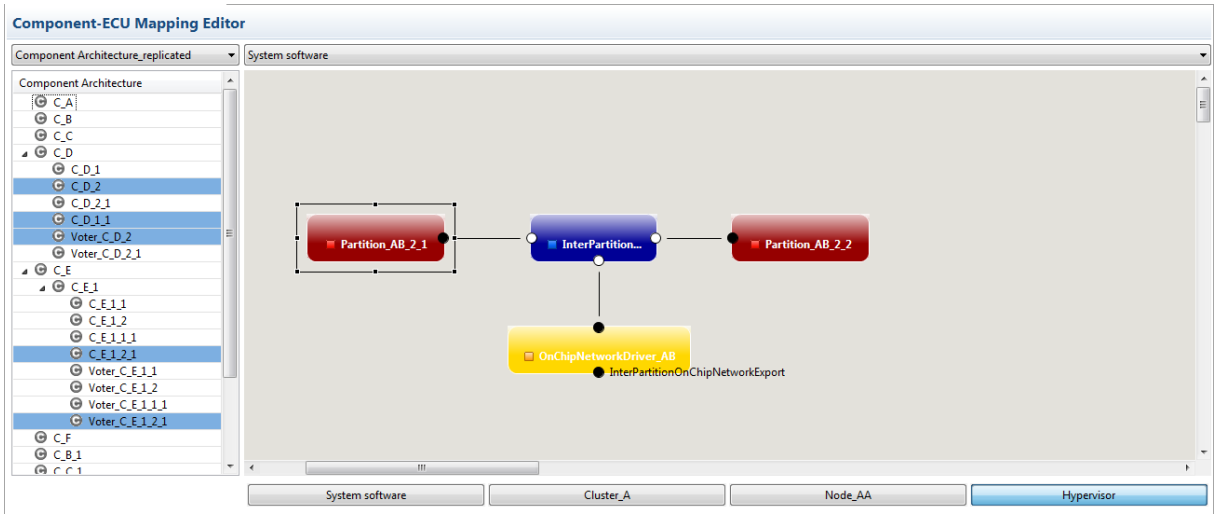


Figure 6.2: Deployment Editor

Figure 6.2 shows the editor that can be used to edit and display mappings of elements of the logical component architecture to elements of the technical platform architecture mappings. Logical `Components` (located in the tree-view on the left side of Figure 6.2) can be allocated to `ExecutionUnits` contained in the platform architecture using drag-and-drop. By double-clicking onto hierarchical platform elements, and using the breadcrumb widget on the bottom of the editor it is possible to navigate to the desired `ExecutionUnit`. The highlighted components in the left listing in Figure 6.2 indicate those `Components` that are allocated to the selected `ExecutionUnit` in the graphical editor. As pointed out in Section, 6.1 the result of such `Component`-to-`ExecutionUnit` mapping is described using a `ComponentAllocation` within the edited deployment model. All `ComponentAllocations` contained in a deployment model can also be viewed in the separate tab *Raw Mappings* (see Figure 6.3).

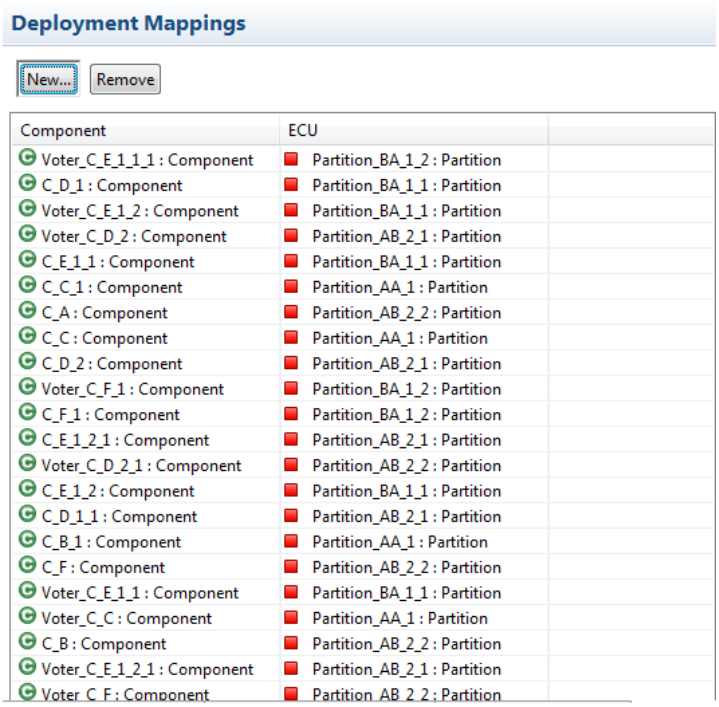


Figure 6.3: Table with "raw" component-partition allocations of a deployment model

The deployment-specific parameters introduced in Section 6.1 and 6.2 can be edited in the annotation view of a selected `Deployment`. As shown in Figure 6.4, each `Component-ExecutionUnit` pair is represented as a row – for each of the deployment-specific parameters, the table contains a dedicated column.

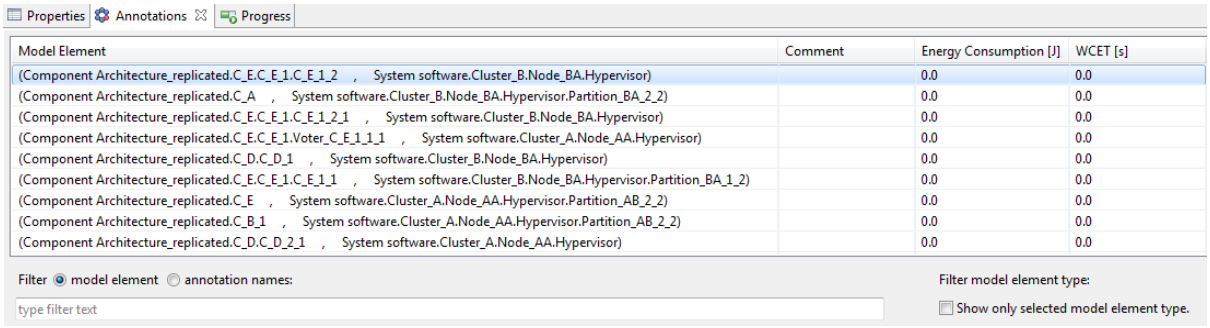


Figure 6.4: Deployment-specific parameters (accessible in Annotation View)

7 Temporal Viewpoint

The temporal viewpoint provides the elements required for the description of the timing properties of a DREAMS application in AutoFOCUS3.

7.1 DREAMS Timing Meta-Model

The temporal viewpoint consists of the timing meta-model described this section. It is based on the Timmo-2-use[1] timing meta-model [10] adapted for the DREAMS specificities.

Name	**DREAMS Timing Meta-Model**	
Description	The goal of the DREAMS timing meta-model is to describe all the timing requirements and properties of a DREAMS application.	
Ecore file	`timing.ecore`	
Plugin	`eu.dreamsproject.rtaw.timing`	
Packages	`eu.dreamsproject.rtaw.timing`	DREAMS timing meta-model
Dependencies	`org.fortiss.af3.component` (see Section 4.1)`org.fortiss.tooling.kernel` (see Section 3.2.6.1)	

Table 7.1: DREAMS Timing Meta-Model

In Figure 7.1, the UML class diagram of the timing meta-model is shown.

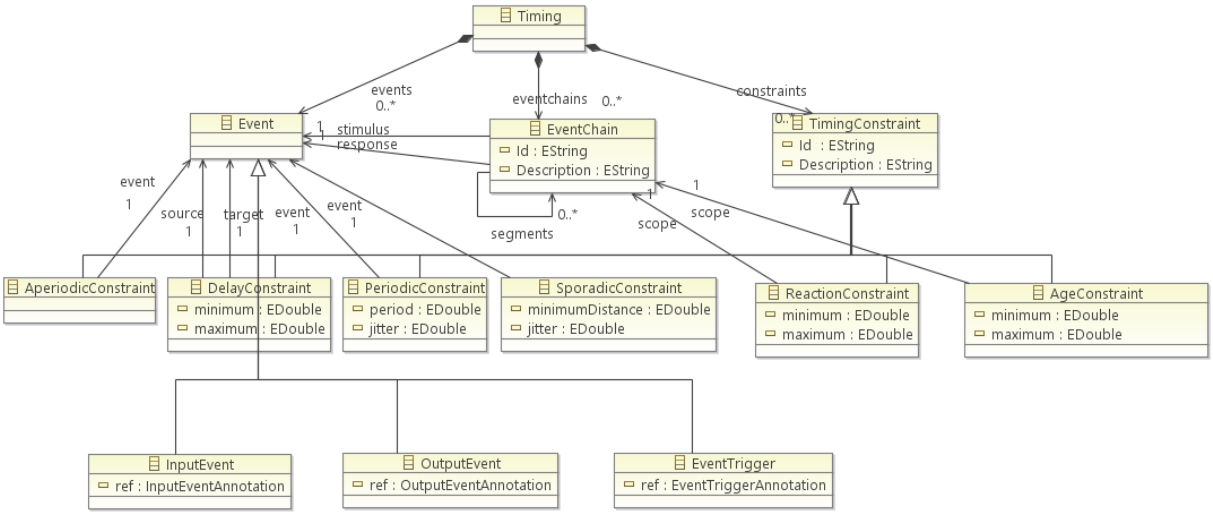


Figure 7.1: Timing Meta-Model (UML Diagram of package `eu.dreamsproject.rtaw.timing.model`)

The meta-model consist of the package `eu.dreamsproject.rtaw.timing.model` that contains the definition of the timing meta-model.

The following classes are defined to describe timing information:

- `Timing`
  - Root element for timing information meta-model. The collection of timing descriptions, namely events and event chains, and the timing constraints imposed on these events and event chains.

- TimingConstraint. This is an abstract element. It is not a design constraint but either a requirement or the result of a validation. TimingConstraint offers several means to constrain the time occurrences of events.
    - **Attributes:**
        - Id: a string identifier to for constraint traceability.
        - Description: a string description of the constraint.
- AgeConstraint:
    - An age constraint defines how long before each response a corresponding stimulus must have occurred. It applies to a TimingChain.
    - **Attributes:**
        - minimum: Minimum value of the AgeConstraint. Value in seconds.
        - maximum: Maximum value of the AgeConstraint. Value in seconds.
        - scope: Reference to the TimingChain on which this constraint applies.
- DelayConstraint:
    - A DelayConstraint imposes limits between the occurrences of an event called source and an event called target.
    - **Attributes:**
        - source **:** Reference to the source Event
        - target : Reference to the target Event
        - lower: Lower value of the DelayConstraint. Value in seconds.
        - upper: Upper value of the DelayConstraint. Value in seconds.
- ReactionConstraint:
    - A ReactionConstraint defines how long after the occurrence of a stimulus a corresponding response must occur.
    - **Attributes:**
        - minimum: Minimum value of the ReactionConstraint. Value in seconds.
        - maximum: Maximum value of the ReactionConstraint. Value in seconds.
        - scope: Reference to the TimingChain on which this constraint applies.
- PeriodicConstraint:
    - A PeriodicConstraint describes an event that occurs periodically.
    - **Attributes:**
        - period**:** The effective ideal separation between two successive occurrences of event without jitter. Value in seconds.
        - jitter**:** Describes the local deviation from the strictly sporadic pattern. Value in seconds.
        - event**:** Reference to the Event on which his constraint applies.
- SporadicConstraint:
    - A SporadicConstraint describes an event that occurs with a minimum interarrival time in between successive occurrences.
    - **Attributes:**
        - minimumDistance**:** The effective minimum distance between any two occurrences of event. Value in seconds.
        - jitter**:** Describes the local deviation from the strictly sporadic pattern. Value in seconds.
        - event**:** Reference to the Event on which his constraint applies.
- AperiodicConstraint: An AperdiodicConstraint describes an event for which only one instance occurs.
- EventChain:

  - An `EventChain` is a container for a pair of events that must be causally related.
  - **Attributes**:
    - `Id`: a string identifier to for event chain traceability.
    - `Description`: a string description of the event chain.
    - `stimulus`: Reference to the `Event` that stimulates the steps to be taken to respond to this event.
    - `response`**:** Reference to the `Event` that is a response to a stimulus that occurred before.
    - `segment`**:** Ordered list of reference to `EventChains` in sequence.
- `Event`: This is a sequence of times indicating the times that each event occurrence is predicted to occur.
- `InputEvent`:
  - This links the timing model elements to component `InputPort`.
  - **Attributes:**
    - `ref`**:** References the `InputPortAnnotation` of an `InputPort` from the logical component architecture meta-model.

- `OutputEvent`:
  - This links the timing model elements to component `OutputPort`.
  - **Attributes**:
    - `ref`**:** References the `OutputPortAnnotation` of an `OutputPort` from the logical component architecture meta-model.
- `EventTrigger`:
  - This links the timing model elements to a `Component`.
  - **Attributes:**
    - `ref`**:** References the `ComponentAnnotation` from the logical component architecture meta-model.

7.2 Interface to other Meta-Models

The DREAMS timing meta-model contains references to the logical component architecture meta-model described in Section 4. In particular, the timing meta-model references:

- `Component` (via `ComponentAnnotations`)
- `InputPorts` (via `InputPortAnnotations`)
- `OutputPorts` (via `OutputPortAnnotations`)

7.3 DREAMS Timing Model Example Instance

The Timing viewpoint is instantiated for the expression of the requirements of a braking system. It is illustrated Figure 7.2.

In this example, the following timing requirements are described:

- End-to-end delay: The vehicle must start decelerating within the driver's reaction time (250ms) after the driver has indicated his wish to do so.
- This End-to-end delay is further decomposed into segments allowing time budget allocation between `InputEvent` and `OutputEvent` on `Components`.
- The `EventTrigger` on the pedal sensor allows the specification of the brake pedal sensing period.

