Modeling and Analysis Suite for Real Time Applications (MAST 2.0.0)

MAST Metamodel

By : César Cuevas Cuesta {cuevasce@unican.es]
José María Drake {drakej@unican.es]
Michael González Harbour {mgh@unican.es]
José Javier Gutiérrez {gutierjj@unican.es]
Patricia López Martínez {lopezpa@unican.es]
Julio Luis Medina {medinajl@unican.es]
José Carlos Palencia {palencij@unican.es]

Contents

1 MAST2 meta-model....5
2 MAST2 primitive types....6
3 MAST2 core classes....9
4 MAST2 model elements....13
4.1 PROCESSING RESOURCES....13
4.1.1 COMPUTING RESOURCES....14
4.1.2 NETWORKS....16
4.1.3 NETWORK SWITCHES AND NETWORK ROUTERS....25
4.2 TIMERS AND SYNCHRONIZATION OBJECTS....33
4.3 SCHEDULERS....37
4.3.1 SCHEDULING POLICIES....39
4.4 SCHEDULABLE RESOURCES....45
4.4.1 SCHEDULING PARAMETERS....49
4.4.2 SYNCHRONIZATION PARAMETERS....64
4.5 MUTUAL EXCLUSION RESOURCES....67
4.6 OPERATIONS....69
4.6.1 Overriden scheduling parameters....73
4.7 END-TO-END FLOWS....75
4.7.1 WORKLOAD EVENTS....79
4.7.2 OBSERVERS....82
4.7.3 STEP EVENT HANDLER....88
4.7.4 TIMED EVENT HANDLERS....89
4.7.5 FLOW CONTROL EVENT HANDLERS....91
4.7.6 MESSAGE EVENT HANDLERS....95

Index of diagrams

Figure 1 – Primitive types defined in MAST2 7
Figure 2 – The core classes in the MAST2 meta-model 10
Figure 3 – Top subclasses of Model_Element 14
Figure 4 – Hierarchy of processing resources 14
Figure 5 – Hierarchy of computing resources 15
Figure 6 – Hierarchy of networks 17
Figure 7 – Hierarchy of drivers 22
Figure 8 – The RTEP_Packet_Driver class 24
Figure 9 – Hierarchy of switches and routers 26
Figure 10 – The Clock_Synchronization_Object class 34
Figure 11 – Hierarchy of timers 35
Figure 12 – Hierarchy of schedulers 38
Figure 13 – Hierarchy of scheduling policies 40
Figure 14 – Hierarchy of schedulable resources 46
Figure 15 – Hierarchy of scheduling parameters 50
Figure 16 – Hierarchy of priority-based scheduling parameters 51
Figure 17 – Hierarchy of EDF scheduling parameters 56
Figure 18- Hierarchy of resource reservation params 57
Figure 19 – Hierarchy of timetable-driven scheduling parameters 64
Figure 20 – AFDX scheduling parameters 65
Figure 21 – Hierarchy of synchronization parameters 65
Figure 22 – Hierarchy of mutual exclusion resources 68
Figure 23 – Hierarchy of operations 70
Figure 24 – Hierarchy of overriden scheduling parameters 74
Figure 25 – End-to-end flow transaction model 76
Figure 26 – Hierarchy of workload events 80
Figure 27 – Hierarchy of observers 83
Figure 28 – The Step event handler 89
Figure 29 – The timed event handlers (Delay and Offset) 90
Figure 30 – The flow control event handlers 92
Figure 31 – The message flow event handlers 96

1 MAST2 meta-model

This document constitutes a readable version of the MAST 2.0.0 meta-model, formulated through the Ecore meta-modeling language.

MAST defines a model to describe the timing behaviour of real-time systems designed to be analyzable via schedulability analysis techniques. MAST also provides an open-source set of tools to perform schedulability analysis or other timing analysis, with the goal of assessing whether the system will be able to meet its timing requirements, and, via sensitivity analysis, how far or close is the system from meeting its timing requirements. Tools are also provided to help the designer in the assignment of scheduling parameters. By having an explicit model of the system and automatic analysis tools it is also possible to perform design space exploration. A discrete event simulator is also provided to obtain statistical performance information of the modeled system.

It is well known that testing cannot generally be used to make guarantees on the timing behaviour of the system even after it is built, because there is usually no guarantee that the worst case was tested. Schedulability analysis techniques are mathematical methods to obtain guarantees on the ability of the system to meet all its timing requirements.

Although schedulability analysis techniques have evolved a lot and may be complex to apply, the MAST toolset allows engineers developing real-time systems to apply advanced schedulability analysis techniques without the need to study all their intricate details, and thus concentrating the efforts in the design and tuning of the actual real-time system. This toolset can be used through all the phases of the design cycle from requirements analysis through design, implementation, test, and integration. Such an approach taken before the system is built increases the confidence that the system will be able to meet its timing requirements and therefore the risk of generating a design that fails to meet the timing requirements is reduced.

2 MAST2 primitive types

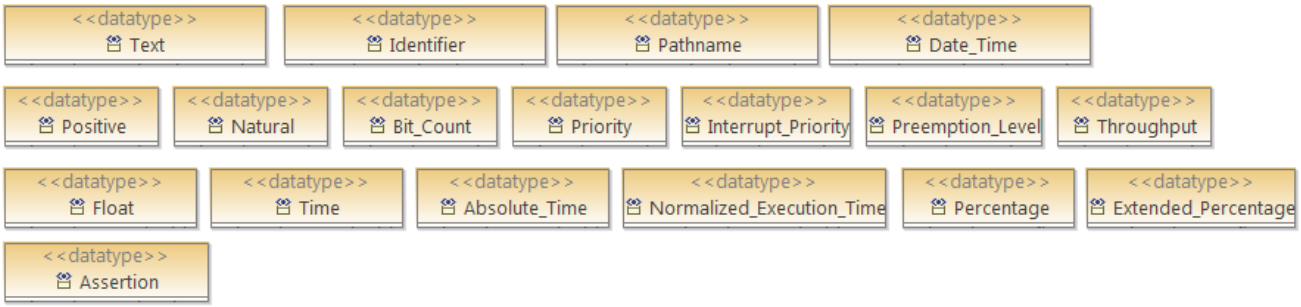

Figure 1 – Primitive types defined in MAST2

Primitive type **Text**

- Semantics. String of arbitrary characters, excluding the double quote character, and delimited within double quotes.

Primitive type **Identifier**

- Semantics. String of characters following these rules. Identifiers or names have similar rules as the Ada composite identifiers : they begin with a letter, followed by letters, digits, underscores ('_') or periods ('.') They can be expressed with or without quotes. A quoted name can be the same as one of the reserved words in MAST, which are the names of the non-<<abstract>> MAST model elements.

Primitive type **Pathname**

- Semantics. String representing a pathname or the URL of a file.

Primitive type **Date_Time**

- Semantics. String representing a date and time (hours, minutes and seconds) in the extended ISO 8601 format with no time zone YYYY-MM-DDThh :mm :ss (e.g., 1997-07-16T19 :20 :30).

Primitive type **Positive**

- Semantics. Integer positive number (excluding zero).

Primitive type **Natural**

- Semantics. Integer number that is greater than or equal to zero.

Primitive type **Bit_Count**

- Semantics. Long non negative integer that represents the size of a message in bits of information. It is converted to normalized execution time (i.e., transmission time) by dividing it by the throughput, measured in bits per time unit.

Primitive type **Priority**

- Semantics. Natural integer of model-defined range that must be within [1 .. 2^{15}-1], defining the scheduling priority of schedulable resources.

