Distributed Real-time Architecture for Mixed Criticality Systems

Metamodels for platform-specific modelling
D 1.6.1

Project Acronym	DREAMS	**Grant Agreement Number**		FP7-ICT-2013.3.4-610640	
Document Version	1.0	**Date**	30.06.2016	**Deliverable No.**	D 1.6.1
Contact Person	Simon Barner	**Organisation**		fortiss	
Phone	+49 (0)89360352222	**E-Mail**		barner@fortiss.org	

Contributors

Name	Partner
Simon Barner	FORTISS
Alexander Diewald	FORTISS
Carmen Carlan	FORTISS
Fernando Eizaguirre	IKL
Lionel Havet	RTAW
Ramon Serna Oliver	TTT
Ali Syed	TUKL
Alfons Crespo	UPV
Patricia Balbastre Betoret	UPV

Table of Contents

Executive Summary

This deliverable describes the DREAMS *Metamodels for platform-specific modelling*. It complements the DREAMS *Metamodels for Application and Platform* described in [2] and consists of metamodels to specify resource utilizations, service configurations, and tool-specific input formats.

The *resource utilization metamodel* can be used for the fine-grained description of the resource consumption of mixed-criticality applications implemented on the DREAMS platform. It consists of a virtual link metamodel that allows to describe the characteristics and routes of DREAMS virtual links, a schedule metamodel for the specification allocation schemes (e.g., time-triggered schedules), and a metamodel for the description of global and local reconfiguration strategies. Further, the timing metamodel [2] is refined to support the decomposition of timing chains in order to ease their verification.

The *service configuration viewpoint* contributes metamodels to describe configurations of building blocks of the DREAMS platform. It consists of a configuration infrastructure that defines the interface to the DREAMS configuration generation framework [10][11], as well as configuration metamodels for the components of the virtual and physical DREAMS platform.

In addition to the resource utilization metamodel that is used as an exchange format between the different offline resource adaptation tools developed in DREAMS, this document also specifies the input formats of tools for partition scheduling (Xoncrete), TTEthernet network scheduling (TTPlan), and transition mode generation (MCOSF) [8][9]. Further, the document also refines the metamodel used by the safety constraints and rules checker [2].

1 Introduction

This deliverable discusses how platform-dependent aspects of the DREAMS platform are modelled. The following sections give some guidance to the reader by summarizing its structure and pointing out its role in the project.

1.1 Structure of the Deliverable

Chapter 2 describes the overall architecture of the DREAMS *platform-specific metamodel*, which is summarized in the following:

- Chapter 3 refines the deployment viewpoint introduced in [2] with tool-independent metamodels for the fine-grained description of the resource consumption of mixed-criticality applications implemented on the DREAMS platform.
- Chapter 4 describes configuration metamodels defined for the building blocks of the DREAMS platform.
- Chapter 5 provides an overview of the input/output formats of the offline resource allocation tools developed in WP4.

Finally, Annex A contains the full listings of XML schemas used to define some of the above metamodels.

In this document, the following typographic conventions are used.

- Package names, plugin names and metamodel types (classes, enumerations, etc) are set in `typewriter`. Where the context is clear, the package name is omitted from class names.
- Instance objects of metamodel classes are set in *italics* (typically used in the discussion of examples).

1.2 Positioning of the Deliverable in the Project

The DREAMS architectural style [1] constitutes the blueprint of the platform developed in DREAMS and hence served as the initial input for the specification of the DREAMS platform-specific model (PSM).

In addition, the following documents describing the fine-grained specification and actual implementation of the building blocks of the DREAMS platform have been considered as input for the definition of the different configuration metamodels:

- [3][17]: On-Chip Local Resource Scheduler
- [2][21]: XtratuM hypervisor
- [5]: TTEthernet off-chip communication
- [6]: Global Resource Management

The resulting configuration metamodels are defined in Chapter 4. They are used as input for the configuration generators for the DREAMS physical platform [10][11] and the DREAMS virtual platform [17][18][19].

As the resource utilization metamodels defined in Chapter 3 serve as the exchange format between the different offline resource allocation tools developed in WP4, the definition of the PSM has been performed in close collaboration with working task T4.1 *"Offline adaptation strategies for mixed criticality"* [8][9] and working task T4.4 *"Tool integration and Demonstrator Support"* [13].

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).

2 Architecture of DREAMS Platform-Specific Metamodel

Figure 1.2.1 gives an overview of the overall architecture of the PSM developed in DREAMS and sketches its integration into the DREAMS tool-chain that is defined in more detail in [13]. The PSM consists of two main parts that will be described in the following.

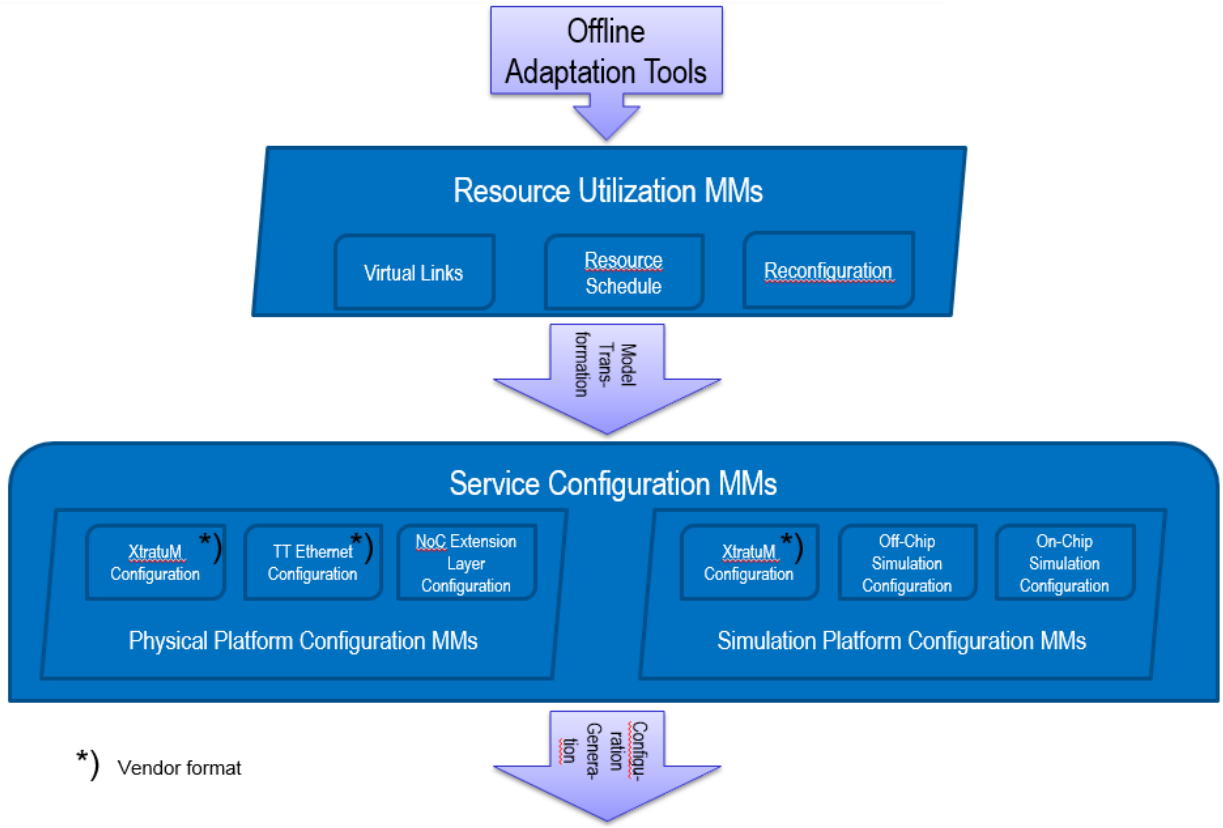


Figure 1.2.1: Architecture of DREAMS platform-specific model.

The *Resource Utilization Metamodel* (see Chapter 3) is an extension of the deployment viewpoint introduced in [2]. Its purpose is to capture the output of the offline resource adaptation tools developed in [8][9] in a tool and device-independent format. It can be used to describe the mapping of elements of the logical architecture (e.g., channels, ports) to platform resources (e.g., execution and transmission units) [2] as well as the allocation schemes (e.g., time-triggered schedules) for the following resource types:

- Computation Resources
  - Logical component to execution unit mapping (already defined in [2]):
  - Partition and task schedules (see Section 3.2)
- Communication Resources
  - On-chip and off-chip schedules (see Section 3.2)
  - DREAMS virtual links [1] and their (static) routes (see Section 3.1).

Furthermore, the resource utilization model comprises a reconfiguration metamodel (see Section 3.3). It can be used to describe the reconfiguration strategies determined by the offline resource adaptation tools developed in [8][9] that enable to react on faults according to the selected fault hypothesis by remapping applications to other platform resources and changing the underlying resource allocation.

The *Service Configuration Viewpoint* (see Chapter 4) contributes metamodels that provide additional implementation-specific parameters on top of the resource utilization metamodel. When a

configuration is generated for a given DREAMS platform building block [10][11][17][19], a model-transformation translates the information from the resource utilization model instance to a service configuration model, performing the required format conversions (e.g., time representation) and setting default values for parameters that are specific to the configuration of a DREAMS building block (e.g., synchronization strategy, use of interrupts or polling, etc.), and are hence not captured by the resource utilization metamodel.

As illustrated in Figure 1.2.1, we distinguish between configuration metamodels that are based on existing vendor configuration formats that have been extended in the scope of the project, i.e., XtratuM Hypervisor Configuration (see Section 4.5) and TTEthernet Network Configuration (see Section 4.6), and configuration metamodels that have entirely been developed in the scope of the project (all other configuration metamodels).

3 Resource Utilization Metamodel

The Resource Utilization Metamodel extends the deployment viewpoint introduced in [2]. Its purpose is to capture the output of the offline resource adaptation tools developed in [8][9] in a tool and device-independent format. It comprises the following sub-metamodels

- The virtual link metamodel (see Section 3.1) allows to describe the characteristics [2] and routes of DREAMS virtual links.
- The schedule metamodel (see Section 3.2) can be used to model allocation schemes (e.g., time-triggered schedules).
- The reconfiguration metamodel (see Section 3.3) can be used to describe local and global reconfiguration strategies [8][9].
- The metamodel presented in Section 3.4 is an extension of the timing metamodel documented in [2] that allows to decompose timing chains in order to ease their verification.

The implementation of the resource utilization metamodels is bundled with the AutoFOCUS3/DREAMS (see [2], Section 3.2.2 for instructions on how to obtain the package).

3.1 Virtual Links Metamodel

3.1.1 Communication Deployment

The virtual links metamodel is an extension to the deployment metamodel presented in [2] that allows to describe the characteristics of DREAMS virtual links as well as the corresponding (static) routes.

As defined in [1], Section 1.1, virtual links are an abstraction over physical networks at different levels of the DREAMS platform architecture. They are defined as end-to-end multicast channels between the `OutputPort` of one logical sender `Component` and the `InputPort`s of multiple logical receiver Components (see [2] for definition of logical `Port`s and `Component`s). Each virtual link has a unique identifier (ID), and defines the message's route, traffic type (time-triggered or rate-constraint) and timing (period or minimum inter-arrival time). Furthermore, the namespace defined in [1] defines the mapping between the logical architecture and the platform architecture.

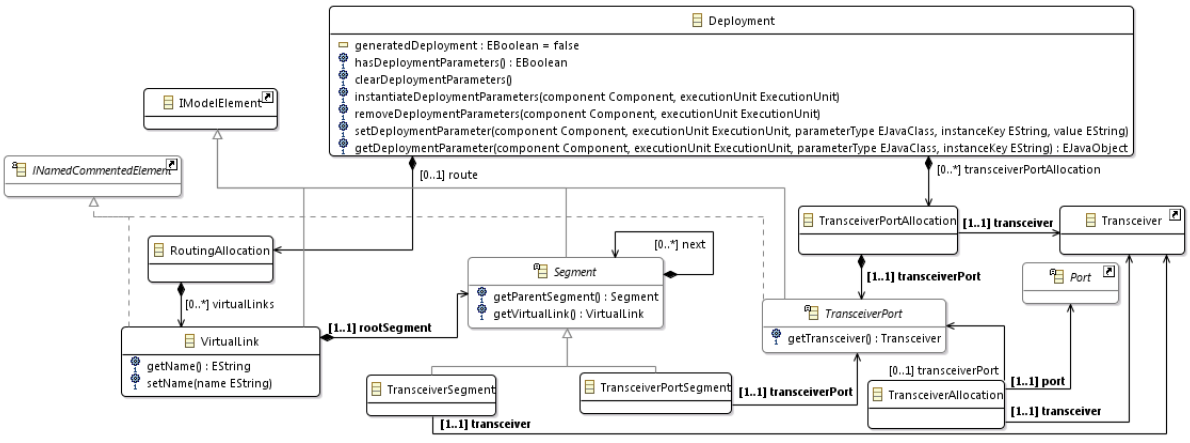


Figure 3.1.1: Extension of Deployment Metamodel (VirtualLinks, TransceiverPorts).

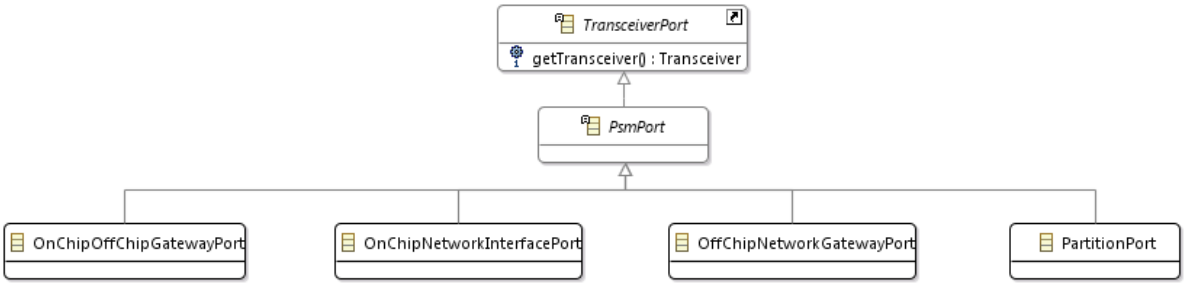


Figure 3.1.2: Platform-specific TransceiverPorts defined for DREAMS platform.

In the following, it will be sketched how the concepts of virtual links summarized above is considered in an extension of the deployment metamodel introduced in [2] (see Figure 3.1.1).

- `VirtualLink`s are contained by the `route: RoutingAllocation` field that has been added to `Deployment`.
- For the definition of routes and physical names, the notion of `TransceiverPort`s has been introduced. They are contained in the `transceiverPortAllocation: TransceiverPortAllocation` field that has been added to `Deployment`.

In order to describe platform-specific parameters of endpoints and waypoints of virtual links, the concept of `TransceiverPort`s has been introduced: For each virtual link, `TransceiverPorts` can be allocated to the `Transceiver`s [2] of the `TransmissionUnit`s [2] involved in the route that define parameters such as names, IDs, etc. Typical examples are:

- `PartitionPort` (allocated to `InterPartitionComPort` [2] of `Partition`s [2]) represent the communication ports of hypervisor partitions.
- `OnChipNetworkInterfacePort` (allocated to `BusOnChipNetworkExport` [2] of `NetworkInterface`s [2]) represent the communication ports of a `Tile`'s [2] network-interface, i.e., the on-chip LRS.

Figure 3.1.2 gives an overview of all `TransceiverPort`s defined for the DREAMS architecture that share the common base class `PsmPort`.

[1] defines the logical name of a message as *<Criticality>.<Subsystem>.<Component>.<Message>*. As pointed out in [2], the underlying architecture is modelled as a `ComponentArchitecture` that contains a logical `Component` for each *Subsystem*. The *Component* is also modelled as logical `Component` that is contained by the respective subsystem-`Component`. The *Message*-based interface of `Component`s is modelled using `OutputPort`s.

Now, the logical name is defined as a 4-tuple of integers (IDs are based on the following annotations introduced below):

- *Criticality*: annotated to *Subsystem* logical `Component` using the `SafetyIntegrityLevel` annotation [2].
- *Subsystem*: `ComponentId` (annotated to *Subsystem* logical `Component`).
- *Component*: `ComponentId` (annotated to *Component* logical `Component`).
- *Message*: `MessageId` (annotated to logical `OutputPort`).

For the physical name *<Cluster>.<Node>.<Tile>.<Port>*, the following annotations have been introduced below (see [2] for definition of `PlatformArchitecture` elements `Cluster`, `Node`, and `Tile` and Section 3.1.3.4 for the definition of the ID types):

- *Cluster*: `ClusterId` (bound to `Cluster`s).
- *Node*: `NodeId` (bound to `Node`s).
- *Tile*: `TileId` (bound to `Tile`s).
- *Port*: `OnChipNetworkInterfacePortId` (bound to `OnChipNetworkInterfacePort`s, see Section 3.1.3.2).

The one-to-one mapping of the logical name to the physical name introduced in [1] is implemented by an extension of the TransceiverAllocation. In [2], the TransceiverAllocation has been introduced as mapping of logical Ports to Transceivers attached to TransmissionUnits. Now, a TransceiverAllocation allows to additionally specify a TransceiverPort allocated to the corresponding Transceiver. Since every logical Component is mapped to a Partition of a Hypervisor, the TransceiverAllocation maps the Component's InputPorts and OutputPorts to the PartitionPort allocated to the Partition's InterPartitionComPort.

VirtualLinks are contained by a Deployment's RoutingAllocation. A VirtualLink is characterized by its endpoints, parameters, and its route.

- Endpoints
  - Given a VirtualLink, its sender / receiver PartitionPorts can be determined using the VlSender / VlReceivers annotation.
  - Using the TransceiverAllocation, the resulting PartitionPorts can be resolved to the corresponding logical OutputPort / InputPorts for which the VirtualLink has been created.
- Parameters
  - VlTrafficType defines the VirtualLink's type (time-triggered or rate-constraint).
  - VlTempRepetition defines the temporal requirements onto the virtual link (period for time-triggered virtual links, minimum inter-arrival time for rate-constraint virtual links).
  - VlPayloadSize defines the maximum payload size for the given virtual link.
- Route
  - VirtualLinks represent multi-cast communication in a distributed system. Hence, its route is represented as a tree of Segments, where a segment represent one hop of a route, i.e., from one Transceiver to another. The tree representation assumes that there is only one possible route between each sender and receiver, which reflects the capabilities of the current implementation of the underlying platform services. However, the VirtualLink class could easily be changed to use a directed graph to represent routes (using Transceivers / TransceiverPorts as vertices, and Segments as edges). E.g. during configuration generation, all routes could be explored by traversing this graph from the route vertex (Transceiver / TransceiverPort of sender) using a depth-first search.
  - A TransceiverPortsSegment represents a Segment that ends in a Transceiver for which a dedicated TransceiverPort concretization has been defined in the metamodel (the corresponding TransceiverPort instance is allocated in TransceiverAllocation).
  - A TransceiverSegment represents a Segment that ends in a Transceiver for which no TransceiverPorts have been defined.

3.1.2 Example

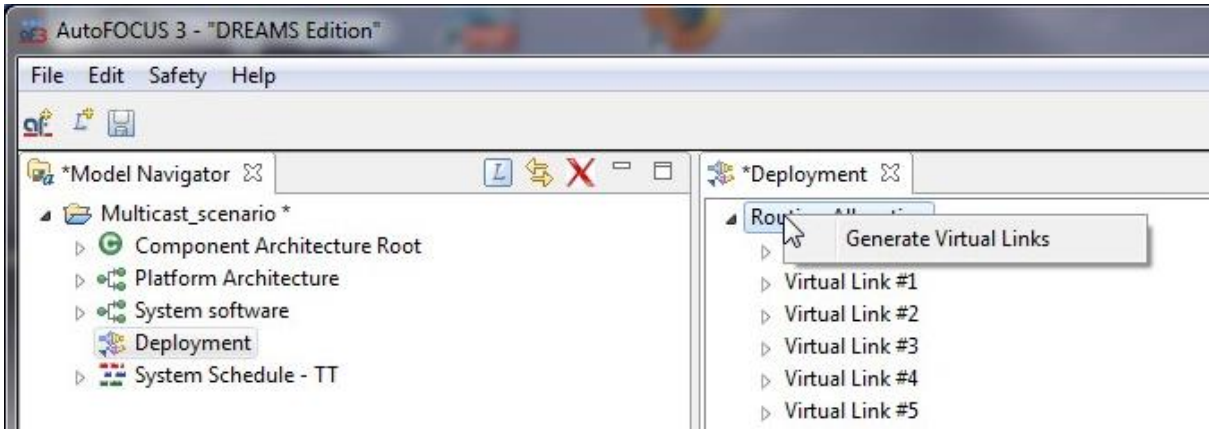


Figure 3.1.3: Starting the Virtual Link generation for a specific deployment model

In this section, the virtual link metamodel and the corresponding tool-support will be illustrated based on the example depicted in Figure 3.1.4. Virtual links are contained in a `RoutingAllocation` of a deployment, which can be accessed via the deployment editor (see [2], Section 6.4). The tab *Virtual Links* opens the corresponding virtual link viewer. The context menu of the empty pane or an existing `RoutingAllocation` allows to (re-)generate the Virtual Links that are implicitly defined via the logical architecture (see [2], Section 4) and the `ComponentAllocation`s of the Deployment (see [2], Section 6). The route of the virtual links is calculated automatically using the shortest path from the sending resource to the receivers. Virtual links are only generated for time-triggered and rate-constraint messages [1]. In order to make the location of the receiver transparent to the configuration generator for the hypervisor, (pseudo-) `VirtualLink` objects are also created for messages that are exchanged between partitions that reside in the same hypervisor (the DREAMS architectural style [1] foresees virtual links only for inter-tile or inter-node communication).

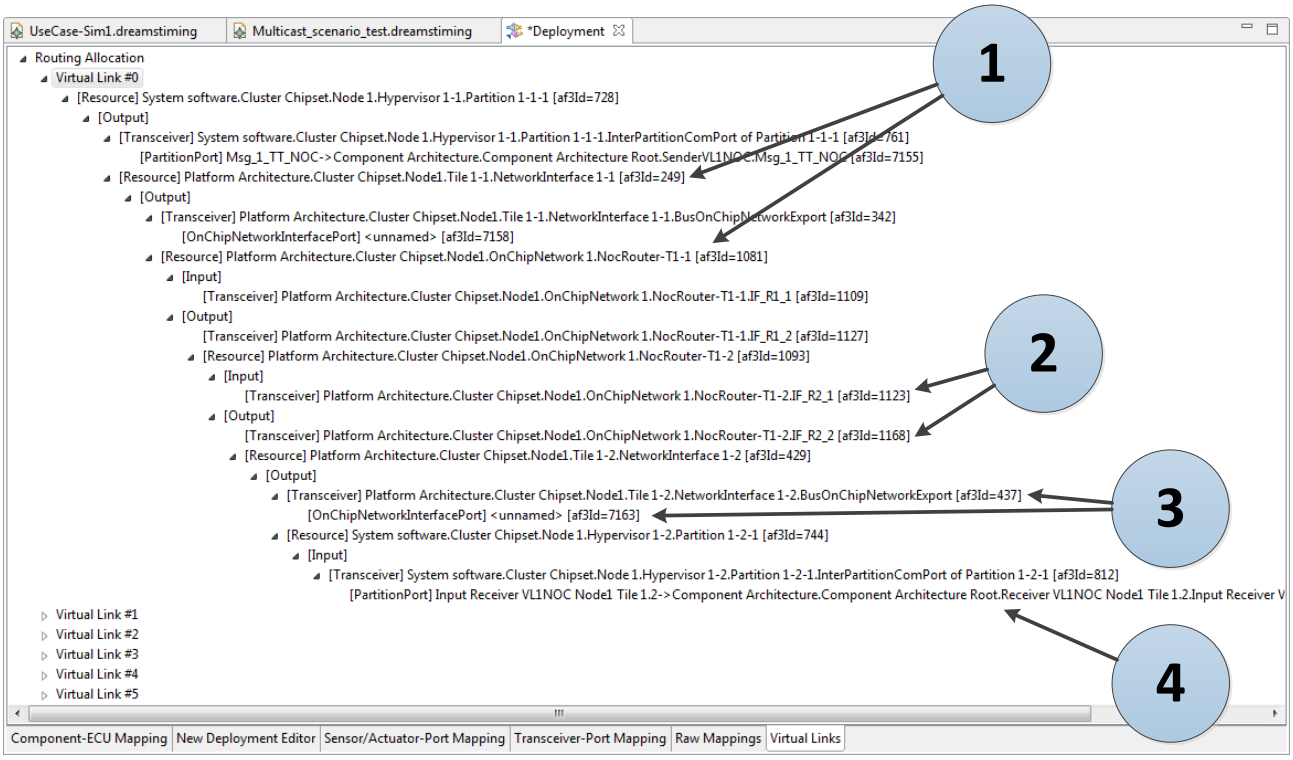


Figure 3.1.4: Virtual Link view integrated in the Deployment editor.

As pointed out in Section 3.1.1, virtual links represent multicast messages, which effectively results in a tree-structure of traversed elements (however note that the virtual link #0 in Figure 3.1.4 has only one receiver). Virtual links provide a bridge between the data exchanged among logical `Component`s and their realization as messages on the target platform. The corresponding elements in the logical application architecture are `OutputPort`s, which define logical senders, and connected `InputPort`s (via logical `Channel`s [2], see Section 3.1.1), which receive the sent data. In the target platform, the equivalent to the mentioned `Port`s are the `Transceiver`s that are defined in the *port mapping* of the deployment. These `Transceiver`s are the communication interfaces of the `ExecutionUnit`s [2] to which the `Component`s [2] containing the sender and receiver ports are deployed.

From the perspective of the target platform, a virtual link is a route from the sender resource (in DREAMS: a `Partition` [2]) through the involved communication resources to the target resources, e.g., the `NetworkInterface` [2] and the `NocRouter` [2] denoted by (1) in Figure 3.1.4. The actual route of virtual links is a tree of traversed `Transceiver`s and `TransceiverPort`s (which reference the traversed `Transceiver`s of the traversed communication resources) in the target platform, since the route of a Virtual Link is not uniquely defined by the involved resources. Hence, the in- and outputs of each traversed communication resource is explicitly defined. The `Transceiver`s and `TransceiverPort`s are encoded by `Transceiver-` and `TransceiverPortSegment`s, respectively, in the `VirtualLink`'s route. For instance, in Figure 3.1.4, the blue circle (2) points to the `Transceiver`s (resp. `TransceiverSegment`s) of the communication resource *NocRouter-T1-2* that receive the virtual links and that emit it to the successor resources. Similarly, the blue circle (3) points to the `TransceiverPort` of the `TransceiverPortSegment` associated with the resource *NetworkInterface 1-2* and to the `Transceiver` referenced by the `TransceiverPort`.

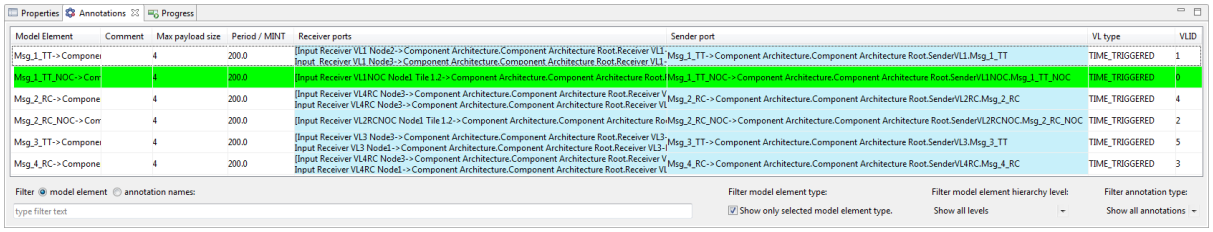


Figure 3.1.5: Properties of a Virtual Link provided as Annotations (see [2]).

The annotation view of Deployment models provides a tabular view on the properties of a virtual link (see Figure 3.1.5). In particular, the properties *maximum payload size*, *Period or MINT (minimum inter-arrival time)*, *Virtual Link Type*, and the *Virtual Link ID* have been implemented as annotations. Also, the *Sender Port* (a *PartitionPort*: derived from the name of the `OutputPort` in the logical architecture during the virtual link generation), and a list of *Receiver Ports* (also `PartitionPort`s) which are derived from the route of the Virtual Link, e.g., the `TransceiverPort` to which the blue circle (4) points, are provided as annotations. Further, the properties *Direction* and *Semantics* of `TransceiverPort`s that are generated along with virtual links are available as annotations (see Figure 3.1.6).

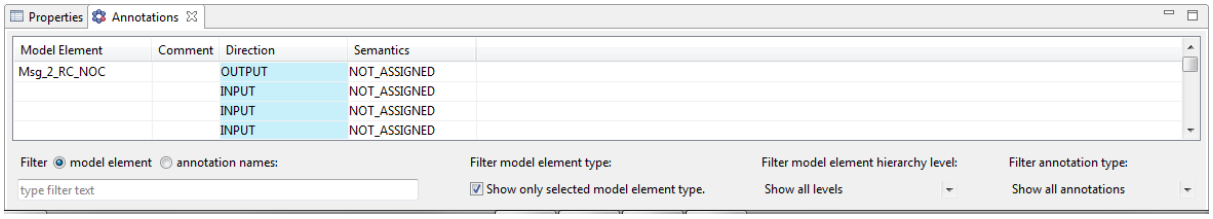


Figure 3.1.6: Properties of `TransceiverPort`s provided as Annotations (see [2]).

3.1.3 Reference Documentation

In the following, the newly introduced/extended packages and classes will be documented.

3.1.3.1 Package org.fortiss.af3.deployment

Table 3.1.1 provides an overall summary of the implementation and dependencies of the deployment metamodel

Name	Deployment Metamodel (extended)
Description	The deployment metamodel establishes the link between the logical architecture and the platform that realizes the logical architecture.
Ecore file	`deployment.ecore`
Plugin	`org.fortiss.af3.deployment`
Packages	`org.fortiss.af3.deployment`

Dependencies	`org.fortiss.af3.component`
	`org.fortiss.af3.platform`
	`org.fortiss.af3.timing`
	`org.fortiss.af3.project`
	`org.fortiss.tooling.base`
	`org.fortiss.tooling.kernel`

Table 3.1.1: Extended Deployment Metamodel.

3.1.3.1.1 Extension of Deployment

The extension of the `org.fortiss.af3.deployment` to implement the concept of virtual links is anchored in the `Deployment` class. Since it has already been introduced in [2], only the extensions are described in the following:

- `Deployment`:
  - "Root" class that contains the mapping between the logical architecture and the hardware platform (see [2]).
  - **Attributes:**
    - `routingAllocation`: Contains the `Deployment's RoutingAllocation`, i.e., the set of `VirtualLink`s corresponding to all messages in the system (see Section 3.1.3.1.3)
    - `transceiverPortAllocation`: Contains the `Deployment's TransceiverPortAllocation` (see Section 3.1.3.1.2)
- `TransceiverAllocation`:
  - Attributes:
    - `port`: Logical `Port` for which the `Transceiver` and optionally also the `TransceiverPort` should be specified. In hierarchic `PlatformArchitecture`s, the `Transceiver` and optionally also the `TransceiverPort` that constitute the endpoints of the communication link are specified. In this case, the `TransceiverPort`s of the intermediate hops can be determined by investigating the corresponding `VirtualLink`.
    - `transceiver`: `Transceiver` to which the logical `Port` is allocated.
    - `transceiverPort`: Resource reservation within the `Transceiver` to which the given logical `Port` is mapped. This field is optional and required for the fine-grained description of communication deployment. `TransceiverPort`s are owned by the `TransceiverPortAllocation` map (see Section 3.1.3.1.2).

3.1.3.1.2 TransceiverPorts

The concept of `TransceiverPort`s is based on the following classes (see Figure 3.1.2 for concrete `TransceiverPort`s defined for the communication components of the DREAMS platform).

- `TransceiverPortAllocation`:
  - Assigns a `TransceiverPort` to a `Transceiver`.
  - **Attributes:**
    - `transceiver`: `Transceiver` for which a `TransceiverPort` has been allocated.
    - `transceiverPort`: `TransceiverPort` allocated for the given `Transceiver`.
- `TransceiverPort`:
  - Resource reservation within a `Transceiver`. This abstract class must be concretized by ports for specific platforms, e.g., to represent ports in hypervisor partitions, etc. (see Section 3.1.3.2).
  - **Operations:**
    - `getTransceiver()`: Obtain `Transceiver` for which this `TransceiverPort` has been allocated.

3.1.3.1.3 Virtual Links

The following classes have been introduced to describe deployment of communication. They are parameterized with universal (see Section 3.1.3.2) and DREAMS-specific annotations (see Section 3.1.3.4).

- `RoutingAllocation`:
  - Specifies how communication is deployed, i.e., the set `VirtualLink`s corresponding to all messages in the system.
  - **Attributes:**
    - `virtualLinks`: Set of `VirtualLink`s required to deploy the communication for the `Deployment` owning this `RoutingAllocation`.
- `VirtualLink`:
  - Specifies how a single (multi-cast) message is deployed in the system. While this model only specifies the message's (static) route, for specific platforms further attributes (IDs, traffic type, temporal properties) can be added using `IAnnotatedSpecifications`.
  - **Attributes:**
    - `rootSegment`: The root-`Segment` of the tree representing this `VirtualLink`.
  - **Operations:**
    - `getName()`: Returns the `VirtualLink's` name.
    - `setName()`: Sets the `VirtualLink's` name.
- `Segment`:
  - Base class to define one hop of a multicast message's (static) route.
  - **Attributes:**
    - `next`: The `Segment`s following this `Segment`.
  - **Operations:**
    - `getParentSegment()`: Returns the parent `Segment` of this `Segment`. For the root `Segment`, `null` is returned.
    - `getVirtualLink()`: Returns the `VirtualLink` that contains this `Segment`.

- `TransceiverSegment`:
  - Concrete `Segment` that is characterized by the sending `Transceiver`. (`TransceiverPorts` cannot be allocated for `Transceiver` types).
  - **Attributes:**
    - `transceiver`: The source Transceiver of this TransceiverSegment.
- `TransceiverPortSegment`:
  - Concrete `Segment` that is characterized by the sending `Transceiver's TransceiverPort`.
  - As described in Section 3.1.3.2, `TransceiverPorts` have only been defined for those `Transceiver` types where message-specific information is required (e.g., name, ID, etc.).
  - **Attributes:**
    - `transceiverPort`: The `TransceiverPort` at the source `Transceiver` of this `TransceiverPortSegment`.

3.1.3.2 Package eu.dreamsproject.psm.model.port

Name	**DREAMS Platform-Specific Ports**
Description	DREAMS-specific `TransceiverPort` concretizations
Ecore file	`psm.ecore`
Plugin	`eu.dreamsproject.psm`
Packages	`eu.dreamsproject.psm.model.port`

Dependencies	`org.fortiss.af3.deployment`
	`org.fortiss.tooling.base`
	`org.fortiss.tooling.kernel`

Table 3.1.2: DREAMS-specific `TransceiverPort` concretizations.

The `org.fortiss.af3.psm.port` package is contained by the `psm.ecore` metamodel and consists of the classes below (see Figure 3.1.2):

- `PsmPort`:
  - Base class for DREAMS platform-specific ports.
- `OffChipNetworkGatewayPort`:
  - Ports defined for `Transceivers` of `OffChipNetworkGateways` [2].
- `OnChipNetworkInterfacePort`:
  - Ports defined for `Transceivers` of `NetworkInterfaces` [2], i.e., for `BusMasterPorts` [2].
- `OnChipOffChipGatewayPort`:
  - Ports defined for `Transceivers` of `OnChipOffChipGateways` [2].
- `PartitionPort`:
  - Ports defined for `Transceivers` of `Partitions` [2].

Parameters are assigned to `PsmPorts` using annotations (see Sections 3.1.3.3.1 and 3.1.3.4.3).

3.1.3.3 Package org.fortiss.af3.deployment.annotation

This package provides annotations for `TransceiverPort`s and `VirtualLink`s that are generally applicable to model communication deployment.

Name	**Deployment Metamodel / Annotations**
Description	Annotations for `TransceiverPort`s and `VirtualLink`s
Ecore file	`deployment.ecore`
Plugin	`org.fortiss.af3.deployment`
Packages	`org.fortiss.af3.deployment.annotation`

Dependencies	`org.fortiss.af3.component`
	`org.fortiss.af3.platform`
	`org.fortiss.af3.timing`
	`org.fortiss.af3.project`
	`org.fortiss.tooling.base`
	`org.fortiss.tooling.kernel`

Table 3.1.3: Annotations for TransceiverPorts and VirtualLinks.

3.1.3.3.1 Annotations for TransceiverPorts

The `org.fortiss.af3.deployment.annotation` package contributes the following annotations for `TransceiverPort`s:

- `TransceiverPortDirection`:
  - `IAnnotatedSpecification` providing the direction of `TransceiverPorts`.
  - **Operations:**
    - `getDerivedValue()`: Returns the annotated `TransceiverPort`'s direction depending on the `VirtualLink` that referenced the `TransceiverPort`. In case the `TransceiverPort` is not referenced by any `VirtualLink`, `TrafficDirection.NOT_ASSIGNED` is returned.
- `TransceiverPortSemantics`:
  - `IAnnotatedSpecification` providing the semantics of `TransceiverPort`s.
  - **Attributes:**
    - `semantics`: Returns the `PortSemantics` assigned to the annotated `TransceiverPort`, i.e., whether it is a state or an event port.
- `TransceiverPortAccessRights`:
  - `IAnnotatedSpecification` defining the access rights for a given `TransceiverPort`, i.e., the list of `ExecutionUnit`s that may read (for `INPUT` ports) or write (for `OUTPUT` ports) the `TransceiverPort`.
  - **Attributes:**
    - `admissibleExecutionUnits`: list of `ExecutionUnits` that may read (for `INPUT` ports) or write (for `OUTPUT` ports) the `TransceiverPort`.

3.1.3.3.2 Annotations for VirtualLinks

Furthermore, the `org.fortiss.af3.deployment.annotation` package contributes the following annotations for `VirtualLink`s:

- `VlTrafficType`:
  - IAnnotatedSpecification providing the `TrafficType` of `VirtualLink`s.
  - **Attributes:**
    - `type`: TrafficType of the annotated `VirtualLink`, i.e., whether it is `TIME_TRIGGERED` or `RATE_CONSTRAINT`.