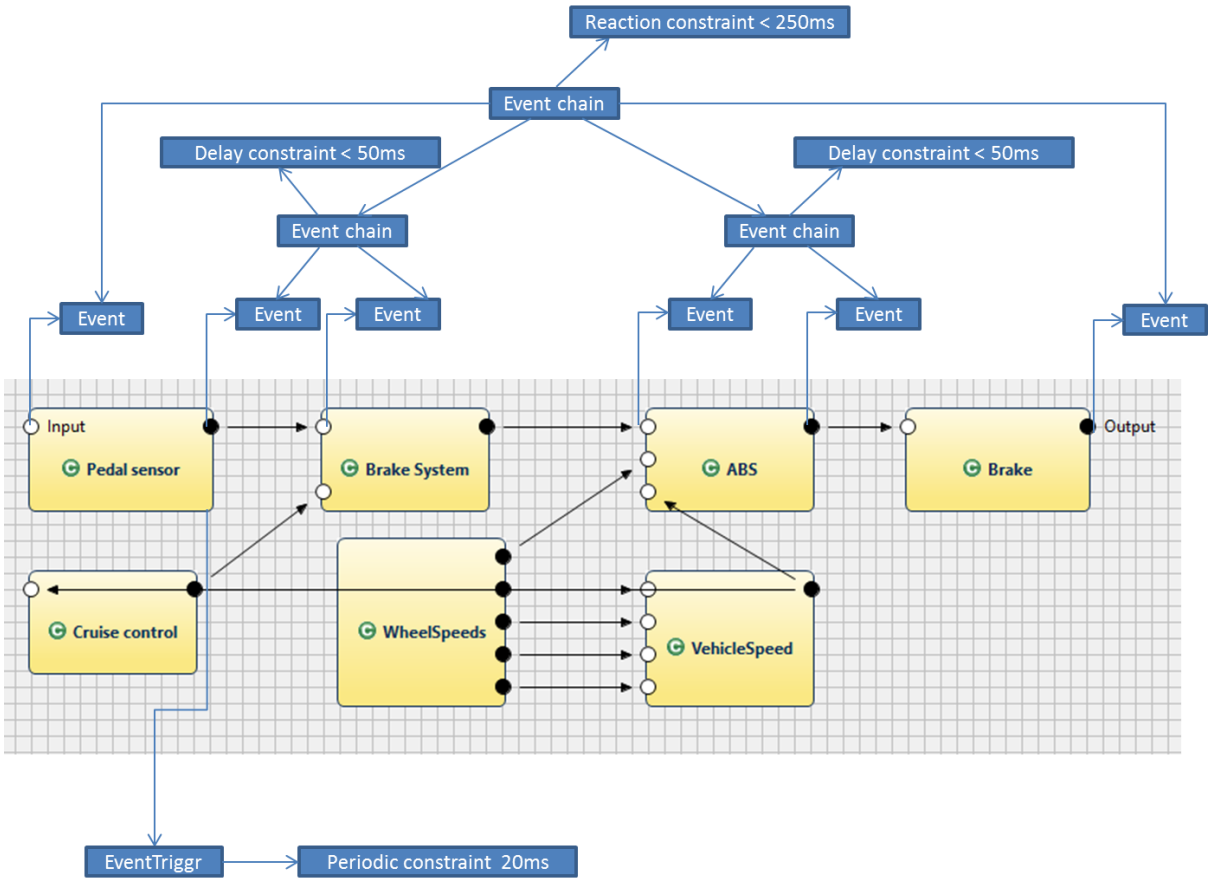


Figure 7.2: Braking System (Timing viewpoint illustration)

8 Extra-functional Viewpoints

8.1 Safety Viewpoint

Safety viewpoint consist of three meta-models that are used to add safety consistency checking functionality to a dreams project.

In order to describe a safety model, three safety meta-models developed in the MultiPARTES project[7], have been adapted and enhanced to the more sophisticated needs.

Name	**Safety Viewpoint** • **IEC 61508 and Diagnostic Techniques and Measures Meta-Model** • **Safety Compliance Meta-Model** • **Safety Compliance Constraint Meta-Model**	
Description	The goal of these hierarchic element meta-model is to: • Provide the basis for the description of IEC61508 SIL levels, IEC 61508 Systematic Capability (IEC 61508-2 and IEC 61508-3) related to measures against systematic faults, and Diagnostic Techniques and Measures in IEC 61508-2, Annex A. • Allow specifying Safety Manuals (with a subset of IEC 61508-2 and IEC 61508-3 Annex D's attributes) for SCItems (Safety Compliance Items related to Component, Platform, and System Software elements. • Allow Safety Consistency Rules to check safety consistency of deployments.	

Ecore file	`IEC61058.ecore`
	`SafetyCompliance.ecore`
	`SafetyComplianceConstraint.ecore`	

Plugin	`eu.dreamsproject.ikerlan.safetystandards`	

Packages	`eu.dreamsproject.ikerlan.safetystandards.IEC61508`
	`eu.dreamsproject.ikerlan.safetystandards.SafetyCompliance`
	`eu.dreamsproject.ikerlan.safetystandards.SafetyComplianceConstraint`
		IEC 61508 standard and Diagnostic Techniques and Measures
		Safety Compliance of a dreams project
		Safety Compliance Constraints generated by a Safety Compliance Specification

Dependencies	`eu.dreamsproject.platform`
	`org.fortiss.af3.platform`
	`org.fortiss.af3.component`
	`org.fortiss.af3.deployment`	

Table 8.1: Safety Meta-Models

The following sections describe these meta-models in detail.

[7] http://www.multipartes.eu/

8.1.1 IEC 61508 and Diagnostic Techniques and Measures

This meta-model is used to represent IEC 61508 based safety standard, SIL integrity levels and Diagnostic Techniques and Measures defined in the standard. Figure 8.1 shows the class diagram defined in the `.ecore` file.

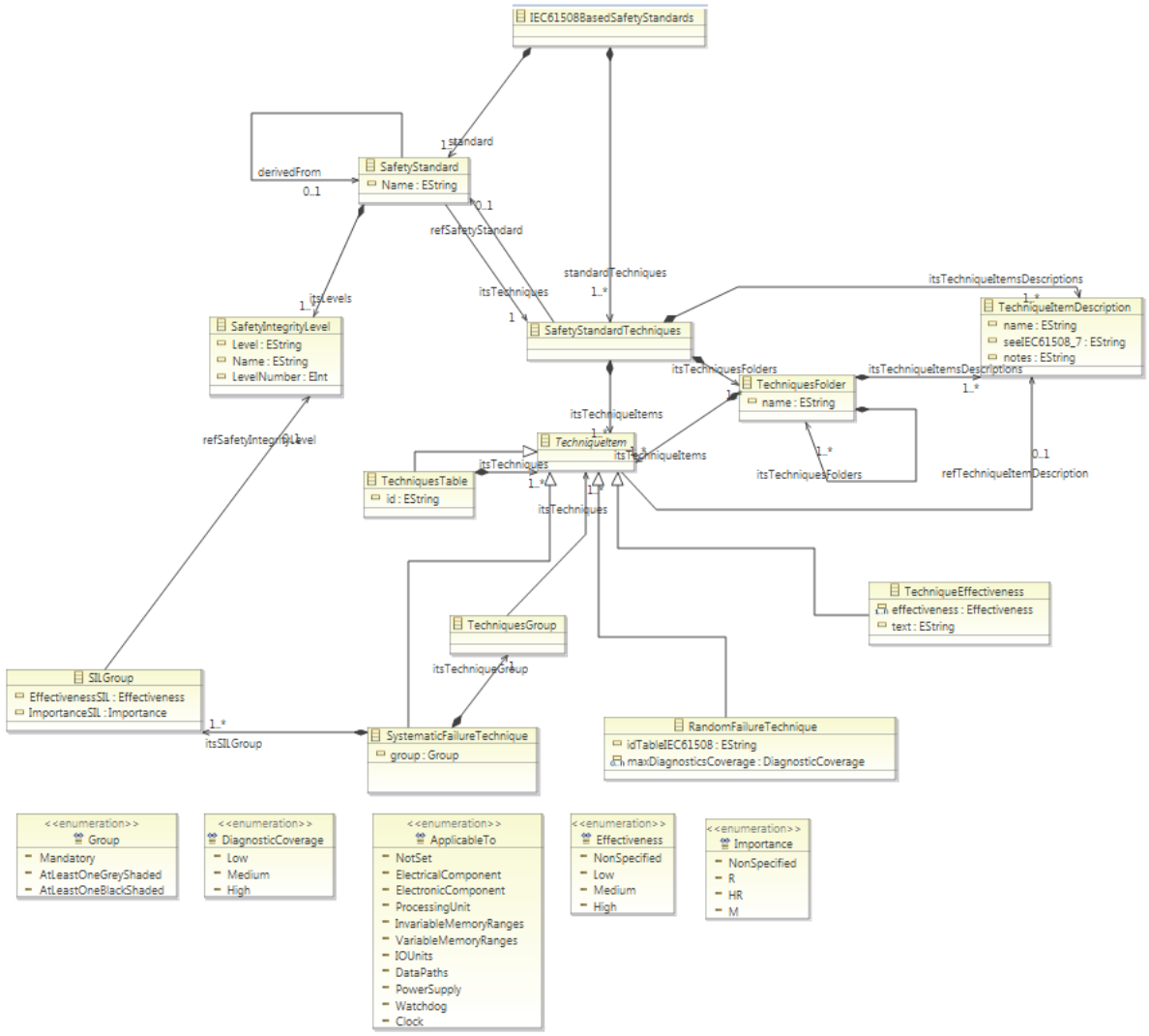


Figure 8.1: Classes of meta-model of IEC 61508 Standard with Diagnostic Techniques and Measures.

The meta-model consist of the package `eu.dreamsproject.ikerlan.safetystandards.IEC61508` that contains the core definition of the standard and its techniques. The following classes are available:

- `IEC61508BasedSafetyStandard`. Root of the meta-model.
- `SafetyStandards`.
  - Each *SafetyStandard* has a name.
  - A *SafetyStandard* has N *SafetyIntegrityLevels.* (i.e., IEC 61508 has 4 integrity levels: "SIL1", "SIL2", "SIL3", and "SIL4").
- `SafetyStandardTechniques` may have several `TechniqueFolder`. Each folder keeps information of one part of the Standard. Examples are "Annex A", "Techniques&Measures" of IEC 61508-2

- TechniqueFolder may have TechniqueFolder inside, allowing to define a recursive structure of TechniqueFolder.
- SafetyStandarTechnique has TechniqueItems. An example of a TechniqueItem is *"Watch-dog with separate time base and time-window".*
- TechniqueItem
  - has a TechniqueItemDescription.
  - Each TechniqueItem belongs to one *TechniqueFolder*.
  - Each TechniqueItem belongs also to one TechniqueTable. An example of a TechniqueTable is *"A.10 – Program sequence (watch-dog)".*
  - Each TechniqueItem has a TechniqueEffectiveness *(Low, Medium, High)*
- TechniqueItems are classified as well, attending to its goal:
  - Some are to control Random Failures. In this case RandomFailureTechnique entity is used, to specify the standard table and the DiagnosticCoverage needed.
  - Others are to control Systematic Failures. In this case SystematicFailureTechnique entity is used. There techniques are grouped in TecniquesGroups.
- There are also some other entities, which are vocabularies:
  - DiagnosticCoverage :
    - Low
    - Medium
    - High
  - Effectiveness: Effectiveness of a technique.
    - NonSpecified
    - Low
    - Medium
    - High
  - Group: This is used to group techniques into a table. The possible values are:
    - Mandatory
    - AtLeastOneGreyShaded
    - AtLeastOneBlackShaded
  - Importance: Importance of a technique. The possible values are:
    - NonSpecified
    - R (recommended)
    - HR (highly recommended)
    - M (mandatory)
  - ApplicableTo: Classes of elements where techniques may be applied to. Values are:
    - ElectricalComponents
    - ProcessingUnits
    - PowerSupply
    - etc.

8.1.2 Safety Compliance Meta-Model

This meta-model is used to represent safety specifications of Component Architecture, Platform Architecture and System Software Architecture.

Figure 8.2 shows the class diagram of the SCItem class (key) and SafetyManual classes defined in the .ecore file.

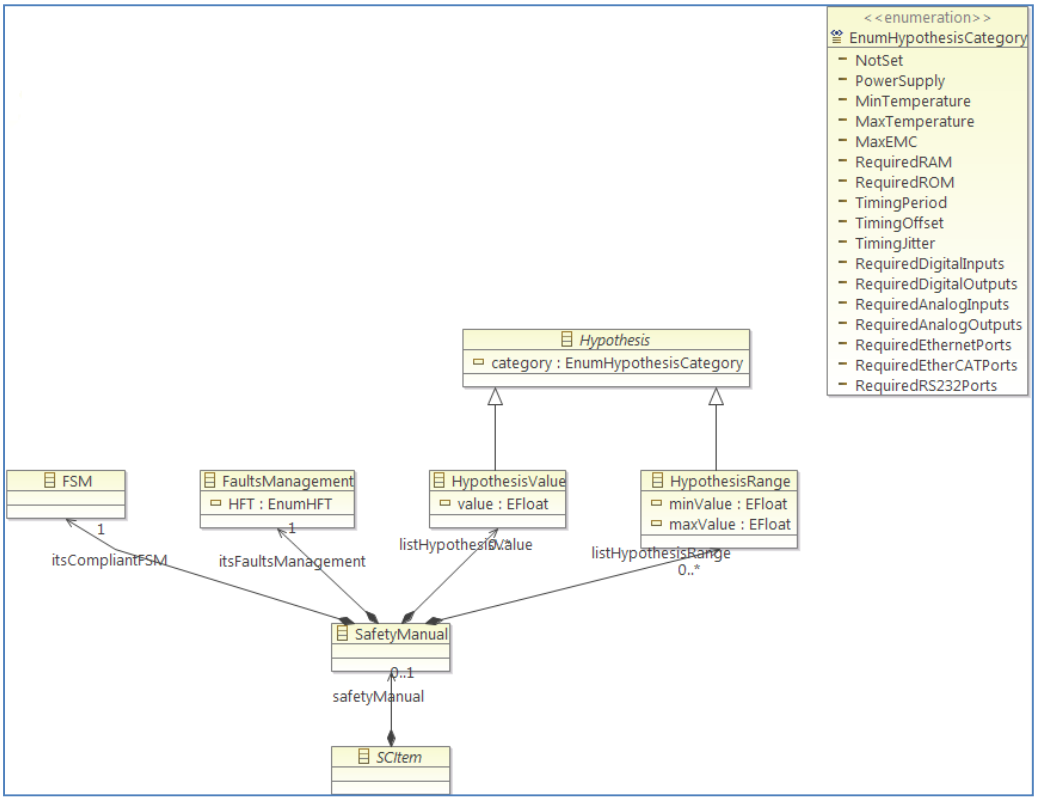


Figure 8.2: Classes of SCItem and Safety Manual meta-model

The meta-model consist of the package `eu.dreamsproject.ikerlan.safetystandards.IEC61508.SafetyCompliance` that contains the core definition. The following classes are available:

- `SCItem`: the key class in the meta-model. Represents a Safety Compliant Item that in DREAMS may be a software Component, hardware Cluster, hardware Node, hardware Tile, software Hypervisor or software Partition. A SCItem may define a Safety Manual.
- `SafetyManual`:
  - `FSM`, defining:
    - Safety Standard (`IEC61508`)
    - Safety Integrity Level (`SIL1`, `SIL2`, ...)
    - Systematic Capability (`SC1`, `SC2`, `SC3`, `SC4`)
  - `FaultsManagement`, defining:
    - `HFT`: Hhardware Fault tolerance level (`HFT0`, `HFT1`, `HFT2`, `HFT3`) in case of hardware nodes
    - `DiagnosticTechniquesItem`: list of IEC61508-2 Annex A (tables A.2 to A.17) Diagnostic and Measures Techniques.
  - A collection of `HypothesisValue` that specify assumptions about the types of faults, the rate at which components fail and how components may fail
  - A collection of `HypothesisRange`s that specify assumptions about the types of faults, the rate at which components fail and how components may fail.
- `Hypothesis`: base class of Hypothesis defining the category of the hypothesis as an enumerated value.

Figure 8.3 shows the classes needed to attach and manage safety manuals to dreams project hierarchy (Component, Platform and System Software) and to prepare the whole structure to tackle with variability.