Primitive type **Interrupt_Priority**

- Semantics. Natural integer of model-defined range that must be within [1 .. 2^{15}-1], defining the scheduling priority of interrupt service routines.

Primitive type **Preemption_Level**

- Semantics. Natural integer in the range [1 .. 2^{15-1}], defining the preemption level of schedulable resources (threads) and mutual exclusion resources, used in the SRP protocol for mutually exclusive access to shared resources.

Primitive type **Throughput**

- Semantics. Long non negative integer that represents the transmission bandwidth of a communication network in bits per time unit in a network with speed factor one.

Primitive type **Float**

- Semantics. It represents any floating point number with the 64 bit IEEE representation.

Primitive type **Time**

- Semantics. Floating point number that represents a time interval in unspecified time units.

Primitive type **Absolute_Time**

- Semantics. Floating point number that represents an absolute time measured from and arbitrary time origin, in unspecified time units (the same units used in Time).

Primitive type **Normalized_Execution_Time**

- Semantics. Floating point number that represents the amount of processing resource capacity that is required for the execution of an operation. It is expressed as the execution time of an

operation, when it is executed by a normalized processing resource of speed factor equal to one. The real execution time is obtained by dividing the normalized execution time by the processing resource's speed factor.

Primitive type **Percentage**

- Semantics. A floating point number representing a percentage in the range 0 to 100, and followed by a "%" character.

Primitive type **Extended_Percentage**

- Semantics. A floating point number representing a percentage, and followed by a "%" character. In some cases (slacks) the notation "> = num%" may be used to indicate that the actual result is greater than the specified number. This case is usually reserved to analysis results that are unbounded or too large to be calculated by practical means. It may represent also negative percentages

Primitive type **Assertion**

- Semantics. It affirms (Yes) or negates (No) any characteristic.

3 MAST2 core classes

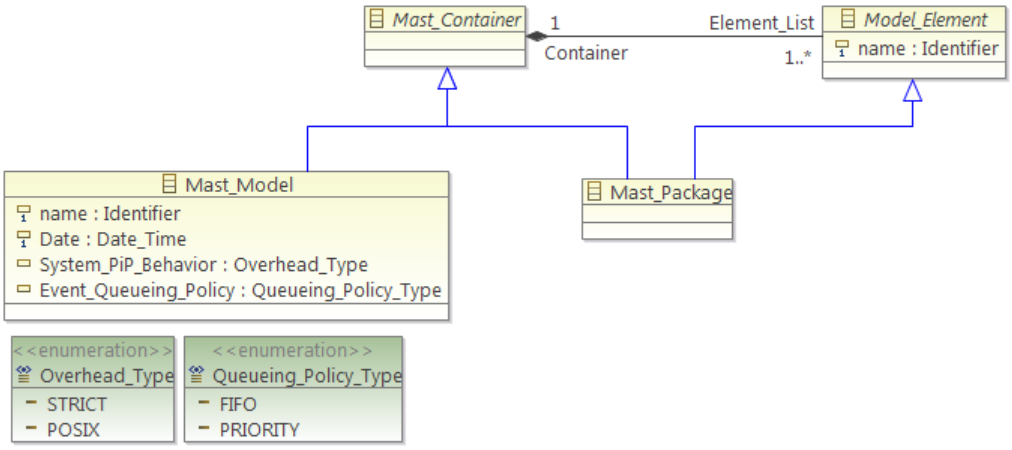

Figure 2 – The core classes in the MAST2 meta-model

<<abstract>> class **Mast_Container**

- Semantics. This class introduces the ability to declare Model_Element instances of the model. Each Mast_Container element defines a new namespace, i.e. different Model_Element instances of the same type declared in different containers can have the same name.
- Ancestors hierarchy: -
- Direct subclasses:
  - Mast_Model
  - Mast_Package
- Attributes: -
- References:
  - <Aggregated> **Element_List :** Model_Element [1 .. *] → List of model elements declared as aggregated in the container.

Class **Mast_Model**

- Semantics. It represents the overall MAST model of a real-time situation that a particular system may have, and that needs to be analyzed. Global information about the real-time situation and the underlying implementation is described in the Mast_Model object.
- Ancestors hierarchy: *Mast_Container*
- Direct subclasses: -
- Attributes:
  - **name :** Identifier [1] → The name of the modeled real-time situation.

- **Date :** Date_Time [1] → The date in which the real-time situation model was created.
- **System_PIP_Behavior :** Overhead_Type [0 .. 1] = STRICT → The behavior of the underlying implementation in regard to the priority inheritance protocol. It can be STRICT, when the implementation strictly follows the original priority inheritance protocol, or POSIX, when the implementation follows the POSIX standard, which allows a more relaxed implementation that can lead to longer blocking times. The default value is STRICT. When a mutex is unlocked, the POSIX specification allows the implementation to pass the lock to the highest priority thread that is waiting to acquire the mutex. However, in the original priority inheritance protocol the mutex must be unlocked and the lock cannot be passed directly to another thread before the scheduler chooses the highest priority thread to be executed.
- **Event_Queueing_Policy :** Queueing_Policy_Type [0 .. 1] = FIFO → This attribute represents the policy used to determine which request must be processed first when several pending event instances are queued in the input of a step. Its complete meaning is explained in the Merge element.

- References:
  - <Aggregated> **Element_List :** Model_Element [1 .. *] → Inherited from *Mast_Container*.

Enumeration **Overhead_Type**

- Semantics. Enumerated type that describes the behaviour of the underlying implementation in regard to the priority inheritance protocol. It can be STRICT or POSIX.
- Values:
  - **STRICT** → The implementation strictly follows the original priority inheritance protocol.
  - **POSIX** → The implementation follows the POSIX standard that allows a more relaxed implementation that can lead to longer blocking times.

Enumeration **Queueing_Policy_Type**

- Semantics. It is the policy used to determine the request to be processed when several pending event instances are queued in the input of the step. It can be FIFO or Priority (using the priority value of the Schedulable Resource that generated the event instance).
- Values:
  - **FIFO** → The request is chosen with a FIFO policy.
  - **PRIORITY** → The request is chosen based on the priority of the Schedulable Resource that generated the activation request.

<<abstract>> class **Model_Element**

- Semantics. It is the common ancestor of the high level classes in the Mast_Model.
- Ancestors hierarchy: -
- Direct subclasses:
  - Mast_Package
  - Processing_Resource
  - Timer
  - Clock_Synchronization_Object
  - Scheduler
  - Schedulable_Resource
  - Operation
  - Mutual_Exclusion_Resource
  - End_To_End_Flow
- Attributes:
  - **name :** Identifier [1] → The identifier of the model element.
- References:
  - <Referenced> **Container** : Mast_Container [1] → The container in which the element is declared.

Class **Mast_Package**

- Semantics. It is a container which is introduced to organize the elements of the model. A Mast_Package element can be added to any container (Mast_Model or Mast_Package)
- Ancestors hierarchy: *Mast_Container* / *Model_Element*
- Direct subclasses: -
- Attributes:
  - **name :** Identifier [1] → Inherited from *Model_Element*
- References:
  - <Aggregated> **Element_List :** Model_Element [1 .. *] → Inherited from *Mast_Container*.