- `VlSender`:
  - `DerivedAnnotation` providing the sender `TransceiverPort` of `VirtualLink`s.
  - **Operations:**
    - `getDerivedValue()`: Returns the sender `TransceiverPort` of `VirtualLink`s.
- `VlReceivers`:
  - `IAnnotatedSpecification` providing the list of receiver `PartitionPort`s of `VirtualLink`s.
  - **Operations:**
    - `getDerivedValue()`: Returns the list of receiver `TransceiverPort`s of `VirtualLink`s.
- `VlTempRepetition`:
  - `IDerivedAnnotation` providing the period (for periodic `VirtualLink`s) or the minimum inter-arrival time (for sporadic `VirtualLink`s).
  - **Attributes:**
    - `periodOrMint`: Period (for periodic `VirtualLink`s) or the minimum inter-arrival time (for sporadic `VirtualLink`s) (in seconds).
- `VlPayloadSize`:
  - `IAnnotatedSpecification` providing the maximum allowed size of the payload associated with a `VirtualLink` (in bytes).
  - **Attributes:**
    - `maxSize`: Maximum allowed size of the payload associated with a `VirtualLink` (in bytes).

3.1.3.4 Package eu.dreamsproject.psm.model.annotation

Name	DREAMS Platform-Specific Metamodel / Annotations
Description	DREAMS-specific annotations for communication deployment.
Ecore file	`psm.ecore`
Plugin	`eu.dreamsproject.psm`
Packages	`eu.dreamsproject.psm.model.annotation`

Dependencies	`org.fortiss.af3.timing`
	`org.fortiss.tooling.base`
	`org.fortiss.tooling.kernel`

Table 3.1.4: DREAMS-specific annotations related to communication deployment.

This package provides annotations required for communication deployment that are specific to the DREAMS platform (mainly element IDs).

3.1.3.4.1 Annotations for ComponentArchitecture elements

- `ComponentId`:
  - `IAnnotatedSpecification` providing the IDs of `Component`s.
  - **Attributes:**
    - `componentId`: Provides the ID of the `Component`.
- `MessageId`:
  - `IAnnotatedSpecification` providing the IDs of logical `OutputPort`s (i.e., messages according to the nomenclature introduced in [1]).
  - **Attributes:**
    - `messageId`: Provides the message ID.

3.1.3.4.2 Annotations for PlatformArchitecture elements

- `ClusterId:`
  - `IAnnotatedSpecification` providing the IDs of `Cluster`s.
  - **Attributes:**
    - `clusterId:` ID of the `Cluster.`
- `NodeId:`
  - `IAnnotatedSpecification` providing the IDs of `Node`s.
  - **Attributes:**
    - `nodeId:` ID of the `Node.`
- `TileId:`
  - `IAnnotatedSpecification` providing the IDs of `Tile`s.
  - **Attributes:**
    - `tileId:` ID of the `Tile.`
- `CoreId:`
  - `IAnnotatedSpecification` providing the ID of `Core`s.
  - **Attributes:**
    - `coreID:` ID of the `Core.`
- `PartitionId:`
  - `IAnnotatedSpecification` providing the IDs of `Partition`s.
  - **Attributes:**
    - `partitionId:` ID of the `Partition.`
- `OnChipNetworkInterfaceId:`
  - `IAnnotatedSpecification` providing the IDs of `OnChipNetworkInterface`s.
  - **Attributes:**
    - `onChipNetworkInterfaceId:` ID of the `NetworkInterface.`
- `OnChipVirtualNetworkId:`
  - `IAnnotatedSpecification` providing the ID of the virtual network served by the annotated `NetworkInterface.`
  - **Attributes:**
    - `virtualNetworkId:` Provides the ID of the virtual network served by the annotated `NetworkInterface.`

3.1.3.4.3 Annotations for TransceiverPorts / PsmPorts

- `OffChipNetworkRouterPortId:`
  - `IAnnotatedSpecification` providing the IDs of `OffChipNetworkRouterPorts.`
  - **Attributes:**
    - `offChipNetworkRouterPortId:` ID of the `OffChipNetworkRouterPort.`
- `OnChipNetworkInterfacePortId:`
  - `IAnnotatedSpecification` providing the IDs of `OnChipNetworkInterfacePorts.`
  - **Attributes:**
    - `onChipNetworkInterfacePortId:` ID of the `OnChipNetworkInterfacePort.`

3.1.3.4.4 Annotations for VirtualLinks

- `Vlid:`
  - `IAnnotatedSpecification` providing the IDs of `VirtualLink`s.
  - **Attributes:**
    - `vlid:` ID of the `VirtualLink.`

3.2 Schedule Metamodel

3.2.1 Hierarchical Resource Schedules

The schedule metamodel is a device- and tool-independent metamodel to represent hierarchical schedules. It is a format that can capture the output of the different device-/platform-service specific scheduling tools and allows to exchange schedules between them (as part of the DREAMS toolchain [13]). Figure 3.2.1 provides an overview of the schedule metamodel.

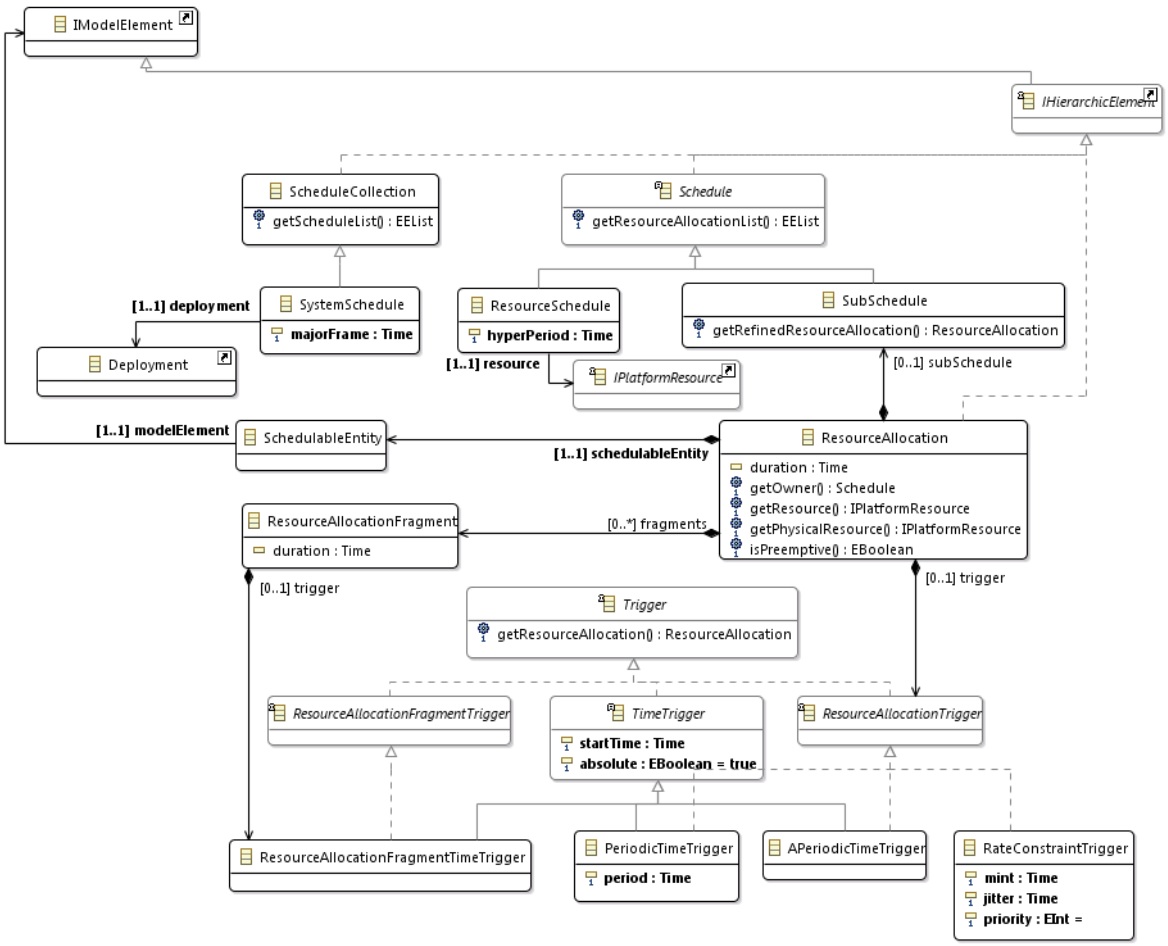


Figure 3.2.1: Schedule Metamodel.

A `SystemSchedule` collects the schedules for all involved resources in its `Deployment` (that defines the mapping of the elements of the logical `ComponentArchitecture` to the `PlatformArchitecture` it references [2]). The Major Frame (MAF) is the period of time to be considered when computing the task allocation and scheduling. The MAF of the `SystemSchedule` is normally equal to the least-common multiple (LCM) of the periods of all `ResourceSchedule`s, but can be less than this value under certain conditions such as harmonicity or geometricity of periods (see [9], Section 6.1.4).

The schedule of each individual resource is described using a `ResourceSchedule`. It has the following main attributes:

- `resource`: a reference to a resource in the physical platform architecture contained by the `Deployment`'s `PlatformArchitecture`. Since in DREAMS, the `Deployment` references a system software `PlatformArchitecture`, the actual physical resources are contained in the `PlatformArchitecture` model of the underlying hardware platform (see [2]).

  - In the case of Partition schedules, the corresponding ResourceSchedule references a Core. For multi-core Partitions, a ResourceSchedule is defined for each core.
  - In the case of on-chip communication schedules, the corresponding ResourceSchedule references an (on-chip) NetworkInterface.
  - etc.
- hyperperiod: Typically the LCM of the periods of all referenced ResourceAllocations, but see comment above about length of MAF.

The allocation of a share of the resource referenced by a ResourceSchedule is expressed using a ResourceAllocation. The containment relationship between Schedules and ResourceAllocations is expressed using the containedElements field inherited from IHierarchicElement. A ResourceAllocation has the following main attributes

- The SchedulableEntity contains a reference to the IModelElement to be scheduled onto the referenced resource: For example:
  - In the case of partition schedules, the ResourceAllocation's SchedulableEntity references a Partition.
  - In the case of task schedules, the ResourceAllocation's SchedulableEntity references a Component (see discussion of SubSchedules below).
  - In the case of communication schedules, the ResourceAllocation's SchedulableEntity references a VirtualLink.
- duration: Time for which the resource is reserved for the referenced SchedulableEntity.
- trigger: A Trigger object is used to specify the temporal activation pattern of the ResourceAllocation (see below for a description of the available trigger types).

The schedule metamodel is designed to express hierarchical schedules: The share of a resource described by a ResourceAllocation allocated in ResourceSchedule can be further sub-divided by declaring a SubSchedule for it. SubSchedules apply the concept recursively, i.e., they also contain ResourceAllocations that reference a SchedulableEntity and a Trigger object. A SubSchedule does not reference a platform resource, since it refines a ResourceAllocation. In DREAMS, this concept is used to define task schedules for the Partitions within partitions schedules.

Triggers are an extensible way to specify the temporal activation pattern of ResourceAllocations. In the scope of DREAMS, the following Triggers are relevant:

- PeriodicTimeTriggers allow to specify the period and phase of a ResourceAllocation and are intended for strictly time-triggered activities.
- APeriodicTimeTriggers provide the start time of aperiodic activities.
- RateConstraintTriggers define the maximum jitter and the minimum inter-arrival time of sporadic activities.

3.2.2 Example

While schedules are typically computed by tools (see [13]), they can be viewed (and instantiated, e.g., for testing), in the AF3 Model Navigator (see Figure 3.2.2).

When a system schedule is selected in the navigation pane, the properties view allows the user to reference a Deployment model for which the schedule is modelled and displays the calculated Major Application Frame (MAF) of the schedule (see Figure 3.2.3).

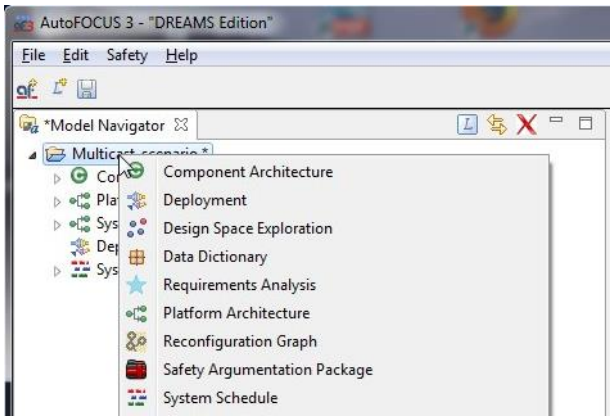


Figure 3.2.2: Create a System Schedule.

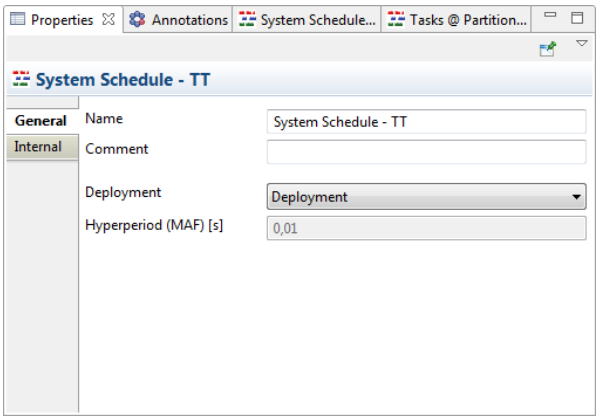


Figure 3.2.3: Properties of a System Schedule.

Sub-elements of a schedule model can be added via the context menu of the model navigator (see Figure 3.2.4). A system schedule is organized as `ResourceSchedule`s of the `IPlatformResource`s that are present in the target hardware platform (see Figure 3.2.5). In the following, *Resource Schedule - Core 1-1-1* will be discussed that specifies the schedules of the hypervisor `Partition`s that are allocated to *Core 1-1-1*. This `ResourceSchedule` contains `ResourceAllocation`s of the `Partition`s (e.g., *Resource Allocation – Partition -1-1-1*) which are assigned to the corresponding `Core`s via a `ResourceLink` annotations in the *System Software Platform* model (see [2], Sections 5.2.6 and 5.3.7). Each `ResourceAllocation` for a `Partition` contains a `Subschedule` (e.g., *Tasks @ Partition 1-1-1*) that represents the schedule of the tasks executed within the `Partition`. These `SubSchedule`s are again a list of `ResourceAllocation`s that point to a `Component` in the logical architecture (e.g., *Resource Allocation – Component Sender VL1, Resource Allocation – Component Sender VL1NOC*, etc.).

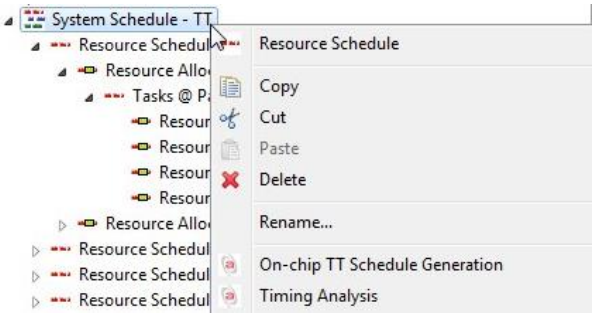


Figure 3.2.4: Adding Elements to a Schedule.

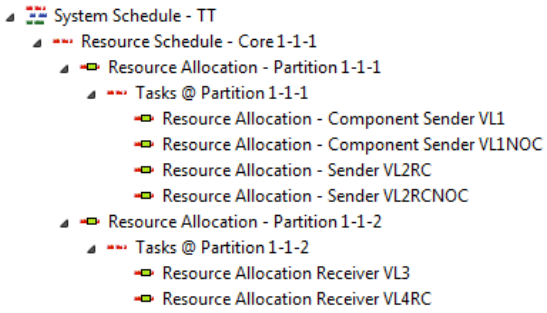


Figure 3.2.5: Tree of Schedule Elements.

The properties of each of the entities mentioned in the previous paragraphs are accessible via the properties view (see Figure 3.2.6 and Figure 3.2.7). This view allows the user to reference the corresponding entities from the logical architecture or the platform architecture models. For instance, one `ResourceSchedule` in the example references *Core 1-1-1* which implies that it defines the schedule executed on this resource (see Figure 3.2.6). Similarly, the contained `ResourceAllocation` references *Partition 1-1-1* implying that this entity is scheduled on *Core 1-1-1* with the corresponding properties (see Figure 3.2.7).

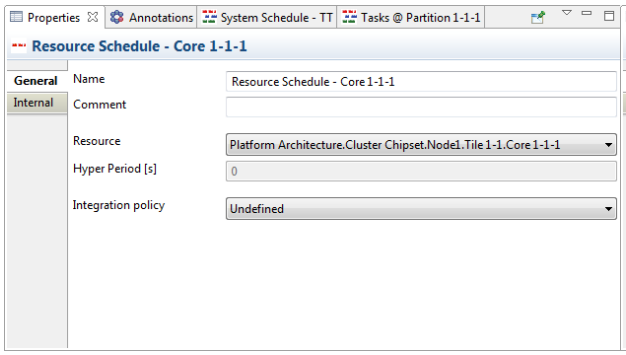


Figure 3.2.6: Properties of a `ResourceSchedule`.

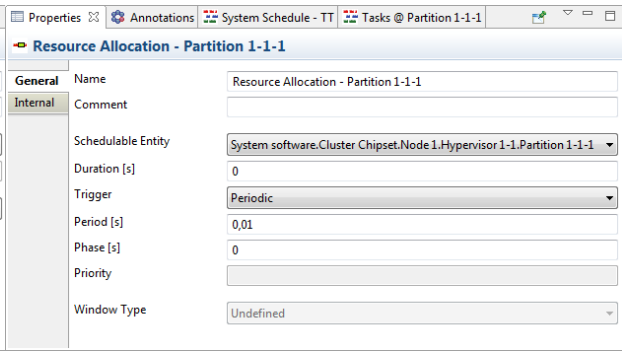


Figure 3.2.7: Properties of a `ResourceAllocation`.

In addition to the tree representation in the model navigator, a Gantt chart viewer can be opened by selecting the *Schedule View* via the quick access menu in the top-right corner of AF3-DREAMS edition (see Figure 3.2.8).

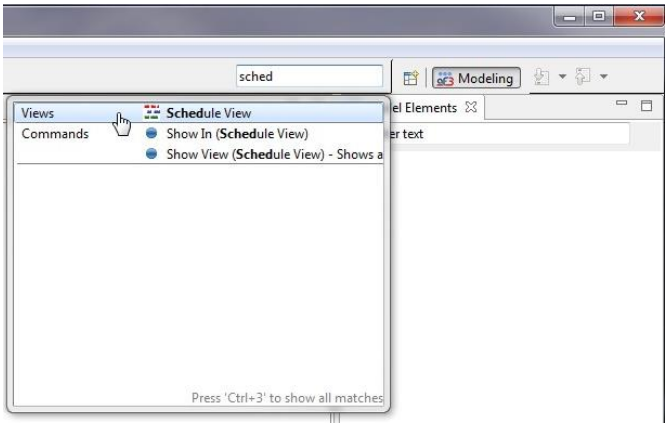


Figure 3.2.8: Opening the Schedule View.

When the *Schedule View* is opened, it displays a Gantt chart of the `SystemSchedule` in the model navigator (see Figure 3.2.9). In the example, the `ResourceAllocation`s *Partition 1-1-1* and *Partition 1-1-2*, which are assigned to and scheduled in *Core 1-1-1*, are shown.

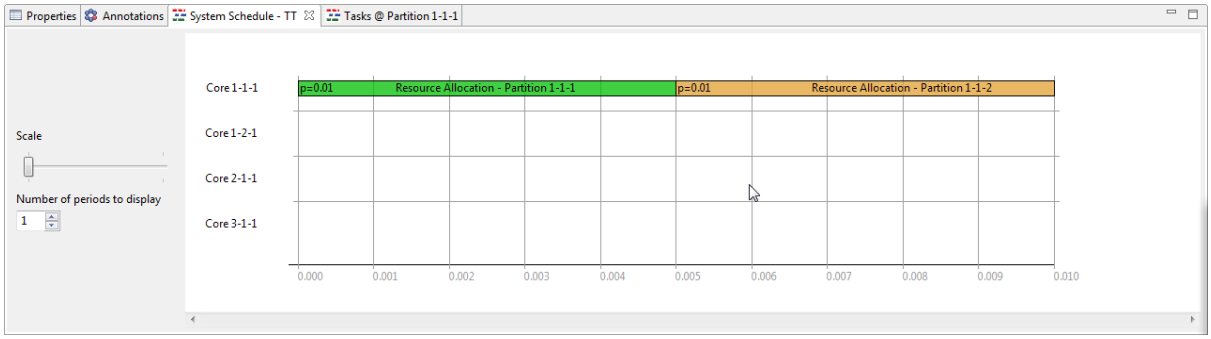


Figure 3.2.9: Gantt Chart of a sample System Schedule.

The `Subschedule` of a `ResourceAllocation` can be opened in a dedicated tab using the middle mouse button. Further, the view can be zoomed to a range selected using the left mouse button, or zoomed in (out) using the left (right) mouse button. Figure 3.2.10 illustrates the task schedule associated to *Partition 1-1-1*.

Figure 3.2.10: Gantt Chart of a sample Task Schedule.

3.2.3 Reference Documentation

3.2.3.1 Package org.fortiss.af3.schedule

Name	Schedule Metamodel
Description	Device-independent hierarchical schedule metamodel.
Ecore file	`schedule.ecore`
Plugin	`org.fortiss.af3.schedule`
Packages	`org.fortiss.af3.schedule`

Dependencies	`org.fortiss.tooling.base`
	`org.fortiss.tooling.kernel`
	`org.fortiss.af3.deployment`
	`org.fortiss.af3.timing`
	`org.fortiss.af3.platform`

Table 3.2.1: Device-independent Hierarchical Schedule Metamodel.

The package `org.fortiss.af3.schedule` implements the hierarchical schedule metamodel introduced in the previous section. Its classes are structured into three groups that will be explained in the following:

3.2.3.1.1 Schedule

The following classes are used to represent hierarchical schedules:

- `ScheduleCollection`:
  - Base class to define collections of `Schedule`s.
  - **Operations:**
    - `getScheduleList()`: Returns the list of `Schedule`s contained in this `ScheduleCollection`.
- `SystemSchedule`:
  - Collection of all required schedules for a given `Deployment`.
  - **Attributes:**
    - `deployment`: The `Deployment` on which this `SystemSchedule` is based on.
    - `majorFrame`: System's major frame (MAF), i.e., usually the LCM of all `ResourceSchedules` owned by this `SystemSchedule`.
- `Schedule`:
  - Base class for `ResourceSchedules` and `SubSchedules`.
  - **Operations:**
    - `getResourceAllocationList()`: Returns the list of `ResourceAllocations` defining this `Schedule`.

- `ResourceSchedule`:
  - Schedule for a single resource of the system (specification how the resource is time-shared between the allocated `SchedulableEntity`s.
  - **Attributes:**
    - `resource`: `IPlatformResource` whose usage is defined by this `ResourceSchedule`.
    - `hyperPeriod`: `ResourceSchedule`'s hyper-period.
- `SubSchedule`:
  - A `SubSchedule` refines the given `ResourceAllocation` by defining a `Schedule` that further divides the use of the corresponding `IPlatformResource`.
  - **Operations:**
    - `getRefinedResourceAllocation()`: Returns the `ResourceAllocation` that is refined by this `SubSchedule`, which further divides the use of the corresponding `IPlatformResource`.

3.2.3.1.2 Resource Allocation

Resource allocations are used to specify the allocation of resources (e.g., time-slot) within a schedule for a given schedulable entity (e.g., task, or message).

- `ResourceAllocation`:
  - Specification how the `Schedule` containing this `ResourceAllocation` reserved a share of its `IPlatformResource` for the referenced `SchedulableEntity`.
  - **Attributes:**
    - `trigger`: Reference to the `Trigger` (see Section 3.2.3.1.3) that defines the activation condition for this `ResourceAllocation`.
    - `schedulableEntity`: Handle to the `IModelElement` for which the allocation of a particular `IPlatformResource` is described by this `ResourceAllocation`.
    - `duration`: Total amount of time reserved on the underlying `IPlatformResource` for the execution of the referenced `SchedulableEntity` (for the time instant specified using the given `trigger`).
    - `subSchedule`: Sub-`Schedule` that divides this `ResourceAllocation` (optional).
    - `fragments`: Optional specification how execution of the referenced `IModelElement` is split. Must not be set for non-pre-emptive schedules.
  - **Operations:**
    - `getOwner()`: Returns the `Schedule` owning this `ResourceAllocation`.
    - `getResource()`: Returns the owning Schedule's `IPlatformResource` that serves this `ResourceAllocation`.
    - `getPhysicalResource()`: Returns the physical `IPlatformResource` that serves this `ResourceAllocation`, i.e., the `IPlatformResource` associated to the top-level `ResourceSchedule`.
    - `isPreemptive()`: Predicate if the execution of the `SchedulableEntity` referenced by the given `ResourceAllocation` is pre-emptive.

- `ResourceAllocationFragment`:
  - A fraction of a `ResourceAllocation` (optional field of `ResourceAllocation` that may only be set for pre-emptive schedules).
  - **Attributes:**
    - `trigger`: Specification of this `ResourceAllocationFragment`'s activation condition.
    - `duration`: `Time` on resource managed by the `ResourceSchedule` containing this `ResourceAllocationFragment`.
- `ScheduableEntity`:
  - Handle to the `IModelElement` for which the allocation of a particular `IPlatformResource` is described by the `ResourceAllocation` containing this `SchedulableEntity`.
  - **Attributes**
    - `modelElement`: Reference to the `IModelElement` whose scheduling is specified by this `SchedulableEntity`.

3.2.3.1.3 Trigger

Triggers provide a temporal-specification when the schedulable entity associated to a resource allocation is executed. The following classes are available to represent triggers:

- `Trigger`:
  - Base class for specification of a `ResourceAllocation`'s activation condition.
  - **Operations:**
    - `getResourceAllocation()`: Returns the `ResourceAllocation` whose activation condition is specified by this `Trigger`.
- `ResourceAllocationTrigger`:
  - Marker interface if a concrete `Trigger` subclass can be used to specify the activation condition of a `ResourceAllocation`.
- `TimeTrigger`:
  - Base class for `Triggers` that are based on a direct specification of the start-time.
  - **Attributes:**
    - `startTime`: Start time / phase (computed by the scheduler).
    - `absolute`: Indicates whether the specified `startTime` is absolute (i.e., relative to the major frame of the `SystemSchedule`), or relative to the period of the `SubSchedule` that refines a particular `ResourceAllocation`.
- `APeriodicTimeTrigger`:
  - A `TimeTrigger` for aperiodic activities.
- `PeriodicTimeTrigger`:
  - A `TimeTrigger` for `ResourceAllocation`s for which a `period` is defined.
  - **Attributes:**
    - `period`: Period of a periodic activity for which the starting time has been specified using the `startTime` field of the `TimeTrigger` base class.
- `RateConstraintTrigger`:
  - Activation condition for rate-constraint / sporadic activation of `ResourceAllocation`s.
  - **Attributes:**
    - `mint`: (Specified) minimum inter-arrival time of a rate-constraint activity.
    - `jitter`: Maximum execution jitter as determined during the scheduling of rate-constraint activity.
    - `priority`: Priority of rate-constraint / sporadic activity.

- `ResourceAllocationFragmentTrigger`:
  - Marker interface if a concrete `Trigger` subclass can be used to specify the activation condition of a `ResourceAllocationFragment`.
- `ResourceAllocationFragmentTimeTrigger`:
  - The start time is relative to the period of the `SubSchedule` that refines a particular `ResourceAllocation`.

3.2.3.2 Package eu.dreamsproject.psm.model.annotation

Name	**DREAMS Platform-Specific Metamodel / Annotations**
Description	DREAMS-specific annotations for schedules
Ecore file	`psm.ecore`
Plugin	`eu.dreamsproject.psm`
Packages	`eu.dreamsproject.psm.model.annotation`
Dependencies	`org.fortiss.tooling.base` `org.fortiss.tooling.kernel`

Table 3.2.2: DREAMS-specific annotations related to schedules.

The following DREAMS specific properties can be annotated to schedule models:

- `IntegrationPolicy`:
  - `IAnnotatedSpecification` providing to `ResourceSchedules` the integration policy applied to resolve media access conflicts, i.e., if a high-priority message arrives while a low-priority message is being processed.
  - **Attributes:**
    - `integrationPolicy`: Integration policy applied to resolve media access conflicts, i.e., if a high-priority message arrives while a low-priority message is being processed. The following values are admissible
      - `UNDEFINED`: The media access resolution policy has not been defined.
      - `TIMELY_BLOCK`: The switch will not forward any message at times when a time-triggered message is expected.
      - `SHUFFLING`: If a low-priority message is relayed when a high-priority message arrives, the high-priority message is delayed until the processing of the low-priority message has finished (i.e., at most for a maximum-sized low priority message).
      - `PREEMPTION`: If a low-priority message is relayed when a high-priority message arrives, the relay process of the low-priority message is stopped. After the minimum time of silence on the transmission channel and a delay defined a priori, the high-priority message is relayed.

3.2.3.3 Package eu.dreamsproject.psm.model.schedule

Name	**DREAMS Platform-Specific Scheduling Metamodel**
Description	DREAMS-specific extensions for schedules
Ecore file	`psm.ecore`
Plugin	`eu.dreamsproject.psm`
Packages	`eu.dreamsproject.psm.model.schedule`

Dependencies	`org.fortiss.af3.schedule`
	`org.fortiss.tooling.base`
	`org.fortiss.tooling.kernel`

Table 3.2.3: DREAMS-specific annotations related to schedules.

The following DREAMS extensions have been provided to the schedule meta-model in order to support the concept of guarding and bypass windows (see configuration meta-model of physical and virtual on-chip network interface in Sections 4.2 and 4.3). These windows result from particular design choices for the DREAMS harmonized platform and are hence provided as separate package.

- `WindowSchedulableEntity`:
  - `SchedulableEntity` specialization used to define reserved windows in a `ResourceSchedule`. In contrast to its base class, it does not reference a particular model element (such as a `Component` or an `OutputPort`), but only reserves a certain amount of time in the schedule.
  - **Attributes:**
    - `windowType`: Type of reserved window. The following values are admissible:
      - `UNDEFINED`: The window type has not been defined.
      - `GUARDING_WINDOW`: Guarding windows used in DREAMS to reserve time-slots in in which no event triggered messages are allowed to be injected into the NoC.
      - `BYPASS_WINDOW`: Bypass windows are used to define slots where the accesses to the TTE and DDR controllers my bypass the on-chip-LRS.

3.3 Reconfiguration Metamodel

3.3.1 Resource Reconfiguration

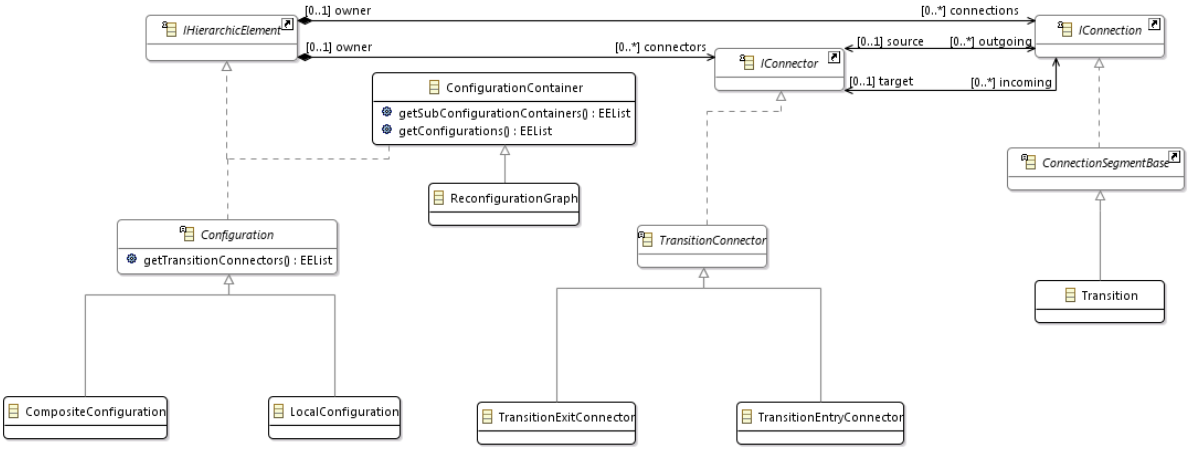


Figure 3.3.1: Reconfiguration Metamodel.

In DREAMS, recovery strategies based on global and local reconfiguration are expressed using reconfiguration graphs (see [6], and [8], Chapter 7). Figure 3.3.1 provides an overview of a reconfiguration metamodel that implements these concepts based on the AutoFOCUS3 Hierarchic Element Metamodel (see [2], Section 3.2.6.2).

`ReconfigurationGraph`s are organized using a hierarchic element model of `ConfigurationContainer`s. All children of a given `ConfigurationContainer` may be of exactly one of the following types:

- `ConfigurationContainer`: Further refinement of `ReconfigurationGraph`'s structure. In DREAMS, `ConfigurationContainer`s are used to define one "folder" for the system's global reconfiguration graph, and one folder for each of the system's `Node`s that contains sub-folders for the local reconfiguration graphs of the respective `Tile`s.
- `LocalConfiguration`: Node in a local reconfiguration graph (see below).
- `CompositeConfiguration` Node in a global reconfiguration graph (see below).

`LocalConfiguration`s define the system's recovery strategy at the `Tile` level. For each `Tile`, the recovery strategy is encoded as a graph (typically a tree), whose vertices are `LocalConfiguration` objects that reference a `ScheduleCollection` (see Section 3.2) using the `ReferencedScheduleCollection` annotation. The edges of the graph are represented by `Transition`s that specify the condition to switch from one local configuration to another (in DREAMS: list of failed resources defined using the `PhysicalPlatformArchitectureElementFailure` annotation). The initial mode is marked using the `InitialConfiguration` annotation and references a `SystemSchedule` that provides a `ResourceSchedule` for all platform resources contained by the `Tile`. Subsequent `LocalConfiguration`s reference partial `SystemSchedule`s that are based on updated `Deployment`s that exclude the list of failed resources and update the `ResourceSchedule` of the remaining resources (`ResourceSchedule`s that did not change in comparison to the predecessor in the graph may be omitted).

The recovery strategy at the system level (see [6]) is expressed as a graph (typically a tree) whose nodes are `GlobalConfiguration`s and whose edges are `Transition`s. The initial `GlobalConfiguration` is specified using the `InitialConfiguration` and references the set of all initial `LocalConfiguration`s using the `ReferencedConfiguration`s annotation. For each subsequent vertex, the `GlobalConfiguration` references the set of new local configurations to be activated (for `Tile`s whose local configuration should not be changed, nothing is specified).

It is important to note that global reconfiguration usually also involves a reconfiguration of the communication. Hence, the `Deployment`s referenced by the `LocalConfiguration`s, which are referenced by the corresponding `GlobalConfiguration`, also update the routes of the underlying `VirtualLink`s and the allocation of `PsmPort`s (see Section 3.1).

3.3.2 Example

While reconfiguration graphs are typically computed by tools (see [13]), they can be viewed (and instantiated, e.g., for testing), in the AF3 Model Navigator (see Figure 3.2.2 that also shows the menu entry for reconfiguration graphs).

Model elements that comprise a `ReconfigurationGraph` can be added via the Model Element library depicted in Figure 3.3.2, or via the context menu in the Model Navigator, similar to the creation of the `ReconfigurationGraph` itself.

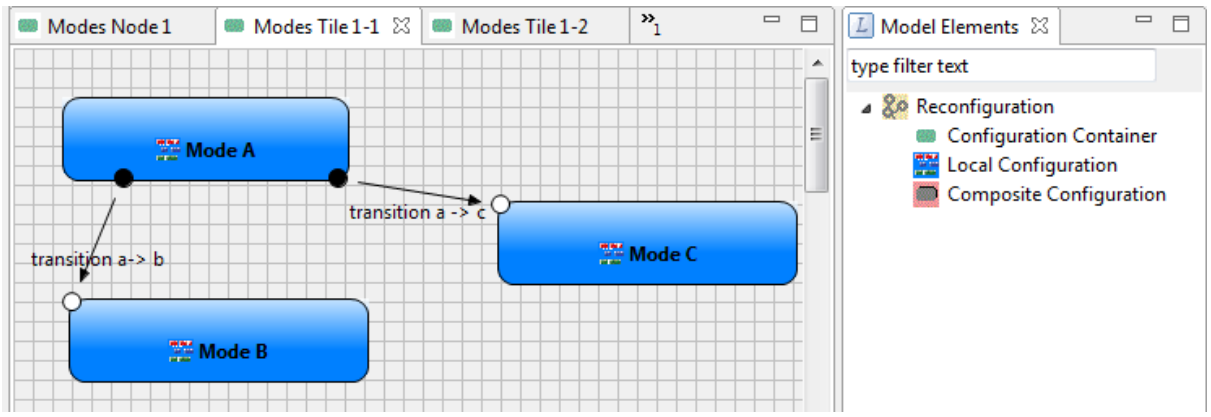


Figure 3.3.2: Additional Elements are added via the Element Library (right).

A `ReconfigurationGraph` is structured using `ConfigurationContainer`s. Figure 3.3.3 shows the typical structure, where one `ConfigurationContainer` contains the global reconfiguration graph, and where there is one `ConfigurationContainer` for each `Node`. The `ConfigurationContainer` for *Node 1* contains two sub-`ConfigurationContainer`s for its `Tile`s *Tile 1-1* and *Tile 1-2*.

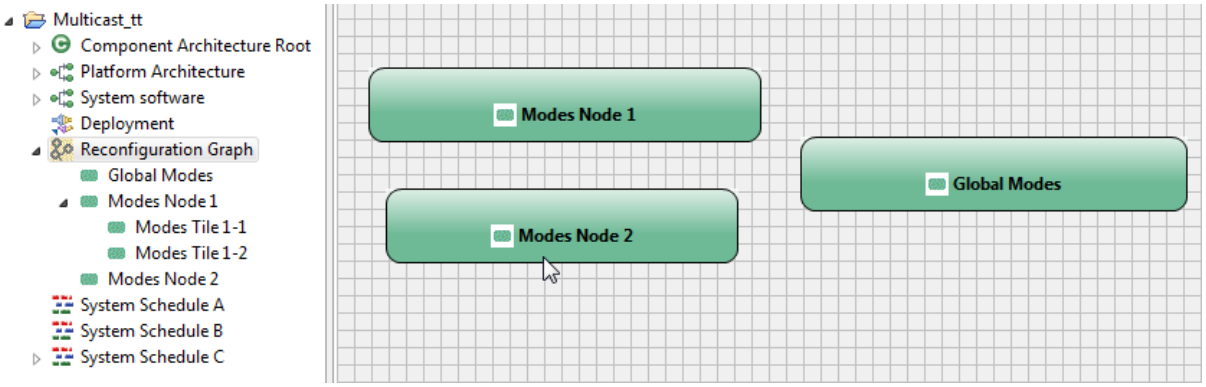


Figure 3.3.3: Global and local modes (`Node`s and `Tile`s) contained in a reconfiguration graph.