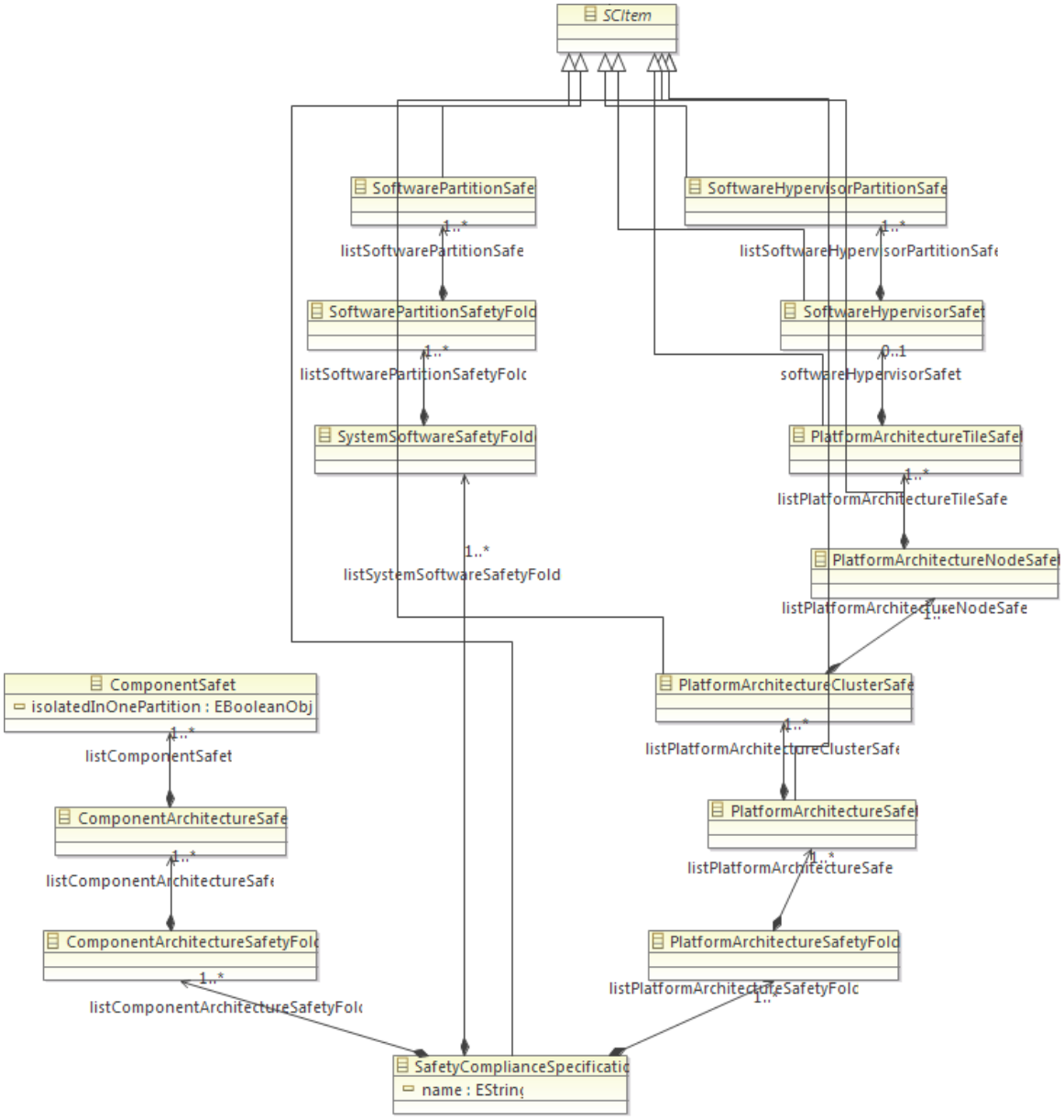


Figure 8.3: Classes of Safety Compliance meta-model

The following main classes are shown in the hierarchy:

- `SafetyComplianceSpecification`: root class of the hierarchy that represents a complete safety specification for a dreams project and its variants.
- `ComponentArchitectureSafetyFolder`: Collection of `ComponentArchitectureSafety` roots.
- `PlatformArchitectureSafetyFolder`: Collection of `PlatformArchitectureSafety` roots.
- `ComponentArchitectureSafety`: Represents a Component or subcomponent of the Logical viewpoint. The following properties are defined:

  - RefComponent: Reference to the Component of the project.
  - Safety Standard and Safety Integrity Level: SIL level claimed for the Component.
  - RefCore: if the safety engineer wants to make sure that any deployments involving this component deploys the component in a given core, then this field contains a reference to the core.
  - RefTile: if the safety engineer wants to make sure that any deployments involving this component deploys the component in a given tile, then this field contains a reference to the tile.
  - Isolated in One Partition (Boolean): True if the safety engineer wants to make sure that any deployments involving this component deploys the component "alone" in one partition (not shared with any other component).
  - NeedAccessListHWResources: List of hardware resources (watchdogs, clocks, tiles, etc.) to which the component need access rights. This is for example needed for a Component that resets a Watchdog and is deployed into a Partition. In this case the Partition has to be configured in the hypervisor having access to those HW resources.
- `PlatformArchitectureSafety`, `PlatformArchitectureClusterSafety`, `PlatformArchitectureNodeSafety`, `PlatformArchitectureTileSafety`: All of them represents `SCIItem` and therefore may contain a `SafetyManual`:
- `SoftwareHypervisorSafety`: It is a `SCItem` (generated by Virtualization Layer) and may contain a `SafetyManual`. In addition to this, may contain a collection of `SoftwareHypervisorPartitionSafety`
- `SoftwareHypervisorPartitionSafety`: Represent a Safety Partition already certified and associated to the hypervisor by construction. It is not a Partition generated by VirtualizationLayer. It is a `SCItem` and may contain a `SafetyManual`.
- `SoftwareHypervisorPartitionSafety`: Represent a Safety Partition already
- `SystemSoftwareSafetyFolder`: Collection of `SystemSoftwareSafetyRoot` roots.
- `SoftwarePartitionSafetyFolder`: collection of `SoftwarePartitionSafety`
- `SoftwarePartitionSafety`: `SCItem` that represents a partition generated by Virtualization layer. It is a `SCItem` and may contain a `SafetyManual`.

8.1.3 Safety Partitioning Restrictions Meta-Model

This meta-model is used to model the constraints to be met the deployment of the system in order to help in ensuring the correctness of the system from the safety point of view (see Figure 8.4).

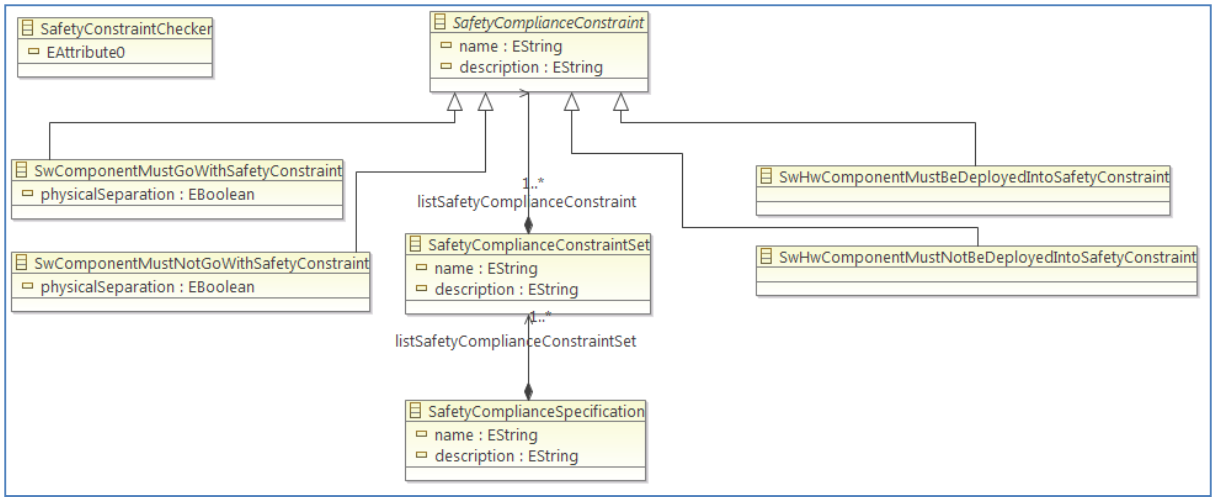


Figure 8.4: Classes of Safety Compliance Constraint meta-model

The following main classes are shown in the hierarchy:

- `SafetyComplianceChecker`: contains the *`evaluateSafetyCompliance`* function that allows checking safety constraint for a given deployment and safety specification.
- `SafetyComplianceSpecification`: contains a collection of `SafetyConstraintSet`.
- `SafetyConstraintSet`: contains a collection of `SafetyConstraint`.
- `SafetyConstraint`: represents a safety compliance constraint and may be of type:
  - `SwComponentMustGoWithSafetyConstraint`: contains the following parameters
    - `model.Component`
    - `model.Component`
    - `boolean physicalSeparation;`
  - `SwComponentMustNotGoWithSafetyConstraint`: contains the following parameters
    - `model.Component`
    - `model.Component`
    - `boolean physicalSeparation;`
  - `SwHwComponentMustBeDeployedIntoSafetyConstraint`: contains the following parameters
    - `model.Component`
    - `model.ExecutionUnit`
  - `SwHwComponentMustNotBeDeployedIntoSafetyConstraint`: contains the following parameters
    - `model.Component`
    - `model.ExecutionUnit`

8.1.4 Interface to other Meta-Models

Figure 8.5 provides an overview of the interfaces of the safety meta-models described in this section to other DREAMS meta-models. Arrows indicate references from one meta-model to another one.

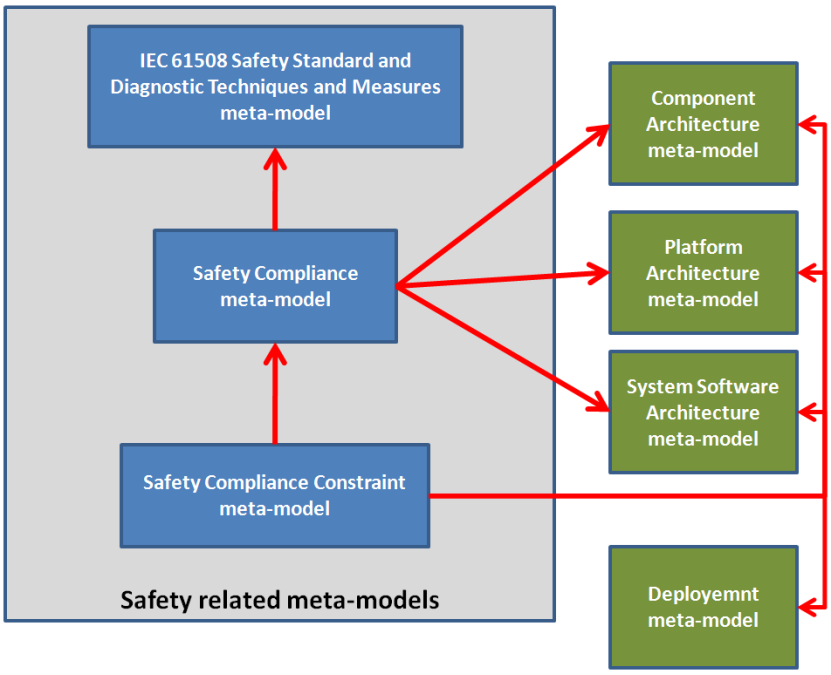


Figure 8.5: Safety Related Meta-Models

The safety meta-models contain references to a `ComponentArchitecture` model, a hardware `PlatformArchitecture` model, a system software `PlatformArchitecture` model and a `Deployment` model. As mentioned before, safety models make "external references" to entities of these models.

- `ComponentArchitecture` model entities:
  - `Component` and sub-`Component`s defined
- Hardware `PlatformArchitecture` model entities:
  - `Cluster`
  - `Node`
  - `Core`
  - `Tile`
  - `RAM/ROM Memory`
  - `GPIOS`s of `Tiles`
  - `Busses` connecting the internals of `Tiles` (and the hardware platform elements connected by the `Bus`)
  - `Clock` (and the hardware platform elements connected to it)
  - `Watchdog` (and the hardware elements connected to the `WatchDog`, and the software element acting on it)
  - `PowerSupply`
- System Software `PlatformArchitecture` Model entities (Virtualization Layer):
  - `Hypervisor`
  - `Partition`
- `Deployment` Model elements, with relation between them
  - `Components` (and `ExecutionUnit` assigned)
  - `Hypervisor` (and `Tiles` assigned)
  - `Partitions` (and `Hypervisor` and `Core` assigned)

The Safety Compliance Constraint meta-model also references external entities. These constraints, as explained in previous sections, are used to check if deployment model is valid from the safety point of view.

These constraints and the referenced entities are the following (these classes are also provided in deliverable D4.1.2 due to the strong connection of these classes with section 4.2.2 Safety Constraint Generation for Partitioning as they are key classes for *SafetyConstraintChecker*):

- Constraint `SwComponentMustGoWithConstraint`
  - Parameters
    - `model.Component`
    - `model.Component`
    - *Boolean* `physicalSeparation`
  - Semantic:
    - Both components must be deployed together in the same partition
    - `physicalSeparation`:
      - If it is `true`, the constraint can be used to describe that only one of the two application `Component`s should be affected by a single physical fault.
      - If it is `false`, the `Partition`s may be hosted by the same `Hypervisor` that protects them from application `Component` software design fault.
- Constraint `SwComponentMustNotGoWithConstraint`
  - Parameters
    - `model.Component`
    - `model.Component`
    - Boolean `physicalSeparation`
  - Semantic:
    - Both `Component`s cannot be deployed together in the same `Partition`
    - `physicalSeparation`: Whether the two `Partition`s must run on sufficiently separated hardware `ExecutionUnit`s in order to withstand physical faults
- Constraint `SwHwComponentMustBeDeployedIntoConstraint`
  - Parameters
    - `model.Component`
    - `model.ExecutionUnit`
    - Semantic: `Component` must be deployed into a given `ExecutionUnit`
- Constraint `SwHwComponentMustNotBeDeployedIntoConstraint`
  - Parameters
    - `model.Component`
    - `model.ExecutionUnit`
  - Semantic: `Component` must not be deployed into a given `ExecutionUnit`

8.2 Security Viewpoint

The security meta-model allows the modelling of the security services in DREAMS. It is based on the AutoFOCUS3 framework described in Section 3.2. Its implementation is contained in the AutoFOCUS3 DREAMS Eclipse RCP installation (see Section 3.2.2).

The security services include confidentiality, integrity and authenticity. DREAMS will provide these security services for the different core services of communication, resource management and execution. The security meta-model is not a separate model, it extends the DREAMS models of the core services, e.g., DREAMS cross-domain application meta-model and DREAMS platform meta-model.

The meta-models of DREAMS have different architectural views and levels of abstraction. Both cover different security related requirements. They are described in the following section.

8.2.1 Security Meta-Model

The DREAMS system model is divided into a logical view and a physical view (D1.2.1, Architectural Style). In the DREAMS meta-model presented in this document, this is reflected by a logical viewpoint and a technical viewpoint which are then mapped into the deployment viewpoints. Each of the viewpoints is implemented in terms of the corresponding meta-models. Each of these meta-models can be augmented with additional information using annotations that can be defined based on a generic annotation meta-model (see Section 3.2.5.3).