4 MAST2 model elements

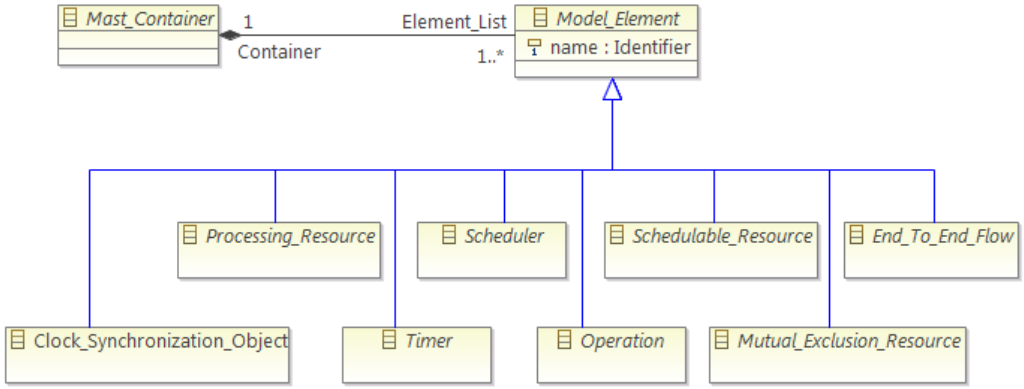

Figure 3 – Top subclasses of Model_Element

4.1 PROCESSING RESOURCES

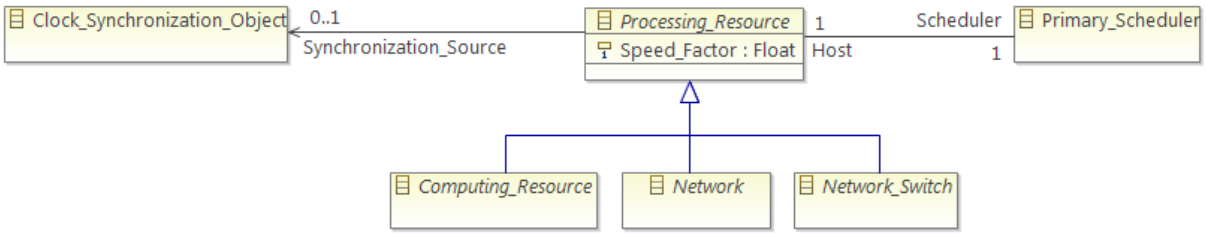

Figure 4 – Hierarchy of processing resources

<<abstract>> class **Processing_Resource**

- Semantics. It models the processing capacity of a hardware component that carries out some of the modeled system activities, which are generally pieces of code to be executed, operations carried out directly by hardware, or messages to be transferred.
- Ancestors hierarchy: *Model_Element*
- Direct subclasses:
  - *Computing_Resource*
  - *Network*
  - *Network_Switch*
- Attributes:
  - **Speed_Factor :** Float [1] = 1.0 → All execution times will be expressed in normalized units. The real execution time is obtained by dividing the normalized execution time by the speed factor. The default value is 1.0.
- References:

- <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
- <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → It references the clock synchronization object used to synchronize the internal system timer of the processing resource. This affects events and timing requirements referencing the system timer, Timetable driven policies, Global EDF scheduling, etc. If this attribute is NULL the processing resource is not synchronized with others. This attribute allows defining groups of processing resources that have clock synchronization services. The default value is NULL.
- <Referenced> **Scheduler** : Primary_Scheduler [1] → The scheduler associated to the processing resource in order to schedule the processing capacity it supplies.

4.1.1 COMPUTING RESOURCES

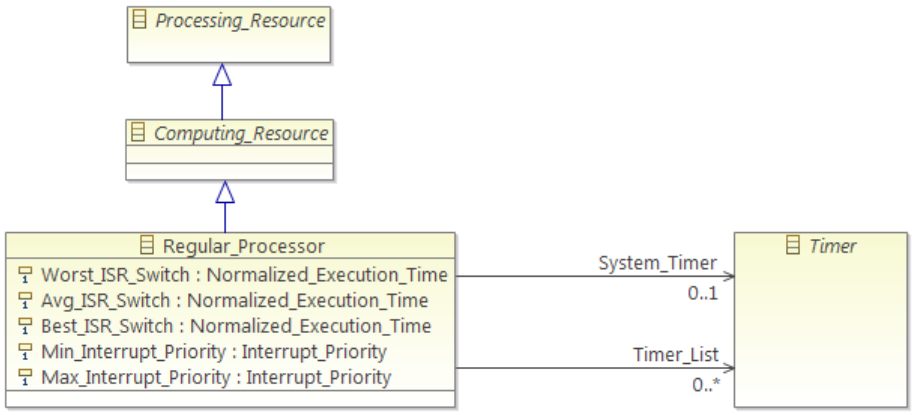

Figure 5 – Hierarchy of computing resources

<<abstract>> class **Computing_Resource**

- Semantics. It models a device capable of executing pieces of application code or operations carried out directly by hardware.
- Ancestors hierarchy: *Model_Element.Processing_Resource*
- Direct subclasses:
  - Regular_Processor
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*
  - **Speed_Factor :** Float [1] = 1.0 → Inherited from *Processing_Resource*
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

- <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
- <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.

Class **Regular_Processor**

- Semantics. It represents a physical processor or a device; it contains attributes that describe the overheads associated with the processor hardware modules, such as interrupt services, timers, etc. A simplified interrupt model is assumed in which interrupts may be serviced at one or more priority levels through a preemptive fixed priority hardware scheduler. The time needed to switch from application code to interrupt code or back is modeled as an overhead called "ISR_Switch" and is modeled by three normalized execution times that define the worst, average, or best case overheads.
- Ancestors hierarchy: *Model_Element.Processing_Resource.Computing_Resource*
- Direct subclases: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*
  - **Speed_Factor :** Float [1] = 1.0 → Inherited from *Processing_Resource*
  - **Worst_ISR_Switch :** Normalized_Execution_Time [1] = 0.0 → Worst overhead. Maximum execution time of an interrupt switch.
  - **Avg_ISR_Switch :** Normalized_Execution_Time [1] = 0.0 → Average overhead. Average execution time of an interrupt switch.
  - **Best_ISR_Switch :** Normalized_Execution_Time [1] = 0.0 → Best overhead. Minimum execution time of an interrupt switch.
  - **Min_Interrupt_Priority :** Interrupt_Priority [1] = 0 → It defines the minimum valid priority value for activities executed as interrupt service routines.
  - **Max_Interrupt_Priority** : Interrupt_Priority [1] = MAXIMUM → It defines the maximum valid priority value for activities executed as interrupt service routines.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
  - <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.
  - <Referenced> **Timer_List :** Timer [*] → List of references to the used hardware timers. They influence the activation time of the timed activities that reference them and specify the processor overhead implicitly introduced by the scheduler or the operating system to

implement its own time management services. An empty list indicates that the timers used by the processing resource have a negligible overhead and perfect resolution. The default value is an empty list.

- <Referenced> **System_Timer :** Timer [0 .. 1] → The system timer is used in the timetable-driven and global EDF scheduling policies and it may also influence the activation time of the timed activities that reference it. It also specifies the processor overhead implicitly introduced by the scheduler or the operating system to implement its own time management services. If not present, this indicates that an implicit system timer is used by the processor and has a negligible overhead and perfect resolution.