In the example of Figure 3.3.4, there exist two global modes: *Global Mode 1* and *Global Mode 2* between which a transition is defined.

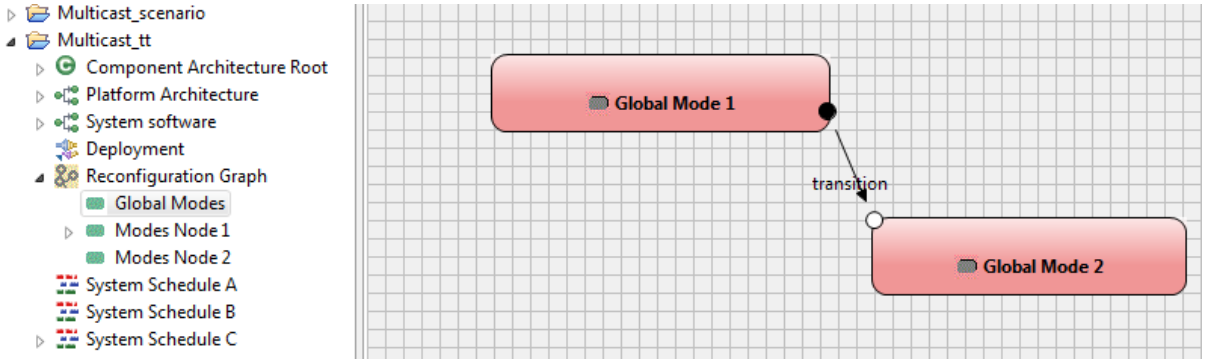


Figure 3.3.4: A global is a graph of `CompositeConfiguration`s.

Figure 3.3.5 shows the `LocalConfiguration`s (i.e., local modes) of *Tile 1-1*. There exist three local modes: *Mode A, Mode B*, and *Mode C.* A transition exists from *Mode A* to *Mode B* and from *Mode A* to *Mode C* and defines when to switch the local mode (see below for specification of transition condition).

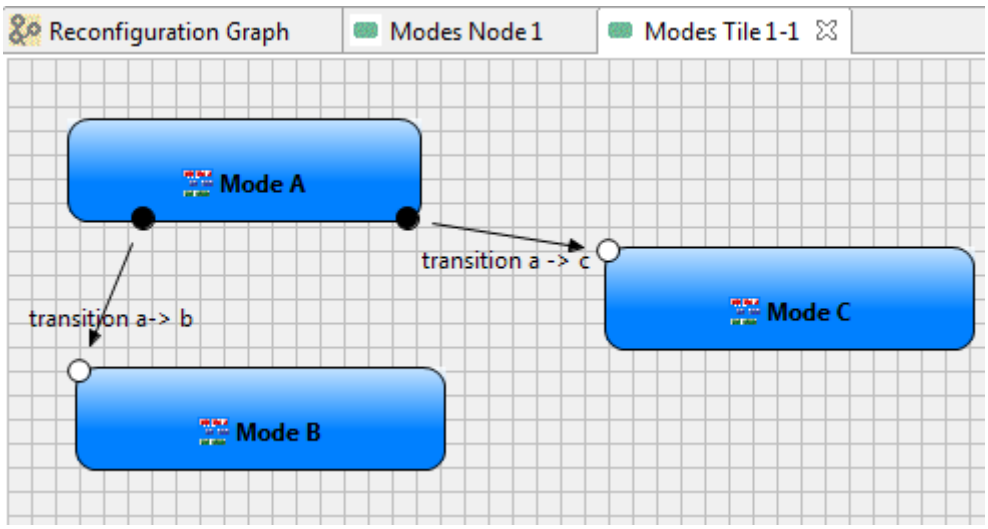
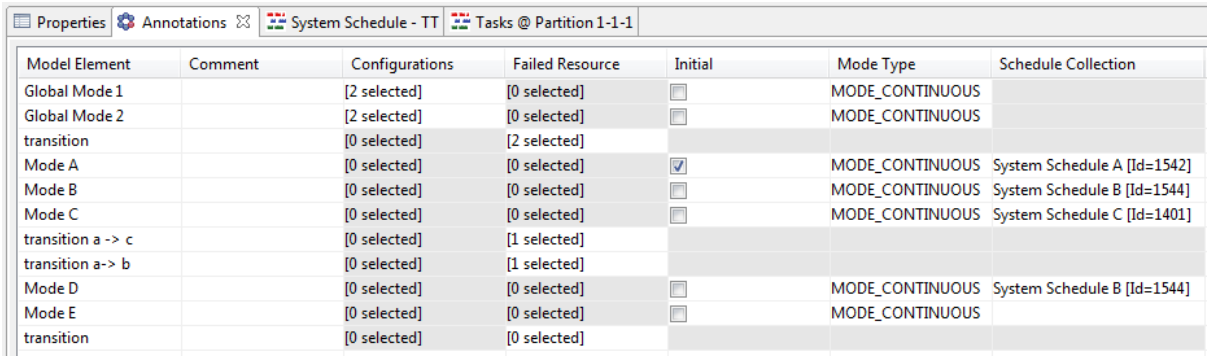


Figure 3.3.5: Local modes and their transitions of a `Tile`.

The main properties of the modes contained in a `ReconfigurationGraph` are defined via annotations (see Figure 3.3.6). `CompositeConfiguration`s have an annotation (*Configurations)* that allows the user to reference other configurations such that global modes are actually defined via its set of local modes. For each mode, it can also be defined whether a particular mode is the *initial* mode of its corresponding resource, and whether it is a continuous or a transition mode (*Mode Type*). The conditions to switch from one mode to another are defined via the *Failed Resource* annotation that is attached to `Transition`s. It allows the user to select the set of resources which need to fail such that the transition is triggered. For each of the local modes, there exists a further annotation which references a system schedule (*Schedule Collection*). Initial modes will reference a complete schedule of the system, whereas alternate modes, which are activated after some failures in the system, reference partial system schedules that provide a schedule for the resources for which the mode is defined.



Model Element	Comment	Configurations	Failed Resource	Initial	Mode Type	Schedule Collection
Global Mode 1		[2 selected]	[0 selected]	☐	MODE_CONTINUOUS	
Global Mode 2		[2 selected]	[0 selected]	☐	MODE_CONTINUOUS	
transition		[0 selected]	[2 selected]			
Mode A		[0 selected]	[0 selected]	☑	MODE_CONTINUOUS	System Schedule A [Id=1542]
Mode B		[0 selected]	[0 selected]	☐	MODE_CONTINUOUS	System Schedule B [Id=1544]
Mode C		[0 selected]	[0 selected]	☐	MODE_CONTINUOUS	System Schedule C [Id=1401]
transition a -> c		[0 selected]	[1 selected]			
transition a-> b		[0 selected]	[1 selected]			
Mode D		[0 selected]	[0 selected]	☐	MODE_CONTINUOUS	System Schedule B [Id=1544]
Mode E		[0 selected]	[0 selected]	☐	MODE_CONTINUOUS	
transition		[0 selected]	[0 selected]			

Figure 3.3.6: Annotations of a reconfiguration graph.

3.3.3 Reference Documentation

3.3.3.1 Package org.fortiss.af3.reconfiguration

Name	Reconfiguration Metamodel
Description	Local and global reconfiguration graph.
Ecore file	`reconfiguration.ecore`
Plugin	`org.fortiss.af3.reconfiguration`
Packages	`org.fortiss.af3.reconfiguration`
Dependencies	`org.fortiss.tooling.base` `org.fortiss.tooling.kernel`

Table 3.3.1: Reconfiguration Metamodel.

The reconfiguration metamodel has the following classes (see Figure 3.3.1):

- `ReconfigurationGraph`:
  - `IProjectRootElement` that hosts a reconfiguration specification.
- `Configuration`:
  - Base class for configurations. The `ReconfigurationGraph` specifies the set of available Configurations, and the Transition conditions between them.
  - **Operations:**
    - `getTransitionConnectors()`: Returns the `TransitionConnectors` that must be added to Configurations in order to be able to define Transitions.
- `LocalConfiguration`:
  - A local configuration. It references a configuration artifact using `IAnnotatedSpecifications` (e.g., `ScheduleCollections`).
- `CompositeConfiguration`:
  - A composite `Configuration` that references a set of other `Configurations`.
- `ConfigurationContainer`:
  - Container that groups several configurations or further sub-`ConfigurationContainers`. AF3 compositors ensure that within a given `ConfigurationContainer`, there can be only child elements of the same type.
  - **Operations:**
    - `getSubConfigurationContainers()`: Returns the sub-`ConfigurationContainers` owned by this `ConfigurationContainer`.
    - `getConfigurations()`: Returns the `Configurations` owned by this `ConfigurationContainer`.

- `TransitionConnector`:
  - `TransitionConnectors` must be added to `Configurations` in order to be able to define `Transitions`.
- `TransitionExitConnector`:
  - The source connector of `Transitions`.
- `TransitionEntryConnector`:
  - The target connector of `Transitions`.
- `Transition`:
  - `Transition` between two `Configurations`. Conditions are specified using `IAnnotatedSpecifications`.

3.3.3.2 Package org.fortiss.af3.reconfiguration

Name	Reconfiguration Metamodel / Annotations
Description	Annotations provided by reconfiguration metamodel.
Ecore file	`reconfiguration.ecore`
Plugin	`org.fortiss.af3.reconfiguration`
Packages	`org.fortiss.af3.reconfiguration.annotation`
Dependencies	`org.fortiss.tooling.base` `org.fortiss.tooling.kernel`

Table 3.3.2: Reconfiguration Metamodel (annotation package).

The reconfiguration metamodel comprises an annotation package, which contains the classes below:

- `ReferencedScheduleCollection`:
  - `IAnnotatedSpecification` to reference a `ScheduleCollection` from a `LocalConfiguration`.
  - **Attributes:**
    - `scheduleCollection`: Referenced `ScheduleCollection`
- `ReferencedConfigurations`:
  - `IAnnotatedSpecification` to reference a set of `Configuration`s from a `CompositeConfiguration`.
  - **Attributes:**
    - `configurations`: Referenced `Configuration`s
- `PhysicalPlatformArchitectureElementFailure`:
  - `IAnnotatedSpecification` to define `TransitionCondition` based on failures of `IPhyscialPlatformArchitectureElements`.
  - **Attributes:**
    - `physcialPlatformArchitectureElement`: The `IPhysicalPlatformArchitectureElement`s that must fail in order enable the `Transition` to which this `IAnnotatedSpecification` is bound.
- `InitialConfiguration`:
  - `IAnnotatedSpecification` to specify the initial `Configuration`.
  - **Attributes:**
    - `initial`: Boolean flag to specify the initial `Configuration`.

3.3.3.3 Package eu.dreamsproject.psm.model.annotation.reconfiguration

Name	Reconfiguration Metamodel / Annotation
Description	DREAMS-specific annotations for local and global reconfiguration graphs.
Ecore file	`psm.ecore`
Plugin	`eu.dreamsproject.psm`
Packages	`eu.dreamsproject.psm.model.annotation.reconfiguration`
Dependencies	`org.fortiss.tooling.base` `org.fortiss.tooling.kernel`

Table 3.3.3: Reconfiguration Annotation Metamodel.

The `eu.dreamsproject.psm.model.annotation.reconfiguration` package has the following classes:

- `BlackoutSlot`:
  - `IAnnotatedSpecification` providing a flag if a given `ResourceAllocation` represents a black-out slot (only applicable for time-triggered slots).
  - **Attributes:**
    - `blackout`: Flag if the annotated `ResourceAllocation` represents a black-out slot (only applicable for time-triggered slots).
- `ConfigurationModeType`:
  - `IAnnotatedSpecification` to specify the `Configuration`'s mode type (see below).
  - **Attributes:**
    - `modeType`: `Configuration` mode type.
- `ModeType`:
  - Enumeration to define the type of mode `Configuration`.
    - `MODE_CONTINUOUS`: A continuous mode is one that is executed cyclically until there is a switching event (e.g., core failure).
    - `MODE_TRANSITION`: A transition mode is one that is executed at most once and switches to a continuous mode when there is a safe point to switch. Safe points are where the black-out parameter of a time slot is false (black-out property is defined by the scheduler).
- `HypervisorSwitchTimeDelay`:
  - `IAnnotatedSpecification` defining the delay time required by a `Hypervisor` to switch its plan.
  - **Attributes:**
    - `delay`: Delay time required by a `Hypervisor` to switch its plan (in seconds).

3.4 Extension of Timing Metamodel

The DREAMS timing metamodel [1] is enriched to provide new elements required for the decomposition of timing chains in a DREAMS application modelled using the DREAMS application metamodel [1]. This section lists the modifications carried out on the DREAMS timing metamodel.

3.4.1 Timing decomposition

A requirement on an end-to-end timing chain may not be easily verified directly when many components are involved in the path. Splitting the timing chain and the timing requirement may ease the verification of the global timing requirement.

This time budgeting decomposition is illustrated in Figure 3.4.1. The initial requirement of 53 ms is split in 4 budgets (either arbitrarily or guided by a tool):

- 16 ms for the scheduling of *A*, *B* and *C* on *Node0*,
- 10 ms for the network on-chip traversal time,
- 17 ms for the network off-chip ,
- 10 ms for the scheduling of *D* and *E* on *Node1*.

Each of these 4 constraints can be more easily verified locally by dedicated tools.

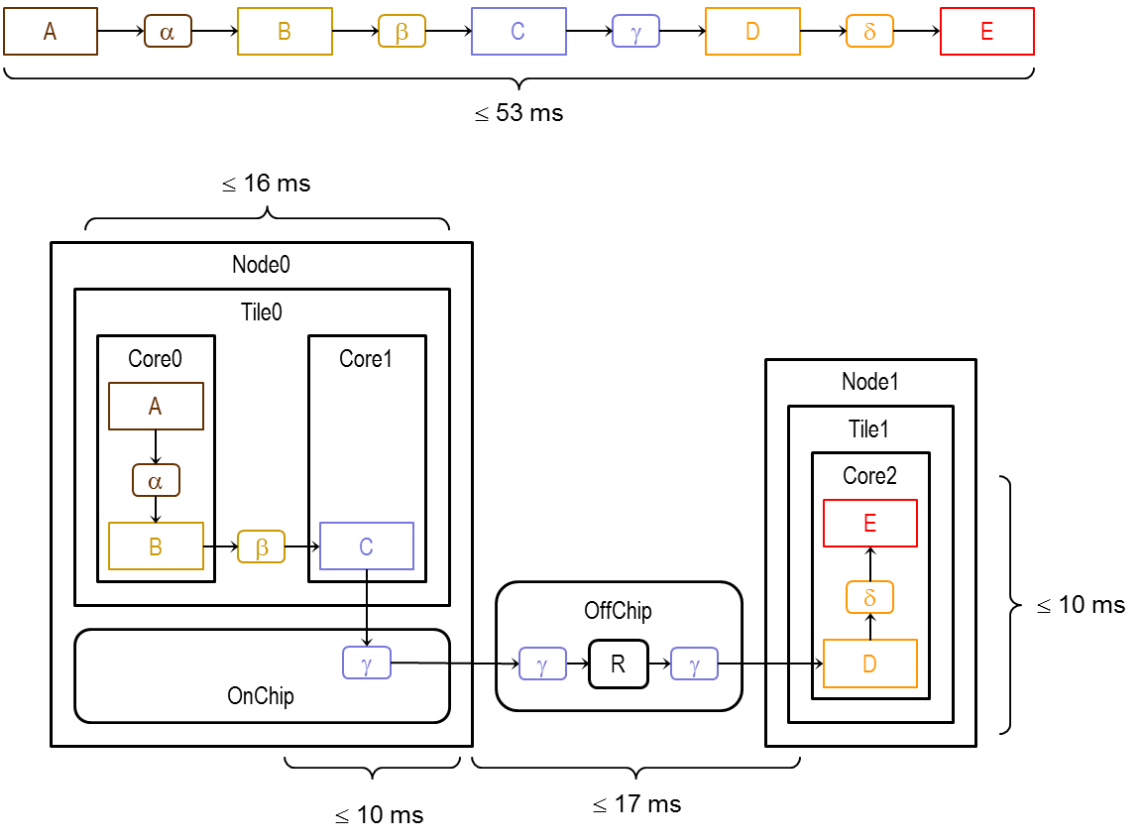


Figure 3.4.1: Timing chain A->E decomposition

3.4.2 Example

The updated timing viewpoint is instantiated for the timing requirements of an on-chip/off-chip scenario, as shown on Figure 3.4.2.

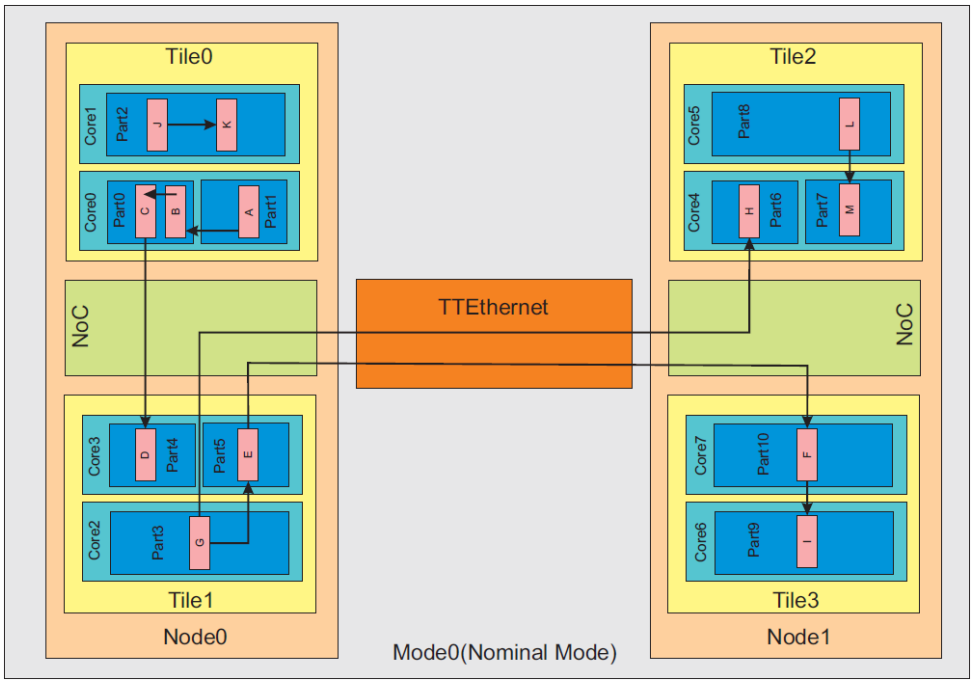


Figure 3.4.2: Timing Model Sample Scenario

The timing metamodel enables the specification of an end-to-end timing constraint at the component level as illustrated in Figure 3.4.3. The timing chain can already be decomposed with the WCET from each component in the path.

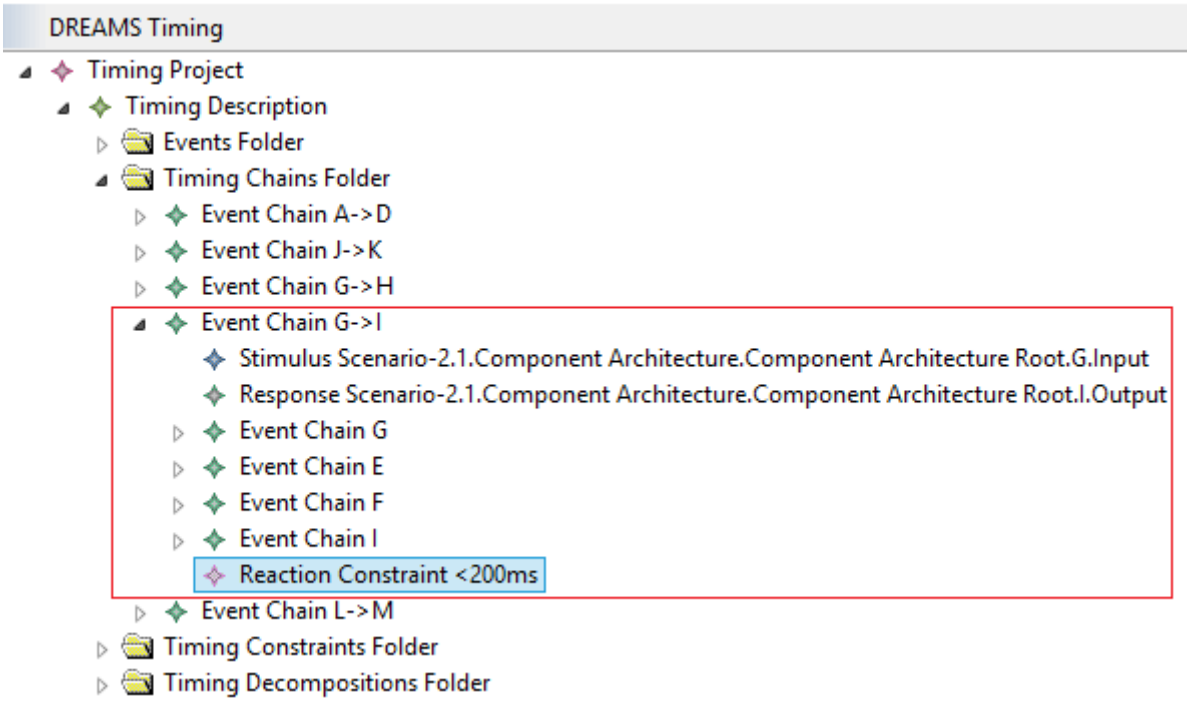


Figure 3.4.3: End to end timing chain from G to I

The refinement principle allows the user to refine timing budgets amongst entities in the path flow of an end-to-end latency constraint for a given deployment.

The latencies introduced by the on-chip network communication of *Node0* and *Node1* and the off-chip communication of TTEthernet are introduced in a refined decomposition related to the chosen deployment. This decomposition is illustrated in Figure 3.4.4. This decomposition of the *G* to *I* end-to-end latency constraint is split in 5 segments:

- The *Tile1* segment, for execution time of components *G* and *E*. This execution time accounts between the stimulus event at input of *G* until the response event at output of *E.*
- The *Noc0* segment, for the transmission time of data over *Node0* on-chip network. From the stimulus event at the output of *E* to the response event at the on-chip/off-chip gateway of *Node0.*
- The off-chip segment, for the transmission time of data over the cluster off-chip network. From the stimulus event at the on-chip/off-chip gateway of *Node0* to the response event at the on-chip/off-chip gateway of *Node1*.
- The *Noc1* segment, for the transmission time of data over *Node1* on-chip network. From the stimulus event at the on-chip/off-chip gateway of *Node0* to the response event at the input of component *F*.
- The *Tile1* segment, for the execution time of components *F* and *I*. This execution time accounts between the stimulus event at input event of *F* until the response event at output of *I*.

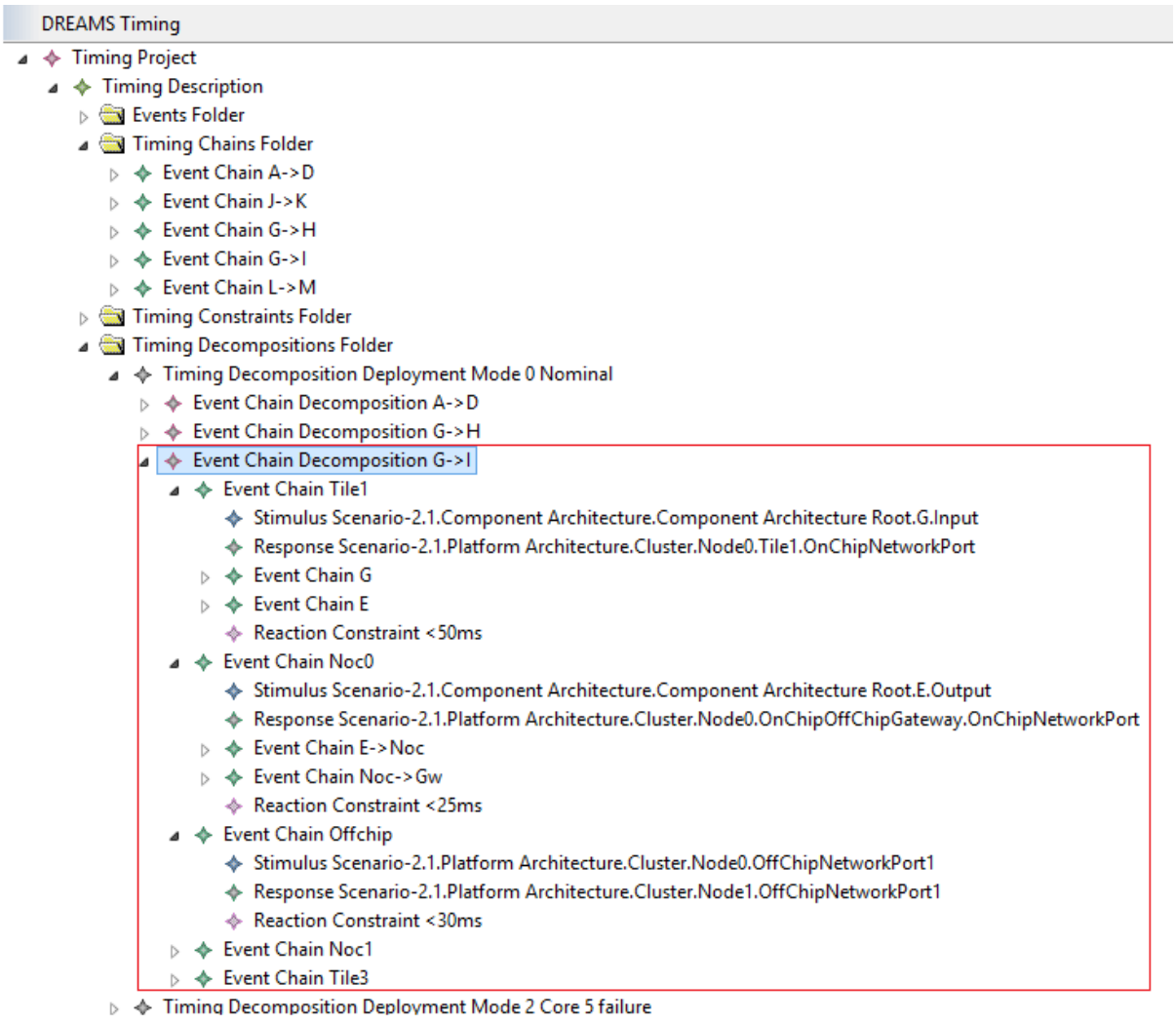


Figure 3.4.4: Timing decomposition

The decomposition allows the designer to split an end-to-end latency constraint into sub-constraints for the different scheduling domains:

- tasks scheduling,
- on-chip communication scheduling,
- off-chip communication scheduling.

This way, the overall scheduling problem is decomposed into sub-problems that can be resolved independently, see [9].

3.4.3 Reference Documentation

The temporal viewpoint consists of the timing metamodel described in this section. It is based on the Timmo-2-use timing metamodel [24] adapted for the DREAMS specificities [2].

Name	**DREAMS Timing Metamodel**
Description	The goal of the DREAMS timing metamodel 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`
Dependencies	`org.fortiss.af3.component` `org.fortiss.tooling.kernel`

Table 3.4.1 : DREAMS Timing Metamodel

The following classes are defined to describe timing information (see Figure 3.4.5 for the class diagram of the metamodel):

- `TimingProject`
  - Root element for timing information metamodel that contains `TimingDescription`s.
- `TimingDescription`
  - Element containing the collections of timing descriptions, namely events, timing chains, timing constraints and timing chains decompositions.
- `EventsFolder`
  - Element containing the collection of `Event` proxy elements which can reference the following elements:
    - A `Port` of a `Component` in the logical `ComponentArchitecture` (`InputEvent`s and `OutputEvent`s, see below).
    - A `Component` of the logical `ComponentArchitecture` (`EventTrigger`, see below).
    - An `OffChipNetworkRouterPort`, or an `OnChipNetworkRouterPort` allocated in an `Deployment` to the `PlatformArchitecture`.
- `TimingConstraintsFolder`
  - Element containing the collection of `TimingConstraint`.
- `TimingChainsFolder`
  - Element containing the collection of `EventChain`.
- `TimingDecomposition`
  - Element containing the collection of `EventChainDecomposition`.
  - **Attributes**
    - `deployment`: references the `Deployment` for which the decomposition is achieved.
    - `decompositions`: collection of `EventChainDecomposition`.

- `EventChain`
  - Element describing an event flow chain between entities.
  - **Attributes**
    - `stimulus`: originating `Event`, start of the chain.
    - `response`: terminating `Event`, end of the chain.
    - `segments`: collection of sub `EventChain` allowing the decomposition of the `EventChain`.
    - `Constraint`: `TimingConstraint` associated to the `EventChain`.
- `EventChainDecomposition`
  - Element describing the decomposition of an `EventChain` for a given `Deployment`.
  - **Attributes**
    - `reference`: references the `EventChain` that is decomposed.
    - `eventChainsSegments`: collection of `EventChain` used for the decomposition.

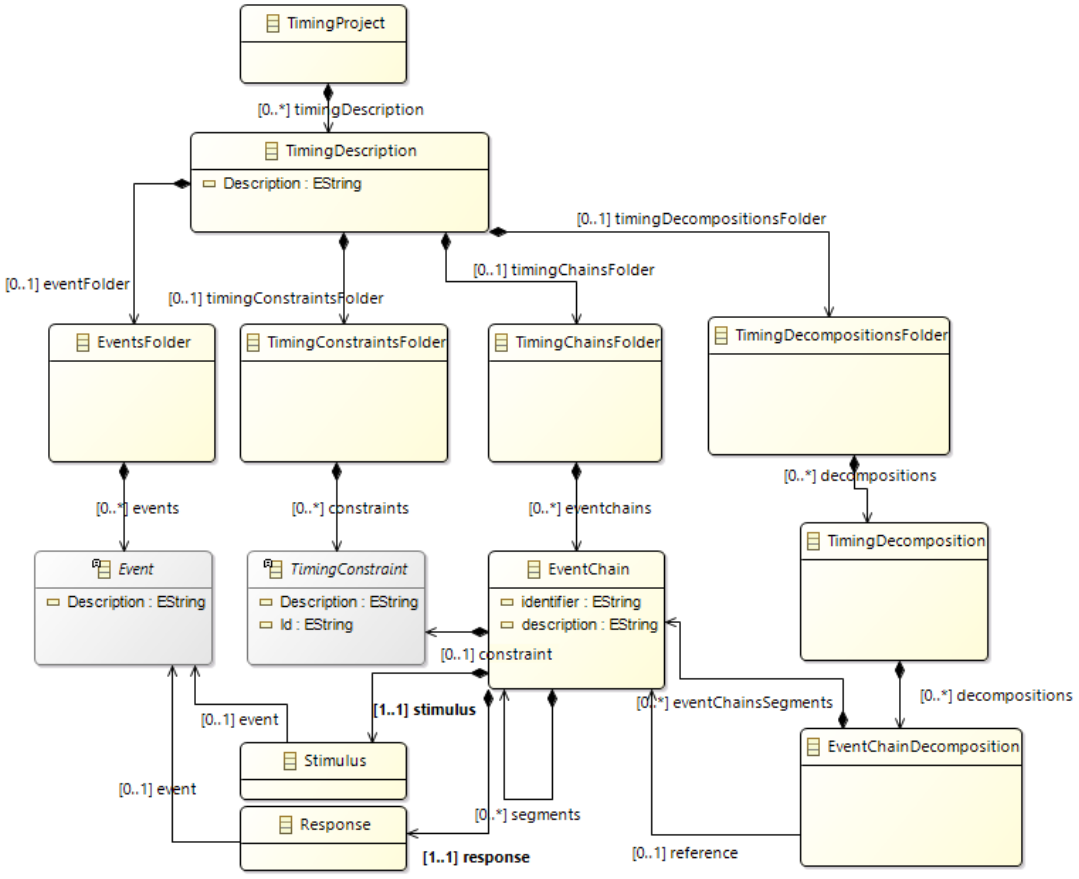


Figure 3.4.5: Timing Metamodel (class diagram of package `eu.dreamsproject.rtaw.timing.model`)

3.4.3.1 Timing Constraints

Timing constraints are almost not modified, please refer to [1].

Only to class `PeriodicConstraint`, a new attribute `offset` has been added which is the effective release of a periodic event with respect to a reference clock (value in seconds).

3.4.3.2 Events

New Events are added in order to link timing model elements to on-chip and off-chip communication elements defined in the `PlatformArchitecture`.

- `OnChipNetworkEvent`:
  - This links the timing model elements to an `OnChipNetworkPort`.
  - **Attributes:**
    - `ref`: References the `OnChipNetworkPortAnnotation` from the platform architecture metamodel.
- `OffChipNetworkEvent`:
  - This links the timing model elements to an `OffChipNetworkPort`.
  - **Attributes:**
    - `ref`: References the `OffChipNetworkPortAnnotation` from the platform architecture metamodel.
- `InputEvent`, `OuputEvent`, `EventTrigger` : please refer to [2].

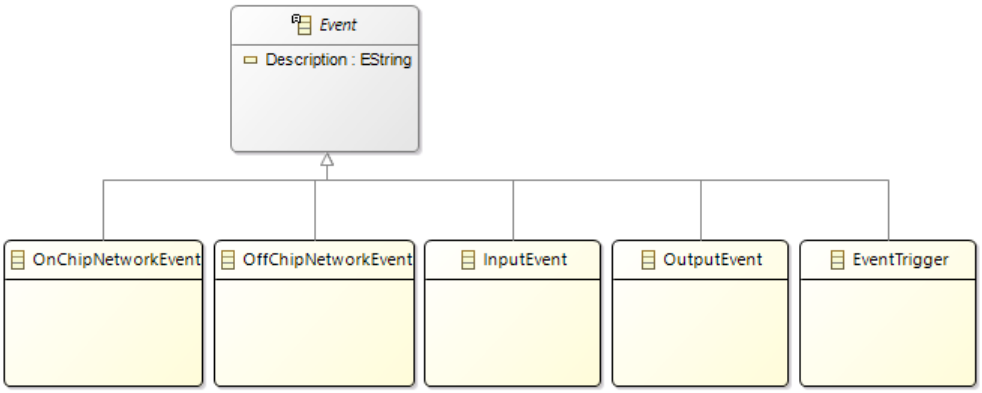


Figure 3.4.6 Timing model Event classes

3.4.3.3 Interface to other Metamodels

The DREAMS timing metamodel contains new references to the platform architecture metamodel and the deployment metamodel, the timing metamodel now references in particular:

- `OnChipNetworkPort` (via `OnChipNetworkPortAnnotation`)
- `OffChipNetworkPort` (via `OffChipNetworkPortAnnotation`)
- `Deployment` (via `DeploymentReference`)

4 Service Configuration Viewpoint

The Service Configuration Viewpoint contributes metamodels that provide additional implementation-specific parameters on top of the resource utilization metamodel.

The rationale for the PSM's two-level architecture (see Figure 1.2.1) is:

- To simplify the resource utilization metamodel by avoiding building block specific parameters. As a result, the unified schedule metamodel is applicable to all types of resources for which offline schedules are required.
- To simplify the implementation of code/configuration templates for configuration generators based on the framework developed in [10][11] since default values for implementation specific parameters can be consistently injected to the resulting configuration model during the model-transformation.
- A separate configuration model makes the configuration generation process more robust since all configuration information is available at the model level where it can be verified more easily based on the coherent interface provided by the metamodel, as opposed to validating the ultimate configuration files required to configure the respective platform services. Checks are either automated, or performed by an expert in the provided PSM model editors. Furthermore, this intermediate level helps mitigate changes in the configuration file format and enables the system integrator to perform manual adjustments using the respective configuration model editor.

In this chapter, the following metamodels will be described

- Configuration infrastructure: Basic metamodels that define the interface to the configuration generation framework described in [10].
- Configuration metamodels for the DREAMS virtual platform. The configuration metamodel for the virtual platform follows the same structure as the implementation of the simulator (see [17], Chapter 2), and hence provides a dedicated metamodel for the following simulator modules:
    - Simulated on-chip network interface (see Section 4.3)
    - Simulated off-chip network components (see Section 4.4)
- Configuration metamodels for the DREAMS physical platform
    - Physical on-chip network interface (see Section 4.2)
    - XtratuM hypervisor (see Section 4.5)
    - TTEthernet network (see Section 4.6)

We will not present examples in this section, since the configuration metamodels are typically not created manually, but are the result of a model-to-model transformation that converts the information contained in the resource utilization metamodel (see Section 3) into the corresponding service-/device-specific format.

The implementation of the service configuration metamodels described in Sections 4.1 to 4.4 is bundled with AutoFOCUS3/DREAMS (see [2], Section 3.2.2 for instructions on how to obtain the package). The configuration metamodels for the XtratuM hypervisor and TTEthernet are bundled with the respective installation package.

4.1 Configuration Infrastructure Metamodel

4.1.1 Overview

The metamodel described in this section is the basis for the configuration metamodels entirely developed in the scope of DREAMS (see Sections 4.2, 4.3, and 4.4). The other configuration metamodels are extensions of existing formats and hence are only loosely integrated into this infrastructure (see Sections 4.5 and 4.6).