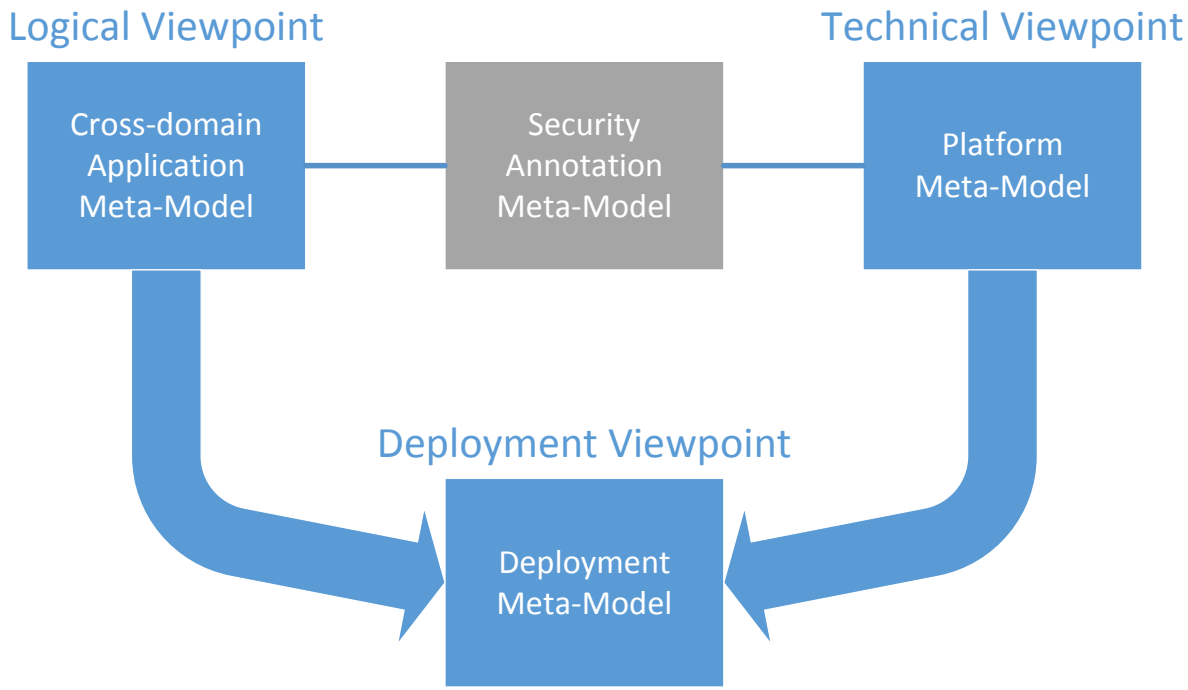


Figure 8.6: Security Annotation Meta-Model

Hence, the logical viewpoint (see Section 4) contains the cross-domain application meta-model which captures the application's architecture and optionally also its behaviour. Using dedicated annotations (realized based on the generic annotation meta-model), the necessity of a security service for a specific component will be expressed, e.g., does a component communication between two applications require confidentiality, integrity and/or authenticity?

The technical viewpoint (see Section 5) contains the platform meta-model. The physical components of the system are modelled. The security meta-model uses annotations to describe the security

services used by physical components communicating with other physical components. Using dedicated annotations, it describes the services provided by the platform. For security, the model expresses the specific algorithms that can be used to implement the required security mechanisms. Only the algorithms that are provided by the particular platform can be used.

In the deployment viewpoint (see Section 6), the application view and the technical view are mapped into the deployment meta-model. This models the concrete security algorithms used on the specific platform to fulfil the security requirements expressed using the corresponding annotations of the application model.

In the following it is described, how the security properties could be handled in the development process, and which models are involved in which step. The example considers security requirements for a resource management configuration message and a communication link between a gateway and a switch.

- Annotations for the application meta-model
  - The annotations for the application meta-model allow selecting which logical connection between two components or applications needs which security services. Confidentiality, Integrity, Authenticity
  - *Example: A configuration message from the GRM to a LRM needs integrity and authenticity, but no confidentiality. Hence, the message will be secured by a message authentication code to provide authenticity and integrity.*
- Annotations for the platform meta-model
  - The platform meta-model models the physical viewpoint. Using the annotations described above, the needs for security services on a physical connection can be modelled. The annotations for the platform meta-model allows to select which security algorithms offered by the platform will be used for a connection between two components (on-chip/off-chip gateways, switches, etc.).
  - *Example: Components of the platform, e.g., `OnChipNetworkDriver` or `OffChipClusterGateway` offers a different set of algorithms. The `OnChipNetworkDriver` could provide SHA-256, AES-CMAC-128 and AES-CMAC-256. AES-CMAC-256 for integrity and authenticity. The `OffChipClusterGateway` uses MACsec for the off-chip communication.*

8.2.2 Extension of the Annotation Meta-Model

The security meta-model extends the annotation meta-model. It allows selecting the security services in the annotation view.

For the logical viewpoint, the following annotations are defined in the security annotation meta-model:

- `LogicalAuthenticity`
- `LogicalConfidentiality`
- `LogicalIntegrity`
- `LogicalMACsec`

In the platform viewpoint, concrete algorithms are selectable. Hence there is a list with the available algorithms:

- `TechnicalAuthenticity`
- `TechnicalConfidentiality`
- `TechnicalIntegrity`
- `TechnicalMACsec`

The UML class diagram of the security meta-model is shown in Figure 8.7.

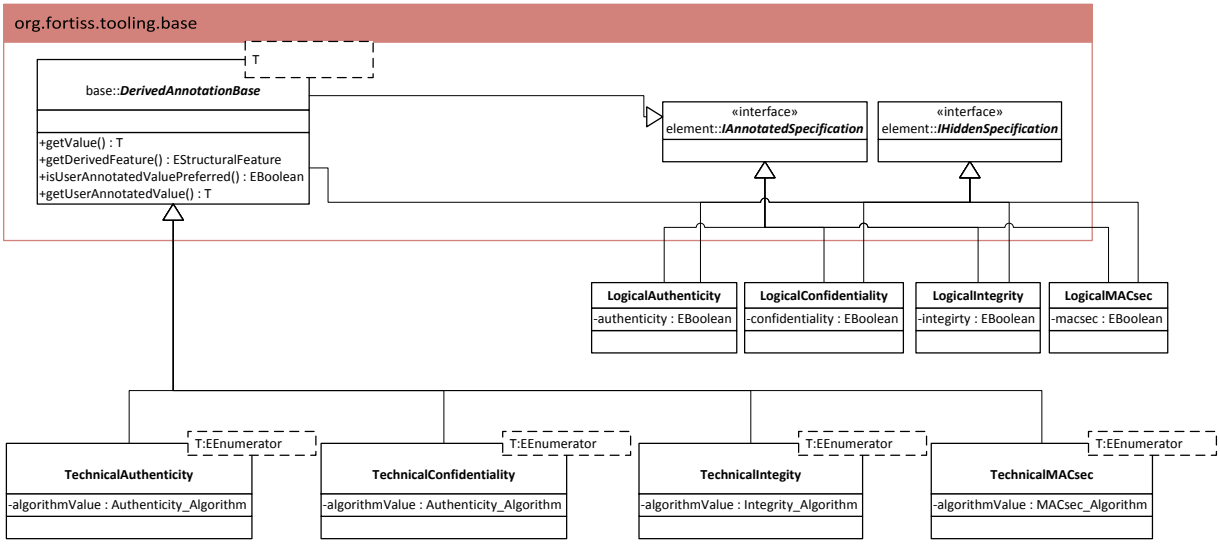


Figure 8.7: UML Class Diagram of Security Annotation Meta-Model

The security viewpoint is used to express the secure communication aspects and uses ports to model the communication between components or applications. Hence, the security services can be selected for `Port` classes:

- `Port`
  - `InputPort`
  - `OutputPort`

In the logical viewpoint, the security services can be selected for ports connecting two logical components.

In the technical viewpoint, the security algorithms can be selected for components that provide security algorithms. The security algorithms for the logical components can be selected using the annotations of the `OnChipNetworkDrivers`. The algorithm used for MACsec can be selected in the `OffChipNetworkGateway`.

This list of available algorithms can be adjusted in the security meta-model. The annotations show only the available algorithms.

8.2.3 Interface to other Meta-Models

The security meta-model extends the annotation meta-model for the logical architecture meta-model and for the platform meta-model. It references the following entities in other meta-models:

- Logical Component Meta-Model
  - `Port`
  - `InputPort`
  - `OutputPort`
- Platform Meta-Model
  - `OnChipNetworkDriver`
  - `OffChipNetworkGateway`

8.2.4 Security Model Example Instances

8.2.4.1 Extension of the Application Meta-Model

In the application meta-model, the security annotation meta-model allows to select the security services for the logical ports of a logical component. For each port the need for confidentiality, integrity and authenticity can be selected.

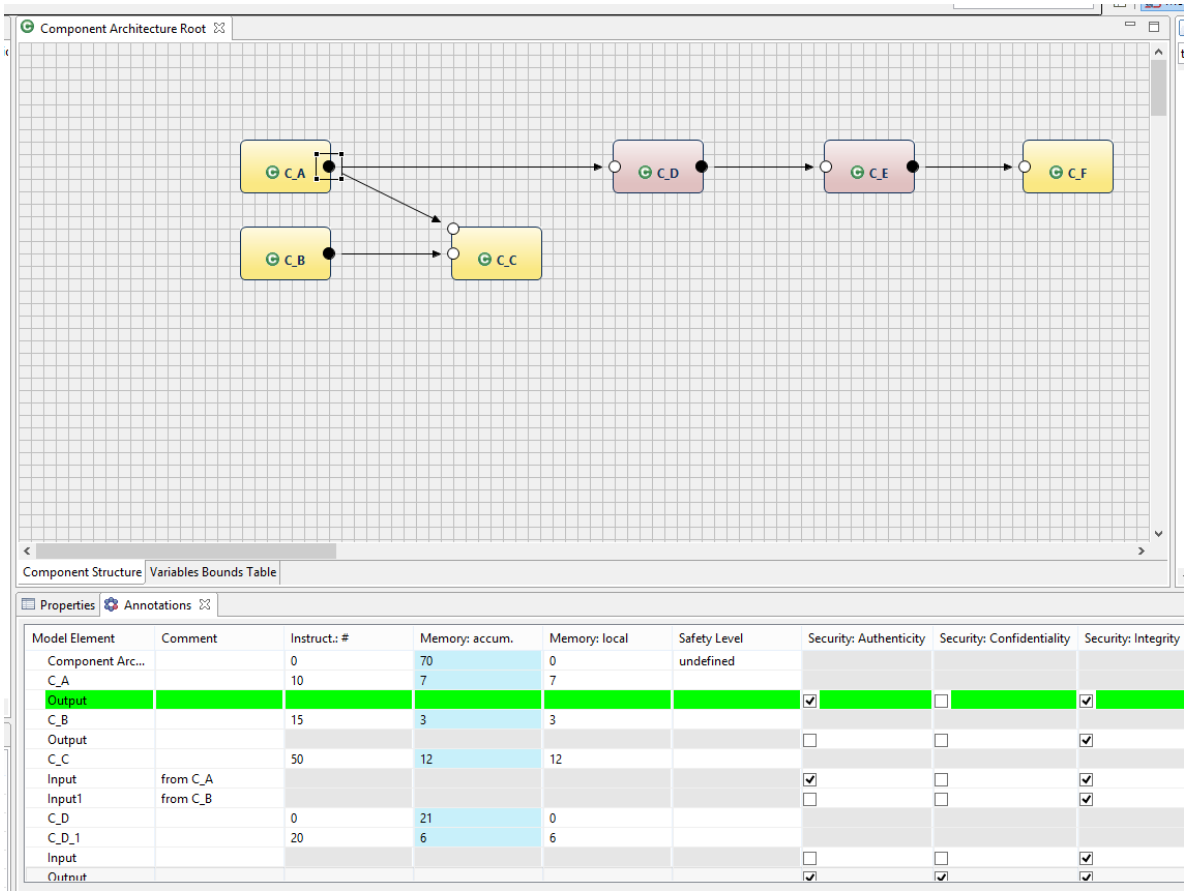


Figure 8.8: Security Annotations in the logical viewpoint

Figure 8.8 shows an example model. The model includes different components with output and input ports. For every connection from an output port to an input port the needs of authenticity, confidentiality and integrity can be selected.

In this example, the connection from the *C_A* to *C_C* and *C_D* needs authenticity and integrity, but no confidentiality. To be consistent with the output port of *C_A*, the input port of *C_C* and the input port of *C_D* (the input port of *C_D* is not shown in the figure) needs also authenticity and integrity, but no confidentiality. The output port of *C_B* and the respective input port of *C_C* need only integrity. The same applies to the input port of *C_D_1*. The output port of *C_D_1* needs authenticity, confidentiality and integrity.

8.2.4.2 Extension of the Platform Meta Meta-Model

In the platform meta-model, the security annotation meta-model allows to select the security algorithms used on connection. The annotations for the `OffChipNetworkGateway` and the `OnChipNetworkDriver` provide selection menu (see Figure 8.9). Here, the used MACsec algorithm can be chosen. In this example, only the *GCM-AES-128* algorithm is selected. As described in section 8.2.2, the list of the available algorithms can be defined in the security meta-model.

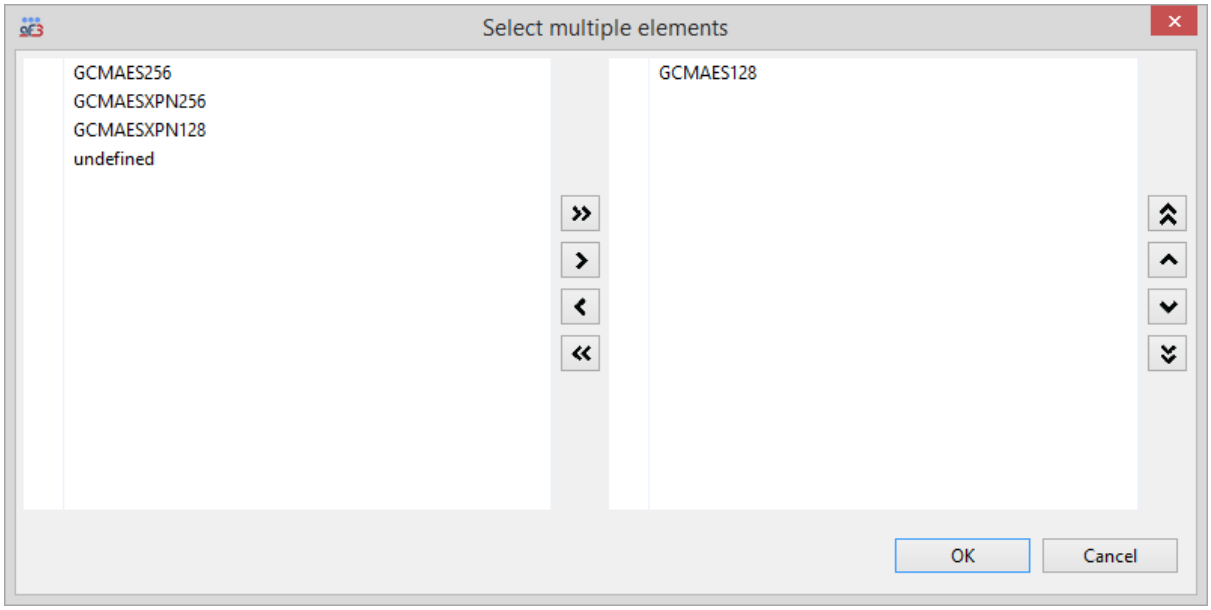


Figure 8.9: Algorithm selection menu

Figure 8.10 shows the security annotations for an `OffChipNetworkGateway`. The selection of the available algorithms and the used algorithm corresponds to the choice of Figure 8.9. One algorithm is selected (*GCM-AES-128*) and consequently this algorithm is also used as the selected algorithm.