4.1.2 NETWORKS

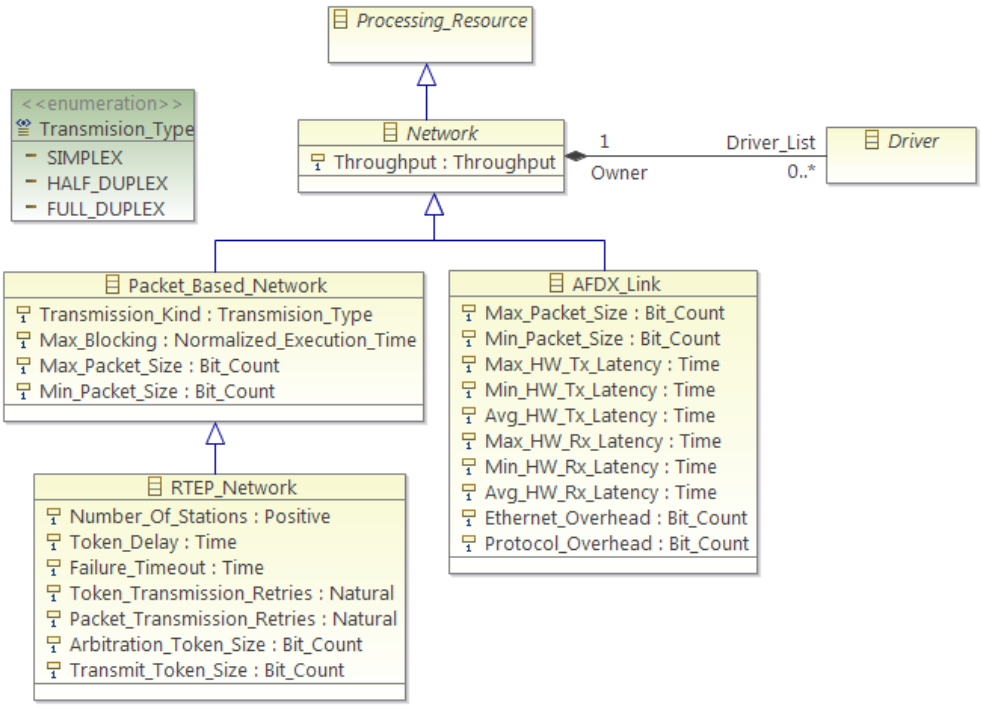

Figure 6 – Hierarchy of networks

<<abstract>> class **Network**

- Semantics. It models a communication system specialized in the transmission of messages among processors or network switches.
- Ancestors hierarchy: *Model_Element.Processing_Resource*
- Direct subclasses:
  - Packet_Based_Network
  - AFDX_Link

- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
  - **Speed_Factor :** Float [1] = 1.0 → Inherited from *Processing_Resource*.
  - **Throughput :** Throughput [1] = 0 → Normalized network bandwidth in bits per time unit. The actual network throughput is affected by the Speed_Factor.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
  - <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.
  - <Aggregated> **Driver_List :** Driver [*]→ A list of network drivers, which contain the processor overhead model associated with the transmission of messages through the network. The default is an empty list.

Enumeration **Transmission_Type**

- Semantics. It describes the capacity of a network for sending messages in different directions.
- Values:
  - **FULL_DUPLEX** → The network allows communication in both directions.
  - **HALF_DUPLEX** → The network allows communication in both directions, but only one at a time.
  - **SIMPLEX** → The network allows only one-way communication.

Class **Packet_Based_Network**

- Semantics. It models a network that uses some kind of real time protocol based on non-preemptible packets for sending messages. There are networks that support priorities in their standard protocols (i.e., the CAN bus), and other networks that need an additional protocol that works on top of the standard ones (i.e., serial lines, ethernet).
- Ancestors hierarchy: *Model_Element.Processing_Resource.Network*
- Direct subclasses:
  - RTEP_Network
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*
  - **Speed_Factor :** Float [1] = 1.0 → Inherited from *Processing_Resource*

- **Throughput :** Throughput [1] = 0 → Inherited from *Network*
- **Transmission_Kind :** Transmission_Type [1] = HALF_DUPLEX → It describes the transmission mode of the network.
- **Max_Blocking :** Time [1] = 0.0 → The maximum blocking that may be experienced before a high priority message can be transmitted, caused by the non preemptability of message packets, including both the application message and the protocol information sent with it. It usually has a value equal to the maximum packet transmission time plus the transmission time of the protocol information. Its default value is zero, indicating a negligible network blocking.
- **Max_Packet_Size :** Bit_Count [1] = MAXIMUM → It describes the maximum amount of data included in a packet, excluding any protocol information. The maximum size is used in the calculation of the number of packets into which a large message is split, calculated as the ceiling of the message size divided by the maximum packet size. This number is multiplied by the packet overhead time of the scheduler to calculate the network overhead.
- **Min_Packet_Size :** Bit_Count [1] = MAXIMUM → It describes the minimum amount of data included in a packet, excluding any protocol information. The Minimum size is used to calculate the shortest period of the overheads associated with the transmission of each packet, and thus has a strong impact on the overhead caused by the network drivers in the processors using the network.

- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
  - <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.
  - <Aggregated> **Driver_List :** Driver [*] → Inherited from *Network*.

Class **RTEP_Network**

- Semantics. It represents a network that uses the RTEP protocol. This is a token-passing protocol with prioritized messages, that uses a two-phase mechanism to send each information packet : in first place there is a priority arbitration phase in which a token is rotated through each station or processing node to determine which is the node with the highest priority message; in the second phase that node is granted permission to transmit a packet with information.
- Ancestors hierarchy: *Model_Element.Processing_Resource.Network*.Packet_Based_Network.
- Attributes:
  - **name :** Identifier [1] → Inherited from *Model_Element*.
  - **Speed_Factor :** Float [1] = 1.0 → Inherited from *Processing_Resource*.

- **Throughput :** Throughput [1] = 0 → Inherited from *Network*
- **Transmission_Kind** : Transmission_Type [1] = HALF_DUPLEX → Inherited from Packet_Based_Network. This value must be HALF_DUPLEX.
- **Max_Blocking :** Time [1] = 0.0 → Inherited from Packet_Based_Network. If not present, it will be calculated automatically by the analysis tools.
- **Max_Packet_Size :** Bit_Count [1] = MAXIMUM → Inherited from Packet_Based_Network.
- **Min_Packet_Size :** Bit_Count [1] = MAXIMUM → Inherited from Packet_Based_Network.
- **Number_Of_Stations :** Positive [1] = MAXIMUM → The number of stations or processors connected with the RTEP network. This attribute is used to determine the token rotation time and several overhead values.
- **Token_Delay :** Time [1] = 0.0 → The configurable delay introduced during the handling of a token to slow down the token transmission time in order to bound the overhead of each of the processors connected to the RT-EP network. The default value is zero, which implies maximum overhead and minimum latency.
- **Failure_Timeout** : Time [1] = MAXIMUM → This is the configurable timeout used to determine that there has been a packet loss due to a failure in the network. The default value is a large time, which would imply no error recovery.
- **Token_Transmission_Retries :** Natural [1] = 0 → Maximum number of retransmissions that we allow for each packet containing a protocol token.
- **Packet_Transmission_Retries** : Natural [1] = 0 → Maximum number of retransmissions that we allow for each packet with user information.
- **Arbitration_Token_Size** : Bit_Count [1] = MINIMUM → Number of bits of the message used for priority arbitration.
- **Transmit_Token_Size :** Bit_Count [1] = MINIMUM → Number of bits of the message used for appointing the next transmit node.

- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
  - <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.
  - <Aggregated> **Driver_List :** Driver [*] → Inherited from *Network*.