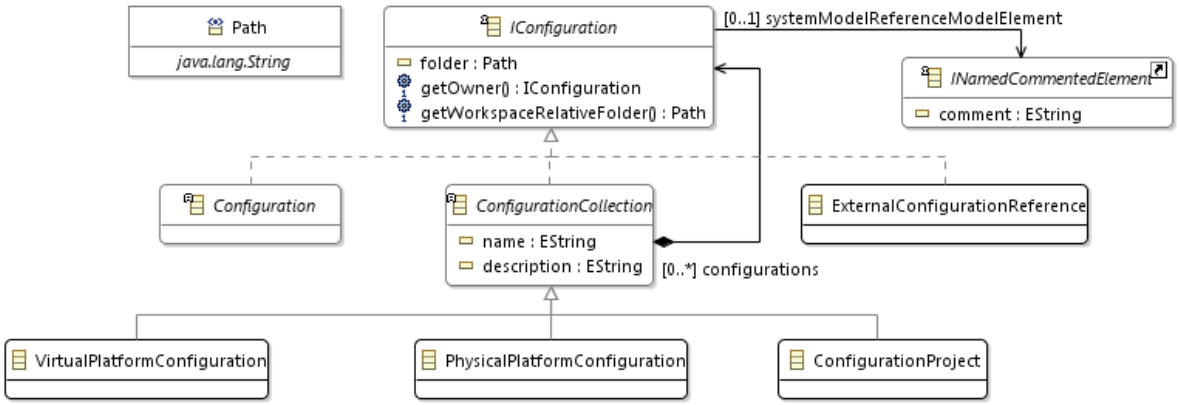


Figure 4.1.1: Service Configuration Metamodel.

The configuration metamodel is intended to support the generation of configuration files for component of the DREAMS platform. Hence, the `IConfiguration` interface provides a link to a `systemModelReferenceElement` and defines the (relative) folder where the configuration should be generated. Since for a platform resource to be configured, potentially more than one configuration file is required, the definition of the file name(s) is performed in the corresponding configuration generators code template (see [10]). The `Configuration` class is the base class for the concrete configurations defined in the next sections. `ConfigurationCollection`s are used to cluster sets of configurations, e.g., the configuration of the DREAMS virtual and physical platform (`VirtualPlatformConfiguration`, `PhysicalPlatformConfiguration`), or the root model element `ConfiguationProject` itself. The `ExternalConfigurationReference` can be used to link an external configuration (e.g., for XtratuM or TTEthernet). That way, the resulting `ConfiguationProject` clusters all configuration artefacts for a particular deployment.

4.1.2 Reference Documentation

Name	**Configuration Metamodel**
Description	Configuration Infrastructure
Ecore file	`psm.ecore`
Plugin	`eu.dreamsproject.psm`
Packages	`eu.dreamsproject.psm.configuration`
Dependencies	`org.fortiss.tooling.kernel`

Table 4.1.1: DREAMS Platform-Specific Metamodel (configuration package).

The configuration infrastructure metamodel consists of the `org.fortiss.af3.psm.configuration` package, which is composed of the classes presented below (see Figure 4.1.1):

- `Path`:
  - Representation of file paths.
- `IConfiguration`:
  - Common interface of all configuration metamodels (including `ConfigurationCollections`).
  - **Attributes:**
    - `folder`: Folder for output generation.
    - `systemModelReferenceModelElement`: Reference to model in the system model whose configuration is described by this `IConfiguration`.
  - **Operations:**
    - `getOwner()`: Returns the `IConfiguration` owning this `IConfiguration`, or `null` if this `IConfiguration` is the top-level `ConfigurationProject`).
    - `getWorkspaceRelativeFolder()`: Returns the workspace relative path where this `IConfiguration` is generated to.
- `ConfigurationCollection`:
  - A collection of `IConfigurations`.
  - **Attributes:**
    - `configurations`: Sub-`IConfiguration`s
    - `name`: Human readable name
    - `description`: Human readable description
- `VirtualPlatformConfiguration`:
  - Configuration of DREAMS virtual platform.
- `PhysicalPlatformConfiguration`:
  - Configuration of DREAMS physical platform.
- `ExternalConfigurationReference`:
  - Reference to external configuration model.
- `ConfigurationProject`:
  - Root node of configuration model. It serves as entry point for the configuration generation for DREAMS systems [10][11].
- `Configuration`:
  - Abstract base class for configuration metamodels (whose instances serve as direct input to the generation of configuration files).

4.2 Physical On-Chip Network Interface Configuration Metamodel

4.2.1 Overview

This section describes a metamodel for the configuration of the LRS for the physical on-chip network interface (see [3] for the implementation of the on-chip LRS, and [11] for the specification of the configuration file format).

As depicted in Figure 4.2.1, the `OnChipNetworkConfiguration` collects information required to generate the configuration of all LRSs of a single physical `OnChipNetwork`. It comprises the following sub-elements:

- Two `OnChipSchedParams` objects to define the scheduling parameters for time-triggered and rate-constraint on-chip network traffic.
- One `OnChipNiLrsConfiguration` for each of the on-chip `NetworkInterface`s connected to the `OnChipNetwork` to be configured.

An `OnChipNiLrsConfiguration` references a `ResourceSchedule` (see Section 3.2) for the respective on-chip `NetworkInterface` and consists of the following sub-configurations:

- A `PortConfiguration` for all on-chip NI ports to be defined for the given `NetworkInterface`. It is a collection of `PortConfigurationItem`s that define low level parameters such as buffer size, queue length, etc. for the referenced `OnChipNetworkInterfacePort` (see Section 3.1).
- A `TimeTriggeredCommunicationSchedule` that defines the configuration of the schedule for the time-triggered traffic handled by the on-chip `NetworkInterface` referenced by the `OnChipNiLrsConfiguration`. In order to match the configuration interface of the physical on-chip NI LRS [3], time-triggered schedules are represented as a set of linked lists of `TimeTriggeredCommunicationScheduleEntry`s that provide operations to obtain the ID of the port to be scheduled, its index in the list, and its phase.
- `EventTriggeredCommunicationSchedule`s define static schedules for rate-constraint traffic as linked lists (see above). However, instead of defining schedules for network interface ports, the `EventTriggeredCommunicationScheduleEntry`s control the on-chip interleaver of the on-chip NI using operations, that control the admission of event-triggered traffic using guarding windows (`GW_OPEN`, `GW_CLOSE`) and by-pass windows for the TTE and DDR controllers (`BP_OPEN`, `BP_CLOSE`).

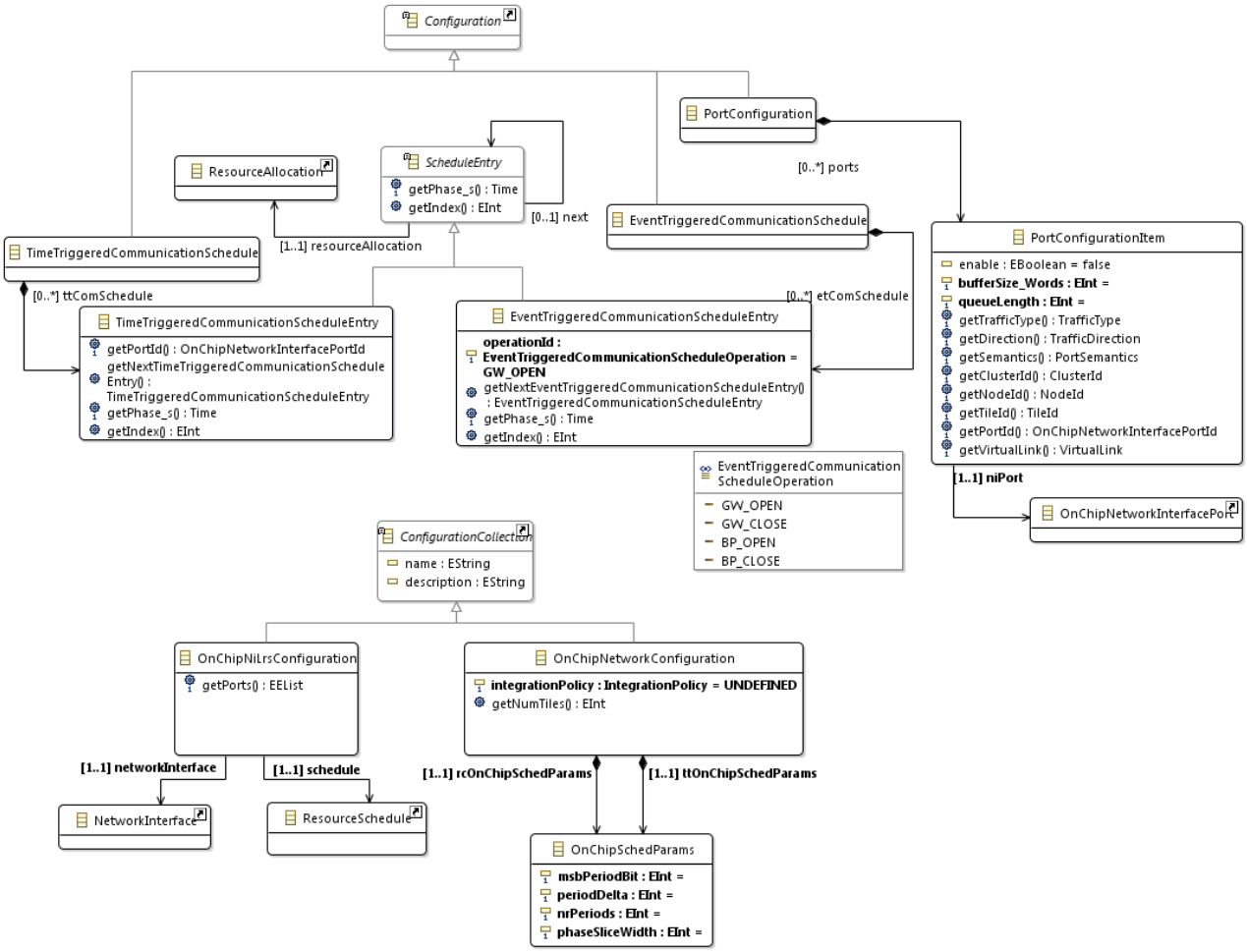


Figure 4.2.1: Physical On-Chip Network Interface Configuration Metamodel.

4.2.2 Reference Documentation

Name	Physical On-Chip Network Interface Configuration Metamodel
Description	Configuration of physical on-chip network-interface's LRS
Ecore file	onchip.phy.ecore
Plugin	eu.dreamsproject.psm
Packages	eu.dreamsproject.psm.onchip.phy
Dependencies	eu.dreamsproject.psm eu.dreamsproject.platform.dreams org.fortiss.af3.timing org.fortiss.af3.schedule org.fortiss.af3.deployment org.fortiss.af3.platform org.fortiss.tooling.base org.fortiss.tooling.kernel

Table 4.2.2: Physical On-Chip Network Interface Configuration Metamodel.

The classes defined in package `eu.dreamsproject.psm.onchip.phy` (see Figure 4.3.2) can be dived into three groups.

4.2.2.1 On-chip Network Configuration

The following classes have been defined to capture the overall configuration of an on-chip network of the physical platform:

- `OnChipNetworkConfiguration`:
  - Hardware configuration of all LRS of a single physical `OnChipNetwork`.
  - **Attributes:**
    - `integrationPolicy`: Integration policy applied to resolve media access conflicts, i.e., if a high-priority message arrives while a low-priority message is being processed. This attribute is of type `IntegrationPolicy` that has been introduced in Section 3.2.3.2.
    - `ttOnChipSchedParams`: Global scheduling parameters for TT messages.
    - `rcOnChipSchedParams`: Global scheduling parameters for RC messages.
  - **Operations:**
    - `getNumTiles()`: Returns the number of `Tiles` referenced by this `OnChipNetworkConfiguration`.
- `OnChipSchedParams`:
  - Global scheduling parameters.
  - **Attributes:**
    - `msbPeriodBit`: This constant refers to the bit in the time format of the global time base that is the period bit of the longest period. All shorter periods are aligned to the right of this bit. The constant `periodDelta` (see below) determines the distance in the time format of the global time base between the period bits of two neighbouring periods.
    - `periodDelta`: This constant specifies the distance (in number of bits) between the period bits of two successive periods.
    - `nrPeriods`: Number of periods that are supported by the NoC.
    - `phaseSliceWidth`: This constant identifies the number that corresponds to the width of a phase slice of a period's phase in the time format of the global time base.

- `OnChipNiLrsConfiguration`:
  - Configuration of the on-chip `NetworkInterface`.
  - **Attributes:**
    - `networkInterface`: The `NetworkInterface` whose configuration is described by this `OnChipNiLrsConfiguration`.
    - `schedule`: `ResourceSchedule` for the NI-LRS configured by this `OnChipNiLrsConfiguration`.
  - **Operations:**
    - `getPorts()`: Returns the `OnChipNetworkInterfacePort`s configured for the referenced `Tile`.

4.2.2.2 Port Configuration

Based on the following classes, the configuration of a single port allocated to an on-chip network interface's LRS is described:

- `PortConfiguration`:
  - Port configuration parameters of on-chip LRS.
  - **Attributes:**
    - `ports`: `PortConfigurationItem`s for the ports defined for the on-chip LRS configured by the `OnChipNetworkConfiguration` that owns this `PortConfiguration`.
- `PortConfigurationItem`:
  - Parameters of a single port of the on-chip LRS (entry of `PortConfiguration`).
  - **Attributes:**
    - `niPort`: `OnChipNetworkInterfacePort` allocated in the `Deployment` model for which both derived and additional configuration parameters are defined in this `PortConfigurationItem`.
    - `enable`: Flag whether this port is enabled or not.
    - `bufferSize_Words`: The size of the port buffer in words.
    - `queueLength`: Determines the queue length for event ports. In case of state ports, this field should be `1`.
  - **Operations:**
    - `getTrafficType()`: Returns the `TrafficType` for which the referenced `OnChipNetworkInterfacePort` is used.
    - `getDirection()`: Returns the `TrafficDirection` of the referenced `OnChipNetworkInterfacePort`.
    - `getSemantics()`: Returns the `PortSemantics` of the referenced `OnChipNetworkInterfacePort`.
    - `getClusterId()`: Returns the `ClusterId` of the `Cluster` that contains the referenced `OnChipNetworkInterfacePort`.
    - `getNodeId()`: Returns the `NodeId` of the `Node` that contains the referenced `OnChipNetworkInterfacePort`.
    - `getTileId()`: Returns the `TileId` of the `Tile` that contains the referenced `OnChipNetworkInterfacePort`.
    - `getPortId()`: Returns the `OnChipNetworkInterfacePortId` of the referenced `OnChipNetworkInterfacePort`.
    - `getVirtualLink()`: Returns the `VirtualLink` that the referenced `OnChipNetworkInterfacePort` corresponds to. For best-effort `OnChipNetworkInterfacePorts`, `null` is returned. This can be used to determine parameters such as the MINT or the virtual link ID associated to the referenced port.

4.2.2.3 Schedule Configuration

Based on the following classes, the time and event triggered schedule configuration for the on-chip network are described:

- `TimeTriggeredCommunicationSchedule`:
  - Cyclic lists of time-triggered message emission times.
  - **Attributes**:
    - `ttComSchedule`: List of `TimeTriggeredCommunicationScheduleEntry`s defining this `TimeTriggeredCommunicationSchedule`.
- `EventTriggeredCommunicationSchedule`:
  - Cyclic lists of gate operations.
  - **Attributes**:
    - `etComSchedule`: List of `EventTriggeredCommunicationScheduleEntry`s defining this `EventTriggeredCommunicationSchedule`.
- `ScheduleEntry`:
  - Base class for a single entry into a `TimeTriggeredCommunicationSchedule` or `EventTriggeredCommunicationSchedule`.
  - **Attributes:**
    - `resourceAllocation`: The `ResourceAllocation` that is contained in the schedule to be represented in the schedule configuration described by a sub-class of `ScheduleEntry`.
    - `next`: Reference to next `ScheduleEntry`.
- `TimeTriggeredCommunicationScheduleEntry`:
  - Schedule configuration for time-triggered traffic.
  - **Operations:**
    - `getPortId()`: Returns the `OnChipNetworkInterfacePortId` of the `OnChipNetworkInterfacePort` whose dequeuing into the NoC is described by this `TimeTriggeredScheduleEntry`.
    - `getNextTimeTriggeredCommunicationScheduleEntry()`: Returns the next `TimeTriggeredCommunicationScheduleEntry`.
    - `getPhase_s()`: Returns the phase (in seconds) at which the `OnChipNetworkInterfacePort` referenced by the `SchedulableEntity` that is referenced by the given `ResourceAllocation` is scheduled.
    - `getIndex()`: Returns the schedule entry's index (i.e., its position in the time-triggered schedule's entry list) that corresponds to the line number in the configuration file. In case the schedule entry is not contained in a `TimeTriggeredCommunicationSchedule`, `-1` is returned.
- `EventTriggeredCommunicationScheduleOperation`:
  - IDs of operations used by the ET-interleaver to eliminate the chance of temporal interferences for the time-triggered messages, by restricting the injection of event-triggered messages (e.g., rate-constraint or best-effort) and accesses to the TTE and DDR controller.
    - `GW_OPEN`: Opening the guarding window, in which no ET messages is allowed to be injected into the NoC.
    - `GW_CLOSE`: Closing the guarding window, which allows that the queued event-triggered messages be injected into the NoC.
    - `BP_OPEN`: Letting the accesses to the TTE and DDR controllers bypass the LRS.
    - `BP_CLOSE`: Prohibiting the accesses to the TTE and DDR controllers bypass the LRS.

- `EventTriggeredCommunicationScheduleEntry`:
  - Schedule configuration for event-triggered traffic.
  - **Attributes:**
    - `operationId`: Operation used by the ET-interleaver to eliminate the chance of temporal interferences for the time-triggered messages, by restricting the injection of event-triggered messages (e.g., rate-constraint or best-effort) and accesses to the TTE and DDR controller.
  - **Operations:**
    - `getNextEventTriggeredCommunicationScheduleEntry()`: Returns the next `EventTriggeredCommunicationScheduleEntry`.
    - `getPhase_s()`: Returns the phase of the given `EventTriggeredCommunicationScheduleEntry's ResourceAllocation`, or `null` if the `ResourceAllocation` does not reference a `WindowSchedulableEntity` (see Section 3.2.3.3). This method evaluates the `EventTriggeredCommunicationScheduleEntry#getOperationId()` in order to distinguish the start and end phase of bypass and guarding windows.
    - `getIndex()`: Returns the schedule entry's index (i.e., its position in the event-triggered schedule's entry list) that corresponds to the line number in the configuration file. In case the schedule entry is not contained in an `EventTriggeredCommunicationSchedule`, `-1` is returned.

4.3 Simulated On-Chip Network Interface Configuration Metamodel

4.3.1 Overview

The simulated on-chip network interface configuration metamodel abstracts the information required to configure the virtual resources of the on-chip network simulator (see [17][18] for the implementation and the configuration file format of the building block).

As shown in Figure 4.3.1, the metamodel provides two `ConfigurationCollection`s: the `OnChipNetworkConfiguration` that captures the information for each DREAMS `OnChipNetwork` that is present in a system, and the `OnChipNiLrsConfiguration` that defines the configuration of a single LRS. Each `OnChipNetworkConfiguration` contains the `OnChipNiLrsConfiguration`s of its connected network interfaces that are associated with the corresponding resource in the platform model and a `ResourceSchedule` (see Section 3.2) for this resource.

The behaviour of the on-chip network developed in DREAMS is defined by the configuration for periodic and rate-constraint virtual links (see [1]). Hence, the `PortConfiguration`, `EgressBridgingUnitTimeTriggeredSchedule`, and the `SerializationUnitSchedule` allow to define the required parts of the network interfaces (see [18], Section 6.3). The `VirtualLinkConfiguration` allows to obtain the required information about the `VirtualLink`s (see Section 3.1) that traverse the on-chip network. The corresponding parameters are defined in [18], Section 6.3, where also a mapping between parameters in the metamodel and the configuration parameters is defined.

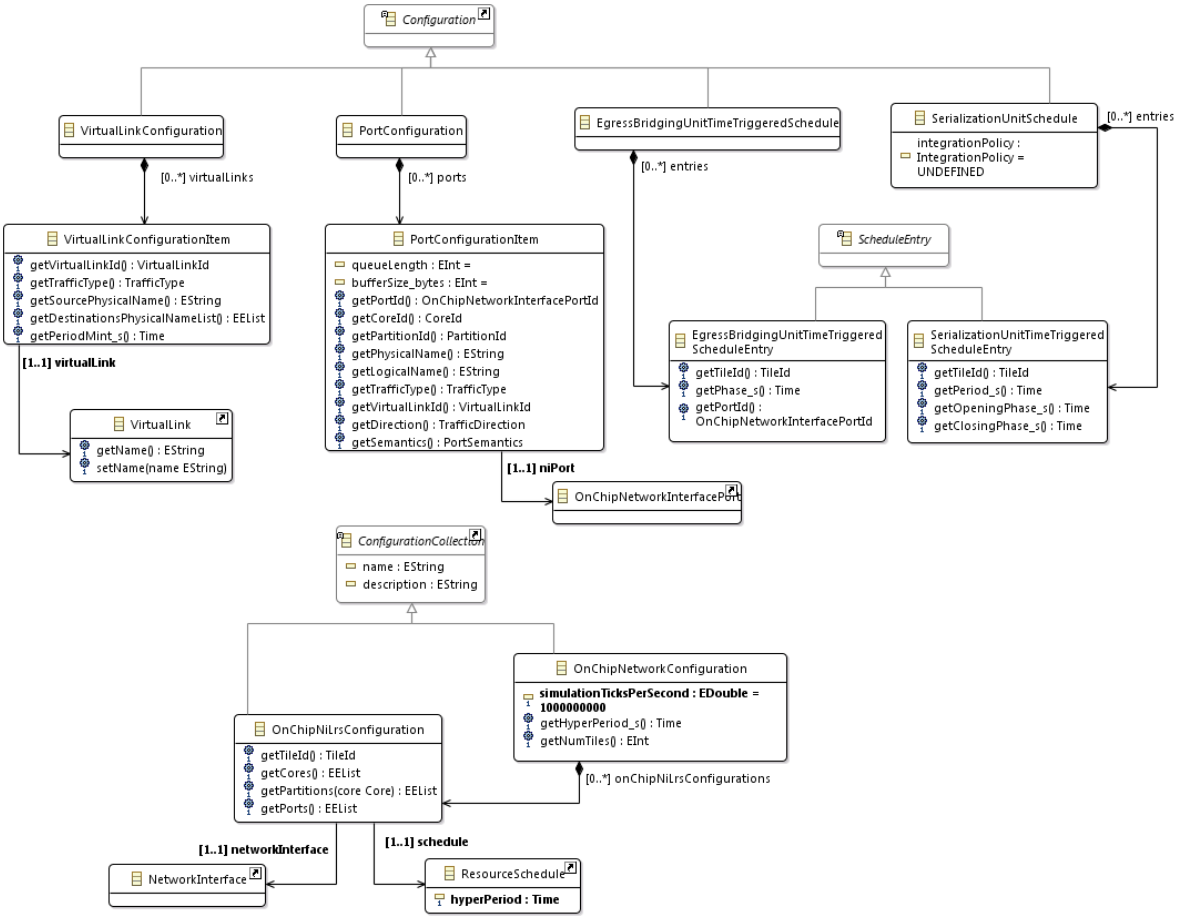


Figure 4.3.1: Simulated On-Chip Network Interface Configuration Metamodel.

4.3.2 Reference Documentation

Name	Simulated On-Chip Network Interface Configuration Metamodel
Description	Configuration of simulated on-chip network-interface's LRS
Ecore file	`onchip.sim.ecore`
Plugin	`eu.dreamsproject.psm`
Packages	`eu.dreamsproject.psm.onchip.sim`

Dependencies	`eu.dreamsproject.psm`
	`eu.dreamsproject.platform.dreams`
	`org.fortiss.af3.timing`
	`org.fortiss.af3.schedule`
	`org.fortiss.af3.deployment`
	`org.fortiss.af3.platform`
	`org.fortiss.tooling.base`
	`org.fortiss.tooling.kernel`

Table 4.3.1: Simulated On-Chip Network Interface Configuration Metamodel.

The classes defined in package `eu.dreamsproject.psm.onchip.sim` (see Figure 4.3.1) can be dived into three groups.

4.3.2.1 On-chip Network Configuration

The following classes have been defined to capture the overall configuration of an on-chip network of the virtual platform:

- `OnChipNetworkConfiguration`:
  - Configuration of all LRS of a single `OnChipNetwork`.
  - **Attributes:**
    - `simulationTicksPerSecond`: Conversion factor to convert 1 s (real-time) into simulation ticks. The current default value of 1000000000 means that one simulation tick corresponds to 1 ns.
    - `onChipNiLrsConfigurations`: The configurations of the on-chip NIs connected to a single `OnChipNetwork`.
  - **Operations:**
    - `getHyperPeriod_s()`: Returns the hyperperiod of all NI-LRS schedules (in seconds).
    - `getNumTiles()`: Returns the number of Tiles referenced by this `OnChipNetworkConfiguration`.
- `OnChipNiLrsConfiguration`:
  - Configuration of the referenced `NetworkInterface`.
  - **Attributes:**
    - `schedule`: `ResourceSchedule` for the NI-LRS configured by this `OnChipNiLrsConfiguration`.
    - `networkInterface`: The `NetworkInterface` whose configuration is described by this `OnChipNiLrsConfiguration`.
  - **Operations:**
    - `getTileId()`: Returns the `TileId` of the `Tile` that owns the referenced `NetworkInterface`.
    - `getCores()`: Returns the `Cores` contained by the referenced `Tile`.
    - `getPartitions()`: Returns the Partitions configured for the given `Core` that is contained by the referenced `Tile`.
    - `getPorts()`: Returns the `OnChipNetworkInterfacePorts` configured for the referenced `Tile`.

4.3.2.2 Port Configuration

Based on the following classes, the configuration of a single port allocated to an on-chip network interface's LRS is described:

- `PortConfiguration`:
  - Port configuration parameters of on-chip LRS.
  - **Attributes:**
    - `ports`: `PortConfigurationItem`s for the ports defined for the on-chip LRS configured by the `OnChipNetworkConfiguration` that owns this `PortConfiguration`.
- `PortConfigurationItem`:
  - Parameters of a single port of the on-chip LRS (entry of `PortConfiguration`).
  - **Attributes**
    - `queueLength`: Number of frames that can be queued.
    - `buffersize`: Port's buffer size (in bytes).
    - `niPort`: `OnChipNetworkInterfacePort` allocated in the `Deployment` model for which both derived and additional configuration parameters are defined in this `PortConfigurationItem`.

- **Operations:**
  - `getPortId()`: Returns the `OnChipNetworkInterfacePortId` of the referenced `OnChipNetworkInterfacePort`.
  - `getCoreId()`: Returns the `CoreId` of the `Core` that has the right to read from / write into the referenced `OnChipNetworkInterfacePort`.
  - `getPartitionId()`: Returns the `PartitionId` of the `Partition` that as the right to read from / write into the referenced `OnChipNetworkInterfacePort`.
  - `getPhysicalName()`: Returns the referenced `OnChipNetworkInterfacePort`'s physical name as a `String`.
  - `getLogicalName()`: Returns the referenced `OnChipNetworkInterfacePort`'s logical name as a String.
  - `getTrafficType()`: Returns the `TrafficType` for which the referenced `OnChipNetworkInterfacePort` is used.
  - `getVirtualLinkId()`: Returns the `VirtualLinkId` of the `VirtualLink` which the referenced `OnChipNetworkInterfacePort` corresponds to. For best-effort `OnChipNetworkInterfacePorts`, `-1` is returned.
  - `getDirection()`: Returns the `TrafficDirection` of the referenced `OnChipNetworkInterfacePort`.
  - `getSemantics()`: Returns the `PortSemantics` of the referenced `OnChipNetworkInterfacePort`.

4.3.2.3 Virtual Link Configuration

The following classes describe the virtual links to be simulated:

- `VirtualLinkConfiguration`:
  - Virtual links relevant for configuration of the given on-chip LRS.
  - **Attributes**:
    - `virtualLinks`: The `VirtualLinkConfigurationItem`s for the `VirtualLink`s configured by this `VirtualLinkConfiguration`.
- `VirtualLinkConfigurationItem`:
  - Configuration of a single virtual link (for configuration of on-chip LRS)
  - **Attributes**:
    - `virtualLink`: The `VirtualLink` whose configuration is described in this `VirtualLinkConfigurationItem`.
  - **Operations:**
    - `getVirtualLinkId()`: Returns the `VirtualLinkId` of the `VirtualLink` referenced by the given `VirtualLinkConfigurationItem`.
    - `getTrafficType()`: Returns the `TrafficType` of the `VirtualLink` referenced by the given `VirtualLinkConfigurationItem`.
    - `getSourcePhysicalName()`: Returns the physical name of the sender `OnChipNetworkInterfacePort` of the `VirtualLink` referenced by the given `VirtualLinkConfigurationItem` as a `String`.
    - `getDestinationsPhysicalNameList()`: Returns the physical names of the receiver `OnChipNetworkInterfacePorts` of the `VirtualLink` referenced by the given `VirtualLinkConfigurationItem` as an `EList` of `String`.
    - `getPeriodMint_s()`: Returns the period of time-triggered `VirtualLinks` / the minimum inter-arrival time (MINT) of rate-constraint

`VirtualLinks` of the `VirtualLink` referenced by the given `VirtualLinkConfigurationItem` in seconds.

4.3.2.4 Schedule Configuration

Based on the following classes, the time and event triggered schedule configuration for the on-chip network are described

- `EgressBridgingUnitTimeTriggeredSchedule`:
  - Configuration of time-triggered schedules of on-chip LRS' Egress Bridging Units (EBUs).
  - **Attributes**:
    - `entries`: `EgressBridgingUnitTimeTriggeredScheduleEntry`s defining this `EgressBridgingUnitTimeTriggeredSchedule`.
- `ScheduleEntry`:
  - Base class for EBU and Serialization Unit schedules.
  - **Attributes**:
    - `resourceAllocation`: The `ResourceAllocation` that is contained in the schedule to be represented in the schedule configuration described by a sub-class of `ScheduleEntry`.
- `EgressBridgingUnitTimeTriggeredScheduleEntry`:
  - A single entry of a time-triggered schedule for the egress bridging unit.
  - **Operations:**
    - `getTileId()`: Returns the `TileId` of the `Tile` that contains the `OnChipNetworkInterface` owning the `OnChipNetworkInterfacePort` whose access from an EBU is described by this `EgressBridgingUnitTimeTriggeredScheduleEntry`.
    - `getPhase_s()`: Returns the phase (in seconds) at which the access of the EBU to the `OnChipNetworkInterfacePort` referenced by the `SchedulableEntity` that is referenced by the given `ResourceAllocation` is scheduled.
    - `getPortId()`: Returns the `OnChipNetworkInterfacePortId` of the `OnChipNetworkInterfacePort` whose access from an EBU is described by this `EgressBridgingUnitTimeTriggeredScheduleEntry`.
- `SerializationUnitSchedule`:
  - Configuration of time-triggered schedule of on-chip LRS' Serialization Units.
  - **Attributes:**
    - `integrationPolicy`: Integration policy applied to resolve media access conflicts, i.e., if a high-priority message arrives while a low-priority message is being processed. This attribute is of type `IntegrationPolicy` that has been introduced in Section 3.2.3.2.
    - `entries`: `SerializationUnitTimeTriggeredScheduleEntry`s defining this `SerializationUnitTimeTriggeredSchedule`.
- `SerializationUnitTimeTriggeredScheduleEntry`:
  - A single entry of a time-triggered schedule for the serialization unit.
  - **Operations:**
    - `getTileId()`: Returns the `TileId` of the Tile to whose Serialization Unit schedule this `SerializationUnitTimeTriggeredScheduleEntry` belongs.
    - `getPeriod_s()`: Returns the period of the guarding window.
    - `getOpeningPhase_s()`: Returns the opening phase of the guarding window (only considered if the corresponding

`OnChipNiLrsConfiguration#mediaAccessConflictResolutionPolicy` is set to `TIMELY_BLOCK`.

- `getClosingPhase_s()`: Returns the closing phase of the guarding window (only considered if the corresponding `OnChipNiLrsConfiguration#mediaAccessConflictResolutionPolicy` is set to `TIMELY_BLOCK`.

4.4 Simulated Off-Chip Network Components Configuration Metamodel

4.4.1 Overview

The simulated off-chip network components configuration metamodel abstracts the configuration parameters for the simulated off-chip network (see [17], Chapter 2, and [18], Section 6.2 for the implementation and the configuration file format of the building block). As illustrated in Figure 4.4.1, the `OffChipNetworkConfiguration` is a `ConfigurationCollection` that clusters the following configurations for the components of the off-chip network simulator:

- `BlackboxNodeConfiguration`s: Nodes that do not have an on-chip network, or whose network is not simulated.
- `SwitchConfiguration`s: Switches of the off-chip network
- `GatewayConfiguration`s: On-/Off-Chip network switches

Each of the above configurations is populated with the respective sub-class of `ConfigurationItem` that represents a single entry of the respective configuration file (`BlackBoxNodeConfigurationItem`, etc.). A `ConfigurationItem` establishes the link with the resource utilization metamodel that hosts the schedule used as a basis for the configuration of the off-chip network simulation by means of a reference to the respective `VirtualLink` (see Section 3.1.1) a `ResourceAllocation` (see Section 3.2).

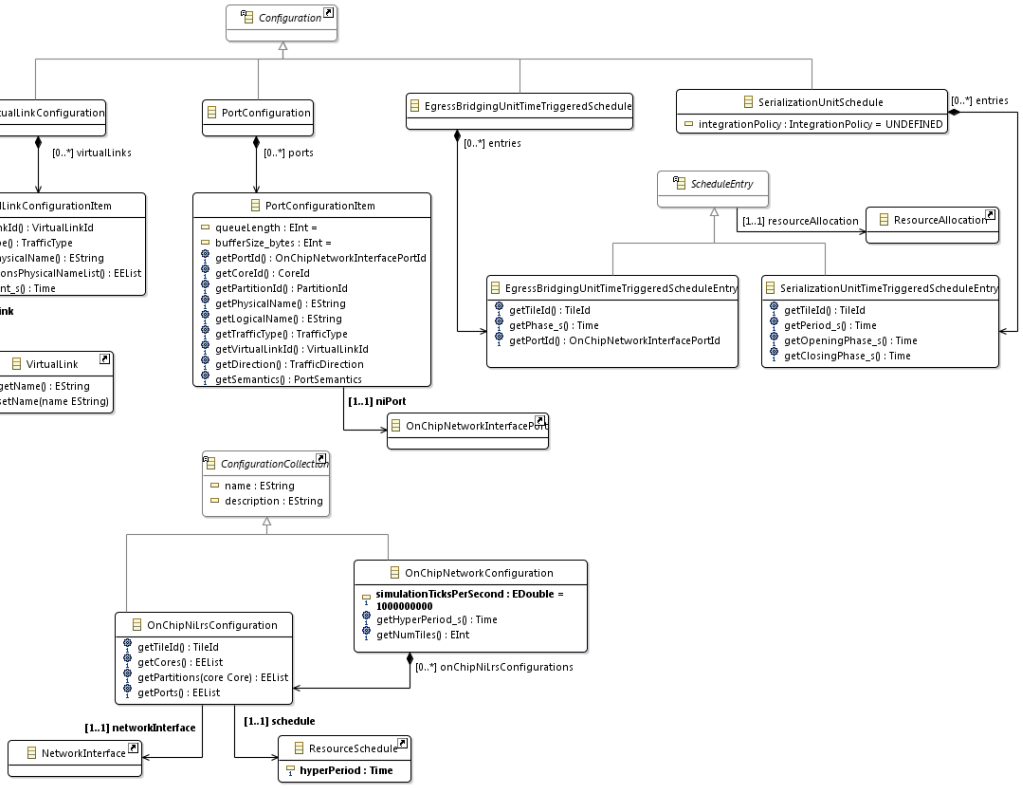


Figure 4.4.1: Simulated Off-Chip Network Interface Configuration Metamodel.

Figure 4.4.2 depicts the simulation configuration metamodel that contains additional information to be provided to the simulator, such as the set of messages to be simulated (cf. `MessageConfiguration`), and their injection times (cf. `TraceConfiguration`) [18].

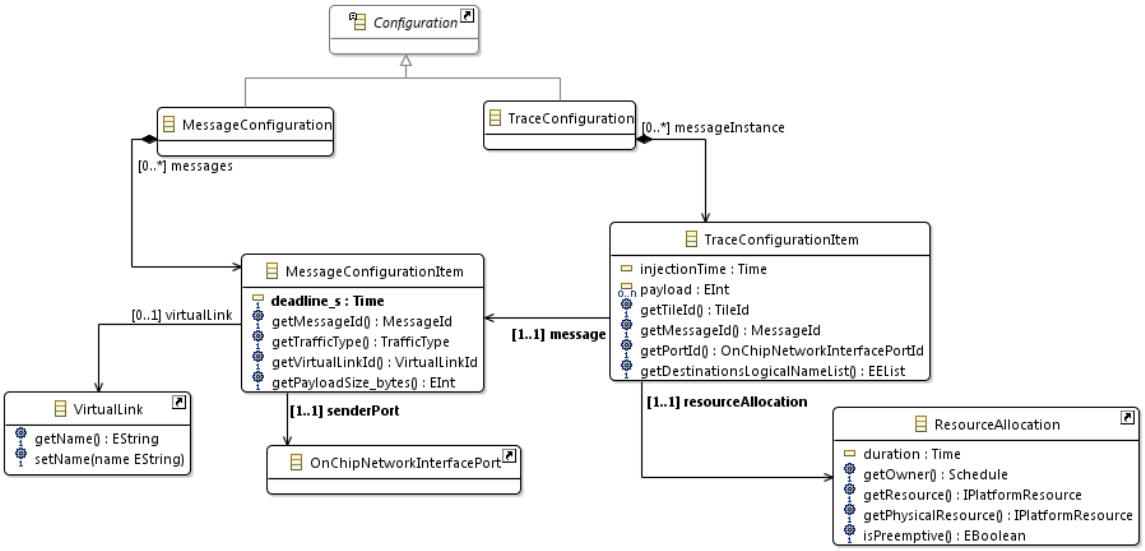


Figure 4.4.2: Simulation Configuration Metamodel.

4.4.2 Reference Documentation

4.4.2.1 Package eu.dreamsproject.psm.offchip.sim

Name	**Simulated Off-Chip Network Components Configuration Metamodel**
Description	Configuration of simulated off-chip network components.
Ecore file	`offchip.sim.ecore`
Plugin	`eu.dreamsproject.psm`
Packages	`eu.dreamsproject.psm.offchip.sim`

Dependencies	`eu.dreamsproject.psm`
	`org.fortiss.af3.timing`
	`org.fortiss.af3.schedule`
	`org.fortiss.af3.deployment`
	`org.fortiss.tooling.base`
	`org.fortiss.tooling.kernel`

Table 4.4.1: Simulated Off-Chip Network Interface Configuration Metamodel.