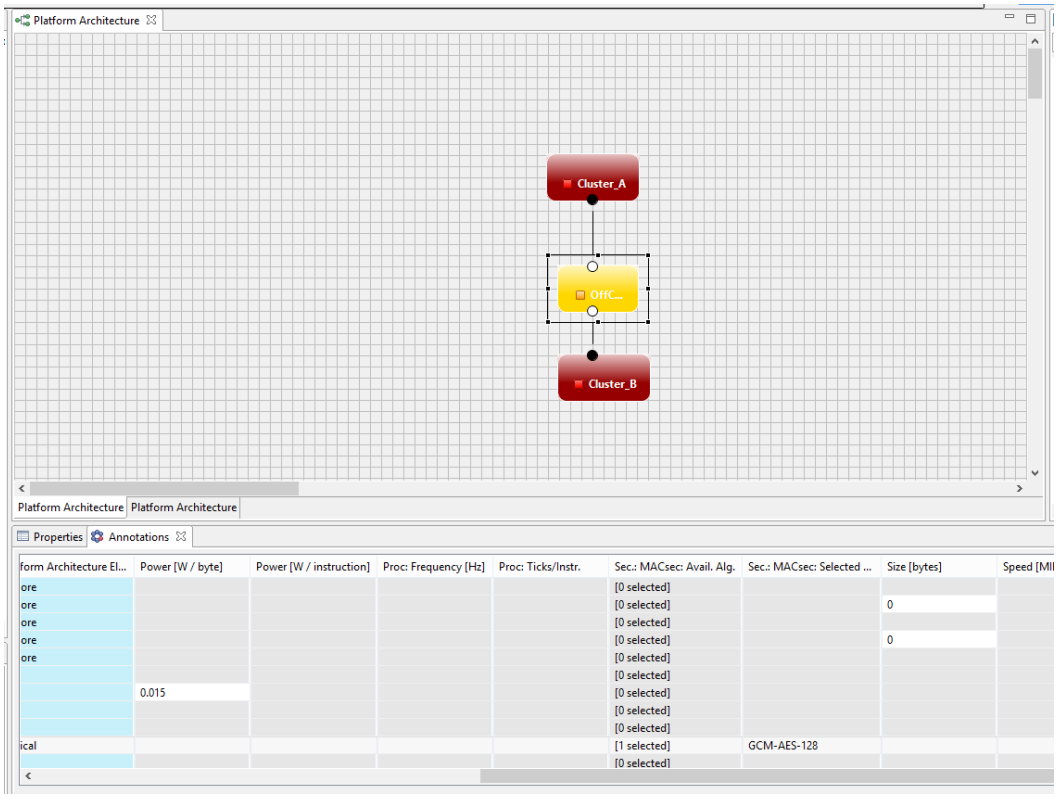


Figure 8.10: Security annotations in the technical viewpoint (`OffChipNetworkGateway`)

The annotations for the OnChipNetworkDriver are shown in Figure 8.10. The available algorithms for authenticity, confidentiality and integrity can be selected. For the selection the same selection menu as shown in Figure 8.9 is used.

In this example, the OnChipNetworkDriver provides *HMAC-SHA-256* for authenticity, *AES-128* for confidentiality and *SHA-256* for integrity.

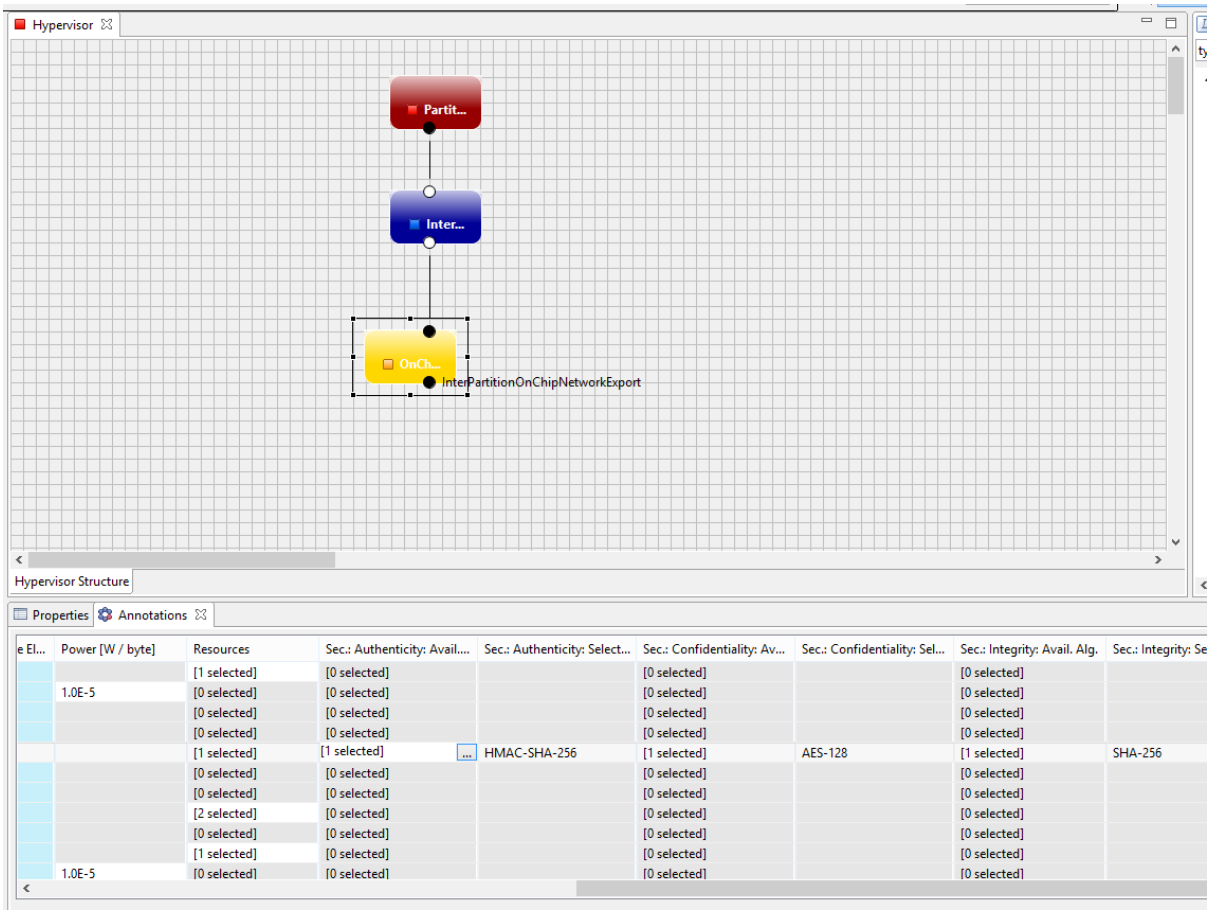


Figure 8.11: Security annotations in the technical viewpoint (`OnChipNetworkDriver`)

8.3 Power Viewpoint

Advanced systems are embedding today one or several interconnects IPs that links all SoC IPs together through a complex, flexible and scalable network. Power architecture exploration and power estimation of application or dimensioning use cases at system level are the most efficient tracks for the power optimization compare to the optimization at implementation level (RTL to layout).

Power architecture analysis at system level must provide power models of all the IP of the system and in particular power models of the interconnect IPs (ICN).

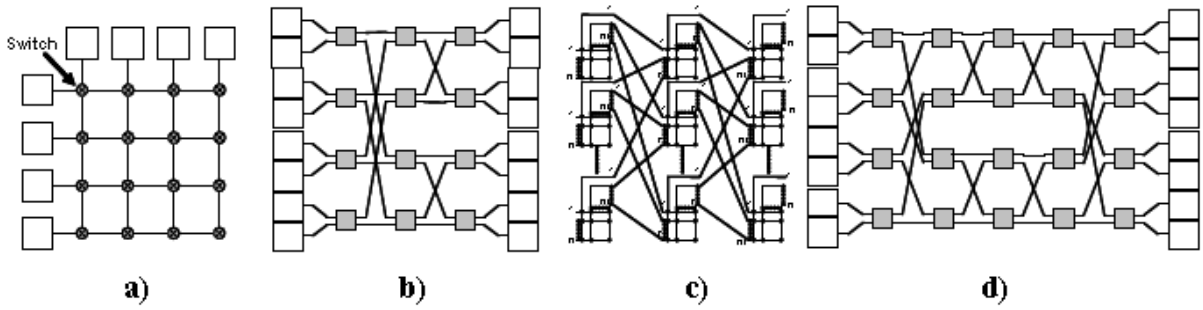
This section presents a solution of ICN power modelling to perform power analysis at system level. This solution of modelling has been developed to be used with the tool Aceplorer[8] provided by the EDA Company Docea Power.

The section starts with the description of the problems and requirements of ICN power modelling, and the presentation of the retained solution, mixing IP power card principle and ICN traffic.

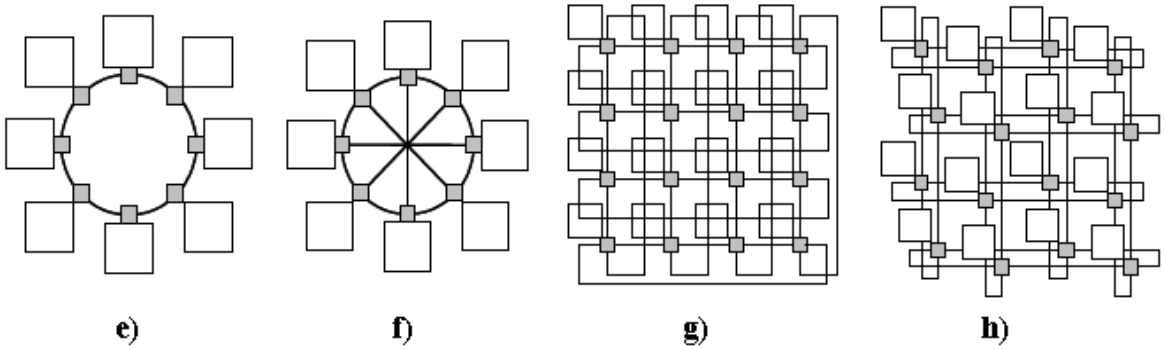
Then, it is explained how it is used in a system, and finally, the concept is validated on a multi-processors project using the new ARM 64 bit architecture[9].

8.3.1 Interconnect Modelling

Different Network on Chips (NoCs) exist and their usage depends on application. In the following, some topology examples are illustrated.



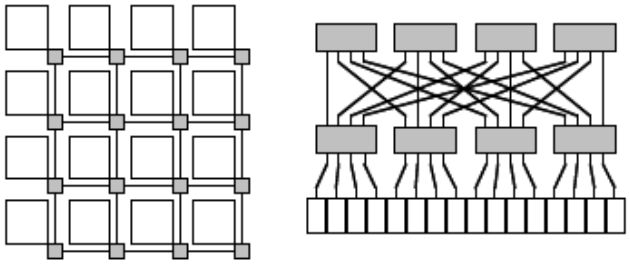
Crossbar Butterfly Clos Benes



Ring Chord Ring Torus Folded Torus

[8] http://www.doceapower.com/

[9] http://www.arm.com/products/processors/cortex-a/cortex-a53-processor.php



i) 2D Mesh

j) Fat Tree

Figure 8.12: NoC Topologies Examples

It would be more accurate and more powerful to model interconnect with its adequate internal topology, but practically it is not easy and may be not feasible to know power figures of each switch (node).

For example in ARM 64 bits project, power figures are given for a full CCN-504 interconnect[10].

It has been decided then to characterize the interconnect IP at its interface, without considering its internal topology. The approach presented in this section defines a power modelling approach for ICN such as the on-chip communication resources defined in the DREAMS Architectural Style (see D1.2.1). In the platform meta-model, an on-chip network is represented by class `OnChipNetwork` (see Section 5.2.3) whose internal structure can be described by the classes provided by the NoC-domain meta-model (see Section 5.2.4).

8.3.2 Variables and parameters of interconnect power calculation principle

For a specified functional mode, the power consumption of an ICN IP is function of the read and write traffics at its interface.

In the scheme below, the traffic to consider to compute the power consumption of the interconnect is:

- For the read traffic:
  - *ReadInputSum = Rd1 + Rd2 + Rd3*
- For the write traffic:
  - *WriteInputSum = Wr1 + Wr2 + Wr3*

[10] http://www.arm.com/products/system-ip/interconnect/corelink-ccn-504-cache-coherent-network.php

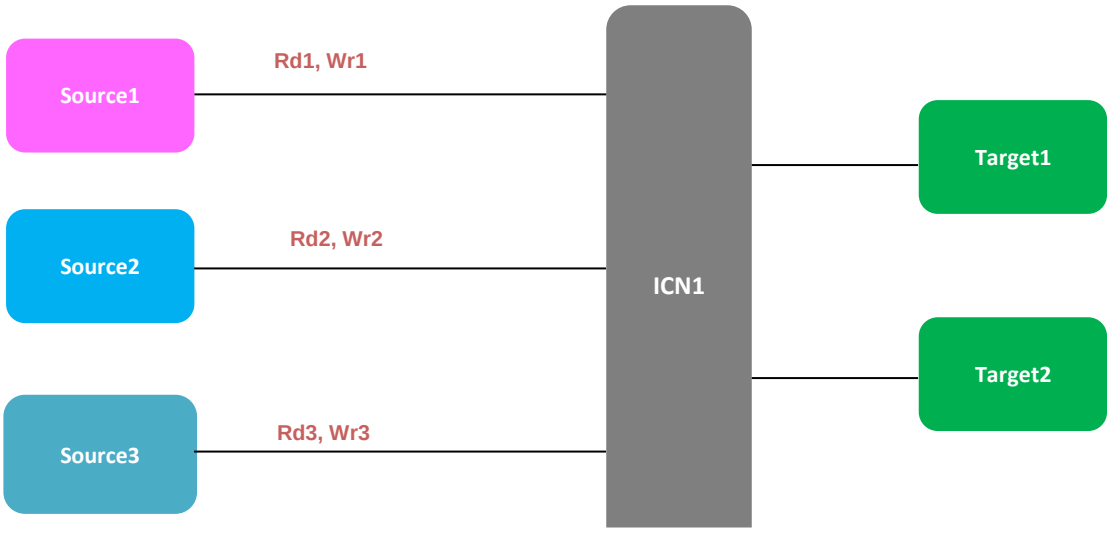


Figure 8.13: ICN power calculation principle

More generally *ReadInputSum* and *WriteInputSum* are respectively read and write traffic activities sums at the ICN interface. They are used as variables in the power consumption calculation.

Then other variables used in power consumption calculation are classical ones that is frequency and supplies voltage. This is always considered in a determined context of temperature, corner, process, and power state (functional mode).

8.3.3 Power Equation

For each Active Power State, Dynamic Power Equation is described as followed:

$$Pdyn = Pref * \frac{Clkin}{Clkref} * \left(\frac{Vin}{Vref}\right)^2 * \frac{ReadInputSum}{ReadInputRef} * \frac{WriteInputSum}{WriteInputRef}$$

with

- *Pref*: Power reference value: i.e. power value at *Clkref*, *Vref*, *ReadInputRef* and *WriteInputRef* → at simulation point
- *Clkin*: input clock
- *Vin*: input supply
- *ReadInputSum*: sum of all READ traffics of all traffic_container inputs.
- *ReadInputRef:* reference full READ traffic value in this Power State.
- *WriteInputSum*: sum of all WRITE traffics of all traffic_container inputs.
- *WriteInputRef*: Reference full WRITE traffic value in this Power State

8.3.4 IP Power Cards management

The reference values of the variables and parameters of the previous equation are obtained with an IP power characterization.