Class **AFDX_Link**

- Semantics. It represents a link between a processor and/or a switch in a network that uses the AFDX (Avionics Full Duplex Switched Ethernet) protocol, which is the protocol used in the ARINC 664 standard. The link allows full duplex communications.
- Ancestors hierarchy: *Model_Element.Processing_Resource.Network*
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
  - **Speed_Factor** : Float [1] = 1.0 → Inherited from *Processing_Resource*.
  - **Throughput** : Throughput [1] = 0 → Inherited from *Network*.
  - **Max_Packet_Size** : Bit_Count [1] = MAXIMUM → It describes the maximum amount of user data included in a packet, excluding any protocol information. The maximum size is used in the calculation of the number of packets into which a large message is split, calculated as the ceiling of the message size divided by the maximum packet size.
  - **Min_Packet_Size** : Bit_Count [1] = MAXIMUM → It describes the minimum amount of user data included in a packet, excluding any protocol information.
  - **Max_HW_Tx_Latency** : Time [1] = MAXIMUM → Maximum technological latency in transmission in the AFDX hardware.
  - **Min_HW_Tx_Latency** : Time [1] = MAXIMUM → Minimum technological latency in transmission in the AFDX hardware.
  - **Avg_HW_Tx_Latency** : Time [1] = MAXIMUM → Average technological latency in transmission in the AFDX hardware.
  - **Max_HW_Rx_Latency** : Time[1] = MAXIMUM → Maximum technological latency in reception in the AFDX hardware.
  - **Min_HW_Rx_Latency** : Time [1] = MAXIMUM → Minimum technological latency in reception in the AFDX hardware.
  - **Avg_HW_Rx_Latency** : Time [1] = MAXIMUM → Average technological latency in reception in the AFDX hardware.
  - **Ethernet_Overhead** : Bit_Count [1] = 160 → It is the number of overhead bytes to be added to each Ethernet frame. Its value is 160 bits (20 bytes).
  - **Protocol_Overhead** : Bit_Count [1] = 376 → It is the number of overhead bytes corresponding to the protocol used for communications. Its value is 376 bits (47 bytes) for the UDP/IP used in AFDX.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

- <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
- <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.
- <Aggregated> **Driver_List :** Driver [*] → Inherited from *Network*.

4.1.2.1 DRIVERS

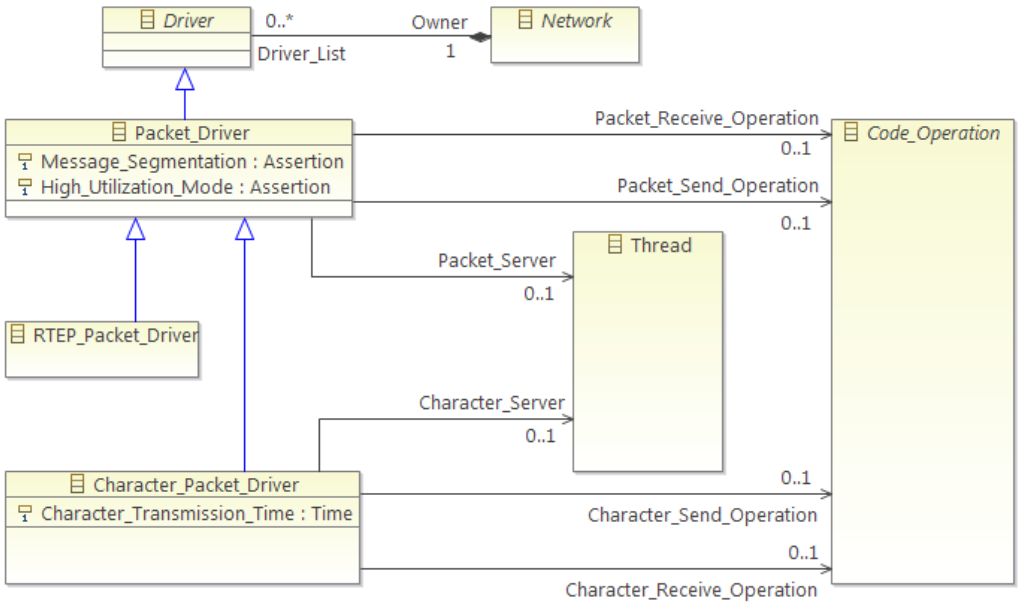

Figure 7 – Hierarchy of drivers

<<abstract>> class **Driver**

- Semantics. It represents activities executed in a processor as a consequence of the transmission or reception of a message or a message packet through a network
- Ancestors hierarchy: -
- Direct subclasses:
  - Packet_Driver
- Attributes: -
- References:
  - <Referenced> **Owner :** Network [1] → The communication network whose processor overhead model due to the transmission of messages through it is featured by the driver.

Class **Packet_Driver**

- Semantics. It represents a driver that is activated at each message packet transmission or reception.
- Ancestors hierarchy: *Driver*
- Direct subclasses:
  - Character_Packet_Driver

- RTEP_Packet_Driver

- Attributes:
  - **Message_Segmentation** : Assertion [1] = Yes → A "Yes|No" value that determines whether or not the driver is capable of partitioning long messages into several packets and rebuilding the message at the other end. This attribute influences the overhead model of the driver.
  - **High_Utilization_Mode :** Assertion [1] = Yes → This parameter specifies whether or not the network is used at a high transmission rate. Its value determines the overhead model that should be used for the driver. If the value is YES, the system has a high network utilization and few messages are being partitioned (i.e., mostly short messages); in this case the driver overhead is modeled as periodic send and receive operations with a period equal to the minimum packet transmission time (this is called the decoupled overhead model). If the value is NO, the driver overhead is modeled as message send and message receive operations that are attached to the transaction that causes the transmission of the message (this is called the coupled overhead model). Both models are pessimistic, but depending on the network utilization one of them produces more accurate results. The default value is YES.

- References:
  - <Referenced> **Owner :** Network [1] → Inherited from *Driver*
  - <Referenced> **Packet_Server :** Thread [0 .. 1] → The reference to the schedulable resource that is executing the part of the driver that is executed for each packet sent or received (which in turn has a reference to the scheduler (and indirectly to the processor), and to the scheduling parameters).
  - <Referenced> **Packet_Send_Operation :** Code_Operation [0 .. 1] → Reference to the operation that is executed by the packet server each time a packet is sent.
  - <Referenced> **Packet_Receive_Operation :** Code_Operation [0 .. 1] → Reference to the operation that is executed by the packet server each time a packet is received.

Class **Character_Packet_Driver**

- Semantics. It is a specialization of a packet driver in which there is an additional overhead associated with sending each character, as happens in some serial lines.
- Ancestors hierarchy: *Driver*.Packet_Driver
- Direct_Subclasses: -
- Attributes:
  - **Message_Segmentation :** Assertion [1] = Yes → Inherited from Packet_Driver.
  - **High_Utilization_Mode** : Assertion [1] = Yes → Inherited from Packet_Driver.

- **Character_Transmission_Time** : Time [1] = 0.0 → Time of character transmission. It determines the period used for the overhead models of the character send and character receive operations.

- References:
  - <Referenced> **Owner :** Network [1] → Inherited from *Driver*
  - <Referenced> **Packet_Server** : Thread [0 .. 1] → Inherited from Packet_Driver.
  - <Referenced> **Packet_Send_Operation** : Code_Operation [0 .. 1] → Inherited from Packet_Driver.
  - <Referenced> **Packet_Receive_Operation** : Code_Operation [0 .. 1] → Inherited from Packet_Driver.
  - <Referenced> **Character_Server** : Thread [0 .. 1] → The reference to the schedulable resource that is executing the part of the driver that is executed for each character sent or received (which in turn has a reference to the scheduler (and indirectly to the processor), and to the scheduling parameters).
  - <Referenced> **Character_Send_Operation** : Code_Operation [0 .. 1] → The reference to the operation that is executed by the character server each time a character is sent.
  - <Referenced> **Character_Receive_Operation** : Code_Operation [0 .. 1] → The reference to the operation that is executed by the character server each time a character is received.