The `eu.dreamsproject.psm.offchip.sim` metamodel contains following classes (see Figure 4.4.1):

4.4.2.1.1 Off-chip network configuration

This group contains the foundation for the configuration metamodel for the simulated off-chip network.

- `QueueId:`
  - Data type to represent buffer identifications (queue IDs).
- `OffChipNetworkConfiguration:`
  - Configuration of simulated off-chip network.

- `ConfigurationItem`:
  - This abstract class represents the common set of attributes associated to the configuration of the simulated platform.
  - **Attributes:**
    - `virtualLink`: `VirtualLink` defined in the `Deployment` model that is used to obtain the following configuration parameters: virtual link ID, traffic type, message size.
    - `resourceAllocation`: The `ResourceAllocation` contained in a `ResourceSchedule` that describes the temporal characteristics of by the configuration described in the concrete `ConfigurationItem` that have been determined by a schedule (phase / jitter / priority).
    - `queueId`: The buffer identification (QueueID).
    - `queueLength`: Number of frames that can be queued.
  - **Operations:**
    - `getVirtualLinkId()`: Returns the `VirtualLinkId` configured using this `ConfigurationItem`.
    - `getTrafficType()`: Returns the `ConfigurationItem`'s traffic type.
    - `getPayloadSize_bytes()`: For time-triggered virtual links, returns the period, for rate-constraint virtual links returns the MINT.
    - `getPeriodMint_s()`: For time-triggered virtual links, returns the period, for rate-constraint virtual links returns the MINT.

4.4.2.1.2 Off-chip network switch configuration

The following classes represent the configuration of off-chip network switches:

- `SwitchConfiguration`:
  - Configuration of simulated off-chip network switch.
  - **Attributes:**
    - `switchConfigurationItems`: List of `SwitchConfigurationItem`s contained by this `SwitchConfiguration` (one per `OffChipNetworkRouter` [2]).
- `SwitchConfigurationItem`:
  - Contribution to `SwitchConfiguration` for a single `VirtualLink`.
  - **Operations:**
    - `getPhaseJitter_s()`: For time-triggered virtual links, returns the phase. For rate-constraint virtual links returns the jitter. Both values are derived from the `ResourceAllocation` referenced by this `SwitchConfigurationItem`.
    - `getSenderPortId()`: Returns the `OffChipNetworkRouterPortId` of the sender port (i.e., the port via which the switch receives the message).
    - `getReceiverPortsIdList()`: Returns the `EList` of `OffChipNetworkRouterPortIds` of the receiver ports (i.e., the ports via which the switch forwards the message) of the referenced `VirtualLink`.

4.4.2.1.3 Black box node configuration

The following classes represent the configuration of black box nodes, i.e., nodes which do not contain an on-chip network, or for which the on-chip network is not simulated:

- `BlackBoxNodeConfiguration`:
  - This class represents the black box node configuration for the simulated platform.
  - **Attributes:**
    - `blackBoxNodeConfigurationItems`: List of `BlackBoxNodeConfigurationItem`s contained by this `BlackBoxNodeConfiguration` (one per `Node` for which the `OnChipNetwork` is not simulated [2]).
- `BlackBoxNodeConfigurationItem`:
  - Contribution to `BlackBoxNodeConfiguration` for a single `VirtualLink`.
  - **Operations:**
    - `getSendingStartTime_s()`: For time-triggered virtual links, it returns the phase at which the message is sent. For rate-constraint virtual links, it returns the sending time of the first message (the sending time of the subsequent messages is simulated according the `VirtualLink's` MINT. Here, the `ResourceAllocation` must be contained in the CPU schedule that contains the sender task.

4.4.2.1.4 On-chip/Off-chip Gateway configuration

The following classes represent the configuration of simulated on-chip/off-chip gateways:

- `GatewayConfiguration`:
  - This class represents the gateway configuration.
  - **Attributes:**
  - `gatewayConfigurationItems`: List of `GatewayConfigurationItem`s contained by this `GatewayConfiguration` (one per `OnChipOffChipGateway` [2]).
- `GatewayConfigurationItem`:
  - This abstract class represents all configuration items for gateways.
  - **Operations:**
    - `getTrafficDirection()`: Returns the direction of the message from the point-of-view of the gateway configured by this `GatewayConfigurationItem`.

- `PeriodicGatewayConfigurationItem`:
  - This class represents the gateway configuration for periodic messages.
  - **Operations:**
    - `getPhase_s()`: Returns the phase defined `ResourceAllocation` in the communication schedule for the gateway to which this `PeriodicGatewayConfigurationItem` belongs.
- `SporadicGatewayConfigurationItem`:
  - This class represents the gateway configuration for sporadic (rate-constraint) messages.
  - **Operations:**
    - `getPriority()`: Returns the priority as defined in the `ResourceAllocation` in the communication schedule for the gateway to which this `SporadicGatewayConfigurationItem` belongs.

4.4.2.2 Package eu.dreamsproject.psm.simulation

Name	**DREAMS Simulation Metamodel**
Description	Configuration of the simulation experiment
Ecore file	`simulation.ecore`
Plugin	`eu.dreamsproject.psm`
Packages	`eu.dreamsproject.psm.simulation`

Dependencies	`eu.dreamsproject.psm`
	`org.fortiss.af3.timing`
	`org.fortiss.af3.schedule`
	`org.fortiss.af3.deployment`
	`org.fortiss.af3.component`
	`org.fortiss.tooling.base`
	`org.fortiss.tooling.kernel`

Table 4.4.2: DREAMS Simulation Metamodel.

The metamodel consists of the package `org.fortiss.af3.psm.annotation.model` which contains following classes (see Figure 4.4.2):

- `MessageConfiguration`:
  - Catalogue of messages to be injected into the simulation.
  - **Attributes**:
    - `Messages`: Messages to be injected into the simulation.
- `MessageConfigurationItem`:
  - A single message to be injected into the simulation.
  - **Attributes:**
    - `virtualLink`: `VirtualLink` for which this message to be injected into the simulation is specified (i.e., for TT and RC messages). If this attribute is non-`null`, the `senderPort` field will be ignored (should not be set in this case).
    - `senderPort`: The `OnChipNetworkInterfacePort` from which a BE message is sent. This field is only respected if the `virtualLink` field is `null`.
    - `deadline_s`: Message deadline.
  - **Operations:**
    - `getMessageId`: Returns the `MessageId` of the message to be injected into the network.
    - `getTrafficType`: Returns the `TrafficType` of the message to be injected into the network.
    - `getVirtualLinkId`: In case of time-triggered and rate-constraint messages, returns the `VirtualLinkId`, `-1` otherwise.
    - `getPayloadSize_bytes`: Returns the referenced `VirtualLink's` payload size.
- `TraceConfiguration`:
  - Trace file that defines the sequence in which messages from the `MessageConfiguration` catalogue are injected into the system.
  - **Attributes:**
    - `messageInstance`: Instance of a message described by a `MessageConfigurationItem`.
- `TraceConfigurationItem`:

- Singe entry into the `TraceConfiguration` file, i.e., a single instance of a message described by a `MessageConfigurationItem`.
- **Attributes:**
  - `message`: `MessageConfigurationItem` classifying the message for which a single instance is described by this `TraceFileConfigurationItem`.
  - `ResourceAllocation`: in the `ResourceSchedule` that defines the injection time of the message.
  - `Receivers`: The logical `InputPort`s which receive a best-effort message, i.e. this field needs only be set for best-effort messages.
  - `payload`: Message payload as sequence of integers.
  - `injectionTime`: Message injection time (in seconds).
- **Operations:**
  - `getTileId`: Returns the `TileId` of the `Tile` from which the message is injected into the network.
  - `getMessageId`: The `MessageId` of the message to be injected into the network.
  - `getPortId`: The `OnChipNetworkInterfacePortId` of the `OnChipNetworkInterfacePort` from which the message is injected into the network.
  - `getDestinationsLogicalNamelist`: For best-effort messages, returns the `EList` of logical names of the receivers as a `String`. For time-triggered and rate-constraint message `null` is returned.

4.5 XtratuM Hypervisor Configuration

The XtratuM hypervisor is configured statically. The configuration is specified in an XML file where the main components of the system are defined [21]. The integration of the configuration file in the deployment phase of a system is depicted in Figure 4.5.1.

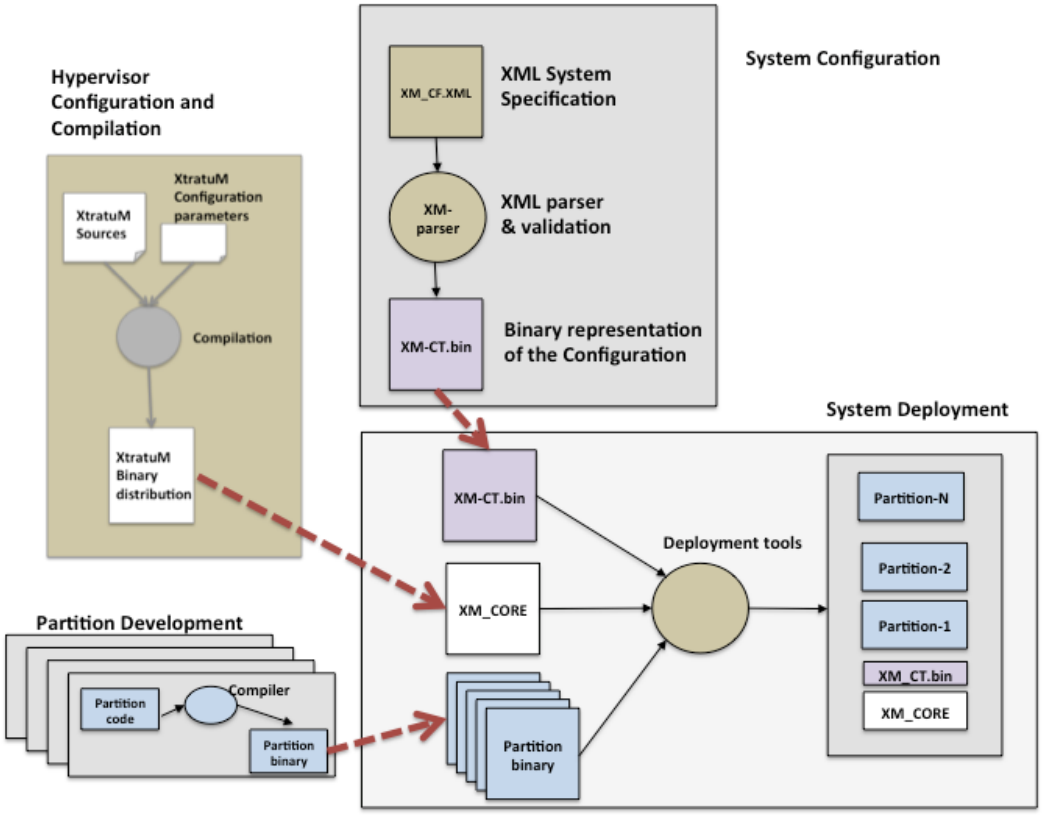


Figure 4.5.1: Configuration of XtratuM during system deployment phase.

Figure 4.5.1 shows the different processes involved in the system development:

- Hypervisor configuration and compilation: It permits to configure XtratuM for a specific processor and board and generates the hypervisor binary or core.
- Partition development: It involves the development of the partitions and the generation of the partition's binaries.
- System configuration: It takes the XML specification of the system and generates a binary representation of the configuration file.
- System deployment: it collects the result of the previous processes and generates the final system to be loaded in the target.

This section is related with the system configuration process and more specifically with the definition of the *XM:CF.XML* file that is the input of this process.

4.5.1 System Specification

The system specification file details the configuration parameters to be managed by the hypervisor to execute the system. The configuration file sections are:

- Hardware description: It details the hardware information such as the memory layout, processor information including the schedules for all processors and devices in the board.
- XM Hypervisor description: it defines the memory allocation and size of the hypervisor in the target.

- Resident Software description: it details the allocation and size of the system bootloader.
- Partition Table description: It provides the memory allocation and size of the partitions in the final system as well as the communication ports that partitions can use for sending or receiving messages.
- Channels description: it defines the links between the partitions' ports.

Figure 4.5.2 shows the scheme representation of the System Description component.

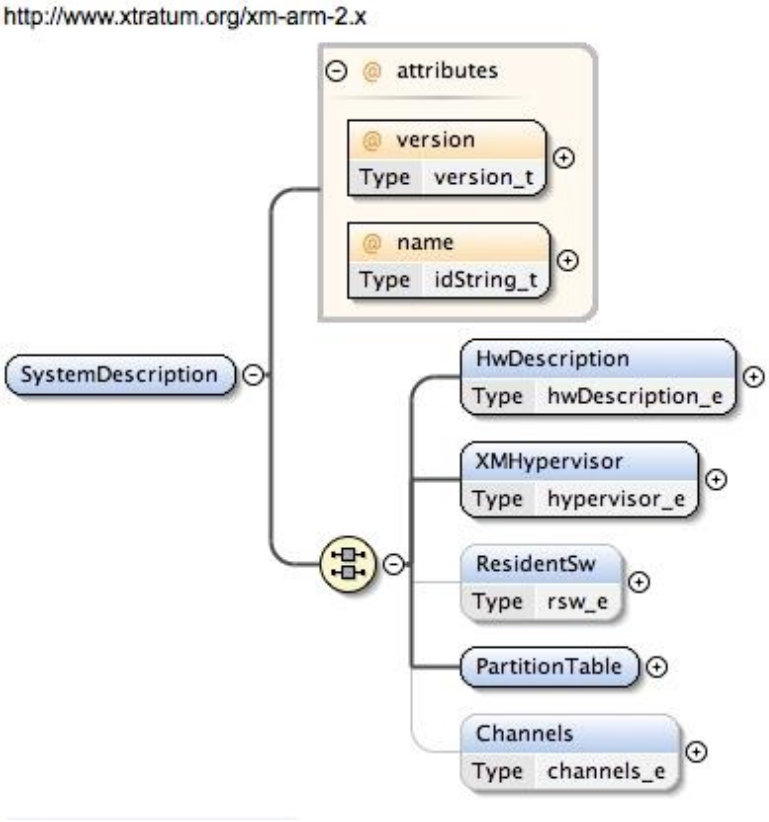


Figure 4.5.2: System Description Schema.

This scheme has been adapted from previous versions of the XtratuM hypervisor and contains specific DREAMS elements that are detailed in the next section.

4.5.2 DREAMS Contributions

The XM schema has been adapted to support the DREAMS boards and the specific hardware related to the STNoC and TTEthernet devices.

These new devices have been included in the schema definition as shown in Figure 4.5.3.

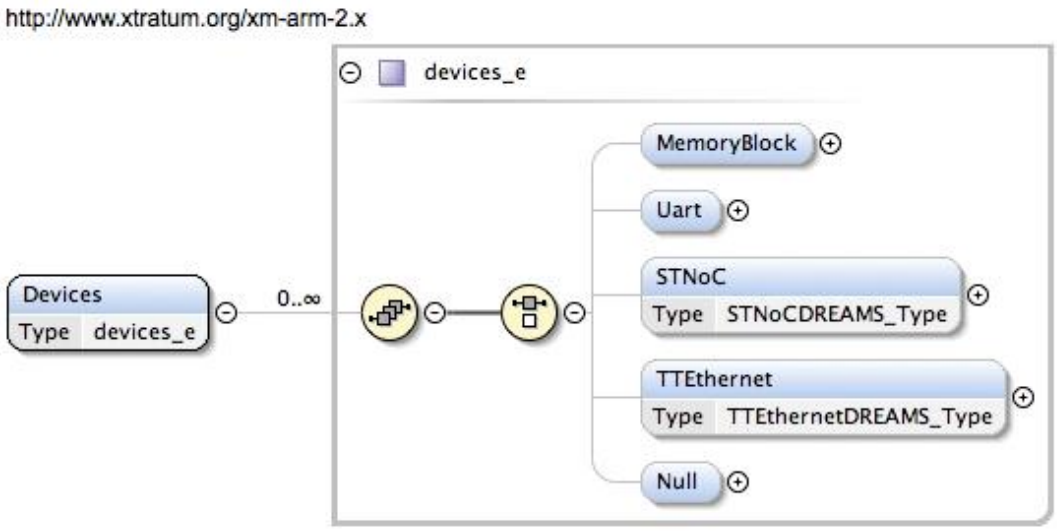


Figure 4.5.3: Devices schema extended with nodes for STNoC and TTEthernet configuration.

An STNoC device is defined as shown in Figure 4.5.4.

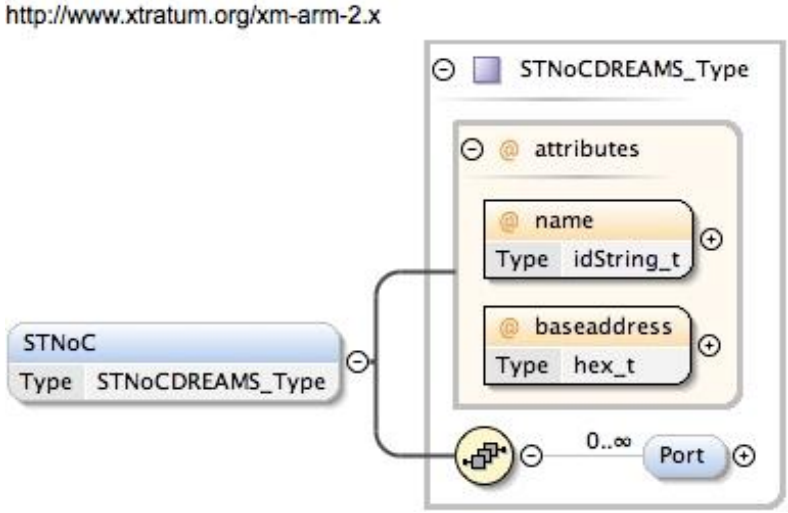


Figure 4.5.4: STNoC configuration schema.

An example of this device is depicted below:

```
<Devices>
     <STNoC name="STNoC_1" baseaddress="0x40000000" >
          <Port name="port_0" type="TRAFFIC_SAMPLING" direction="destination"
                offsetaddress="0x00000000" size="64B" />
          <Port name="port_1" type="TRAFFIC_QUEUING" direction="destination"
                offsetaddress="0x00010000" size="64B" />
          <Port name="port_2" type="TRAFFIC_SAMPLING" direction="destination"
                offsetaddress="0x00020000" size="64B" />
          <Port name="port_3" type="TRAFFIC_QUEUING"  direction="destination"
                offsetaddress="0x00030000" size="64B" />
          <Port name="port_4" type="TRAFFIC_SAMPLING" direction="source"
                offsetaddress="0x00040000" size="64B" />
          <Port name="port_5" type="TRAFFIC_QUEUING"  direction="source"
                offsetaddress="0x00050000" size="64B" />
          <Port name="port_6" type="TRAFFIC_SAMPLING" direction="source"
                offsetaddress="0x00060000" size="64B" />
          <Port name="port_7" type="TRAFFIC_QUEUING"  direction="source"
                offsetaddress="0x00070000" size="64B" />
     </STNoC>
</Devices>
```

Figure 4.5.5: XtratuM STNoC Configuration Example

TTEthernet devices are defined according the schema drawn in Figure 4.5.6.

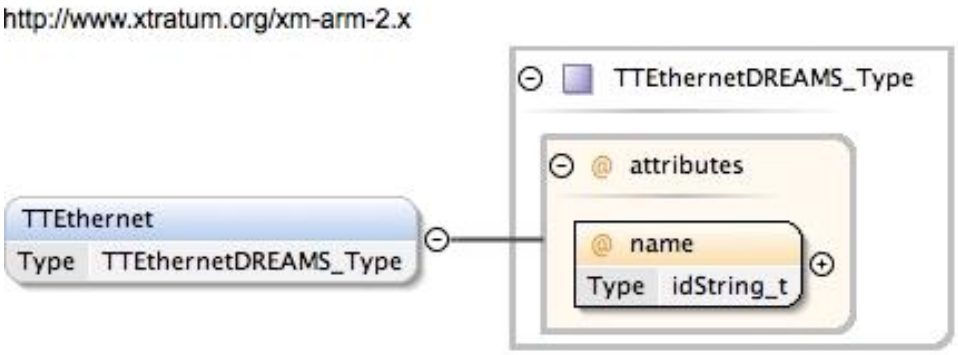


Figure 4.5.6: XTratuM TTEthernet configuration schema.

These devices enable communication among partitions allocated in different tiles. In order to allow the partitions to send or receive messages, the communication port concept at partition level has been extended to include in the Port Table the specific ports provided by these devices.

STNoC Ports now extend the Sampling Ports while TTEthernet ports are an extension of Queuing Ports.

The connection between ports is specified in the Channels element. A Sampling Channel can be a Source (partition in the same tile) or an STNoC port or a TTEthernet Port. Destination ports in a channels can be one or several ports in the same tile (Source) or connected through STNOC or TTEthernet Ports. Figure 4.5.7 shows the definition of Sampling Channels.

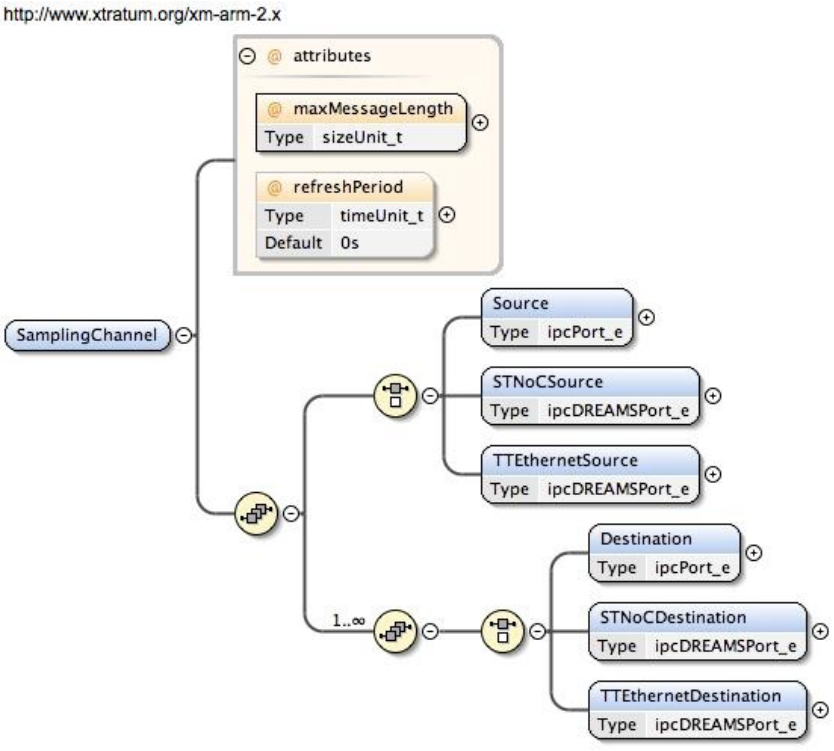


Figure 4.5.7: Sampling channel configuration schema.

The same approach has been used for the definition of Queuing Channels. Figure 4.5.8 provides the Queuing Channel element schema.

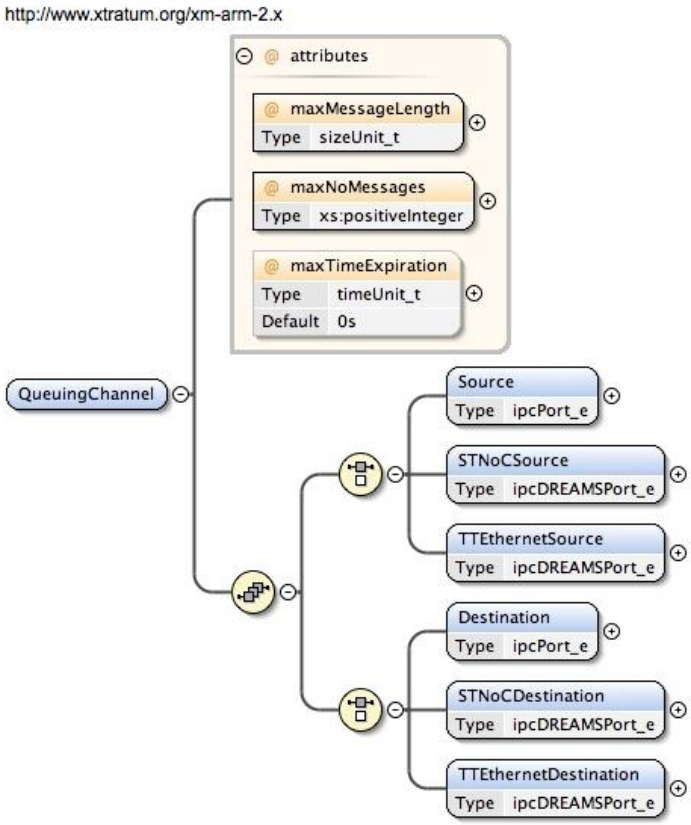


Figure 4.5.8: Queuing channel configuration schema.

4.5.3 Configuration File Binary Representation

As described previously, the system configuration process receives the XML description of a system and generates a binary representation of the configuration file (see Figure 4.5.9).

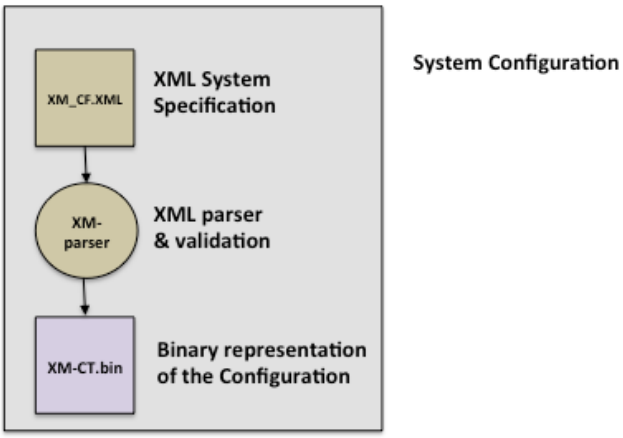


Figure 4.5.9: Generation of binary XtratuM configuration

The *XM-parser* tool consumes the configuration file, checks its correctness and its coherence with the specification and generates a set of data structures in the C language that contain the specification file parameters.

XM-parser validates the configuration file both syntactically and semantically and performs the following non-syntactical checks:

- Memory region overlapping: Checks that the memory regions (board memory blocks) are coherent and there is not overlapping.
- Memory area overlapping: Checks that the memory areas allocated to partitions exist and they do not overlap.
- Memory area inside any region: Checks that the memory areas allocated to a partition exist and their attributes.
- Duplicated partition's name and ID: Checks the coherence of partition names and identifiers.
- Allocated CPUs: Checks the coherence of the CPUs definition and allocation.
- Replicated port's name and ID: Checks the coherence of port names and use as source or destination.
- Cyclic scheduling plan: Checks that all partitions are allocated in the scheduling plan and there is not slot overlapping.
- Cyclic scheduling plan slot partition IDs: Checks the slot identifiers in the scheduling plan.
- Hardware IRQs allocated to partitions: Checks the Hardware IRQ allocation to partitions and their coherence.
- I/O port alignment and partition allocation: Checks the I/O port allocation overlapping and alignment.
- Ports allocated to STNoC and TTEthernet devices: Checks the available defined ports in the device definition and their allocation to partitions.
- Allowed health monitoring actions: Checks that the health monitor events and actions are defined in the hardware model.
- Channels allocation: Checks the coherence of channels and ports defined in partitions.

The generated data structures and variables are compiled to provide a XM-CT.bin file to be integrated in the final system using the deployment tools.

Data structures generated are coherent with the hypervisor and will be used in the system execution as input parameters for the hypervisor.

4.6 TTEthernet Network Configuration

[10] describes the TTEthernet configuration workflow from system requirements to device configuration artefacts as depicted in Figure 4.6.1.

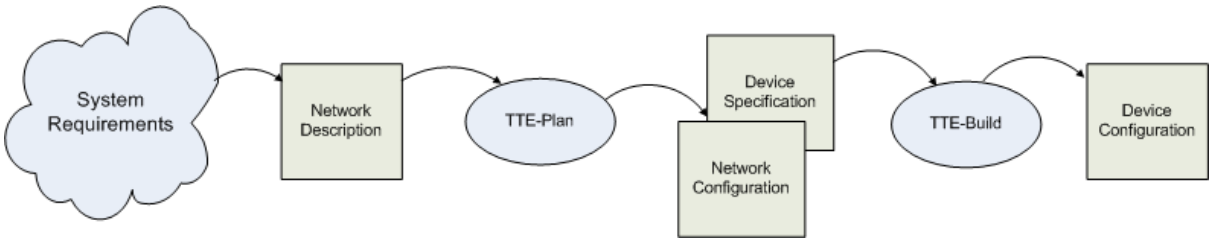


Figure 4.6.1: Configuration Workflow for a TTEthernet Device Configuaration

The *network configuration* (NC) file, generated upon successful execution of TTE-Plan, is a human-readable representation of the configuration artefact that is converted to the corresponding binary artefact loaded into the TTEthernet device. A detailed description of the relevant configuration items in the network configuration for the DREAMS specific configuration is provided in Section 4.2.4 of [10].

5 Tool-Specific Formats

The resource utilization metamodel presented in Chapter 3 is an exchange format between the different offline resource adaptation tools developed in DREAMS (and hence is capable to capture their output). This section presents those input formats of the respective offline scheduling and resource adaptation tools that have been formalized as meta-models or XML schemas ([9] details further tool input formats, e.g., for GRec, [9], Section 7.1). These file formats that will serve as an input for the definition of the DREAMS tool-chain in [13].

Further, this section describes an update of the metamodel for the safety constraints and rules checker introduced in [2].

5.1 Xoncrete

Xoncrete is a tool to define a partitioned system and perform the resource allocation of this system. The way to introduce the description of the entire system is through a web browser. Therefore, the first step to work with *Xoncrete* is with an empty project. The user defines all the elements of the system using *Xoncrete*'s web interface. Once all elements have been defined and all resources have been correctly allocated, *Xoncrete* allows to export the final configuration to a XMCF file for XtratuM.

Xoncrete maintains its own format to save all the data into a project file. This format is known as "*eprj*" format file. Normally, the user does not have to edit this file, but only to save and load it. Nevertheless, in this project the user has to know the *eprj* format since *Xoncrete* is integrated into a set of tools [13] and the information of the system managed by *Xoncrete* will not be entered in an empty project but through an *eprj* file.

In the rest of this section the *eprj* format will be introduced. The complete description of the format can be found in [23]. The main sections of the eprj format are shown in Figure 5.1.1.

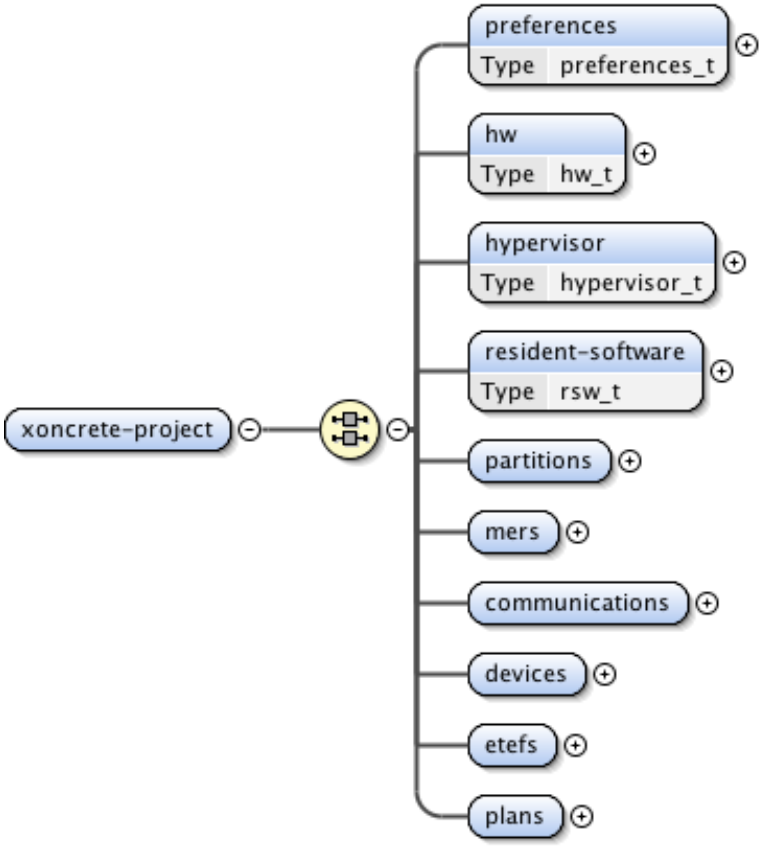


Figure 5.1.1: Main sections of *Xoncrete eprj* format.

The next subsections will show the contents of the above elements. As *Xoncrete* generates the XML configuration file for XtratuM, this format has a lot of common elements with XMCF format. The elements that are obvious will not be explained. There is complete information in the *Xoncrete User Manual* [22].

5.1.1 Preferences

These are the options that will apply to a *Xoncrete* project.

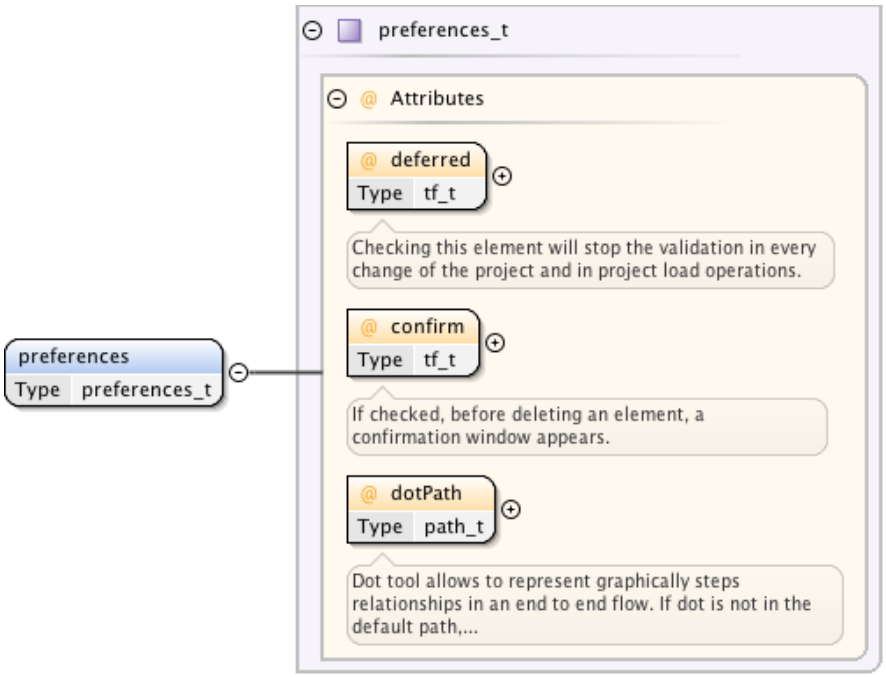


Figure 5.1.2: Xoncrete Preferences.

5.1.2 Hardware

This is the information stored about the hardware of the system.

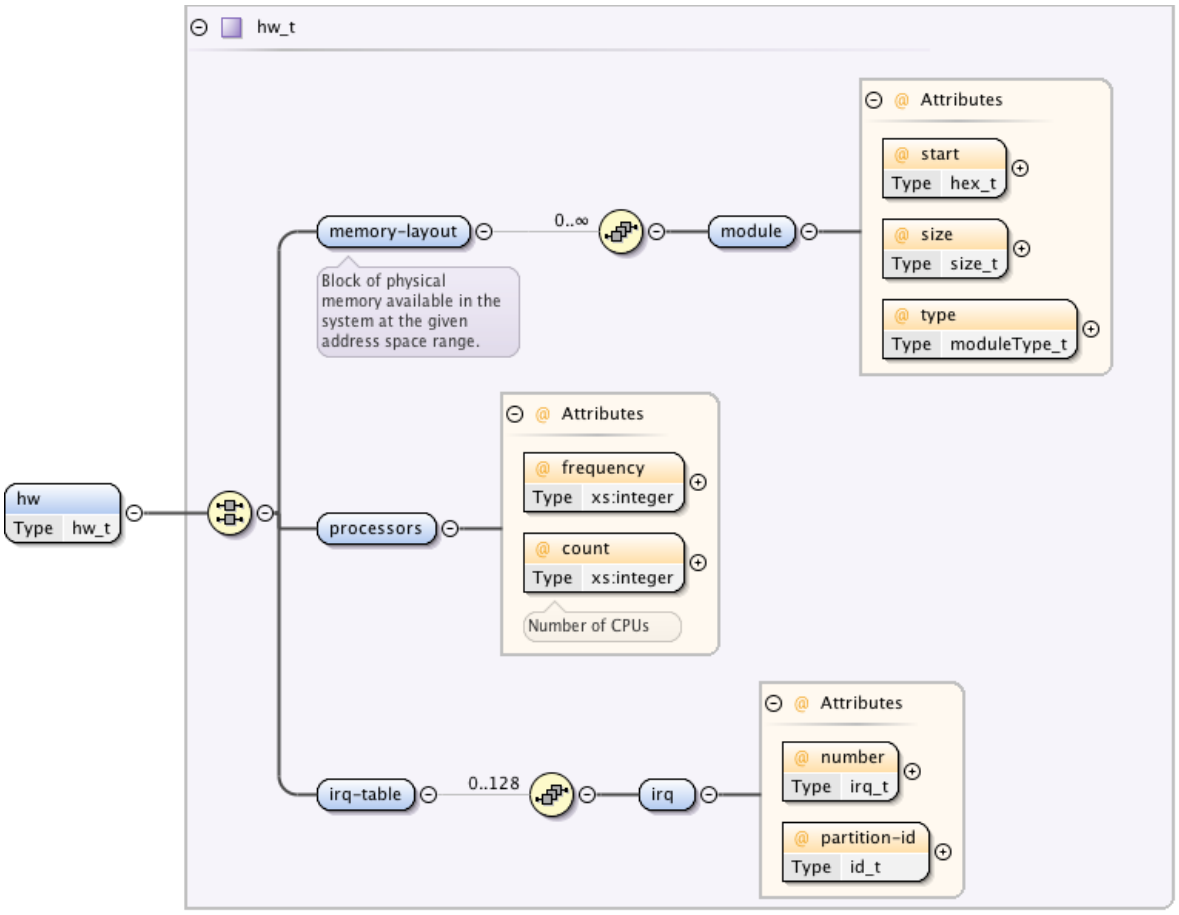


Figure 5.1.3: Xoncrete system hardware model.