A power characterization is done by doing a set of power estimation with relevant test benches at the top interface of the interconnect IP. It can be done on the RTL design with tool like SpyglassPE (Atrenta) or Power Artist (Apache), or at gate level with PrimetimePX (Synopsys).

During an IP power characterization, the variables and parameters issued from the power estimations are classed in IP tables that are called power cards.

As said before, the information, classed per power state (or functional mode), is:

- Voltage
- Frequency
- Activity: in the case of NoC, the activity is the read and write traffics in MIPS/MHz or DMIPS/MHz

Other information are environment parameters: process, temperature, and corner.

8.3.5 Requirements of the interconnect power model

The particularity of an ICN power model is not only to compute the power consumption as explained before, but also **to describe the traffic transfer**. The model must be adaptable to any configuration without being modified. Below are some usage examples:

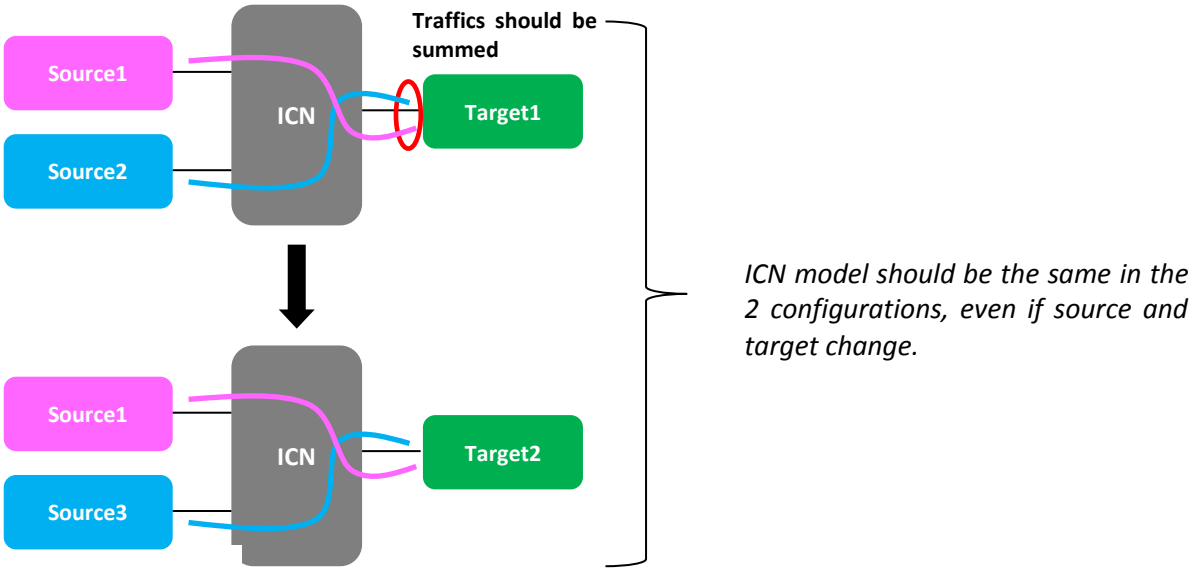


Figure 8.14: Case of a power architecture exploration

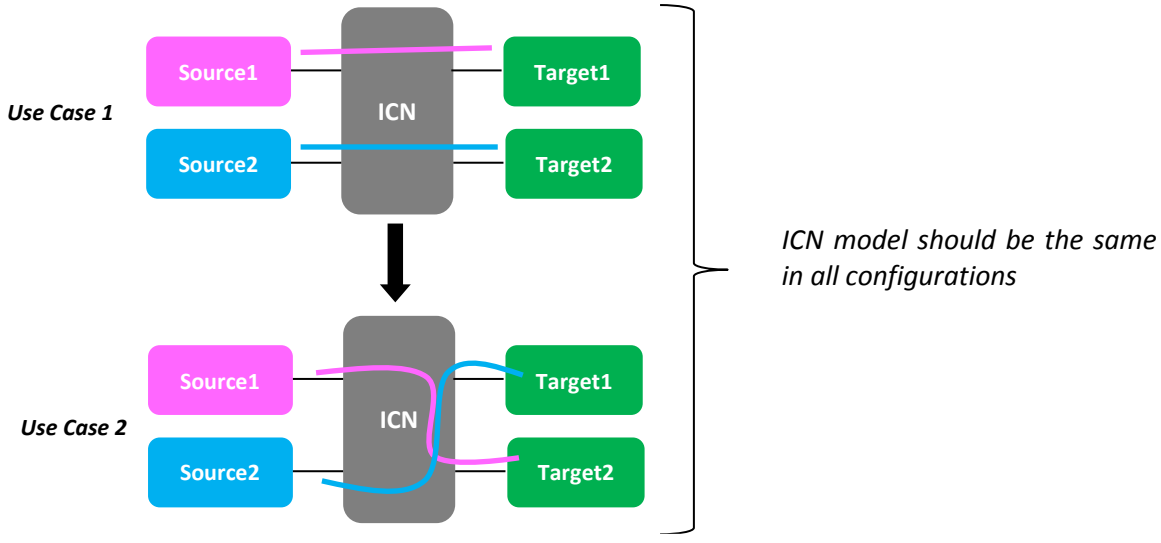


Figure 8.15: Case of a different traffics transfers from Sources to Targets

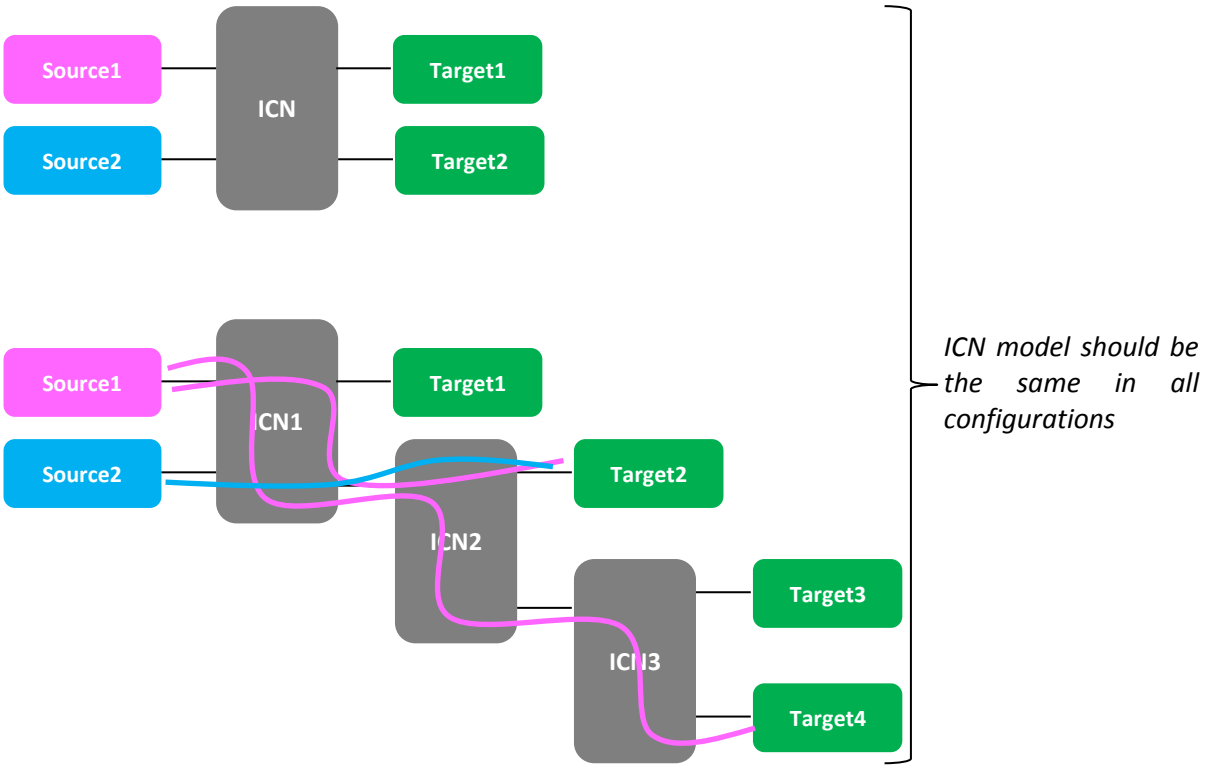


Figure 8.16: Case of cascade of interconnects

The requirements of an ICN model are:

- **Interconnect model should be the same in all configurations**.
- 2 traffics reaching the same Target IP they should be **summed**.
- Each source can have several targets.
- The traffic transfer is defined in the use case.

8.3.6 Interconnect power model

Aceplorer is the tool in which are developed the ICN power model as well as all the models of the complete system to be analysed. The power analysis of the system is also done with Aceplorer.

8.3.6.1 Traffic container

Into Aceplorer, a STMicroelectronics customization has been done to define a dedicated type of data that allows describing the traffic transfer in the ICN: **traffic_container**.

This new type permits **to define in a scenario** (scenario = test bench applied at the interface of the ICN IP) the traffic from Source to Targets at the ICN I/Os.

Traffic_container is a ***container,*** or a list, of several traffics.

Format: [[Source, Target1, Rd1, Wr1], …, [Source, Targeti, Rdi, Wri], …] where:

- **Source:** IP generating traffic (Core, High Speed Interface…)
- **Target:** IP receiving this traffic (RAM, DDR, Low Speed Interface…)
- **Rd, Wr:** Read & Write traffic in bit/sec, or Mbit/sec, or Mbytes/sec…

On top of defining traffic value on traffic pin, “traffic_container” type permits to define several traffics independently and where these traffics should go.

8.3.6.2 Tasks of the Interconnect Power Model

The ICN power model has to achieve two tasks:

- To compute the power consumption in the ICN IP as explained in Section 8.3.3.
- To distribute the traffic between its I/Os

These tasks are done on the fly, depending of the traffic container defined in the scenario thanks to python functions used in Aceplorer.

Python functions have been developed to compute the power equation variables ReadInputSum and WriteInputSum:

- `TrafficReadPerTrafficInput()` function is used. This function sums all Read part ($\sum Rdi$) of all traffic_container inputs. *ReadInputSum* static is only used to calculate Power.
- `TrafficWritePerTrafficInput()` function is used. This function sums all Write part ($\sum Wri$) of all traffic_container inputs. *WriteInputSum* is only used to calculate Power.

For the traffic distribution, **inside the ICN Power Model, it is only done traffic multiplexing and sums thanks to specific python utilities** developed at STMicroelectronics:

- `ICNOutputGen()` function: define output traffic value for each traffic output pin.

8.3.6.3 Traffic distribution examples

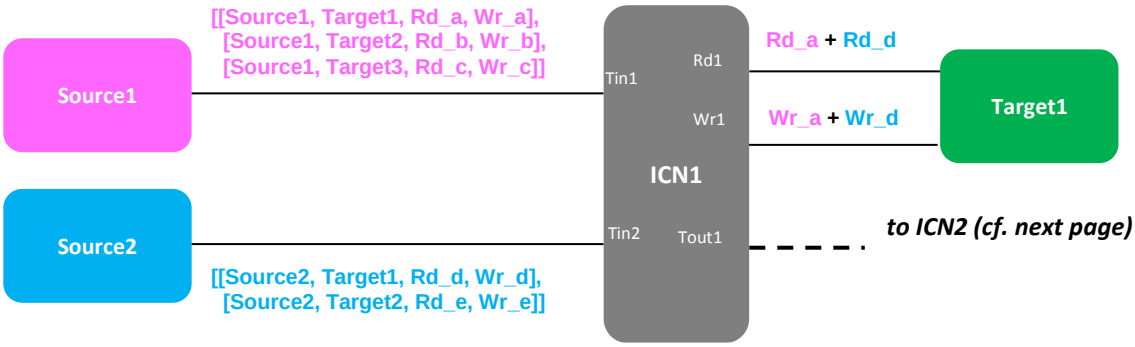


Figure 8.17: Traffic distribution example 1

Rd1 output description: *Rd1 = ICNOutputGen(Tin1, Tin2)*

- *Rd1* output pin type is “*traffic_read*”
- Name of component connected to *Rd1* is “*Target1*” (leaf cell)
- *Tin1* and *Tin2* values are parsed. All READ values having “*Target1*” as Target IP are summed → in this case *Rd1 = Rd_a + Rd_d*

Wr1 output description: *Wr1 = ICNOutputGen(Tin1, Tin2)*

- *Wr1* output pin type is “*traffic_write*”
- Name of component connected to *Wr1* is “*Target1*” (leaf cell)
- *Tin1* and *Tin2* values are parsed. All WRITE values having “*Target1*” as Target IP are summed → in this case *Wr1 = Wr_a + Wr_d*

Tout1 output description:

- If ICN output is “traffic_container” type, it means that output is connected to another interconnect

So `ICNOutputGen()` function get name of components connected to outputs of 2nd interconnect. This function is **recursive** and ICN number is not limited as in the example below:

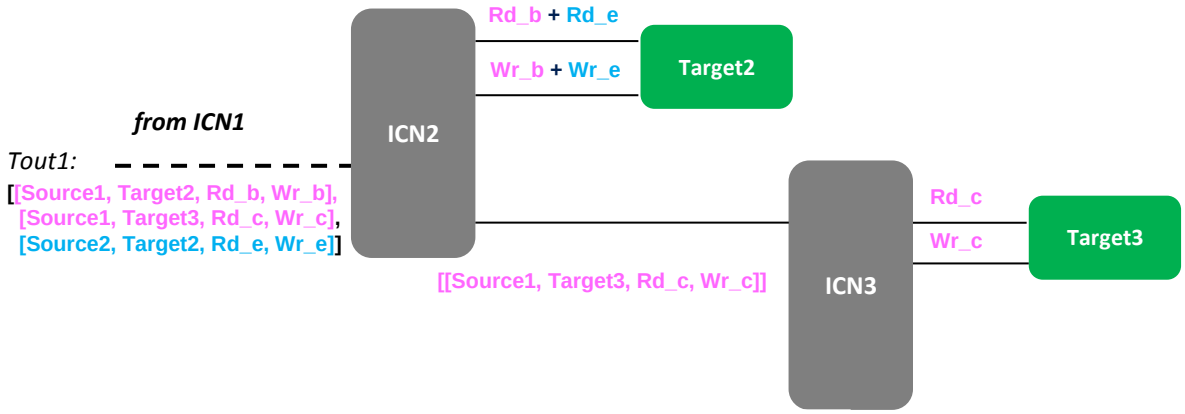


Figure 8.18: Traffic distribution example 2

Tout1 output description: *Tout1 = ICNOutputGen(Tin1, Tin2)*

- *Tout1* output pin type is "*traffic_container*"
- Names of leaf cells potentially receiving traffic from *Tout1* are "*Target2*" and "*Target3*"
- *Tin1* and *Tin2* values are parsed. All TRAFFIC descriptions having "*Target2*" and "*Target3*" as Target IP are concatenated.
  In this case Tout1 = [[Source1, Target2, Rd_b, Wr_b],
  [Source1, Target3, Rd_c, Wr_c],
  [Source2, Target2, Rd_e, Wr_e]]

The same methods as pointed out above are also applied on ICN2 and ICN3.

8.3.7 Conclusion

In this section, a flexible and scalable power model allowing to perform system level architecture power analysis and exploration has been presented. An example that is based on real case, will be presented in the integration report D1.5.1. The work could be extended in the topic of characterization of the ICN.