Class **RTEP_Packet_Driver**

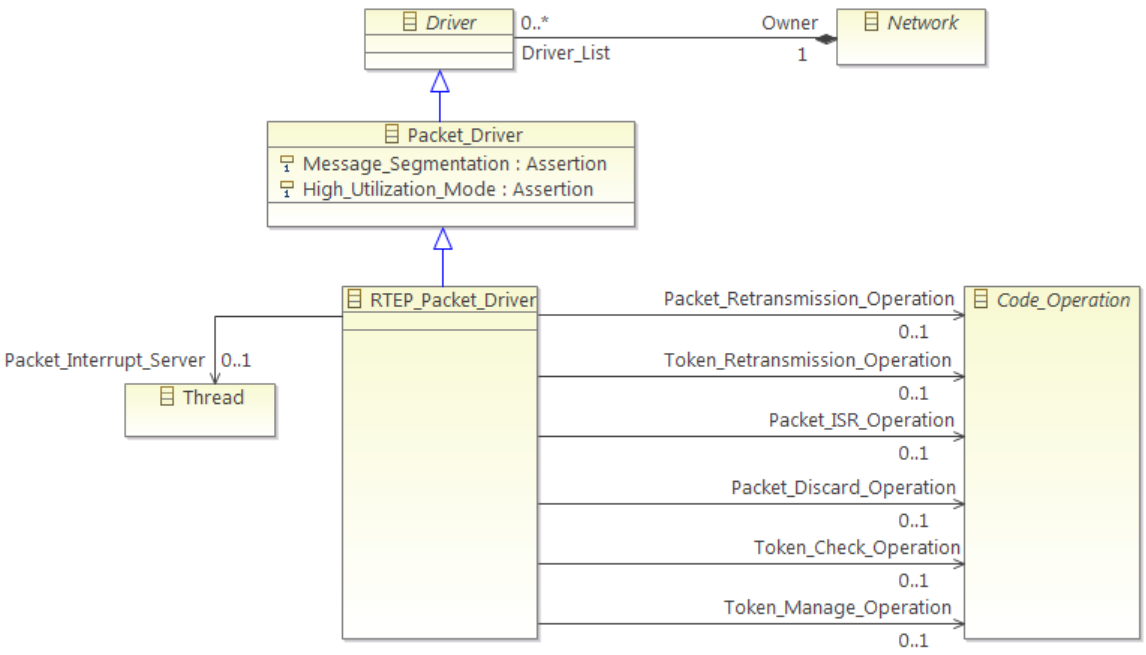

Figure 8 – The RTEP_Packet_Driver class

- Semantics. It is a specialization of a packet driver that characterizes the Real-Time Ethernet Protocol, RTEP. This is a token-passing protocol with prioritized messages, that uses a two-phase mechanism to send each information packet : in first place there is a priority arbitration phase in which a token is rotated through each station or processing node to determine which is the node with the highest priority message; in the second phase that node is granted permission to transmit a packet with information. Compared to a regular packet driver, there are additional overheads associated with the transmission and reception of the tokens, as well as with the error recovery mechanisms. In the Decoupled RTA Overhead Model, the period of the overhead transactions used to model the message send and receive operations is not the minimum packet transmission time, as with the Packet_Driver, but rather a combination of several attributes. See the RT-EP documentation for a description of the overhead model.
- Ancestors hierarchy: *Driver*.Packet_Driver
- Direct subclasses: -
- Attributes:
  - **Message_Segmentation :** Assertion [1] = Yes → Inherited from Packet_Driver.
  - **High_Utilization_Mode** : Assertion [1] = Yes → Inherited from Packet_Driver.
- References:
  - <Referenced> **Owner :** Network [1] → Inherited from *Driver*
  - <Referenced> **Packet_Server** : Thread [0 .. 1] → Inherited from Packet_Driver.
  - <Referenced> **Packet_Send_Operation** : Code_Operation [0 .. 1] → Inherited from Packet_Driver
  - <Referenced> **Packet_Receive_Operation** : Codel_Operation [0 .. 1] → Inherited from Packet_Driver.
  - <Referenced> **Packet_Interrupt_Server** : Thread [0 .. 1] → Reference to the schedulable resource that is executing the interrupt service routine that handles each incoming packet, independently of whether it is a packet with information or a protocol token.
  - <Referenced> **Packet_ISR_Operation** : Code_Operation [0 .. 1] → Reference to the operation executed by the packet server to send a token.
  - <Referenced> **Token_Check_Operation** : Code_Operation [0 .. 1] → Reference to the operation executed by the packet server to receive and check a token packet
  - <Referenced> **Token_Manage_Operation** : Code_Operation [0 .. 1] → Reference to the operation executed by the packet server to send a token.
  - <Referenced> **Packet_Discard_Operation** : Code_Operation [0 .. 1] → Reference to the operation executed by the packet server when a packet is received that is addressed to another processing node.

- <Referenced> **Token_Retransmission_Operation** : Code_Operation [0 .. 1] → Reference to the operation executed by the packet server when a lost token is retransmitted
- <Referenced> **Packet_Retransmission_Operation** : Code_Operation [0 .. 1] → Reference to the operation executed by the packet server when a packet with information that was lost is retransmitted.

4.1.3 NETWORK SWITCHES AND NETWORK ROUTERS

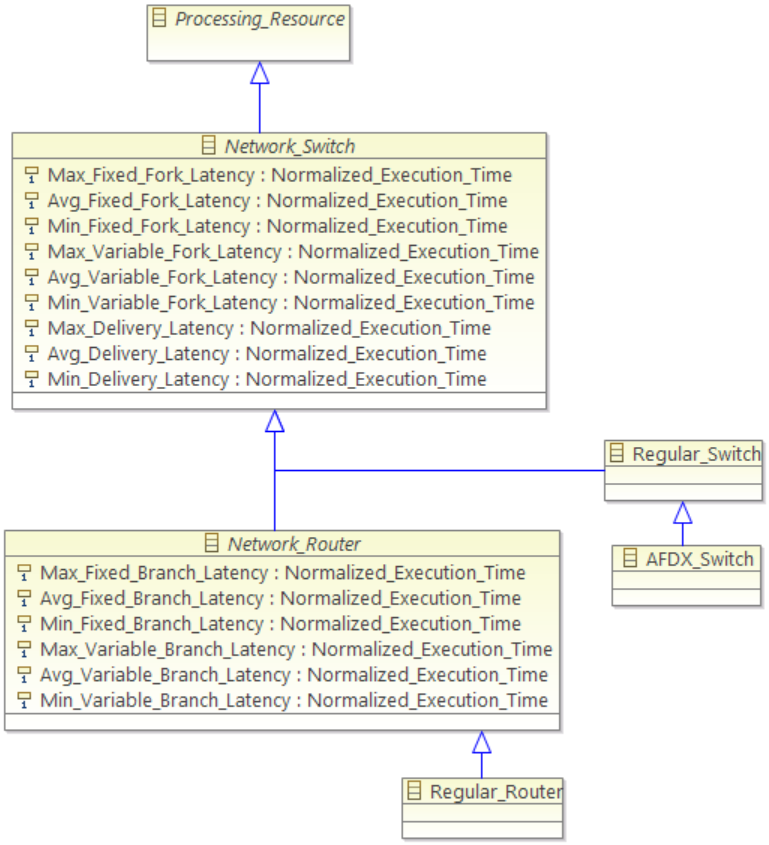

Figure 9 – Hierarchy of switches and routers

<<abstract>> class **Network_Switch**

- Semantics. It models a communication system specialized in the transfer of messages among networks. It is capable of delivering messages arriving at an input port to one or more output ports. The delivery operations are specified through special-purpose message event handlers including a simple port to port delivery, as well as a message fork handler. The destination port or ports are implied in the message stream. MAST does not require to define the network topology, since it is implicitly defined in the transactions. The overhead of a fork operation contains a fixed latency and a variable latency that depends on the number of output ports (Fork_Latency = Fixed_Fork_Latency + (Num_Of_Output_Ports * Variable_Fork_Latency). The overhead model assumes a shared memory switch in which messages do not need to be copied, with special-purpose hardware that makes it possible to operate all ports simultaneously. Contention occurs of course when several messages need to be sent to the same output port, in which case a queue is used at the output port.
- Ancestors hierarchy: *Model_Element.Processing_Resource*
- Direct subclasses:

- Regular_Switch
- *Network_Router*

- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
  - **Speed_Factor** : Float [1] = 1.0 → Inherited from *Processing_Resource*.
  - **Max_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute represents the maximum value for the fixed part of the latency introduced by the switch when an input message is replicated in multiple output ports. It is the part that is independent of the number of replicas.
  - **Avg_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute represents the average value for the fixed part of the latency introduced by the switch when an input message is replicated in multiple output ports. It is the part that is independent of the number of replicas.
  - **Min_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute represents the minimum value for the fixed part of the latency introduced by the switch when an input message is replicated in multiple output ports. It is the part that is independent of the number of replicas.
  - **Max_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute is used to calculate the additional part of the delay introduced by the switch when an input message is replicated in multiple output ports. It represents the maximum value of the additional delay that is introduced for each generated replica.
  - **Avg_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute is used to calculate the additional part of the delay introduced by the switch when an input message is replicated in multiple output ports. It represents the average value of the additional delay that is introduced by each generated replica.
  - **Min_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute is used to calculate the additional part of the delay introduced by the switch when an input message is replicated in multiple output ports. It represents the minimum value of the additional delay that is introduced by each generated replica.
  - **Max_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Maximum Latency introduced by the switch for each port-to-port message delivery.
  - **Avg_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Average Latency introduced by the switch for each port-to-port message delivery.
  - **Min_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Minimum latency introduced by the switch for each port-to-port message delivery

- References:

- <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
- <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
- <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.

Class **Regular_Switch**

- Semantics. Concrete and simple switch in which the output queues are assumed to work according to the policy of output network (for instance, priority-based in case of a fixed priority policy). The switch is assumed to have routing information preconfigured, so there is no need for additional network traffic originated by the switch.
- Ancestors hierarchy: *Model_Element.Processing_Resource.Network_Switch*
- Direct subclasses:
  - AFDX_Switch
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*
  - **Speed_Factor** : Float [1] = 1.0 → Inherited from *Processing_Resource*
  - **Max_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Min_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Max_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Min_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Max_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Min_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.

- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
  - <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.

Class **AFDX_Switch**

- Semantics. Concrete and simple switch working according to the AFDX specification.
- Ancestors hierarchy*: Model_Element.Processing_Resource.Network_Switch*.Regular_Switch
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
  - **Speed_Factor** : Float [1] = 1.0 → Inherited from *Processing_Resource*.
  - **Max_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Min_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Max_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch.*
  - **Min_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Max_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Min_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

- <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
- <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.

<<abstract>> class **Network_Router**

- Semantics. It models a communication system specialized in routing messages among networks. It is like a network switch but, in addition, it is able to route a message through a destination port that may be dynamically obtained. In addition to the message event handlers supported by the switches (message delivery and message fork), routers support the message branch event handler, which is capable of delivering a message to one output port chosen form of a set of possible outputs. The overhead model for the message branch event handlers is similar to the model for message fork. It contains a fixed latency and a variable latency that depends on the number of possible output ports (Message_Branch_Latency = Message_Fixed_Branch_Latency + (Num_Of_Possible_Output_Ports * Message_Variable_Branch_Latency)).
- Ancestors hierarchy: *Model_Element.Processing_Resource.Network_Switch*
- Direct subclasses:
  - Regular_Router
- Attributes :
  - **name** : Identifier [1] → Inherited from *Model_Element*.
  - **Speed_Factor** : Float [1] = 1.0 → Inherited from *Processing_Resource*.
  - **Max_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Min_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Max_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Min_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Max_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.

- **Avg_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
- **Min_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
- **Max_Fixed_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute represents the maximum value for the fixed part of the delay introduced by the router when an input message is retransmitted through an output port. It is the part that is independent of the number of possible output ports.
- **Avg_Fixed_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute represents the average value for the fixed part of the delay introduced by the router when an input message is retransmitted through an output port. It is the part that is independent of the number of possible output ports.
- **Min_Fixed_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute represents the minimum value for the fixed part of the delay introduced by the router when an input message is retransmitted through an output port. It is the part that is independent of the number of possible output ports.
- **Max_Variable_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute is used to calculate the additional part of the delay introduced by the router when an input message is retransmitted through an output port. It represents the maximum value of the additional delay that is introduced by each possible output port.
- **Avg_Variable_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute is used to calculate the additional part of the delay introduced by the router when an input message is retransmitted through an output port It represents the average value of the additional delay that is introduced by each possible output port
- **Min_Variable_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → This attribute is used to calculate the additional part of the delay introduced by the router when an input message input message is retransmitted through an output port. It represents the minimum value of the additional delay that is introduced by each possible output port.

- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
  - <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.

Class **Regular_Router**

- Semantics. Concrete and simple router in which the output queues are assumed to be FIFO queues.

- Ancestors hierarchy:
  - *Model_Element.Processing_Resource.Network_Switch.Network_Router*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
  - **Speed_Factor** : Float [1] = 1.0 → Inherited from *Processing_Resource*.
  - **Max_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Min_Fixed_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Max_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Min_Variable_Fork_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Max_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Avg_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Min_Delivery_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Switch*.
  - **Max_Fixed_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Router*.
  - **Avg_Fixed_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Router*.
  - **Min_Fixed_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Router*.
  - **Max_Variable_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Router*.
  - **Avg_Variable_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Router*.

- **Min_Variable_Branch_Latency** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Network_Router*.

- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Referenced> **Synchronization_Source :** Clock_Synchronization_Object [0 .. 1] → Inherited from *Processing_Resource*.
  - <Referenced> **Scheduler** : Primary_Scheduler [1] → Inherited from *Processing_Resource*.

4.2 TIMERS AND SYNCHRONIZATION OBJECTS

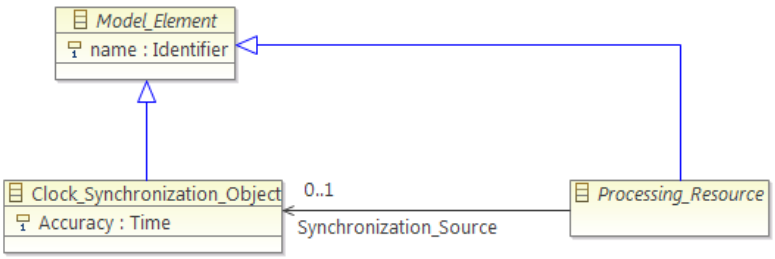

Figure 10 – The Clock_Synchronization_Object class

Class **Clock_Synchronization_Object**

- Semantics. It models a clock synchronization mechanism among clocks of different processing resources. Processing resources that reference the same clock synchronization object are assumed to have a clock synchronization mechanism among them and are considered as a group of synchronized resources with a global reference for the implicit or explicit system timer. A system may contain several of these groups.
- Ancestors hierarch : *Model_Element*
- Direct subclasses: -
- Attributes:
  - **name :** Identifier [1] → Inherited from *Model_Element*.
  - **Accuracy :** Time [1] = 0.0 → Precision of the synchronization mechanism for this timer. It is the maximum time deviation of each timer synchronized with it. Its default value is 0.0.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