5.1.3 Hypervisor

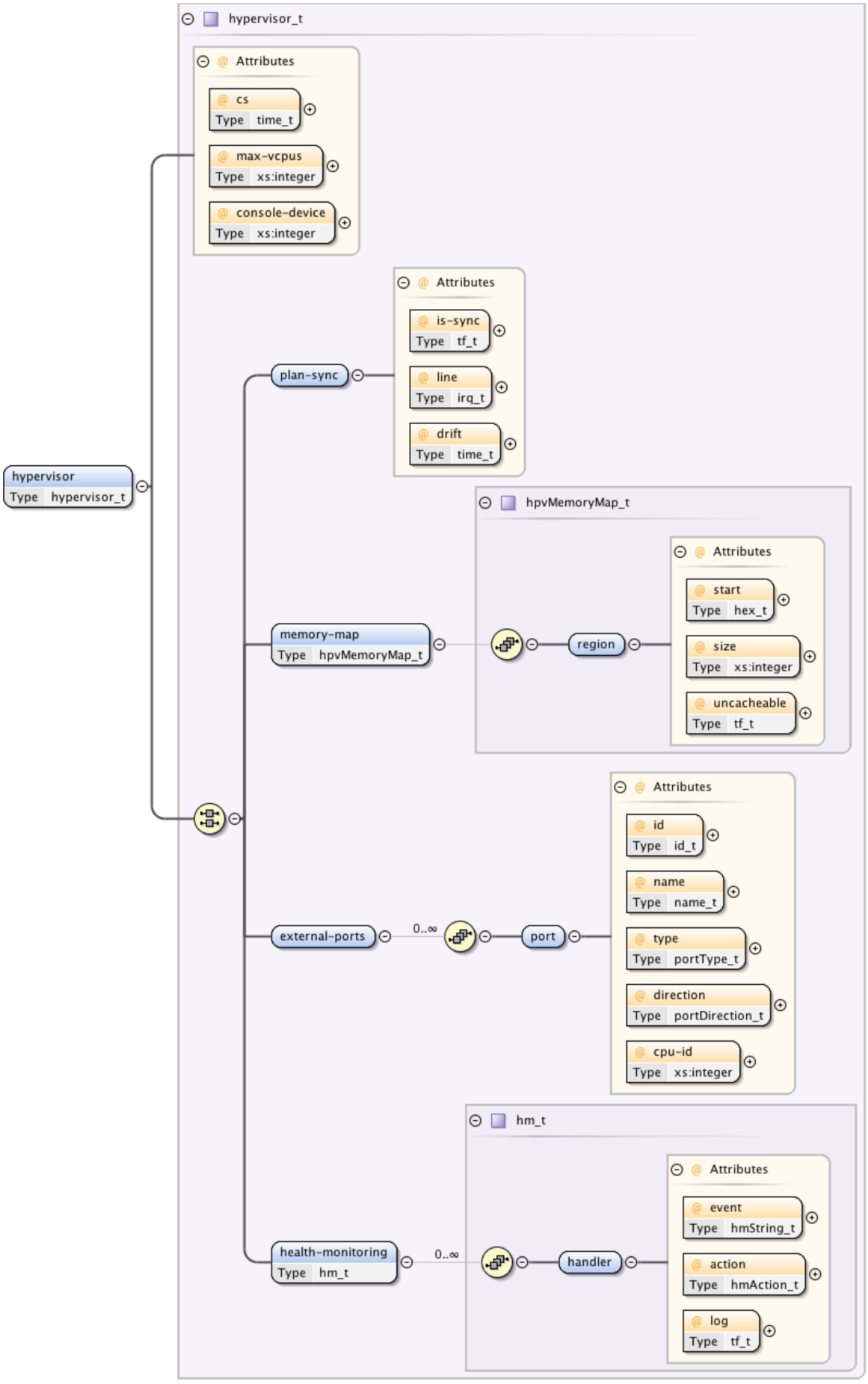


Figure 5.1.4: Xoncrete hypervisor model.

The following elements are described using the schema shown in Figure 5.1.4:

- **Partition Context Switch (CS):** Worst case time required to perform a partition context switch.
- **Maximum number of virtual CPUs (max-vcpus):** Maximum number of Virtual CPUs (vCPU) available for each partition, according to the configuration parameters of XtratuM.
- **Console device:** To which console device events reported to the hypervisor will be logged to.
- **Plan syn-Is sync:** Whether or not XtratuM must synchronise the plan with an externally provided event. If it is set to TRUE, the following options are taken into account:
    - **Synchronisation line:** Interrupt line through which the synchronisation event is signalled.
    - **Maximum clock drift:** Maximum drift between the XtratuM's reference clock and the external clock source of the synchronisation event.
- **Uncacheable:** Specifies that a memory area allocated to a partition will not be cached.
- **External Ports:** Each external port is an endpoint of a communication channel and belongs to an external source.
    - **Name:** Name of the port.
    - **Type:** Type of the port that is either SAMPLING or QUEUING.
    - **Direction:** If the port is a SOURCE or DESTINATION port.
    - **Cpu Id:** The identifier of the core where the external port is available.
- **Health monitoring**: For the hypervisor (and for partitions) there exist a table to control the *health monitoring* mechanism. The table contains a maximum of 29 entries, each of them containing the following elements:
    - **Event:** Unexpected event or failure.
    - **Action:** Action to take.
    - **Log:** If the event occurrence has to be logged for a latter examination.

5.1.1 Resident software

This element only contains its memory map.

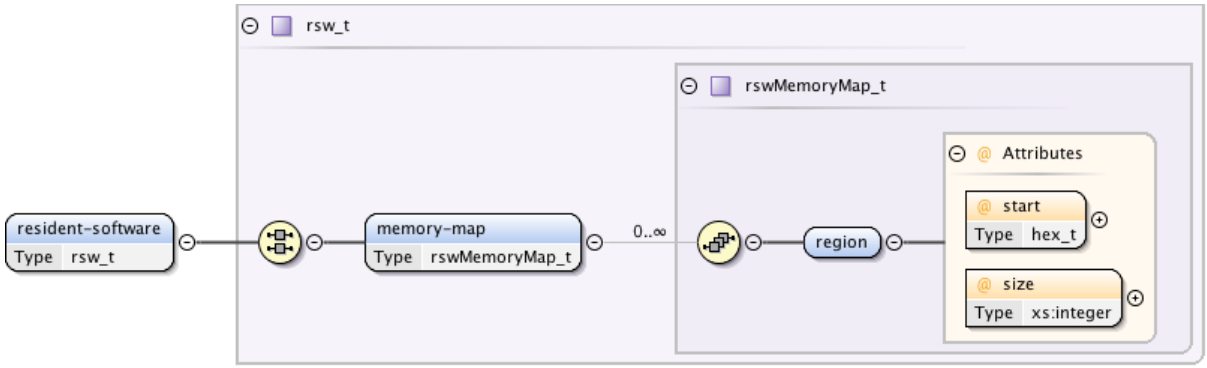


Figure 5.1.5: Xoncrete resident software model.

5.1.2 Partitions

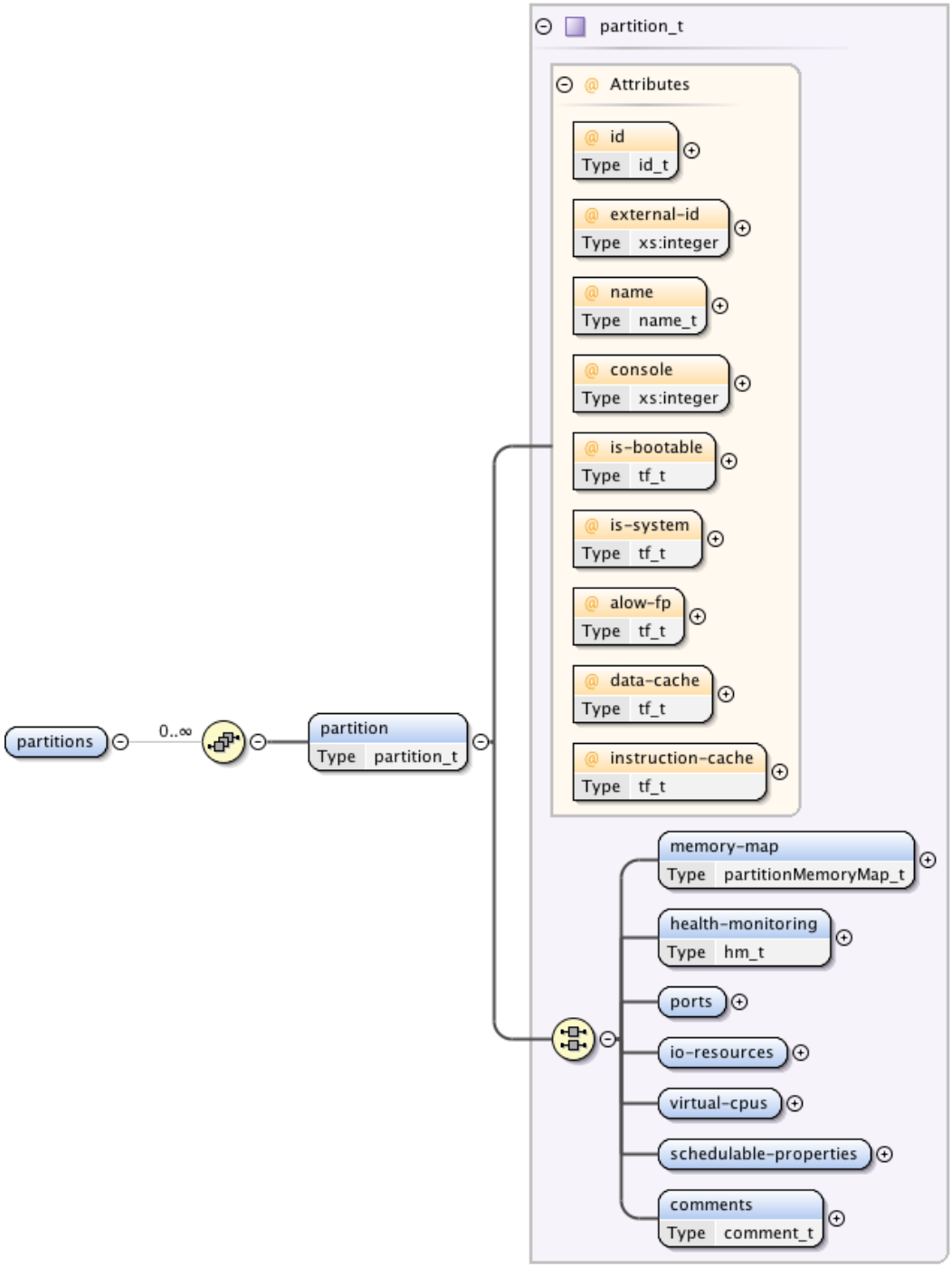


Figure 5.1.6: XtratuM partition model.

An XtratuM partition model has the following attributes (see Figure 5.1.6):

- **allow-fp:** The partition is allowed to use floating point instructions.
- **is-system:** The partition is marked as system partition.
- **is-bootable:** The partition is marked as boot partition.
- **data-cache:** Enable state for data cache.
- **instruction cache:** Enable state for instruction cache.
- **console:** To which console device events reported by the partition to the hypervisor will be logged to.

5.1.2.1 Memory Map

A list of assigned memory areas, each of them with the following attributes (see Figure 5.1.7):

- **Address space:** Portion of memory assigned to place code and variables.
- **Unmapped:** The area will not be mapped.
- **Rom:** The area will be rom.
- **Uncacheable:** The area will not be cached.
- **Read-only:** The area will be read-only.
- **Mapped-at** (optional): When specified, the partition will be mapped starting at this address inside its virtual memory map.

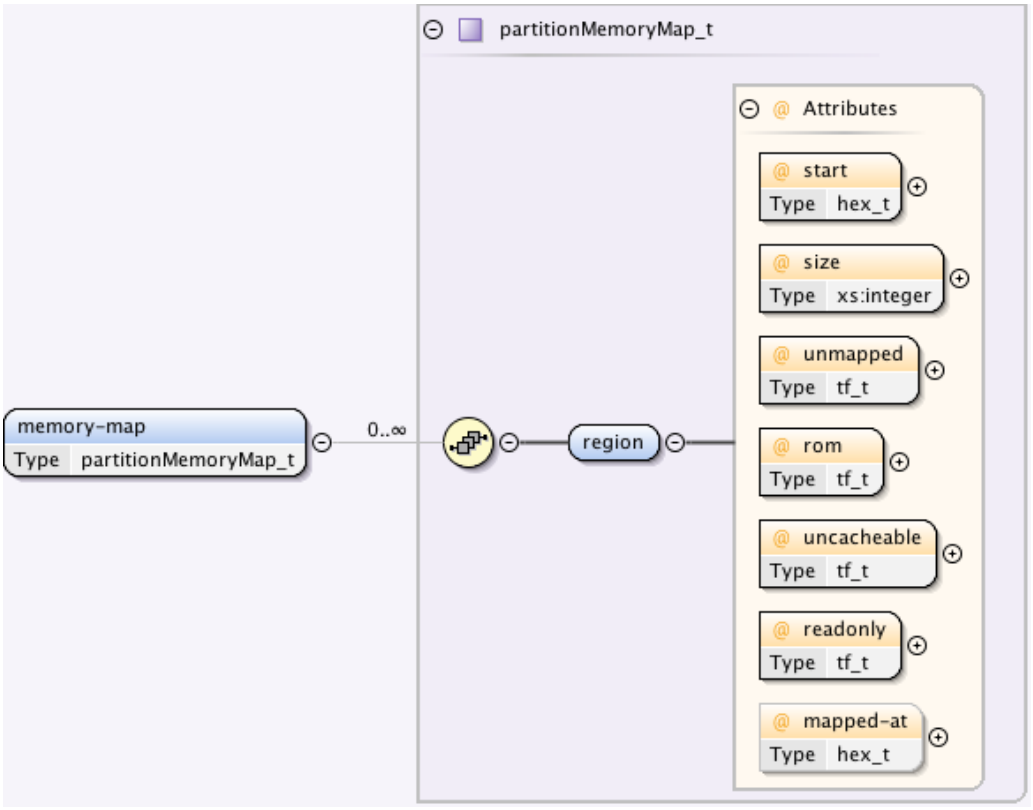


Figure 5.1.7: Xoncrete memory map model.

5.1.2.2 Health Monitoring

This element is the same as for hypervisor.

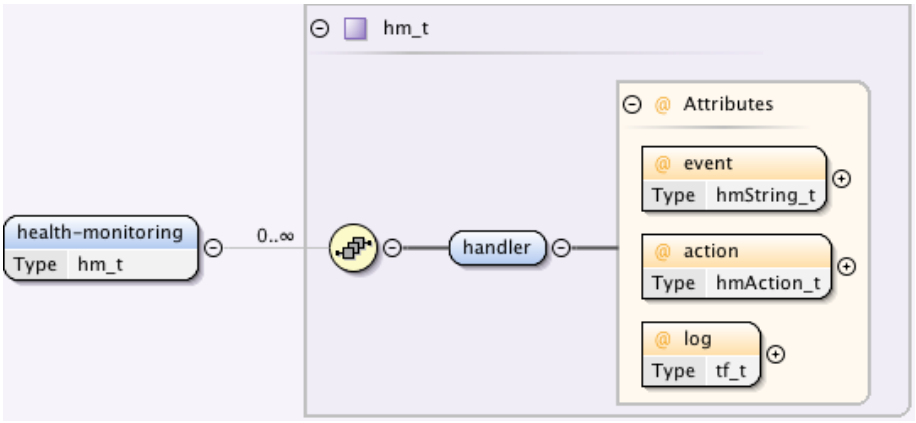


Figure 5.1.8: Xoncrete health monitoring model.

5.1.2.3 Ports

This element is the same as for hypervisor.

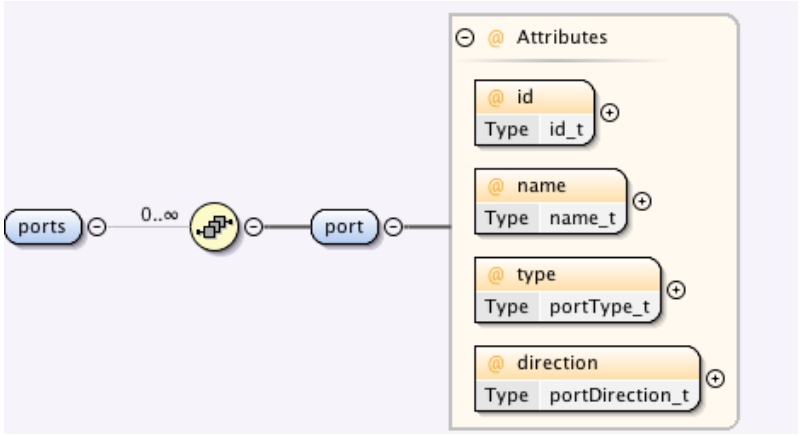


Figure 5.1.9: Xoncrete ports model.

5.1.2.4 I/O Resources

The access to the whole I/O space is also partitioned. Each partition may only access the hardware devices that it is in charge of. I/O space is partitioned by defining I/O address ranges, where all the addresses within this range are granted access from one partition, or by defining I/O masked registers, where a set of bits of an I/O address is assigned to a partition. Many partitions can have access to the same I/O masked registers, but their access mask (assigned bits) must be disjoint.

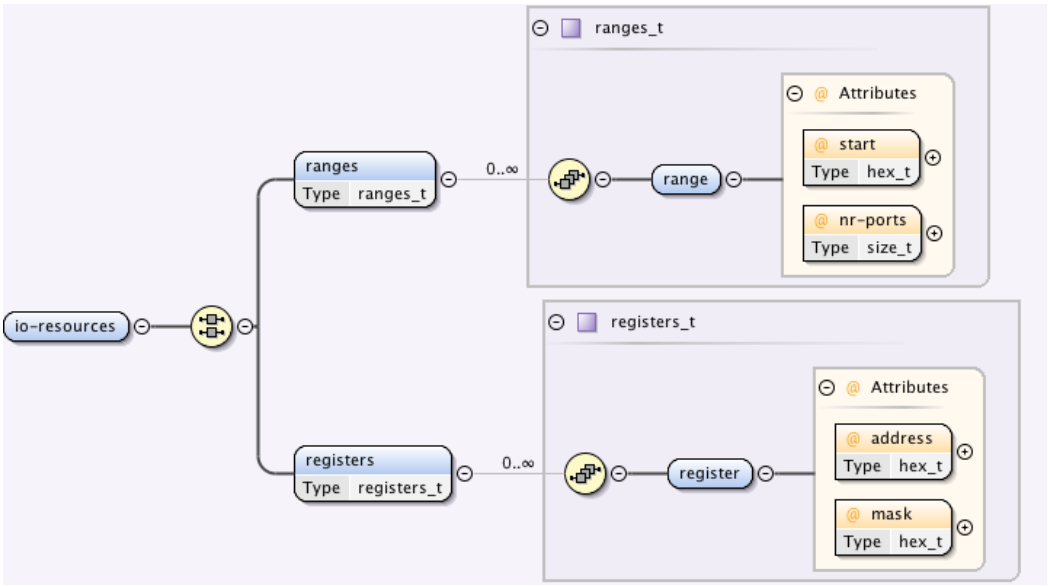


Figure 5.1.10: Xoncrete I/O resources model.

5.1.2.5 Virtual CPUs

The allocation of Virtual CPUs to specific CPUs.

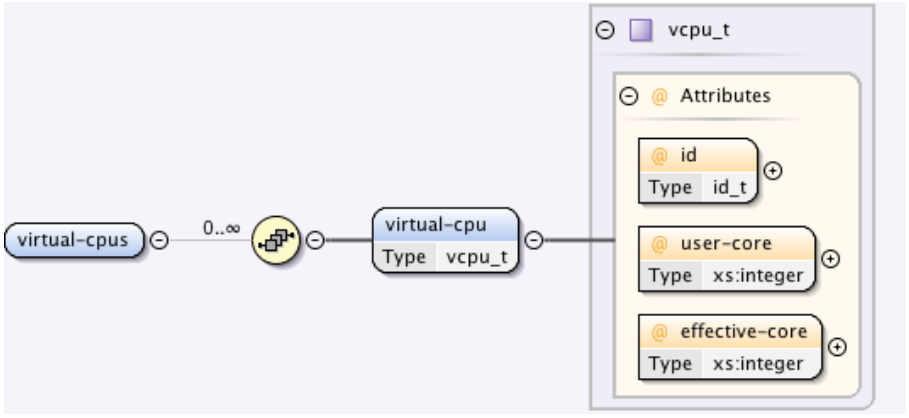


Figure 5.1.11: Xoncrete virtual CPUs model.

5.1.2.6 Schedulable Properties

Regarding the attributes of this element:

- **Longest-hypercall:** Time to be left unused after each slot of the partition.
- **Slot**: Overhead associated with internal processing for each slot assigned to the partition.

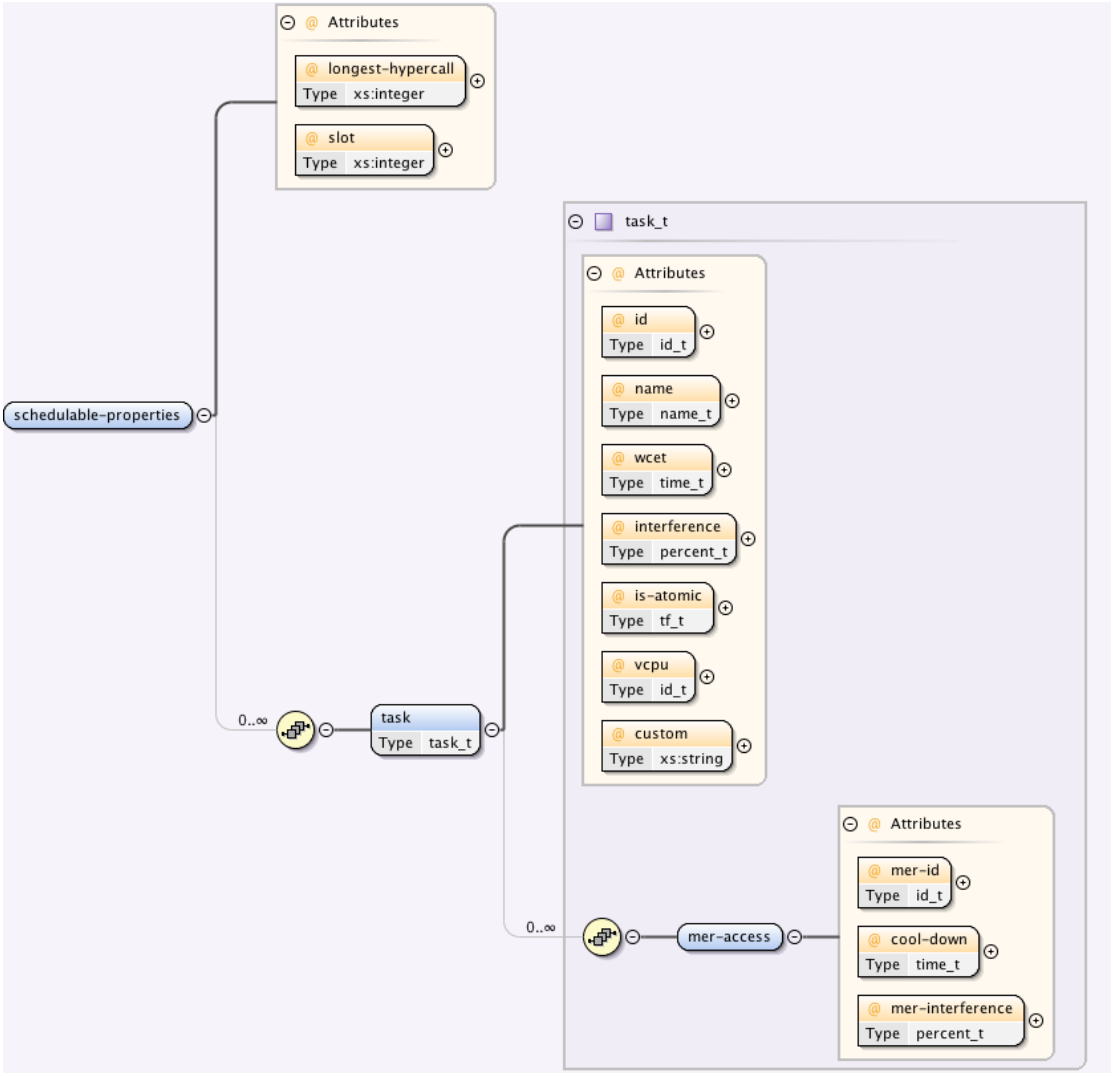


Figure 5.1.12: Xoncrete schedulable properties model.

Regarding the task attributes:

- **Is atomic:** The task cannot be pre-empted.
- **Virtual CPU:** Virtual CPU where the task shall be executed.
- **WCET:** Worst Case Execution Time.
- **Interference:** Fraction of the task's WCET applied as interference generated by the task.
- **Mutual-exclusion-resources:** Usage of mutual exclusion to protect critical sections. Each tuple in the list contains a mutual-exclusion-resources name, a value called *cooldown time*, and a percentage of this value or *interference*.

5.1.3 Mutual exclusion resources (MERs)

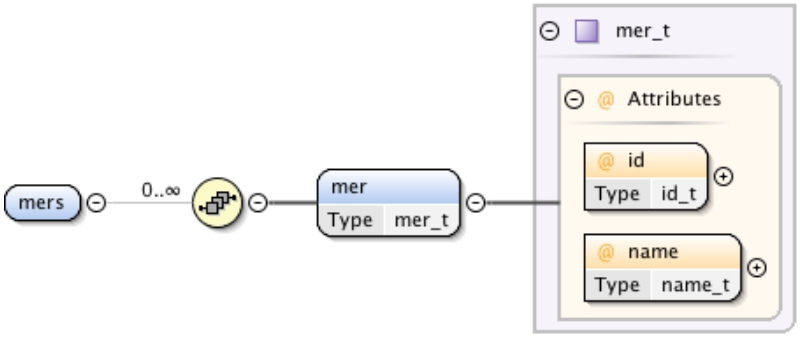


Figure 5.1.13: Xoncrete mutual exclusion resources model.

5.1.4 Communications

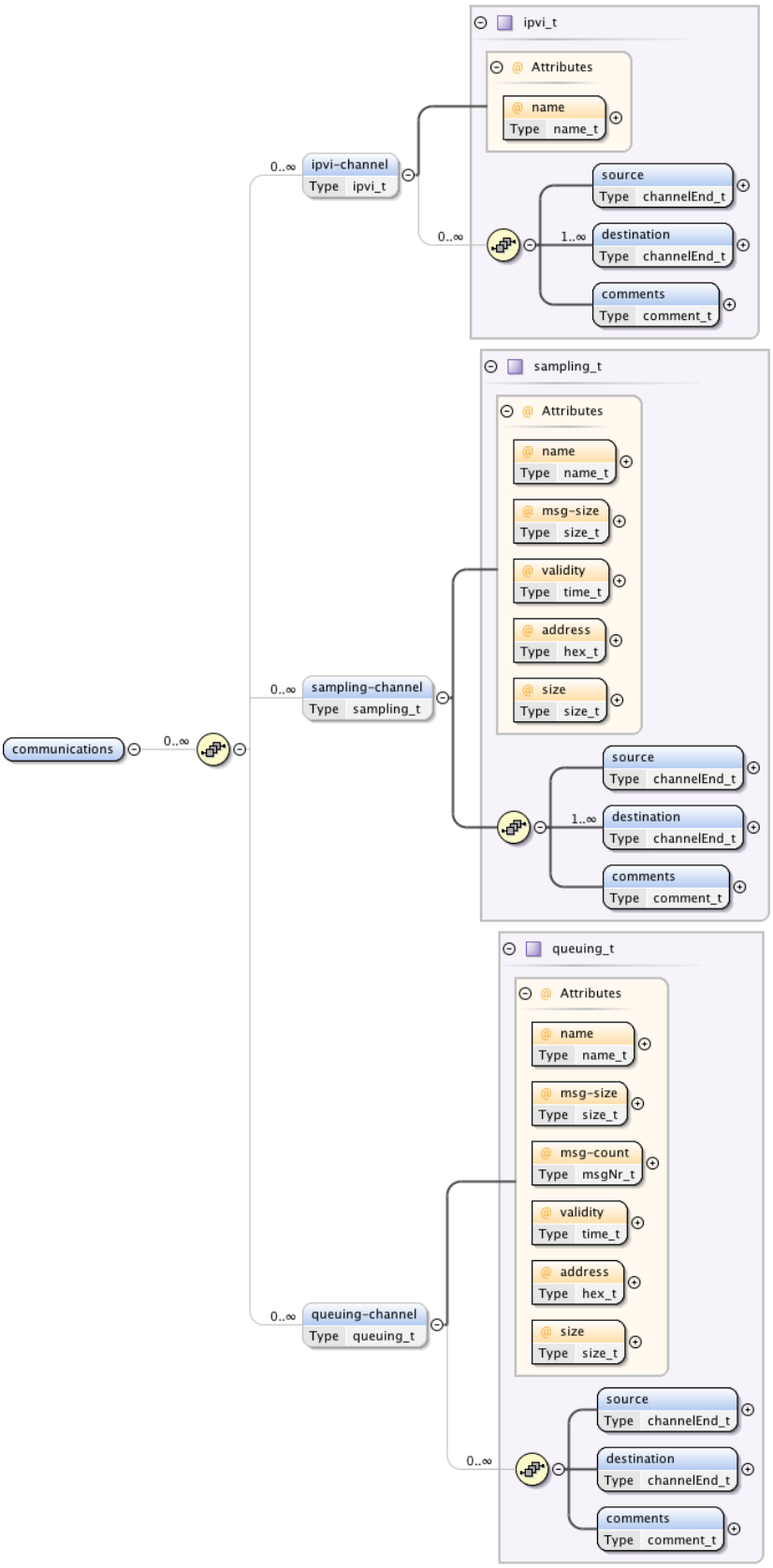


Figure 5.1.14: Xoncrete communications model.

In Figure 5.1.14, `channelEnd_t` represents the identifier of the port.

5.1.5 Devices

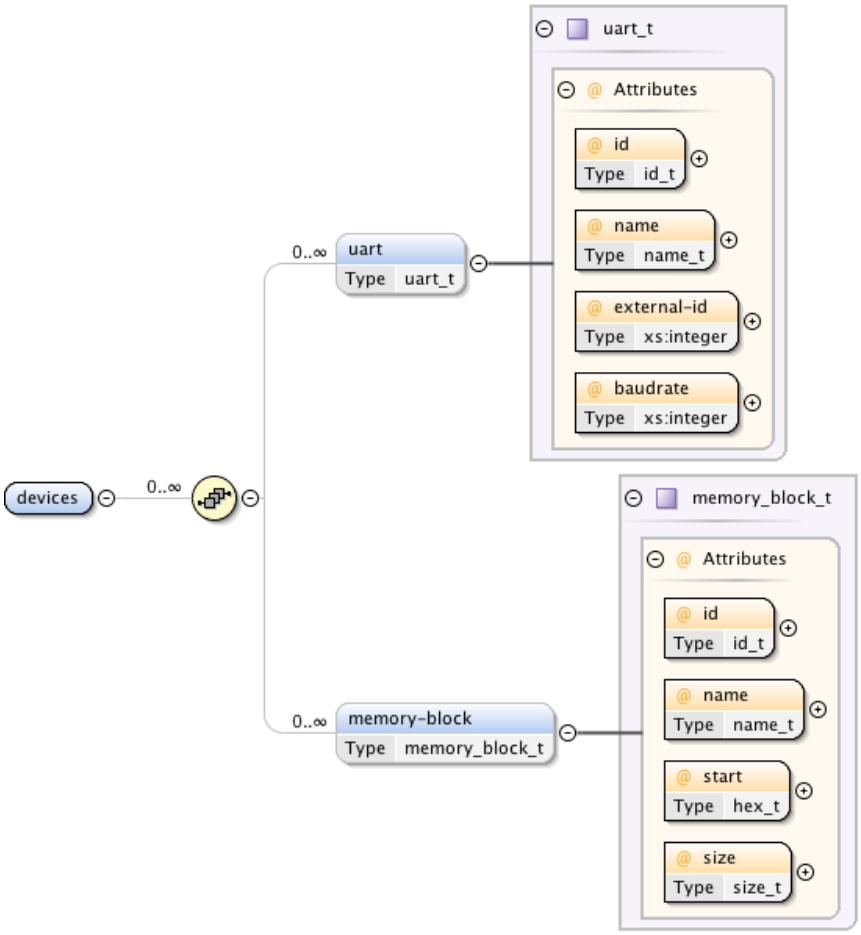


Figure 5.1.15: Xoncrete devices model.

5.1.6 End to end flows (ETEFs)

The attributes and description of an end-to-end flow are described in [22].

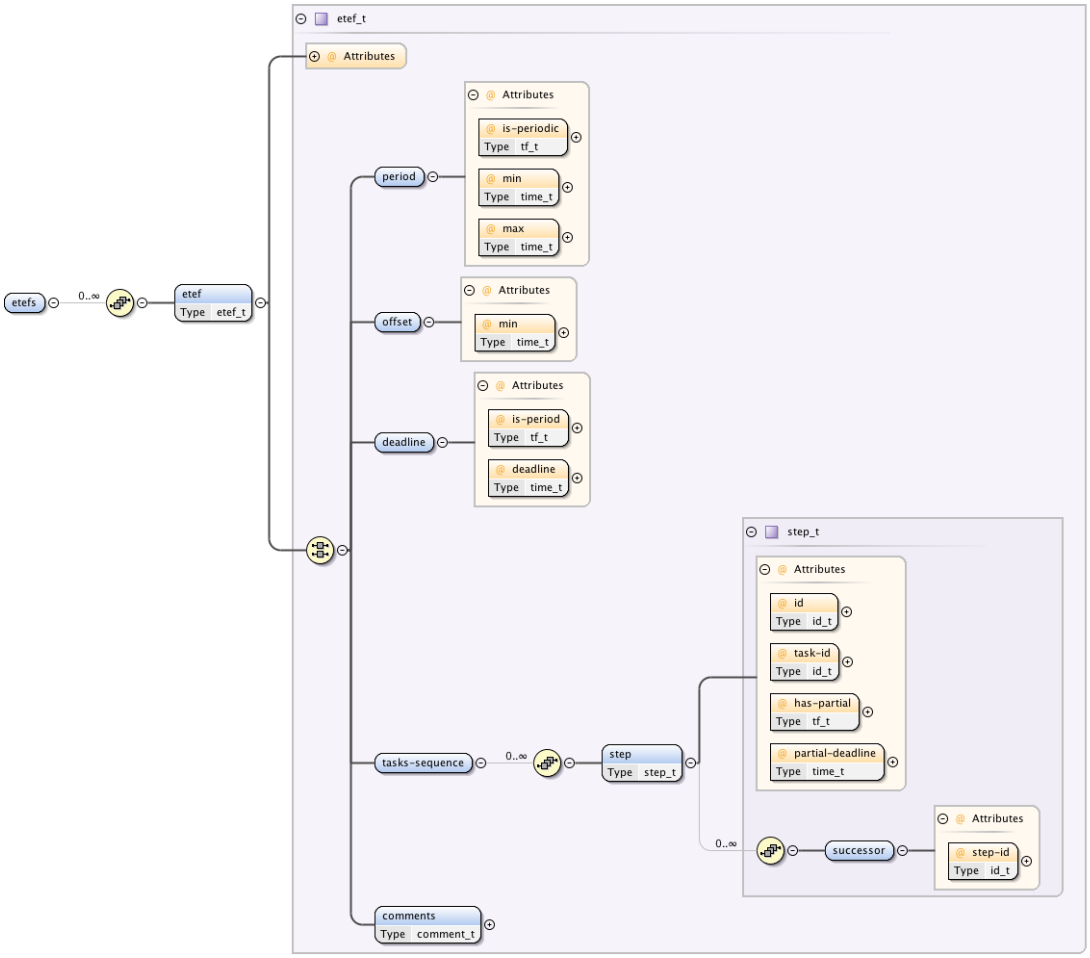


Figure 5.1.16: Xoncrete end-to-end flows model.

5.1.7 Plans

This element stores all the information about scheduling plan. It also stores the plans itself if the scheduling has been run and there exist a temporal allocation.

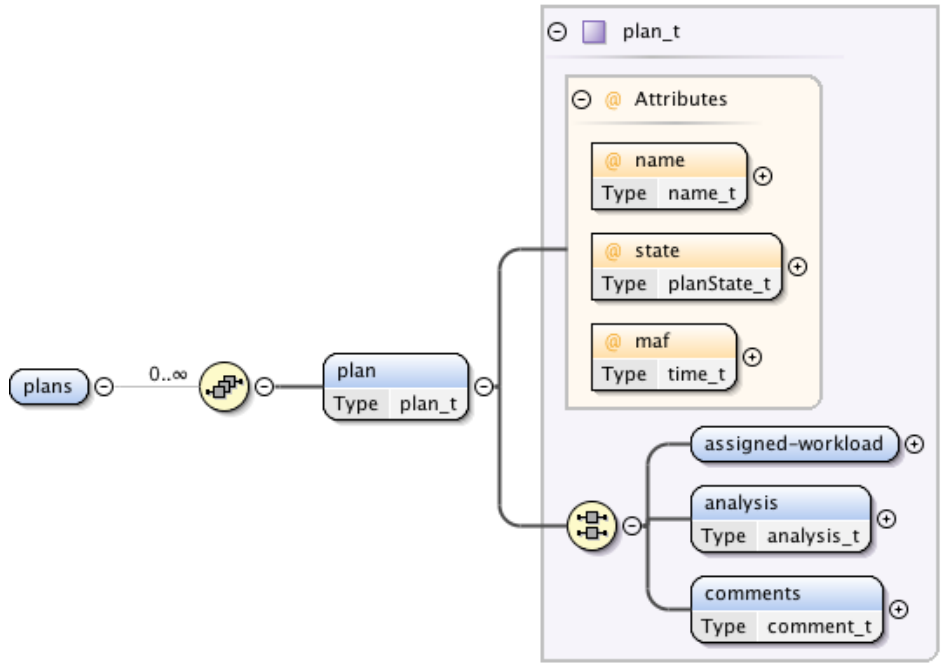


Figure 5.1.17: Xoncrete plans model.

5.1.7.1 Assigned Workload

This element stores the ETEF identifiers that belong to a specific plan.