9 Variability Viewpoint

9.1 BVR Meta-Model

9.1.1 Overview

This section describes selected parts of the meta-model underlying BVR. The BVR meta-model formalizes all the concepts (and their relationships) that can be defined in a BVR model, and we extracted here the parts relevant for the DREAMS project. We refer the reader to the BVR manual [8] for comprehensive description of the meta-model.

Figure 9.1 below shows the connection between the top level concepts of a BVR model. A *BVR model* defines a set of *variation points* that characterizes a given base model. The base model is referenced through an *object handle*, which contains a reference to an Ecore[11] object. In addition the base model also contains the related *resolutions*. By analogy with the classical product-lines terminology, a *VSpec* object stands for the feature tree that characterizes a given product line (the BVR model), and a *VSpecResolution* object stands for the set of concrete choices that characterizes a derived product.

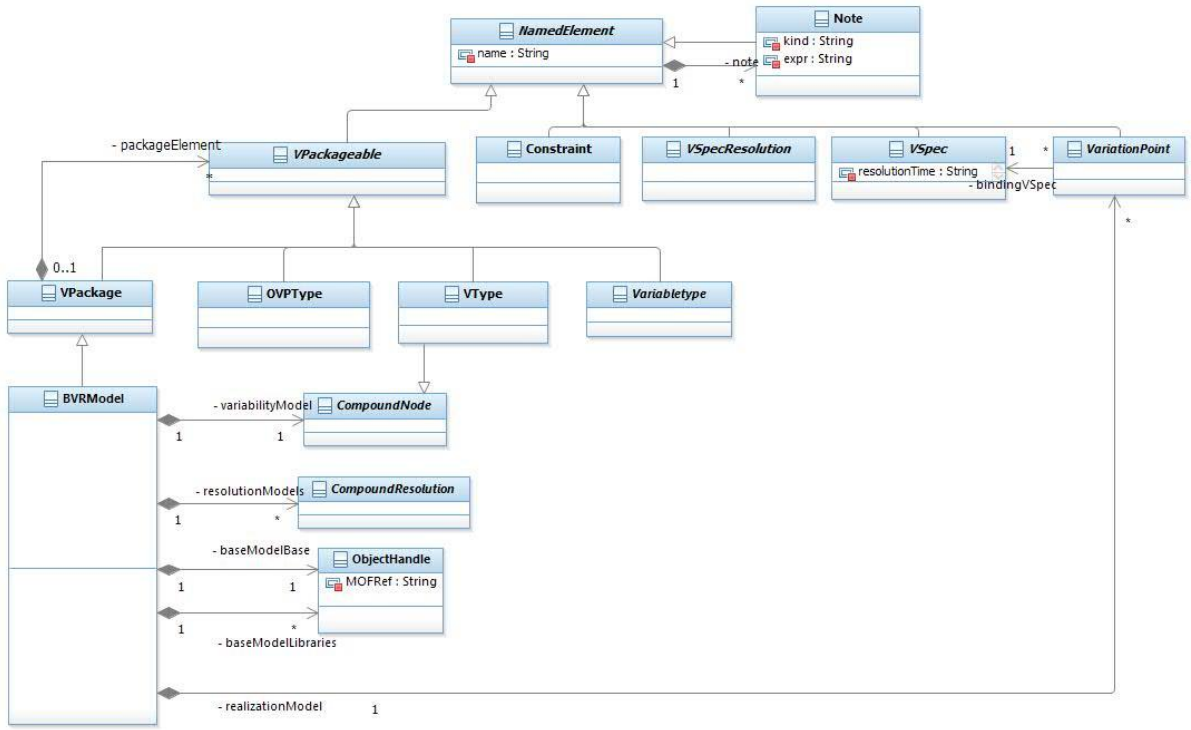


Figure 9.1: Main concepts in BVR

9.1.2 VSpec and VSpecResolution

VSpec and *VSpecResolution* are the key concepts in BVR. A *VSpec* organizes all the decision points that govern the derivation of a product in a tree like structure, as usually done in feature

[11] Ecore is the name of the implementation of the MOF standard provided by EMF

diagrams [11]. A *VSpecResolution* captures the result of a particular derivation, where each single decision point is given a particular value. As shown in Figure 9.2 below, BVR supports several types of decision points: especially *Choice*, *Variable* and *VClassifier*, which are primarily used.

- *Choices* represent Boolean decision points, and can be resolved by either true or false.
- *Variables* represent parameters that are resolved by a particular value, whose type matches the one of the variable.
- A *VClassifier* is a *VSpec* whose resolution requires instantiating it zero or more times and then resolving its sub-tree for each instance. When a repeatable variation point is bound to a *VClassifier*, it will be applied once for each instance of the *VClassifier* during materialization.

One can see on Figure 9.2 that the *VSpecResolution* structure follows the *VSpec* structure. For each type of *VSpec* (e.g., *Choice*) there exists an equivalent *VSpecResolution* (e.g., *ChoiceResolution*) in the meta-model.

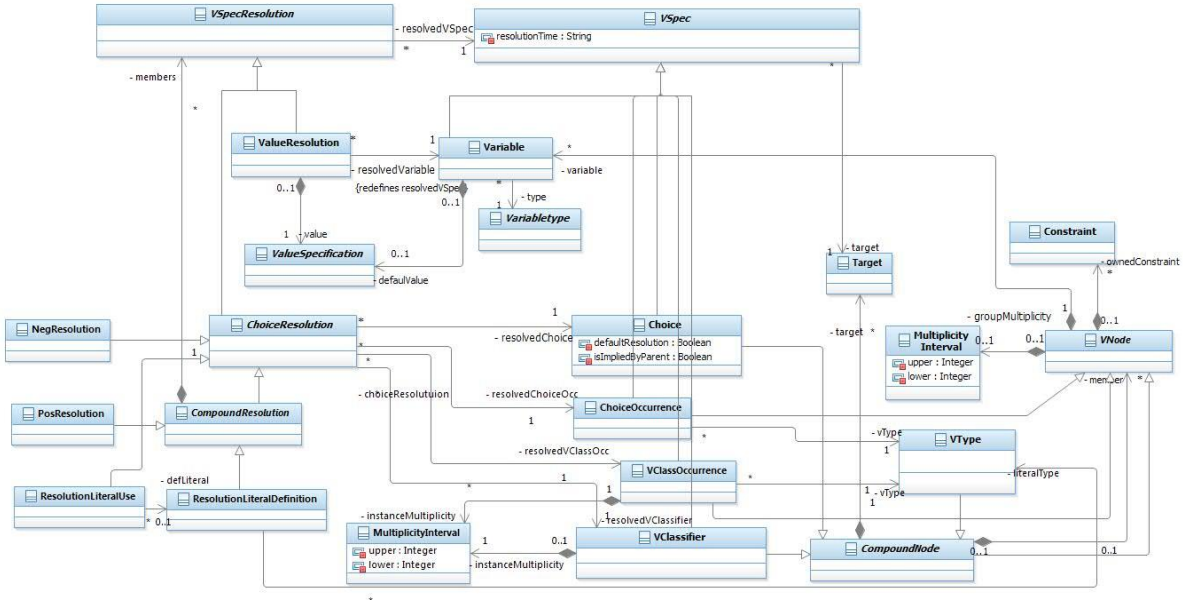


Figure 9.2: Decision points and decision in BVR: VSpec and VSpecResolution

9.1.3 Constraining VSpec Trees

Expressing variability as a tree of variation points is seldom sufficient to properly capture all the constraints within a given application domain. A feature tree results from one of the many possible decompositions of the problem, and therefore only captures a subset of these constraints. Additional constraints have to be encoded as logical formulae that restrict the set of legal choices that can be made within a single feature tree. Figure 9.3 below illustrates the abstract syntax of the logical constructs supported by BVR. In a nutshell, BVR supports logical conditions, numerical assertions (see *NumericalLiteralExp* and its subclasses), and basic comparisons of text literals (see *StringLiteralExp*) as well as assertion of empty values. Note that BVR only relies on propositional formulae and therefore does not support for logical quantifiers (i.e., exists and forall).

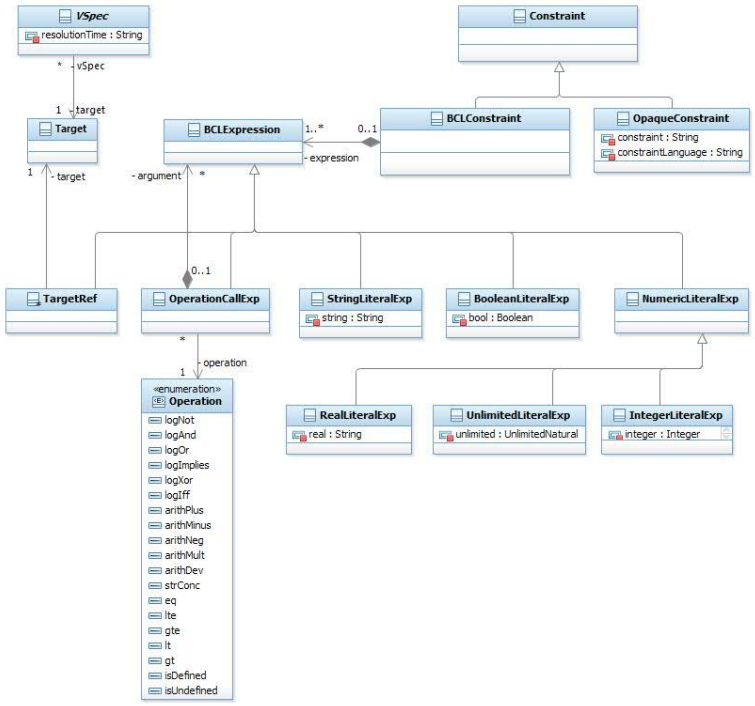


Figure 9.3: Constraining BVR variability models

9.1.4 Fragment Substitution

For BVR to be effective, the user shall detail how BVR can "inject" the selected solution for each variation points into the base model. This injection operation, so called "fragment substitution", is at the heart of the BVR tooling. Figure 9.4 portrays the concepts of placements and replacements that underpin fragments substitution. A *placement* denotes the specific subset in the base model which corresponds to a specific choice, as opposed to the associated *replacement* which specifies the fragment of model that shall be injected.

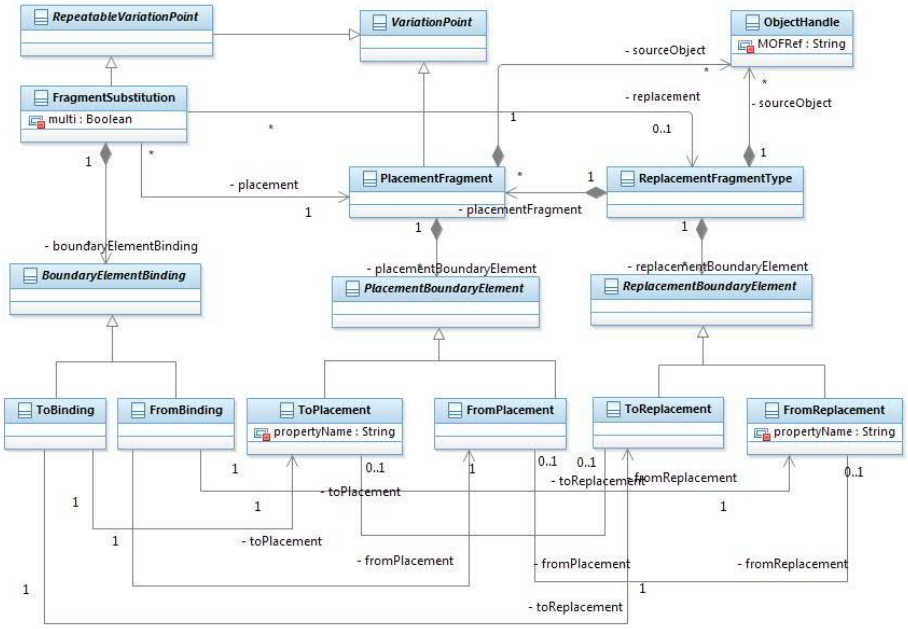


Figure 9.4: Specifying placements and replacements to perform fragments substitution

9.2 Variability Workflow

9.2.1 Modelling Variability

Intuitively, modelling variability is like creating a template document, where some well-identified parts are marked for later replacement with some *ad hoc* content. Templates are often initially derived from an existing document, where the parts to be specialized are stripped out or identified as samples sections. The process associated with the use of BVR follows a very similar scheme [12]:

1. ***Preparing the product line*** can be done by looking at existing models (i.e., products) and looking for similarities and differences. The parts that vary often map to the variation points, as opposed to the parts that remain unchanged, which form the backbone of the future product line.
2. ***Choosing a base model*** consists in promoting one single existing product to be the matrix of subsequent products. BVR will use the model of this product, as input to a model-to-model transformation, which replaces each variation point with the associated model fragment and yields a valid new product, by substitution.
3. ***Identifying a library of reusable fragments*** significantly simplifies the use of the product lines, but enabling derivation of new products by feature selection. Variation points often have several possible solutions, which exist as model fragments, and which should be available for future injection even if they are not included into the selected base model.
4. ***Creating a BVR model*** formalizes the variation points, the base models, and the library of reusable fragments. It helps capitalize on the domain specific knowledge captured in the product line and to proceed with further product derivation.
5. ***Generating products*** is the final step, where one can generate new product by the sole prescription of the base model and the set of features to activate.

9.2.2 Exploiting Variability

Automated product derivation is the most emphasized feature of software product lines. By giving the user the possibility to select the features that one needs, it becomes possible to check the consistency of the whole product line, check the consistency of a given feature prescription, and to assemble the prescribed products.

Checking the consistency of a product line as a whole consists in ensuring that there exists at least one single product that meets all the constraints embedded in the associated feature model. Interestingly feature models can be reduced to propositional logic formulae [13], and their validation thus boils down to the satisfiability problem (SAT). Although SAT is well-known to be a NP-Complete problem, recent advances in SPL [14] showed that industrial size SPLs form a very specific subset of SAT instance, which existing SAT solver can address.

Checking the validity of a specific feature prescription ensures that the prescribed features meet the constraints carried by the feature model. The prescription is valid if and only if the underlying variable assignment satisfies the associated logical formulae. SPL thus permits to detect automatically invalid configurations that will not work in practice.

Finally, assuming a given feature prescription is consistent with its enclosing SPL, it is possible to automated — possibly only partially — the construction and the validation of the associated products. This construction step is tightly coupled with the reuse capabilities of the underlying execution platform.

The BVR Tool Bundle currently supports product sampling based on generating so-called covering arrays [15]. The support is provided via SPLCATool intergraded into the BVR tool.

9.3 Variability Model Example Instances

9.3.1 Variation of DREAMS System Models

We illustrate here the usage of BVR to express the variability inherent in an illustrative example model shipped with AutoFOCUS3. It is a system model describing an automatic cruise control module, hereafter denoted by ACC. In a nutshell, the ACC module computes the acceleration of the car from the actual and desired speeds of the car, and the actual and desired distance to the next vehicle. Figure 9.5 shows the logical view of such system, modelled in AF3. The current speed and the current distance are provided by specific sensors (see inputs *SensedSpeed* and *SensedDist* on the left hand side). Both measures are smoothed to detect irrelevant measures (see the two *plauzibilization* sub components), before to be fed into the *Speed Controller* and the *Distance Controller* units, respectively. The accelerations outputted by these two controllers are aggregated by the *Acceleration controller*, which eventually emits the final acceleration command.