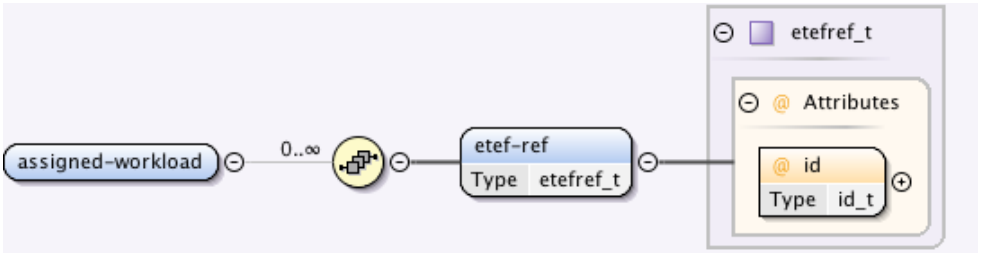


Figure 5.1.18: Xoncrete assigned workload model.

5.1.7.2 Analysis

This element stores the scheduling plan. The schedule element contains the time slices, i.e., the temporal windows with the starting time, the duration and identifier of the task instance of an ETEF (step). The schedule also contains the deadline misses if they exist.

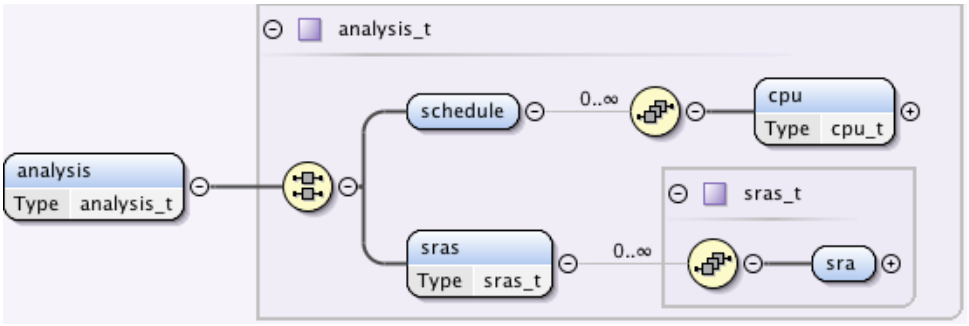


Figure 5.1.19: Xoncrete analysis model.

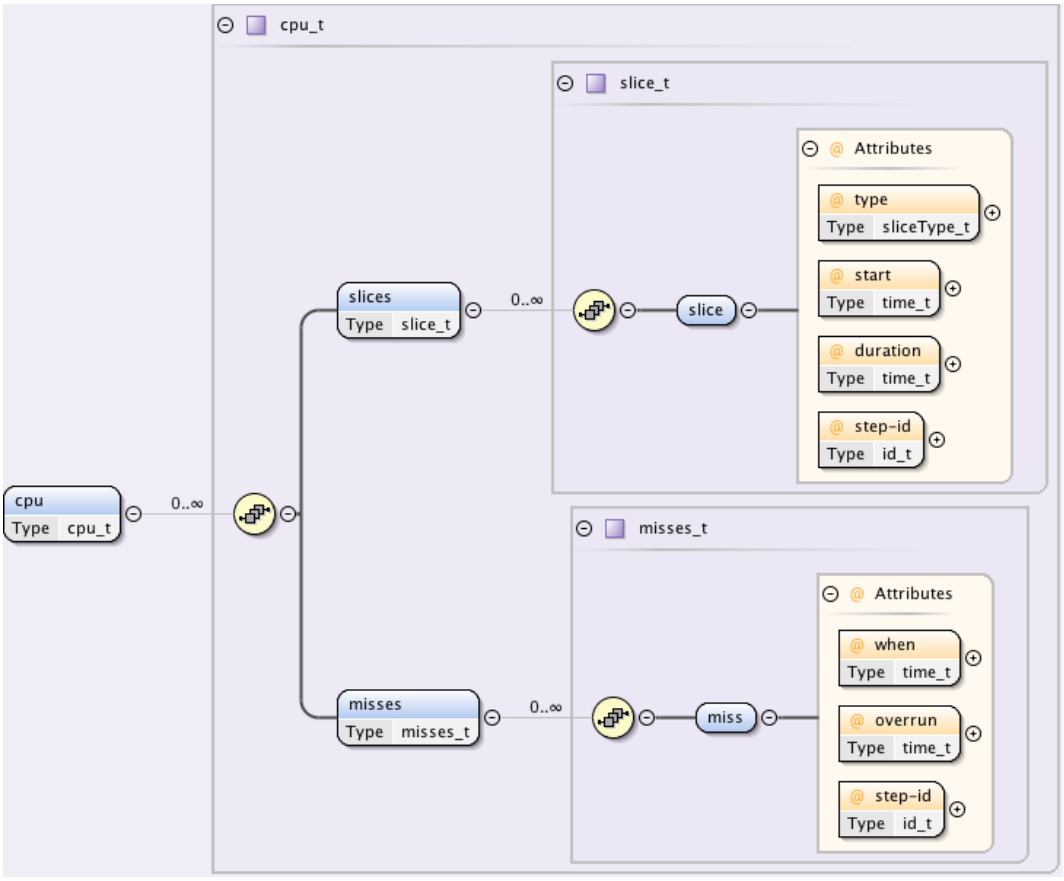


Figure 5.1.20: Xoncrete analysis CPU element model

The *sra* element contains the time slices in which a mutual exclusion resource is executing.

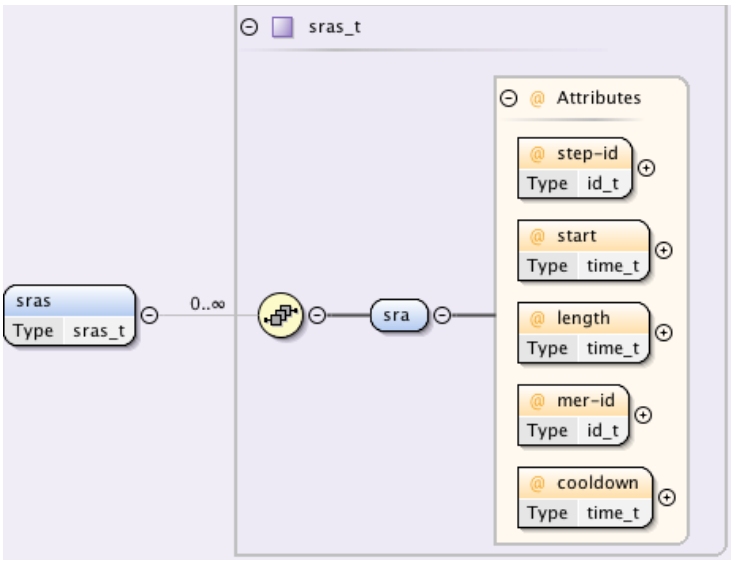


Figure 5.1.21: Xoncrete analysis SRAS element model

5.2 TTPlan

The TTEthernet toolchain and its configuration interfaces is described in detail in [10], Section 4.2. Relevant to this deliverable are the description of the network configuration generator tool (TTE-Plan, see [10], Section 4.2.2) as well as the detailed description of the *network description* (ND) interface (see [10], Section 4.2.3), both depicted in the toolchain workflow in Figure 5.2.1.

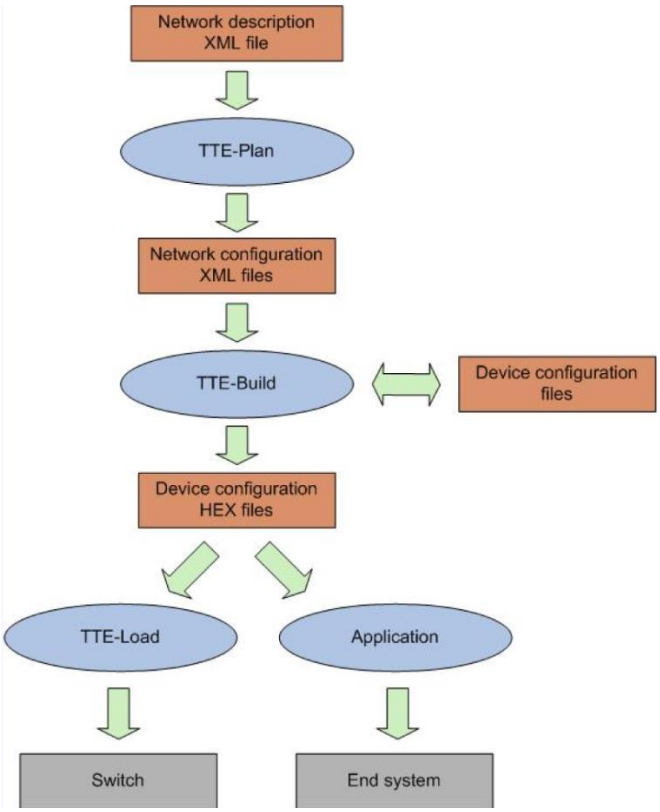


Figure 5.2.1: TTEthernet Tool-Chain Workflow.

5.3 MCOSF

MCOSF is a command line tool that has 3 tasks: (1) generate transition modes to reduce the mode switching delay, (2) embellish the reconfiguration graphs with the generated transition modes and (3) provide information for online incorporation of event-triggered activities in a time-triggered schedule.

The tool MCOSF takes an XML file as input and generates an XML file. The input XML file contains the information of the system hardware, system software, and the nominal modes of operation and software components that are to be scheduled. The output XML file contains the information of the modes of operation including the transition modes and flexibility information. The input and output XML formats are tailored to the DREAMS project and hence the complete details can be found in Annex A.

Before we discuss the input and output files of the tool in detail, it is beneficial to have some basic knowledge of the following terms:

5.3.1 Mode Types

From the perspective of scheduling, there are two types of modes of operation. The online scheduler (i.e., LRS) needs to distinguish between the two modes of operation (these modes are defined in detail in [8][9]). A nominal/continuous mode is a mode that is executed cyclically until there is a switching event (e.g., core failure). This type of mode is defined by the system designer. On the other hand, a transition/switching mode is a mode that is executed at most once and switches to a continuous mode when there is a safe point to switch. In the case of DREAMS, a transition mode is defined by the scheduler to improve mode switching delay.

5.3.2 Black-out Slots

The Black-out parameter of a slot defines a safe point in time to execute a mode switch. For an independent task-set, safe points can be trivially defined at time points when no task results in partial execution upon switching. Either it should be completely executed or not executed at all. However, for the case of dependent task-sets, defining this point in time is not as trivial. For more details, please see [8][9]. This parameter is defined by the scheduler during MAF. In the case of DREAMS, the end of MAF is always a safe point to switch.

5.3.3 Input Schema

The input data elements are defined in a hierarchical structure. The trivial elements of this hierarchy will not be defined in this section, instead they can be found in Annex A.1.

The root element of the hierarchy is defined in Figure 5.3.1. It has only one attribute, i.e., `Name`. It contains the following sub-elements: platform description, set of partitions, local and global modes of operations, set of tasks, and a set of messages communicated between tasks.

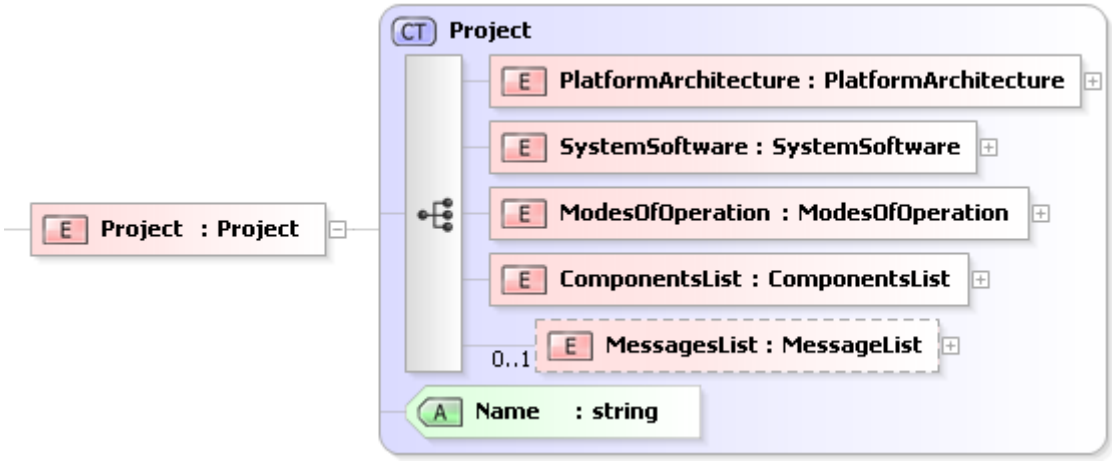


Figure 5.3.1: Project Root for MCOSF input.

5.3.3.1 Platform Architecture

The `PlatformArchitecture` element defines a collection of processing nodes. Note that this element does not define the off-chip network. This information is not required by MCOSF, since the off-chip network schedule cannot change during the operation of the system. Furthermore, scheduling the same message with multiple phases for multiple modes can seriously overload the off-chip network.

Figure 5.3.2: Platform architecture for MCOSF input.

5.3.3.1.1 Processing Node

The processing node is defined as a collection of tiles with an identifier, i.e., `Name`. Note that the on-chip network is not present in this structure.

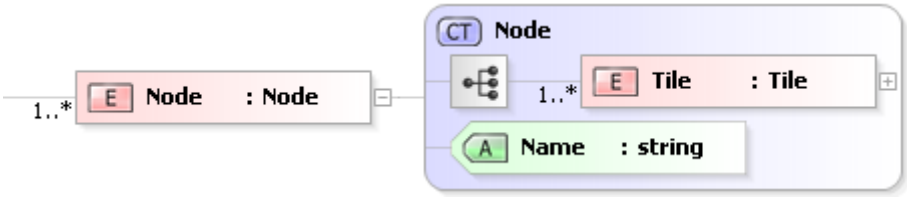


Figure 5.3.3: Processing node element for MCOSF input.

5.3.3.1.2 Tile

A tile is defined to be a collection of cores. The communication between cores within a tile is achieved using shared memory and/or shared caches. Since there is no need of such fine control over caches, which can vary based on the mode of operation, this information is not available in the `Tile` element.

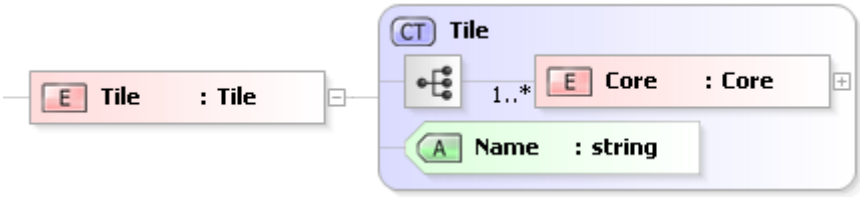


Figure 5.3.4: Tile element for MCOSF input.

5.3.3.1.3 Core

The processing core is defined using the `Core` element in the input XML of MCOSF. An identifier (i.e., a `Name`) is used to mention the allocation of partitions to these cores in a certain mode.

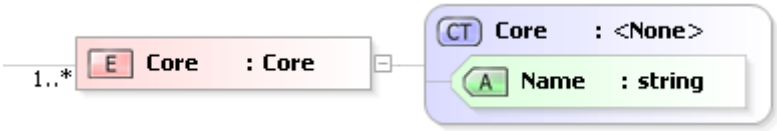


Figure 5.3.5: Core element for MCOSF input.

5.3.3.2 System Software

The system software is defined by a set of partitions. The information regarding the hypervisor and its containing partitions is not required for MCOSF. The `Partition` element defines an identifier, i.e., a `Name`.

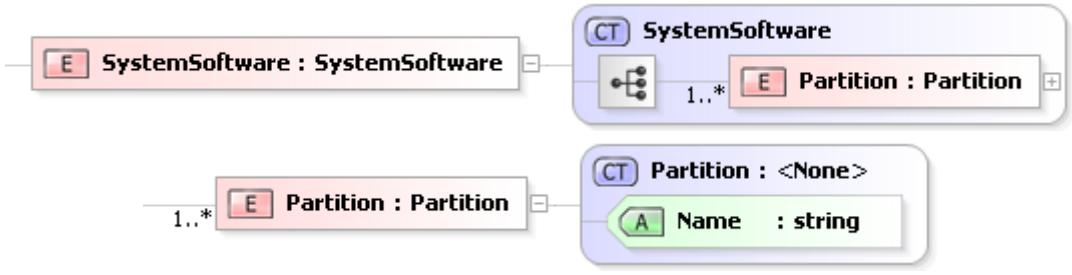


Figure 5.3.6: System Software element for MCOSF input.

5.3.3.3 Modes of Operation

The `ModesOfOperation` element of the input XML file is used to define all local and global modes and their transitions/reconfigurations. The attribute `InitialGlobalMode` provides the name of the global mode of operation in which the system starts.

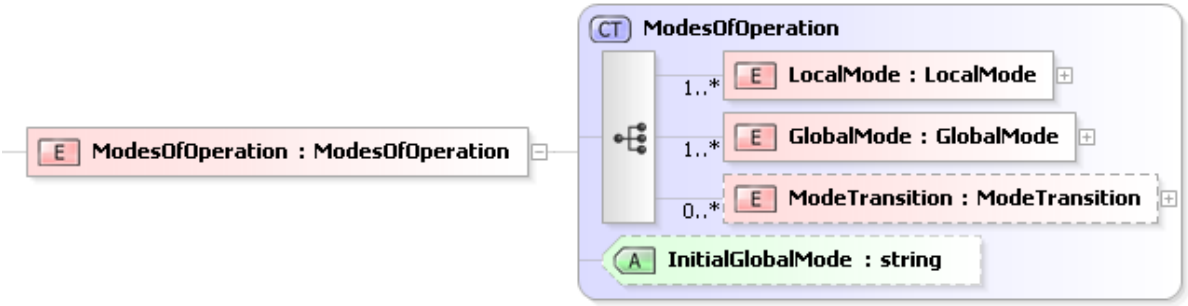


Figure 5.3.7: Modes of operation element for MCOSF input.

5.3.3.3.1 Local Mode

A local mode of operation defines a mode for a tile. It is identified by the set of core schedules used for executing tasks. It has the following attributes:

- `Name`: Provides a name for identification in the reconfiguration graphs.
- `Type`: Defines the type of local mode as per Section 5.3.1.
- `Tile`: The name of the tile for which the local mode is defined.
- `MafUs`: The size of major application frame (MAF) in microseconds.

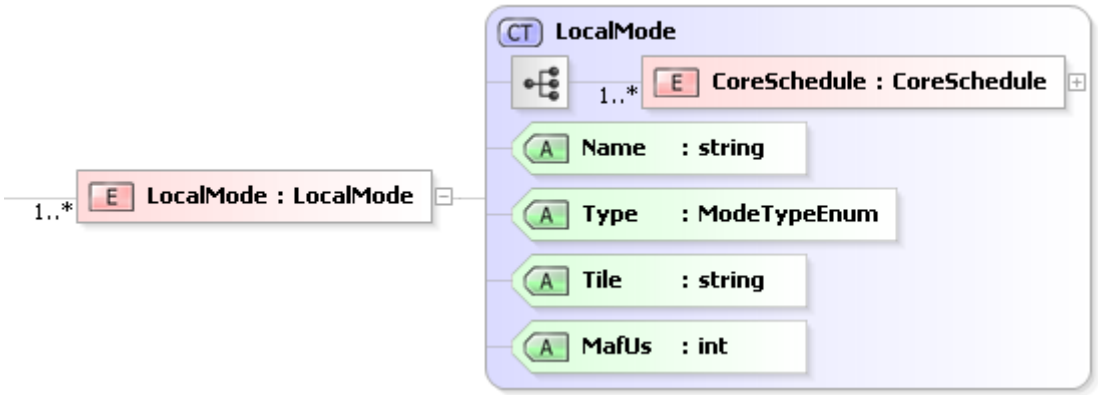


Figure 5.3.8: Local Mode element for MCOSF input.

5.3.3.3.2 CoreSchedule

The core schedule element defines a set of slots available for executing tasks related to specific partitions. The element `CoreSchedule` also defines the core name on which these slots execute. A visual representation of the element is provided below.

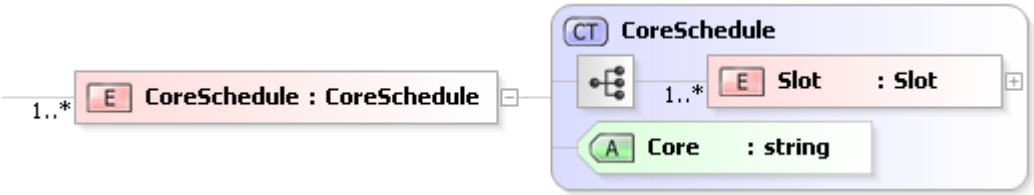


Figure 5.3.9: Core schedule element for MCOSF input.

5.3.3.3.3 Slot

A partition slot is defined by the `Slot` element. It defines a set of components that are executed during this slot. It has the following attributes:

- `Id`: A unique identifier
- `StartUs`: The start time of the slot from the start of current MAF.
- `DurationUs`: The length of the slot.
- `Partition`: Identifier of the partition which that this slot. By definition, only the tasks belonging to this partition can execute during this slot.

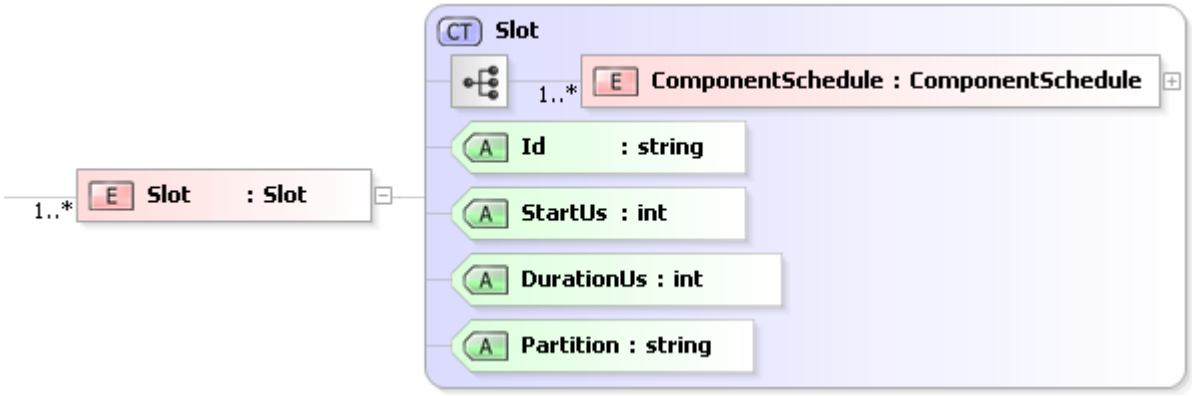


Figure 5.3.10: Slot element for MCOSF input.

5.3.3.3.4 Component Schedule

The schedule for the component/task is defined by the element `ComponentSchedule`. It defines a component identifier for which the schedule is defined and the start time for execution from the start of MAF in microseconds. In DREAMS, the tasks are non-pre-emptive and hence no other information is required to define a schedule for a component.

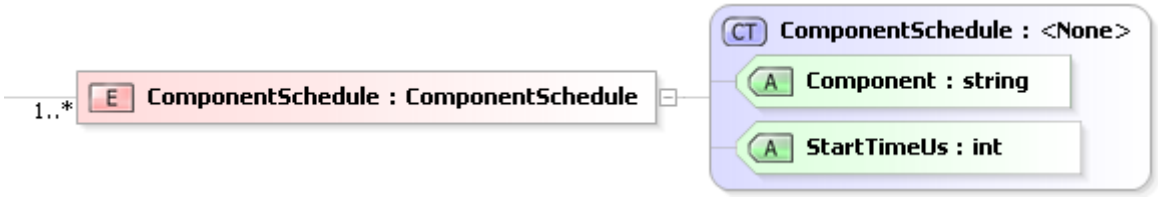


Figure 5.3.9: Component schedule element for MCOSF input.

5.3.3.3.5 Global Mode

The global mode of operation is a collection of local modes of operation that are executed in different tiles at the same time. It has an identifier to signify the current state in the reconfiguration graphs.

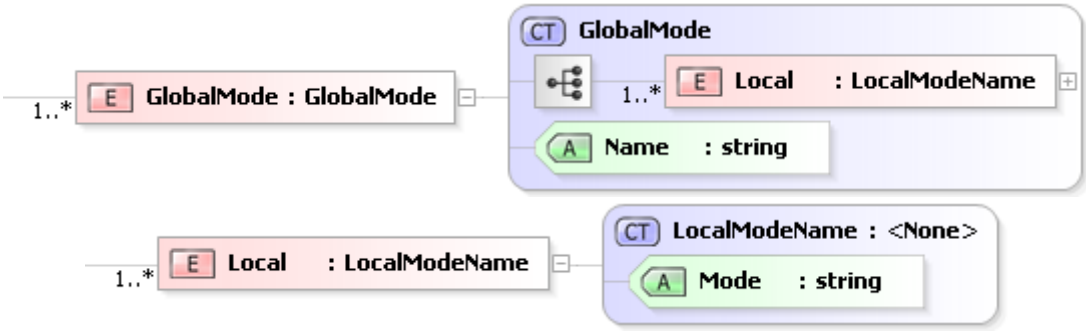


Figure 5.3.10: Global Mode element for MCOSF input.

5.3.3.3.6 Mode Transitions

The reconfiguration graph defines a tree of local and global modes starting at a root mode. The edges of this reconfiguration graph are represented in the input XML by `ModeTransition`s. Therefore, the only attributes of this element are source and destination modes.

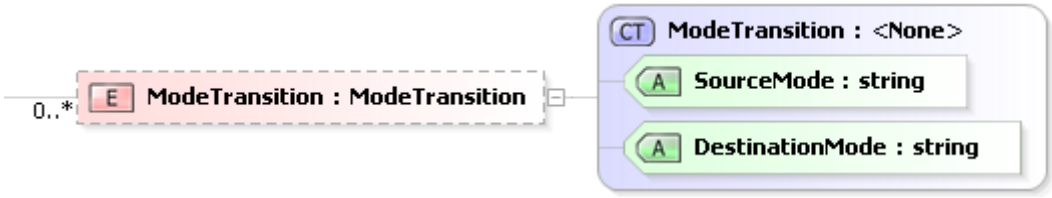


Figure 5.3.11: Mode Transition element for MCOSF input.

5.3.3.4 Component List

A component list is a collection of components/tasks. Each component is defined as shown in the figure below. It has the following attributes:

- `Name`: The name of the component.
- `WcetUs`: The worst-case execution time of the component in microseconds.
- `PeriodUs`: The period of this component in microseconds.
- `MinExecutionOffsetUs`: The minimum execution offset of the component in microseconds.
- `ReactionConstraintUs`: The constraint that defines how long after the start of the period the component needs to be completed. The time units used are microseconds.
- `Partition`: The name of the partition to which this component is associated to.
- `SafetyLevel`: The safety level of the component (i.e., DAL-A/B/C/D).

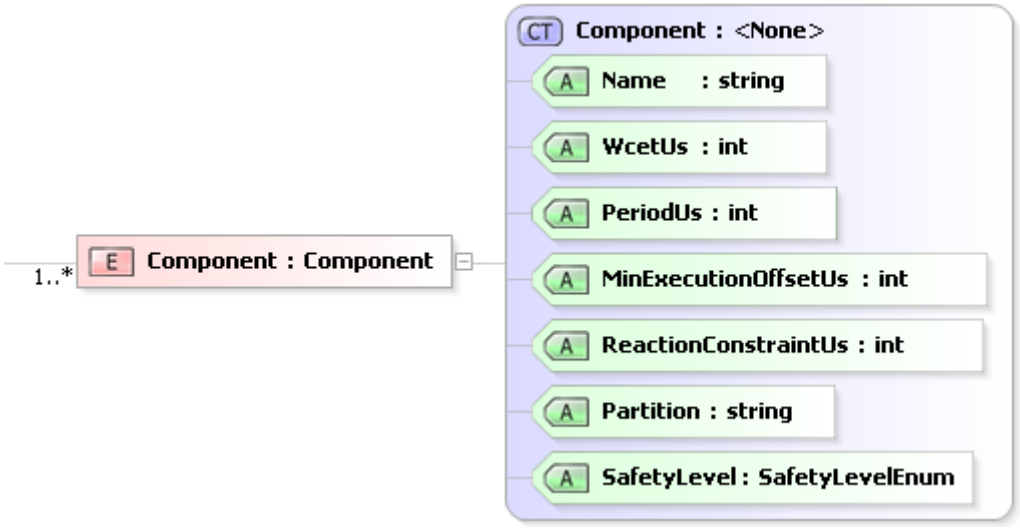


Figure 5.3.12: Component element for MCOSF input.

5.3.3.5 Message List

The message list is a collection of messages that are sent/received between the components. The attributes of a message are defined as follows:

- `Id`: Message identifier.
- `SourceComponent`**:** Name of the source component.
- `Destination`**:** A collection of the destination components (to support broadcast messages)

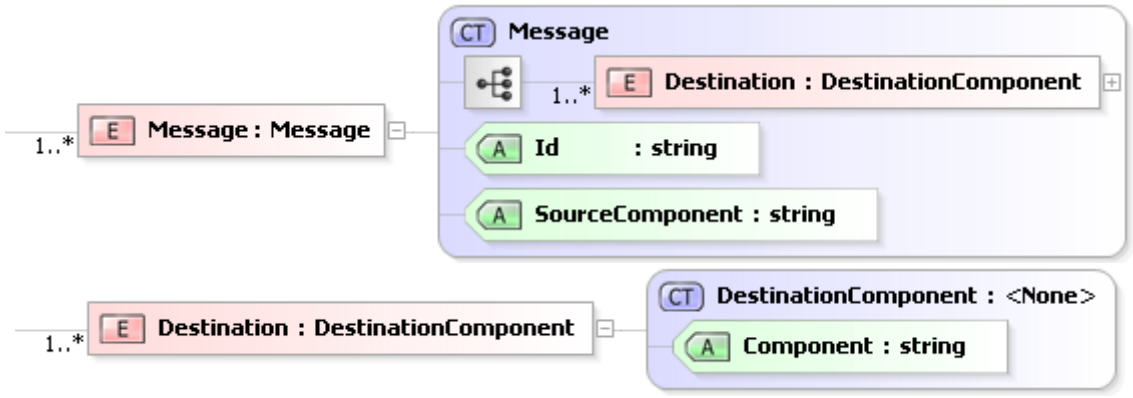


Figure 5.3.13: Message element for MCOSF input.

5.3.4 Output Schema

The output data elements are defined, like input, in a hierarchical structure. The trivial elements of this hierarchy will not be defined in this section, instead they can be found in Annex A.2.

Moreover, the elements similar in the input and output files are not redefined. The interested reader may refer to Section 5.3.3.

The root element of the hierarchy is defined in the figure below. It defines a collection of modes of operations embellished with the new information of *transition modes* and *flexibility information*. The element `ModesOfOperation` in the output file is the same as defined in Section 5.3.3.3.

Figure 5.3.14: Project root in MCOSF output.

5.3.4.1 Component Schedule

The component schedule in the output file is similar to the one in the input file. An attribute is added to provide a notion of flexibility in executing this component. The flexibility defines how much this component can be delayed before it misses its deadline. The attribute is specified in microseconds.

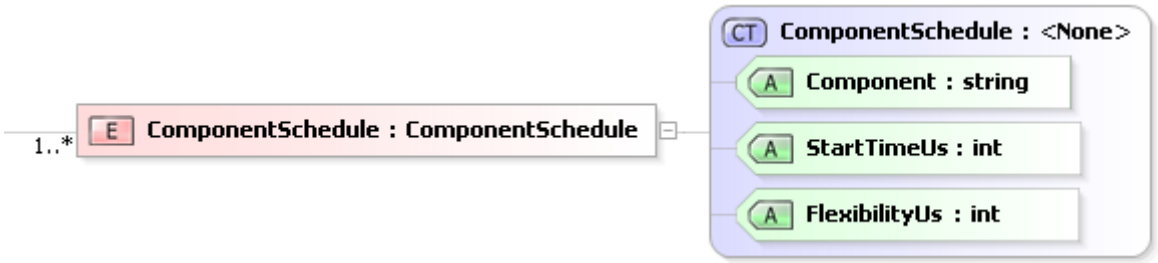


Figure 5.3.15: Component Schedule element in MCOSF output.

5.3.4.2 Slot

The partition slot in the output file is similar to the one in the input file. An attribute is added which is defined as follows:

- `ModeChangeBlackOut`: It defines if the slot is a blackout slot (see Section 5.3.2), i.e., a Boolean flag.

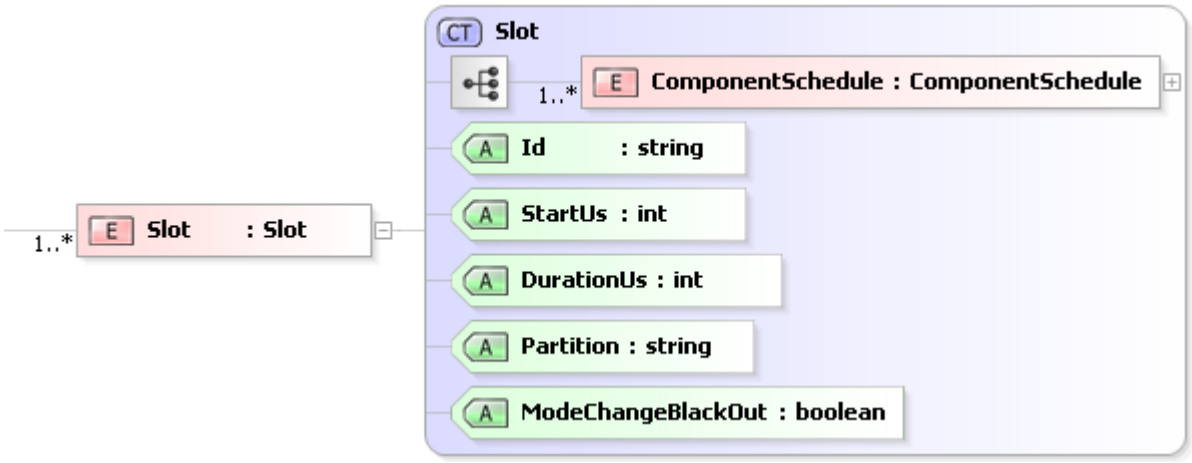


Figure 5.3.16: Slot element in MCOSF output.

5.4 Extension of Safety Compliance Metamodel

5.4.1 Overview

Extensions to Safety Compliance Metamodel are devoted to enhance WP4 safety toolset with the ability of:

- ***Producing safety argumentations*** according to GSN Standard [20] for a specific system and platforms according to WP5 safety certification methodology and modular safety case patterns [14][15][16].
- ***Refining safety rules*** with more refined checks in specific software/hardware elements.

The enhancements consist of a set of new classes and extensions to already existing classes of safety related models (that will be exploited in the final implementation of the safety toolset [12]). These new classes and extensions enable:

- *Representing GSN standard based argumentations*, allowing integrating together:
  - Safety Compliance Metamodel
  - Safety Compliance Constraint Checker and Rule checker with GSN arguments.
  - Existing AF3 GSN classes and AF3 GSN tree editor.

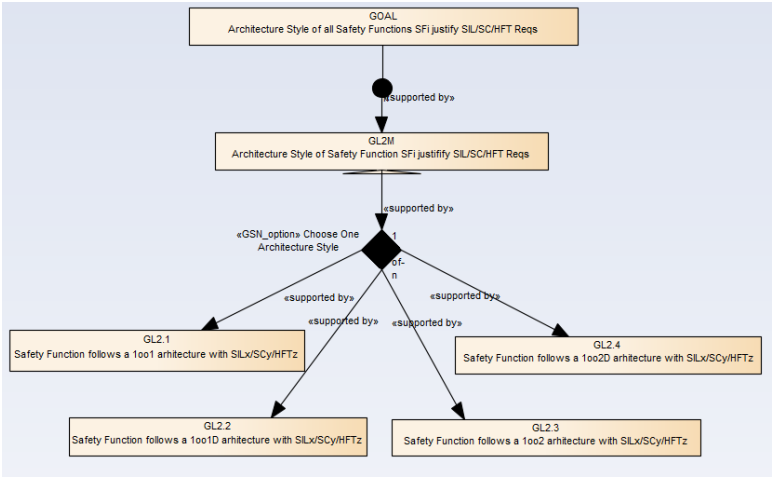


Figure 5.4.1: GSN argumentation example in Enterprise Architect.

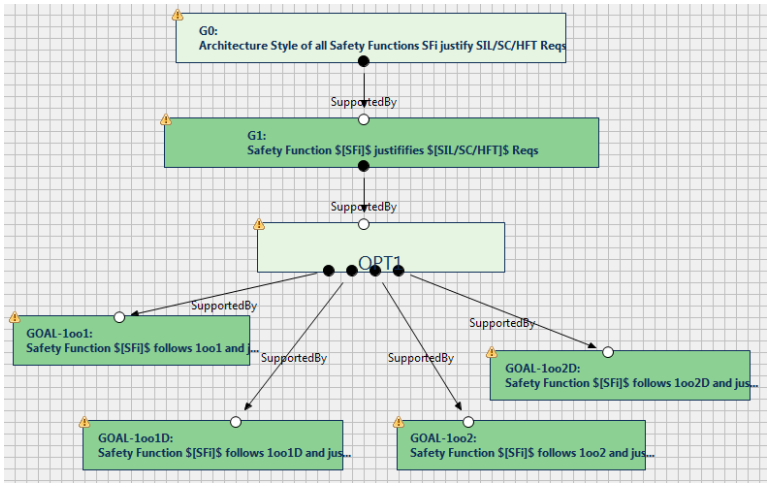


Figure 5.4.2: Example of GSN argumentation in AF3.

- *Building a library of IEC 61508 based Safety Cases* to produce specific argumentations for a specific product (see Figure 5.4.3). The WP4 toolset will be able to instantiate arguments, apply safety constraints checks and rule checks to produce instances of the GSN argument patterns of WP5, etc.

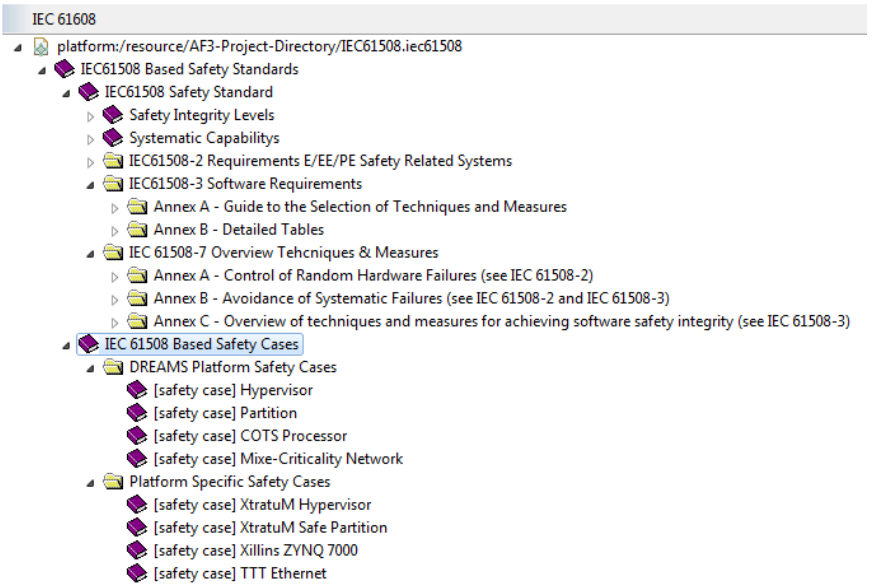


Figure 5.4.3: Library of Safety Case Patterns.

- *Enhancing the verification capabilities* of the WP4 Toolset Safety Constraints and Rules checker by defining entities more closely related to rules of IEC 61058, as for example:
  - If IEC 61508 Double-RAM based techniques are selected for a given system it will be possible to define the precise RAM item involved in the rule and their required features, thus, allowing more refined checking.
  - The Usage-Constraints in current Safety Manual will be enhanced so that WP4 can verify usage properties of a given system.

5.4.2 New classes for Safety Case Argumentations

Name	Safety Compliance Metamodel GSN Standard Extensions
Description	GSN standard based classes to implement Safety Case Argumentations
Plugin	eu.dreamsproject.ikerlan.safetystandards
Packages	eu.dreamsproject.ikerlan.gsn eu.dreamsproject.ikerlan.safetycase.argumentation
Dependencies	org.fortiss.af3.*

These classes are used to represent GSN argumentations of Safety Cases. The GSN graphs can be imported from other GSN editor tools (import from Enterprise Architect has been implemented). The Safety Compliance Constraints and Rules Checker construct GSN graphs as result of the checks. These graphs are later used to generate part of the Safety Case Report.

AF3 Safety Case package and its GSN Editor tool are used as privileged input/output tool for the safety constraints and rules checker. Figure 5.4.4 shows this concept.

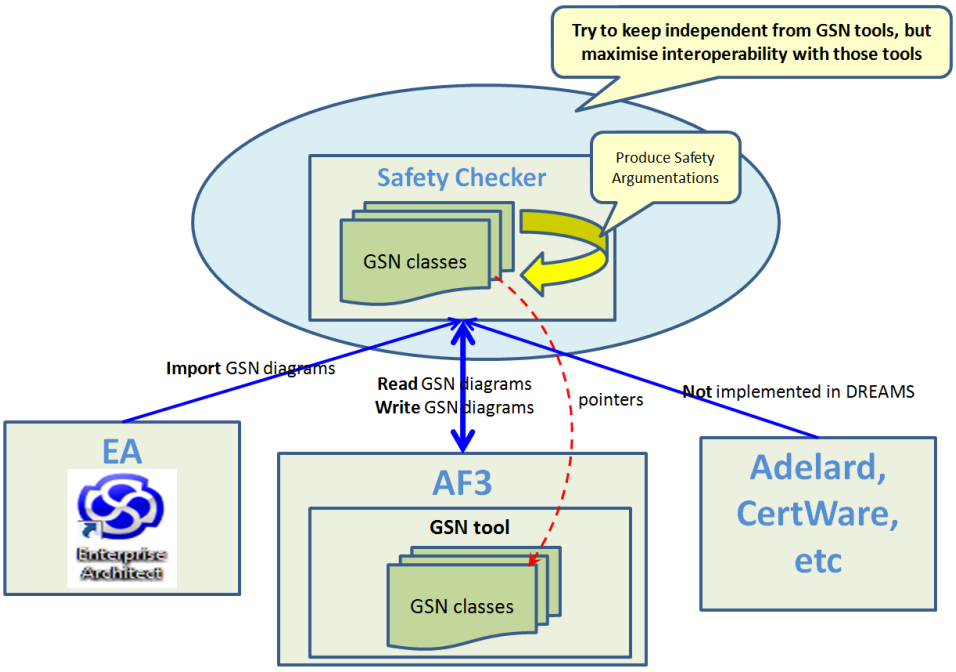


Figure 5.4.4: Safety Compliance Metamodel GSN classes and its relationship to GSN tools.

5.4.2.1 Package eu.dreamsproject.ikerlan.gsn

This package contains the classes to represents GSN standard based graphs. Figure 5.4.5 shows the classes and the main attributes of each class.

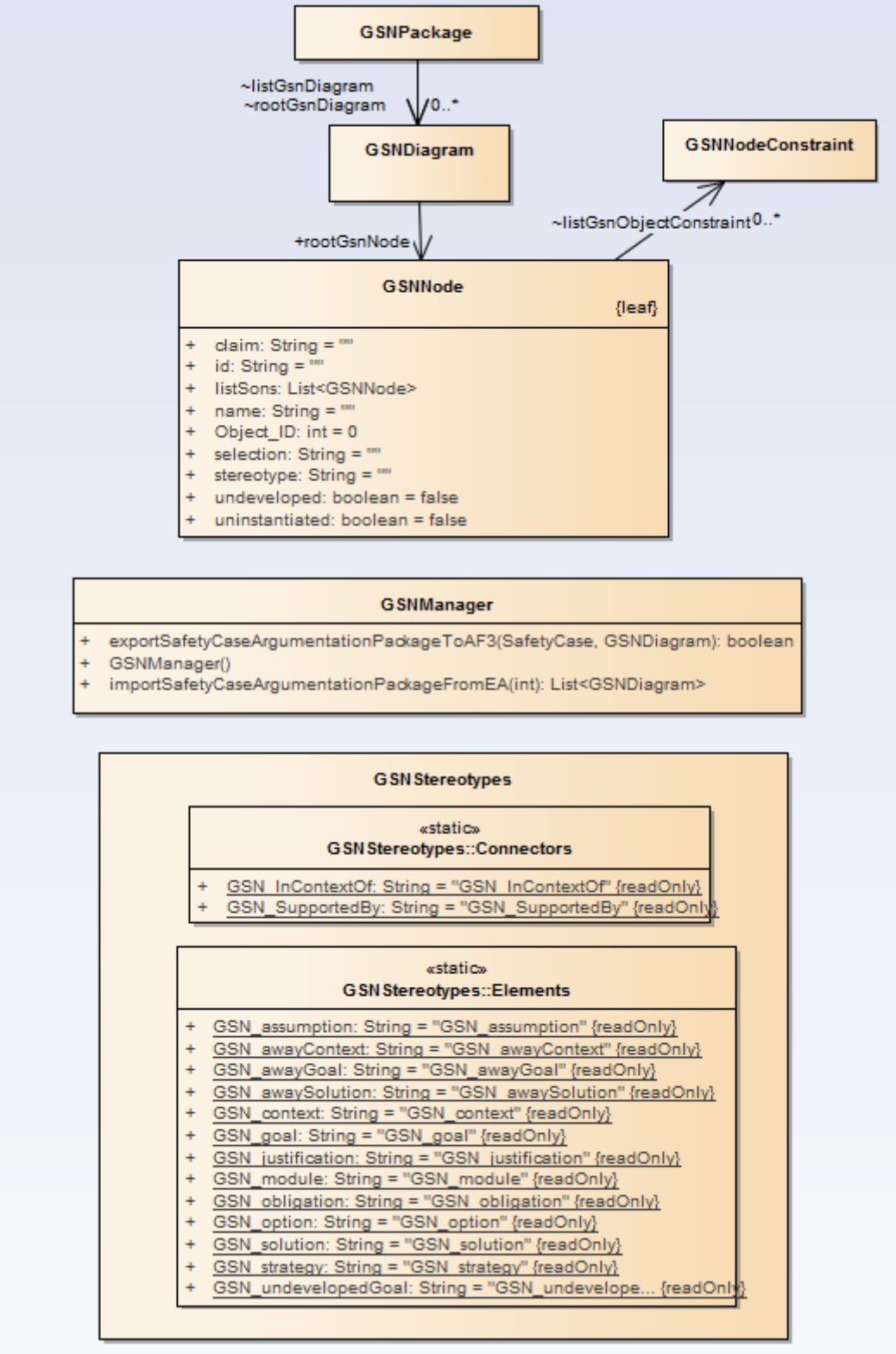


Figure 5.4.5: Main classes of package eu.dreamsproject.ikerlan.gsn.

The description of each class follows:

- GSNDiagram
  - This class represents a GSN standard based diagram.
  - **Attributes:**
    - argumentModuleAF3: Link to ArgumentModule that represents this diagram in AF3.
    - Diagram_ID: Unique identification of the diagram.
    - listAwayGsnDiagram: List of diagrams linked by away elements referenced by this diagram.
    - name: Name of the diagram.
    - rootGsnNode: Root GSN node of diagram. This node is the root goal.
  - **Operations:**
    - GSNDiagram (): Constructs an empty GSN Diagram.
    - printToConsole (): Prints the diagram description and nodes to the console.
    - printToConsoleBody (): Auxiliary recursive function of printToConsole function.
- GSNManager
  - This class manages GSN diagrams. Implements functions to import GSN diagrams from Enterprise Architect and AF3, and implements functions to export GSN diagrams to AF3.
  - **Attributes:**
    - argumentModule: Auxiliary argument module.
    - dbName: DataBase name of Enterprise Architect Repository of GSN Diagrams.
    - dbURL: Connection to the server hosting the repository.
    - MIN_GSN_AF3_ELEMENT_HEIGHT: Graphic constant.
  - **Operations:**
    - createGsnDiagramArgumentElementAF3 (): Auxiliary function to create argumentElements in AF3.
    - createGsnDiagramArgumentElementLinksAF3 (): Auxiliary function to create argumentElements connections in AF3.
    - createGsnDiagramAwayArgumentReferencesAF3 (): Auxiliary function to create away references to argumentElements in AF3.
    - exportGsnDiagramToAF3 (): This function exports to AF3 under the safetyCase all the GSNDiagram passed in the list.
    - exportSafetyCaseArgumentationPackageToAF3 (): This function exports to AF3 under the safety case the GSNDiagram passed as parameter
    - GSNManager (): Constructs an empty GSNManager.
    - importSafetyCaseArgumentationPackageFromEA (): This function imports form Enterprise Architect all the GSN Diagrams contained in the package which ID is passes as parameter.
- GSNNode
  - This class represents a GSN Node in a GSN-standard based diagram.
  - **Attributes:**
    - argumentElementAF3: Argument Element in AF3 that corresponds to the node.
    - awayArgumentElementAF3: Away Argument Element in AF3 that corresponds to the node.
    - awayparent: Away parent GSN node in the hierarchy in GSN Graph

- awaySon: Away son in the hierarchy in GSN Graph (reference to an away goal).
- claim: Claim of the node.
- id: GSN ID of the node.
- listAF3Node: List of AF3 elements (Components, Nodes, Tiles, Hypervisors, Partitions, etc.) related to the node.
- listGsnObjectConstraint: List of GSN Constraints.
- listSafetyComplianceNode: List of Safety Compliance elements related to the node.
- listSons: List of sub nodes in the hierarchy in GSN Graph
- maxNumChoicesRequired: Cardinality for option nodes - maximum number of allowed choices.
- minNumChoicesRequired: Cardinality for option nodes - minimum number of required choices.
- multipleInstanceXOffset: Offset to apply for second and later instances.
- multipleInstanceYOffset: Offset to apply for second and later instances.
- name: Name of the node.
- Object_ID: Unique identification of the node.
- parent: Parent GSN node in the hierarchy in GSN Graph.
- rectBottom: Graphic position of the node (may come from Enterprise Architect or AF3).
- rectLeft: Graphic position of the node (may come from Enterprise Architect or AF3).
- rectRight: Graphic position of the node (may come from Enterprise Architect or AF3).
- rectTop: Graphic position of the node (may come from Enterprise Architect or AF3).
- selection: Cardinality of the relation in textual form (e.g., 1-of-n) in case of option nodes.
- stereotype: Stereotype of the node (Goal, Strategy, Solution, Assumption, etc.).
- undeveloped: True if the node is undeveloped and false otherwise
- uninstantiated: True is the node is not instantiated and false otherwise
- xOffset: Graphic offset. This is generated when one node is split in multiple instances.
- yOffset: Graphic offset. This is generated when one node is split in multiple instances.

- **Operations:**
  - addConstraint (): Adds a constraint (in textual form) to the node.
  - copyBasePropertiesFromArgumentElement (): Copy base properties from ArgumentElement to a GSN Node. Used when importing AF3 GSN diagrams.
  - createArgumentElementAF3 (): Creates the AF3 Argument Element.
  - createArgumentElementLinksAF3 (): Creates the links of the AF3 Argument Element.
  - GSNNode (): Constructs an empty GSN Node.
  - setAwayArgumentReferenceAF3 (): Sets an away argument reference to the AF3 Argument Element.
  - traceToConsole (): Miscellaneous.

- GSNNodeConstraint
  - This class represents a generic GSN Node constraint imposed to a node
  - **Attributes:**
    - constraintExpr: Constraint in textual form.
  - **Operations:**
    - getConstraint (): Gets the text of the constraint.
    - GSN_objectConstraint (): Construct an empty constraint.
    - setConstraint (): Sets the text of the constraint.
- GSNPackage
  - This class represents a set of GSN Diagrams and a list of sub-GSNPackages. It is not part of the GSN standard.
  - **Attributes:**
    - listGsnDiagram.
    - listGSNPackage: GSNPackage contained.
    - packageID: Package ID.
    - rootGsnDiagram: GSN diagrams contained.
  - **Operations:**
    - GSNPackage (): Constructs an empty GSNPackage.
- GSNStereotypes
  - This class contains the stereotypes of Connectors and Elements types defined by the GSN Standard. It contains the elements of the following types (see below):
    - Connectors
    - Elements
- Connectors
  - This class contains the stereotypes of Connectors types defined by the GSN standard.
  - **Attributes:**
    - GSN_InContextOf.
    - GSN_SupportedBy.
- Elements
  - This class contains the stereotypes of Element types defined by the GSN standard.
  - **Attributes:**
    - GSN_assumption.
    - GSN_awayContext.
    - GSN_awayGoal.
    - GSN_awaySolution.
    - GSN_context.
    - GSN_goal.
    - GSN_justification.
    - GSN_module.
    - GSN_obligation.
    - GSN_option.
    - GSN_solution.
    - GSN_strategy.
    - GSN_undevelopedGoal.

5.4.2.2 Package eu.dreamsproject.ikerlan.safetycase.argumentation

This package contains the class to import, export, generate and manage GSN standard based argumentations for specific systems generated with the DREAMS toolset. Figure 5.4.6 shows the class and the main attributes of the class.

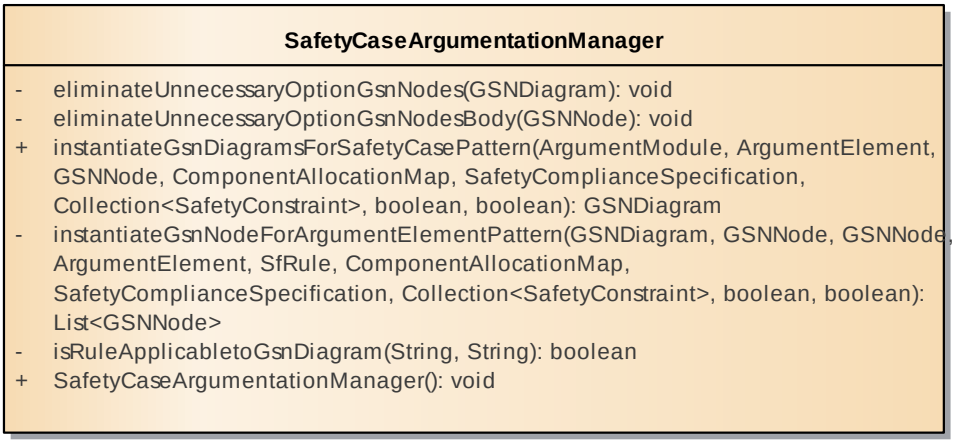


Figure 5.4.6: Main classes of package eu.dreamsproject.ikerlan.safetycase.argumentation.

The description of each class follows:

- `SafetyCaseArgumentationManager`:
  - This class contains the argument generator logic
  - **Attributes:**
  - **Operations:**
    - `instantiateGsnDiagramsForSafetyCasePattern` (): This function instantiates a given GSN argumentation pattern into an instance of argumentation.
    - `instantiateGsnNodeForArgumentElementPattern` (): This function instantiates a node of a GSN Node pattern into an instance of GSN Node. Recursively calls to the sub nodes instantiation
    - `isRuleApplicabletoGsnDiagram` (): Auxiliary functions that checks if a given safety rule is applicable to a GSN Diagram
    - `eliminateUnnecessaryOptionGsnNodes` (): This function eliminates from GSN diagrams those option nodes which selection has been made
    - `eliminateUnnecessaryOptionGsnNodesBody` (): Auxiliary recursive function for eliminateUnnecessaryOptionGsnNodes function
    - `SafetyCaseArgumentationManager` ():

5.4.3 Class members added to existing classes

Name	Safety Compliance Metamodel GSN Standard Extensions
Description	`Class members added to existing Safety Compliance Model classes`
Plugin	`eu.dreamsproject.ikerlan.safetystandards`
Packages	`eu.dreamsproject.ikerlan.safetystandards.SafetyCompliance`
Dependencies	`org.fortiss.af3.*`

5.4.3.1 Package eu.dreamsproject.ikerlan.safetystandards.SafetyCompliance

This package contains the Safety Compliance Metamodel classes that have been modified to implement a refining of safety rules with more fine-grained checks in specific software/hardware elements. Figure 5.4.7 shows the classes and the main attributes of the classes.

Figure 5.4.7: Main modifications to classes of package eu.dreamsproject.ikerlan.safetystandards.SafetyCompliance.

This package contains the following classes (only added class member are listed):

- `IEC61508_2_RandomFailureControlTechniqueMeasureRef`
  - **Attributes:**
    - `listHierarchicElementBase`: List of DREAMS Platform AF3 Elements involved in the rule.
  - **Operations:**
    - `getListHierarchicElementBase ()`: Returns the list of DREAMS Platform AF3 Elements involved in the rule.

- `IEC61508_2_SystematicFailureAvoidanceTechniqueMeasureRef`:
  - **Attributes:**
    - `listHierarchicElementBase`: List of DREAMS Platform AF3 Elements involved in the rule.
  - **Operations:**
    - `getListHierarchicElementBase ()`: Returns the list of DREAMS Platform AF3 Elements involved in the rule.
- `IEC61508_2_SystematicFailureControlTechniqueMeasureRef`:
  - **Attributes:**
    - `listHierarchicElementBase`: List of DREAMS Platform AF3 Elements involved in the rule.
  - **Operations:**
    - `getListHierarchicElementBase ()`: Returns the list of DREAMS Platform AF3 Elements involved in the rule.
- `IEC61508_3_SystematicFailureAvoidanceTechniqueMeasureRef`:
  - **Attributes:**
    - `listHierarchicElementBase`: List of DREAMS Platform AF3 Elements involved in the rule.
  - **Operations:**
    - `getListHierarchicElementBase ()`: Returns the list of DREAMS Platform AF3 Elements involved in the rule.
- `SafetyManualImpl`
  - **Attributes:**
    - `safetyCase`: Safety case corresponding to the system.
- `SafetyCaseImpl`
  - **Attributes:**
    - `safetyCaseGsn`: Safety case GSN object.
    - `safetyCaseReport`: Safety case report generated.
- `SafetyCaseGsnImpl`:
  - **Attributes:**
    - `destinationSafetyCase`: AF3 Safety Case containing the argumentation generated by the Safety Compliance Model.
    - `sourceRootArgumentModulePattern`: AF3 root Argument Module pattern that has been taken as pattern to produce the argumentation.
    - `sourceRootArgumentModuleGoalPattern`: AF3 root Goal of the root Argument Module pattern that has been taken as pattern to produce the argumentation.

6 Bibliography

[1] D1.2.1 – *Architectural Style of DREAMS*, DREAMS Consortium, 7/2014.

[2] D1.4.1 – *Metamodels for Application and Platform*, DREAMS Consortium, 3/2015.

[3] D2.1.3 – *RT-level design specifications of a) virtualization and memory interleaving support of the Spidergon STNoC backbone at the network interface layer and b) a bus-to-noc bridge for seamlessly interconnecting STNoC to the network gateway from WP3*, DREAMS Consortium, 9/2015.

[4] D2.3.1 - *XtratuM support of enhanced hypervisor layer services: description and interface*, DREAMS Consortium, 3/2015.

[5] D3.1.2 – *First Implementation of Mixed-Criticality Cluster Communication Services*, DREAMS Consortium, 3/2015.

[6] D3.2.2 – *First Implementation of Global Resource Management Services*, DREAMS Consortium, 3/2015.

[7] D3.3.2 – *First Implementation of Cluster-level Safety and Security services*, DREAMS Consortium, 3/2015.

[8] D4.1.2 – *Definition of Offline Adaptation Strategies for Mixed-Criticality and Initial Implementation*, DREAMS Consortium, 3/2015.

[9] D4.1.3 – *Final Implementation and Improvement of the Offline Adaptation Strategies for Mixed-Criticality*, DREAMS Consortium, 7/2016.

[10] D4.2.1 – *Specification and first implementation of a platform configuration files generator*, DREAMS Consortium, 11/2015.

[11] D4.2.2 – *Final implementation of a platform configuration files generator*, DREAMS Consortium, 7/2016.

[12] D4.3.3 – *Final Implementation and Improvement of Variability and Testing Techniques for Mixed-criticality Systems*, DREAMS Consortium, 7/2016.

[13] D4.4.1 – *Tools feature map and interoperability capabilities*, DREAMS Consortium, 7/2016.

[14] D5.1.2 – *Modular Safety Case for COTS processor*, DREAMS Consortium, 7/2015.

[15] D5.1.1 – *Modular Safety Case for Hypervisor,* DREAMS Consortium, 1/2015.

[16] D5.1.3 – *Modular safety case for selected Mixed Criticality Networks,* DREAMS Consortium, 11/2015.

[17] D5.2.1 – *Specification of Simulation Framework*, DREAMS Consortium, 03/2015.

[18] D5.2.2 – *Prototype implementation of simulation framework for DREAMS architecture*, DREAMS Consortium, 11/2015.

[19] D5.2.3 – *Fault injection framework*, DREAMS consortium, 7/2016.

[20] *GSN Community Standard Version 1, 11/*2011, Available: http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf.

[21] *XtratuM Software User Manual*. XM-ARM. (Reference: 14-035-03.005.sum.01), 2016.

[22] *Xoncrete User Manual*. Version 3.0. (Reference: fnts-xe-um-14b). Fent Innovative Software Solutions, 05/2016.

[23] *Schema documentation for eprj-2.6.1.xsd*. Fent Innovative Software Solutions, 2016.

[24] *Language syntax, semantics, metamodel V2*, TIMMO-2-USE Project, 2012.

A. Annex

This annex lists the following XML schemas described in this document:

- MCOSF Input Schema (documentation: see Section 5.3.3)
- MCOSF Output Schema (documentation: see Section 5.3.4)

A.1 MCOSF Input Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
	xmlns="http://rts.eit.uni-kl.de/projects/dreams/mcosf_input"
	xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
	xmlns:xs="http://www.w3.org/2001/XMLSchema"
	attributeFormDefault="unqualified"
	ecore:nsPrefix=""
	ecore:nsURI="http://rts.eit.uni-kl.de/projects/dreams/mcosf_input"
	ecore:package="de.unikl.eit.rts.mcosf.input"
	elementFormDefault="qualified"
	targetNamespace="http://rts.eit.uni-kl.de/projects/dreams/mcosf_input">

	<!-->Defining all ENUMs</!-->
	<xs:simpleType name="ModeTypeEnum">
		<xs:annotation>
			<xs:documentation>Defines type of mode. A mode/schedule which executes repetitively until
			there is a switch request is called NOMINAL mode. A mode/schedule which executes at most
			once is called TRANSITION mode.</xs:documentation>
		</xs:annotation>
		<xs:restriction base="xs:string">
			<xs:enumeration value="NOMINAL"/>
			<xs:enumeration value="TRANSITION"/>
		</xs:restriction>
	</xs:simpleType>

	<xs:simpleType name="SafetyLevelEnum">
		<xs:annotation>
```

```
            <xs:documentation>Defines safety level confirming to DO-178B.</xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:string">
            <xs:enumeration value="DAL-A"/>
            <xs:enumeration value="DAL-B"/>
            <xs:enumeration value="DAL-C"/>
            <xs:enumeration value="DAL-D"/>
            <xs:enumeration value="DAL-E"/>
      </xs:restriction>
</xs:simpleType>

<!-->Defining all collections</!-->
<xs:complexType name="PlatformArchitecture">
      <xs:annotation>
            <xs:documentation>Defines the platform architecture with nodes.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="Node" type="Node" maxOccurs="unbounded"/>
      </xs:sequence>
</xs:complexType>

<xs:complexType name="Node">
      <xs:annotation>
            <xs:documentation>Defines a node with tiles.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="Tile" type="Tile" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Tile">
      <xs:annotation>
            <xs:documentation>Defines a tile with cores.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="Core" type="Core" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Name" type="xs:string" use="required"/>
```

```
</xs:complexType>

<xs:complexType name="Core">
      <xs:annotation>
            <xs:documentation>Defines core/processing unit.</xs:documentation>
      </xs:annotation>
      <xs:attribute type="xs:string" name="Name" use="required"/>
</xs:complexType>

<!--> ********************************************************************************** </!-->
<xs:complexType name="SystemSoftware">
      <xs:annotation>
            <xs:documentation>Defines a collection of partitions.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="Partition" type="Partition" maxOccurs="unbounded"/>
      </xs:sequence>
</xs:complexType>

<xs:complexType name="Partition">
      <xs:annotation>
            <xs:documentation>Defines properties of a partition. The allocation parameter defines the
            core/processor/tile/node on which it is allocated. According to the T1.6, 4.3 and 4.4 joint
            meetings, a partition is allocated to a tile in the metamodel. But MCOSF will be invoked
            after Xoncrete which means that the information of partition to core allocation is available
            in the metamodel.</xs:documentation>
      </xs:annotation>
      <xs:attribute type="xs:string" name="Name" use="required"/>
</xs:complexType>

<!--> ********************************************************************************** </!-->
<xs:complexType name="ModesOfOperation">
      <xs:annotation>
            <xs:documentation>Provides information related to modes of operation of the
            system.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="LocalMode" type="LocalMode" maxOccurs="unbounded"/>
            <xs:element name="GlobalMode" type="GlobalMode" maxOccurs="unbounded"/>
```

```
            <xs:element name="ModeTransition" type="ModeTransition" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="InitialGlobalMode" type="xs:string"/>
</xs:complexType>

<xs:complexType name="CoreSchedule">
      <xs:annotation>
            <xs:documentation>Defines a set of slots on a core for a certain local mode of
            operation.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="Slot" type="Slot" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Core" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="LocalMode">
      <xs:annotation>
            <xs:documentation>Defines a set of core schedules for a certain local mode of operation.
            Note that there may exist any number of local modes for a specific tile. Time units are
            microseconds.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="CoreSchedule" type="CoreSchedule" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Name" type="xs:string" use="required"/>
      <xs:attribute name="Type" type="ModeTypeEnum" use="required"/>
      <xs:attribute name="Tile" type="xs:string" use="required"/>
      <xs:attribute name="MafUs" type="xs:int" use="required"/>
</xs:complexType>

<xs:complexType name="Slot">
      <xs:annotation>
            <xs:documentation>Defines a slot (i.e., a reservation for a partition). Note that this assumes
            that the partition is allocated to a core or all the cores on this tile are identical (i.e.
            homogeneous processor), otherwise it would be difficult to see where does this partition execute.
            If the assumption is not correct, an attribute called Core must be appended to this complexType
            and the one in the output XSD. Time units are microseconds.</xs:documentation>
      </xs:annotation>
```

```
        <xs:sequence>
            <xs:element name="ComponentSchedule" type="ComponentSchedule" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="Id" type="xs:string" use="required"/>
        <xs:attribute name="StartUs" type="xs:int" use="required"/>
        <xs:attribute name="DurationUs" type="xs:int" use="required"/>
        <xs:attribute name="Partition" type="xs:string" use="required"/>
    </xs:complexType>

    <xs:complexType name="ComponentSchedule">
        <xs:annotation>
            <xs:documentation>Defines schedule of a task/component within a partition. StartTimeUs
            is relative phase from the start of the MAF. Time units are microseconds.</xs:documentation>
        </xs:annotation>
        <xs:attribute type="xs:string" name="Component" use="required"/>
        <xs:attribute type="xs:int" name="StartTimeUs" use="required"/>
    </xs:complexType>

    <xs:complexType name="GlobalMode">
        <xs:annotation>
            <xs:documentation>Defines a global mode as a collection of local modes for all the available
            nodes in the PlatformArchitecture. Note that all the local modes of a global mode must be
            of the same ModeTypeEnum.</xs:documentation>
        </xs:annotation>
        <xs:sequence>
            <xs:element name="Local" type="LocalModeName" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute type="xs:string" name="Name" use="required"/>
    </xs:complexType>

    <xs:complexType name="LocalModeName">
        <xs:annotation>
            <xs:documentation>Defines a single local mode as part of a global mode.</xs:documentation>
        </xs:annotation>
        <xs:attribute type="xs:string" name="Mode" use="required"/>
    </xs:complexType>

    <xs:complexType name="ModeTransition">
        <xs:annotation>
```

```
            <xs:documentation>Defines a mode transition.</xs:documentation>
      </xs:annotation>
      <xs:attribute type="xs:string" name="SourceMode" use="required"/>
      <xs:attribute type="xs:string" name="DestinationMode" use="required"/>
</xs:complexType>

<!--> ********************************************************************************** </!-->
<xs:complexType name="ComponentsList">
      <xs:annotation>
            <xs:documentation>Defines a collection of tasks/components.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="Component" type="Component" maxOccurs="unbounded"/>
      </xs:sequence>
</xs:complexType>

<xs:complexType name="Component">
      <xs:annotation>
            <xs:documentation>Defines properties of a task/component. Jitter is the execution jitter (-1 when
            not specified). Offset is not the release phase of the task. It is the offset after which the
            task should start execution, even when it is ready (used for latency constraints, aka ETEF
            offset). Time units are microseconds.</xs:documentation>
      </xs:annotation>
      <xs:attribute type="xs:string" name="Name" use="required"/>
      <xs:attribute type="xs:int" name="WcetUs" use="required"/>
      <xs:attribute type="xs:int" name="PeriodUs" use="required"/>
      <xs:attribute type="xs:int" name="MinExecutionOffsetUs" use="required"/>
      <xs:attribute type="xs:int" name="ReactionConstraintUs" use="required"/>
      <xs:attribute type="xs:string" name="Partition" use="required"/>
      <xs:attribute type="SafetyLevelEnum" name="SafetyLevel" use="required"/>
</xs:complexType>

<!--> ********************************************************************************** </!-->
<xs:complexType name="MessageList">
      <xs:annotation>
            <xs:documentation>Defines a collection of messages.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="Message" type="Message" maxOccurs="unbounded"/>
```

```
		</xs:sequence>
	</xs:complexType>

	<xs:complexType name="DestinationComponent">
		<xs:annotation>
			<xs:documentation>Defines the destination task/component for a message.</xs:documentation>
		</xs:annotation>
		<xs:attribute name="Component" type="xs:string" use="required"/>
	</xs:complexType>

	<xs:complexType name="Message">
		<xs:annotation>
			<xs:documentation>Defines properties of a message. Time units are microseconds.</xs:documentation>
		</xs:annotation>
		<xs:sequence>
			<xs:element name="Destination" type="DestinationComponent" maxOccurs="unbounded"/>
		</xs:sequence>
		<xs:attribute name="Id" type="xs:string" use="required"/>
		<xs:attribute name="SourceComponent" type="xs:string" use="required"/>
	</xs:complexType>

	<!--> ********************************************************************************** </!-->
	<xs:complexType name="Project">
		<xs:annotation>
			<xs:documentation>Defines the complete system.</xs:documentation>
		</xs:annotation>
		<xs:sequence>
			<xs:element name="PlatformArchitecture" type="PlatformArchitecture"/>
			<xs:element name="SystemSoftware" type="SystemSoftware"/>
			<xs:element name="ModesOfOperation" type="ModesOfOperation"/>
			<xs:element name="ComponentsList" type="ComponentsList"/>
			<xs:element name="MessagesList" type="MessageList" minOccurs="0"/>
		</xs:sequence>
		<xs:attribute type="xs:string" name="Name" use="required"/>
	</xs:complexType>

	<xs:element name="Project" type="Project"/>
</xs:schema>
```

A.2 MCOSF Output Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
	xmlns="http://rts.eit.uni-kl.de/projects/dreams/mcosf_output"
	xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
	xmlns:xs="http://www.w3.org/2001/XMLSchema"
	attributeFormDefault="unqualified"
	ecore:nsPrefix=""
	ecore:nsURI="http://rts.eit.uni-kl.de/projects/dreams/mcosf_output"
	ecore:package="de.unikl.eit.rts.mcosf.output"
	elementFormDefault="qualified"
	targetNamespace="http://rts.eit.uni-kl.de/projects/dreams/mcosf_output">

	<!-->Defining all ENUMs</!-->
	<xs:simpleType name="ModeTypeEnum">
		<xs:annotation>
			<xs:documentation>Defines type of mode. A mode/schedule which executes repetitively until there is
			a switch request is called NOMINAL mode. A mode/schedule which executes at most once is called
			TRANSITION mode.</xs:documentation>
		</xs:annotation>
		<xs:restriction base="xs:string">
			<xs:enumeration value="NOMINAL"></xs:enumeration>
			<xs:enumeration value="TRANSITION"></xs:enumeration>
		</xs:restriction>
	</xs:simpleType>

	<!-->Defining complex types</!-->
	<xs:complexType name="ModesOfOperation">
		<xs:annotation>
			<xs:documentation>Provides information related to modes of operation of the
			system.</xs:documentation>
		</xs:annotation>
		<xs:sequence>
			<xs:element name="LocalMode" type="LocalMode" maxOccurs="unbounded"/>
			<xs:element name="GlobalMode" type="GlobalMode" maxOccurs="unbounded"/>
			<xs:element name="ModeTransition" type="ModeTransition" minOccurs="0" maxOccurs="unbounded"/>
		</xs:sequence>
		<xs:attribute name="InitialGlobalMode" type="xs:string"/>
```

```
</xs:complexType>

<xs:complexType name="CoreSchedule">
      <xs:annotation>
            <xs:documentation>Defines a set of slots on a core for a certain local mode of
            operation.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="Slot" type="Slot" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Core" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="LocalMode">
      <xs:annotation>
            <xs:documentation>Defines a set of core schedules for a certain local mode of operation.
            Note that there may exist any number of local modes for a specific tile. Time units are
            microseconds.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="CoreSchedule" type="CoreSchedule" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Name" type="xs:string" use="required"/>
      <xs:attribute name="Type" type="ModeTypeEnum" use="required"/>
      <xs:attribute name="Tile" type="xs:string" use="required"/>
      <xs:attribute name="MafUs" type="xs:int" use="required"/>
</xs:complexType>

<xs:complexType name="Slot">
      <xs:annotation>
            <xs:documentation>Defines a partition slot with blackout (i.e., mode switch is not allowed
            during this slot).</xs:documentation>
      </xs:annotation>
      <xs:sequence>
            <xs:element name="ComponentSchedule" type="ComponentSchedule" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Id" type="xs:string" use="required"/>
      <xs:attribute name="StartUs" type="xs:int" use="required"/>
      <xs:attribute name="DurationUs" type="xs:int" use="required"/>
```

```
		<xs:attribute name="Partition" type="xs:string" use="required"/>
		<xs:attribute name="ModeChangeBlackOut" type="xs:boolean" use="required"/>
	</xs:complexType>

	<xs:complexType name="ComponentSchedule">
		<xs:annotation>
			<xs:documentation>Defines schedule of a task within a partition. PhaseUs is relative phase from
			the start of the MAF. FlexibilityUs defines how much this task can be delayed from defined phase
			without missing any deadline in the system.</xs:documentation>
		</xs:annotation>
		<xs:attribute type="xs:string" name="Component" use="required" />
		<xs:attribute type="xs:int" name="StartTimeUs" use="required" />
		<xs:attribute type="xs:int" name="FlexibilityUs" use="required" />
	</xs:complexType>

	<xs:complexType name="GlobalMode">
		<xs:annotation>
			<xs:documentation>Defines a global mode as a collection of local modes for all the available nodes
			in the PlatformArchitecture. Note that all the local modes of a global mode must be of the same
			ModeTypeEnum.</xs:documentation>
		</xs:annotation>
		<xs:sequence>
			<xs:element name="Local" type="LocalModeName" maxOccurs="unbounded"/>
		</xs:sequence>
		<xs:attribute type="xs:string" name="Name" use="required" />
	</xs:complexType>

	<xs:complexType name="LocalModeName">
		<xs:annotation>
			<xs:documentation>Defines a single local mode as part of a global mode.</xs:documentation>
		</xs:annotation>
		<xs:attribute type="xs:string" name="Mode" use="required" />
	</xs:complexType>

	<xs:complexType name="ModeTransition">
		<xs:annotation>
			<xs:documentation>Defines a mode transition.</xs:documentation>
		</xs:annotation>
		<xs:attribute type="xs:string" name="SourceMode" use="required" />
```

```
            <xs:attribute type="xs:string" name="DestinationMode" use="required" />
      </xs:complexType>

      <xs:complexType name="Project">
            <xs:annotation>
                  <xs:documentation>Defines a complete schedule for the system.</xs:documentation>
            </xs:annotation>
            <xs:sequence>
                  <xs:element name="ModesOfOperation" type="ModesOfOperation" maxOccurs="1"/>
            </xs:sequence>
      </xs:complexType>

      <xs:element name="Project" type="Project" />
</xs:schema>
```