The distance control component can actually offer two alternative level of performance. The *Eco* mode aims at reducing the fatigue and the fuel consumption by avoiding strong acceleration/deceleration. By contrast, the *Sport* mode allows for a more aggressive driving. These two modes are also captured in the AF3 logical view of the distance controller component, shown in Figure 9.6. The distance controller can transition between these modes, as requested by the driver.

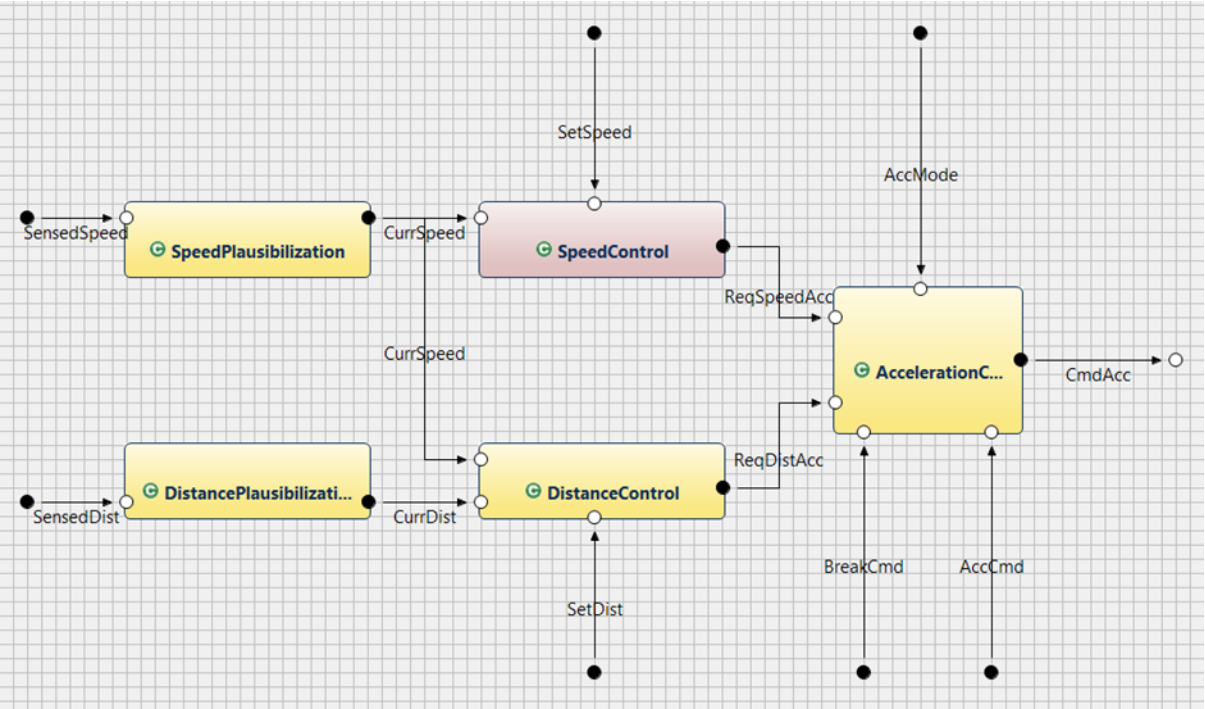


Figure 9.5: The logical view of the ACC system, described in AF3

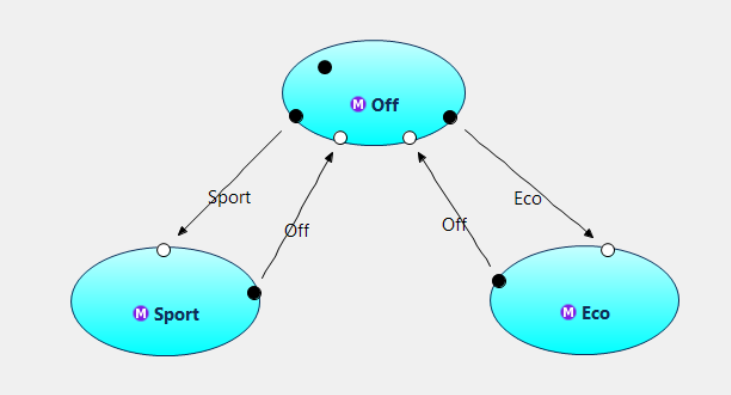


Figure 9.6: Internal behaviour of the "Distance Control" unit (modelled as a mode automaton)

In the following, we will consider this model as our *base model* (see Section 9.2.1 above) and identify variation points and the related variants. A first variation point in this ACC model is the distance control part. A simpler ACC model may need not include a distance controller, in which case the distance controller and the distance plauzibilization can be replaced by a single constant value. Per se, three distance controllers could be built, depending on the modes they offer. The more complex one permit three modes (*Sport*, *Eco* and *Off*), but any combinations of these modes lead to an alternative distance controller.

This variability can be captured in a BVR variability model, as shown on Figure 9.7. This model express the fact that four features are found in the ACC, namely the *Speed Controller*, the *Speed Plauzibilization*, the *Distance Controller* and the Distance *Plauzibilization*. While other features are mandatory, the *Distance Controller* is an optional feature. It is further decomposed into any combination of its two sub features, *Sport* and *Economic*. However, if one selects the *Distance Controller*, one has to also select the related distance plauzibilization feature. This is captured by the constraint on the left hand side of the BVR diagram.

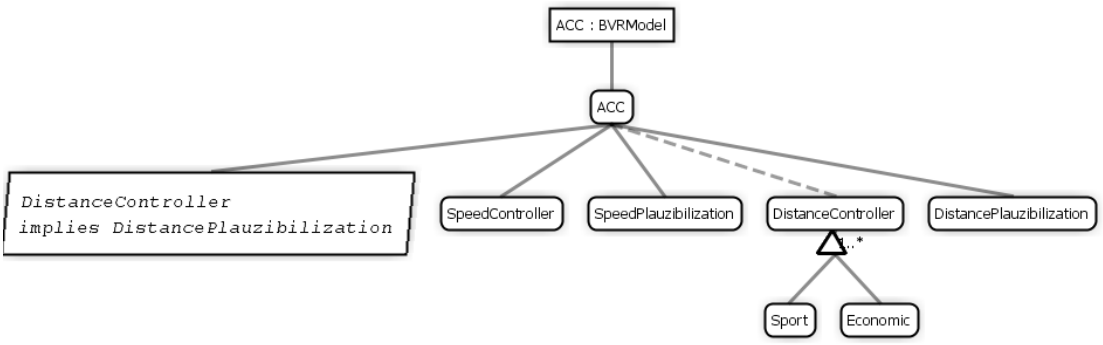


Figure 9.7: Modelling the variability inherent to the ACC system in BVR

By selecting the features that makes the ACC variation of interest BVR can generate the associated AF3 model, which can they be used as any other regular model. In this example, BVR will replace the model elements associated with each variation point with the model elements associated with each the selected variant. For instance, Figure 9.8 illustrates the diagram associated with the model where all distance related features are disabled. They are replaced by a constant distance fed into the *Acceleration Controller*.

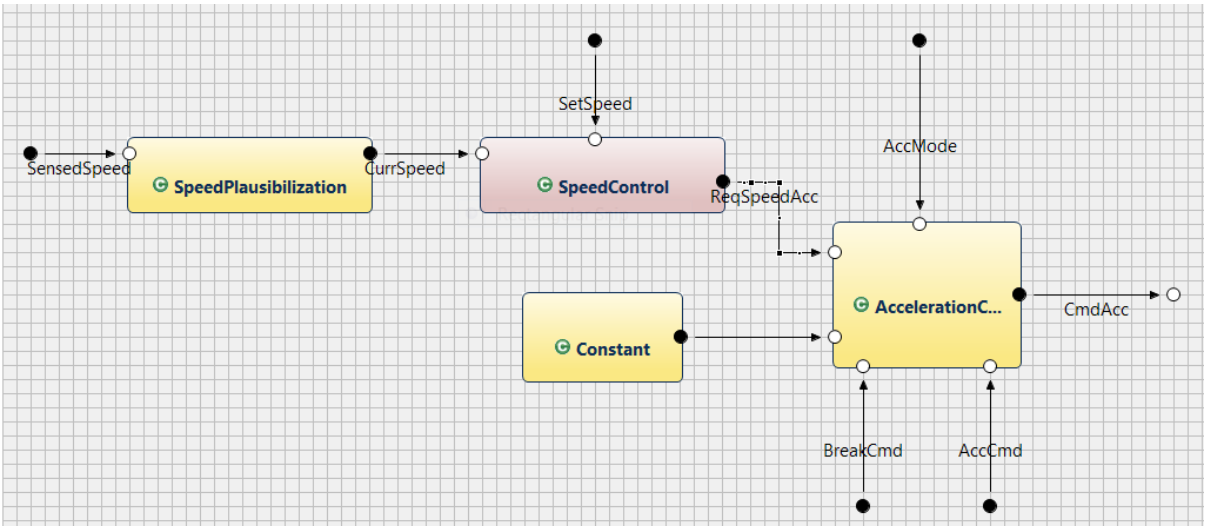


Figure 9.8: A simpler ACC model, where distance control is removed

9.3.2 Variation of Safety Consistency Variability Models

The variation of Safety Consistency Models is modelled using BVR tool from SINTEF. Following this approach, the variability of any model is modelled in a separate model. Then a mapping between the variability model and the original model is performed. After that, the system is able to create concrete instances of the models with variability using replacement mechanisms.

In the safety meta-models, the variability comes for two points (a) the deployment and (b) the *Safety Compliance* model itself. When modelling *Safety Compliance* models using the editor, a variability model will be defined with BVR tool, to express the variability.

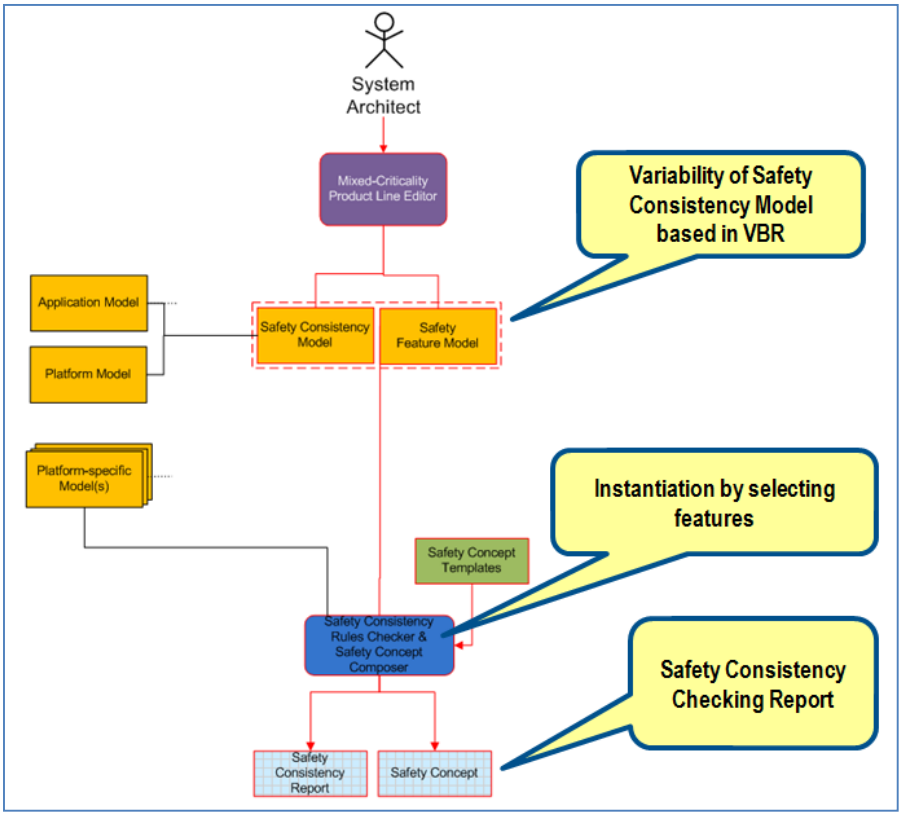


Figure 9.9: Variation of Safety Consistency Models (Example)

Figure 9.9 illustrates the process with an example: The variability of Safety Compliance model is exploited in the following way:

- A *Safety Compliance* model is defined with the corresponding editor.
- A *Safety Feature* model (variation points) is defined and modelled with BVR editor.
- Then, the *Safety Compliance* model is instantiated for a specific use case. This is done by selecting options (features) in the BVR model, and performing the replacement in the model.
- Once the specific instance of the *Safety Compliance* model is generated, the process follows by checking the safety consistency for a given deployment (application and platform models), creating the safety consistency report, etc.

10 Bibliography

[1] International Electrotechnical Commission, „ISO/IEC 42010, Systems and software engineering - Architecture description,“ 2011.

[2] International Electrotechnical Commission, „IEC 61508-1: Functional safety of electrical/electronic/programmable electronic safety-related systems part 1: General requirements,“ 2010.

[3] International Electrotechnical Commission, „IEC 61508-2: Functional safety of electrical/electronic/programmable electronic safety-related systems part 2: Requirements for electrical / electronic / programmable electronic safety-related systems,“ 2010.

[4] International Electrotechnical Commission, „IEC 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems part 3: Software requirements,“ 2010.

[5] D. Steinberg, F. Budinsky, M. Paternostro and E. Merks, EMF: Eclipse Modeling Framework, Amsterdam: Addison-Wesley Longman, 2008.

[6] M. Broy and K. Stølen, Specification and development of interactive systems: focus on streams, interfaces, and refinement, Secaucus, NJ, USA: Springer, 2001.

[7] Ø. Haugen and O. Øgård, "BVR – Better Variability Results," *Proceedings of the 8th International Conference on System Analysis and Modeling: Models and Reusability (SAM '14), LNCS,* vol. 8769, pp. 1-15, 29-30 Sept. 2014.

[8] Ø. Haugen, "BVR - The Language, VARIES Project D4.2," 2014.

[9] Object Management Group, "Meta Object Facility (MOF) Core Specification v2.4.1," 2014.

[10] TIMMO-2-USE Project, „Language syntax, semantics, metamodel V2,“ 2012.

[11] P.-Y. Schobbens, P. Heymans and J.-C. Trigaux, "Feature Diagrams: A Survey and a Formal Semantics," *Proceedings of the 14th IEEE International Conference on Requirements Engineering,* pp. 139-148, 11-15 Sept. 2006.

[12] A. Svendsen, X. Zhang, R. Lind-Tviberg, F. Fleurey, Ø. Haugen, Møller-Pedersen, Birger and G. K. Olsen, "Developing a Software Product Line for Train Control: A Case Study of CVL," *Proceedings of the 14th International Conference on Software Product Lines (SPLC 2010), LNCS,* vol. 6287, pp. 106-120, 13-17 Sept. 2010.

[13] D. Batory, "Feature Models, Grammars, and Propositional Formulas," *Proceedings of the 9th International Conference on Software Product Lines (SPLC '05), LNCS,* vol. 3714, pp. 7-20, 26-29 Sept. 2005.

[14] M. F. Johansen, Ø. Haugen and F. Fleurey, "Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible," *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS '11), LNCS,* vol. 6981, pp. 638-652, 16-21 Oct. 2011.

[15] M. F. Johansen, Ø. Haugen and F. Fleurey, "An Algorithm for Generating T-wise Covering Arrays from Large Feature Models," *Proceedings of the 16th International Software Product Line Conference (SPLC '12),* vol. 1, pp. 46-55, 2-7 Sept. 2012.