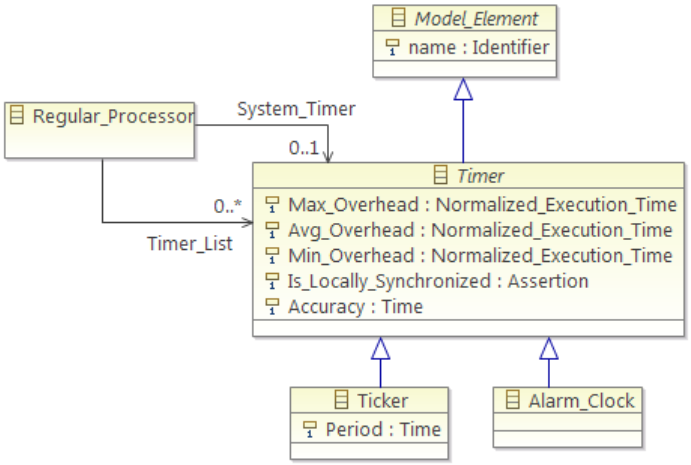

Figure 11 – Hierarchy of timers

<<abstract>> Class **Timer**

- Semantics. It models a hardware timer and defines the overhead associated with the management of the timer and, possibly, its time resolution. Each Timer defines a time domain, although some of these domains may be synchronized. A timer is useful to represent the overheads associated to the management of hardware timers and also to model the timing effects caused by its resolution and by clock synchronization services. When a timer appears as the System_Timer attribute of a Regular_Processor it is used for the timetable-driven and global EDF policies. Timers that are not system timers may be synchronized with the system timer of its processor if Is_Locally_Synchronized is Yes, with the precision given in the Precision attribute.
- Ancestors hierarchy: *Model_Element*
- Direct subclasses:
  - Ticker
  - Alarm_Clock
- Attributes:
  - **name :** Identifier [1] → Inherited from *Model_Element*.
  - **Max_Overhead :** Normalized_Execution_Time [1] = 0.0 → This is the maximum overhead of the timer interrupt, which is assumed to execute at the highest interrupt priority.
  - **Avg_Overhead :** Normalized_Execution_Time [1] = 0.0 → This is the average overhead of the timer interrupt, which is assumed to execute at the highest interrupt priority.
  - **Min_Overhead :** Normalized_Execution_Time [1] = 0.0 → This is the minimum overhead of the timer interrupt, which is assumed to execute at the highest interrupt priority.

- **Is_Locally_Synchronized :** Assertion [1] = No → A "Yes|No" value that determines whether or not the timer is locally synchronized with the processor's system timer. This attribute can only be set to Yes for non system timers.
- **Accuracy :** Time [1] = 0.0 → Precision of the local synchronization mechanism for this timer. It is the maximum time deviation with respect to the system timer. Its value is only meaningful for timers that are locally synchronized. Its default value is 0.0.

- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

Class **Ticker**

- Semantics. It models a timer implemented as a periodic ticker, i.e., a periodic interrupt that arrives at the system. When this interrupt arrives, all those timed events referencing the ticker whose expiration time has already passed, are activated. In this model, the overhead caused by the timer interrupt is localized in a single periodic interrupt, but jitter is introduced in all the associated timed events, because the resolution is the ticker period.
- Ancestors hierarchy: *Model_Element.Timer*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
  - **Max_Overhead :** Normalized_Execution_Time [1] = 0.0 → Inherited from *Timer*.
  - **Avg_Overhead :** Normalized_Execution_Time [1] = 0.0 → Inherited from *Timer*.
  - **Min_Overhead :** Normalized_Execution_Time [1] = 0.0 → Inherited from *Timer*.
  - **Is_Locally_Synchronized :** Assertion [1] = No → Inherited from *Timer*.
  - **Accuracy :** Time [1] = 0.0 → Inherited from *Timer*.
  - **Period :** Time [1] = MAXIMUM → Period of the ticker interrupt
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

Class **Alarm_Clock**

- Semantics. It models a timer implemented by a hardware timer interrupt that is always programmed to generate the interrupt at the time of the closest associated timed event. Consequently, each event generates its own interrupt that causes an overhead.
- Ancestors hierarchy: *Model_Element.Timer*

- Direct subclasses: -
- Attributes:
  - **Name** : Identifier [1] → Inherited from *Model_Element*.
  - **Max_Overhead :** Normalized_Execution_Time [1] = 0.0 → Inherited from *Timer*.
  - **Avg_Overhead :** Normalized_Execution_Time [1] = 0.0 → Inherited from *Timer*.
  - **Min_Overhead :** Normalized_Execution_Time [1] = 0.0 → Inherited from *Timer*.
  - **Is_Locally_Synchronized :** Assertion [1] = No → Inherited from *Timer*.
  - **Accuracy** : Time [1] = 0.0 → Inherited from *Timer*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

4.3 SCHEDULERS

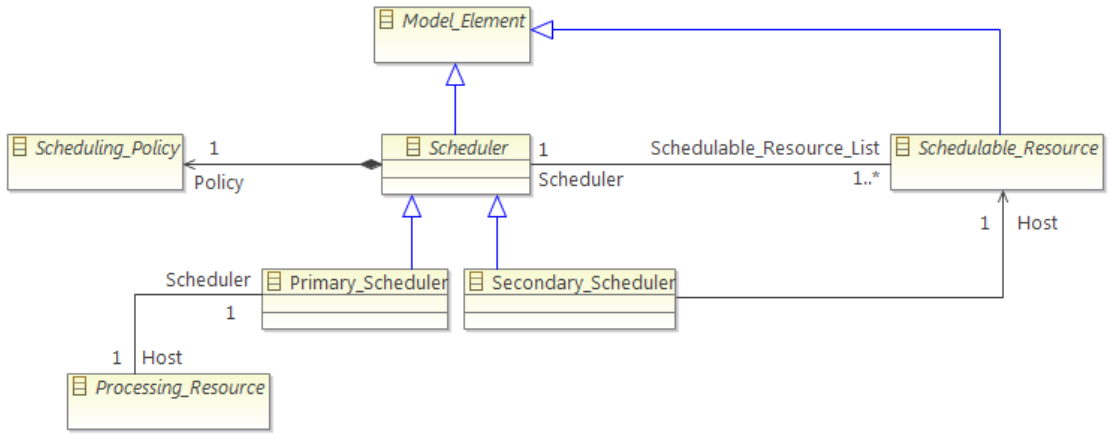

Figure 12 – Hierarchy of schedulers

<<abstract>> class **Scheduler**

- Semantics. It models the operating system or network subsystem objects that implement the appropriate scheduling strategies to manage the amount of processing capacity that has been assigned to them. Its scheduling strategy is defined by means of the associated Scheduling_Policy object. Schedulers can have a hierarchical structure. A Primary Scheduler operates by offering the whole processing capacity of its associated base processor to its associated schedulable resources. A Secondary Scheduler is allowed to deliver to its schedulable resources just the processing capacity that it receives from its associated schedulable resource, scheduled in turn by some other scheduler.
- Ancestors hierarchy: *Model_Element*
- Direct subclasses:
  - Primary_Scheduler
  - Secondary_Scheduler
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Policy** : Scheduling_Policy [1] → It represents the scheduling policy, which is the basic strategy that is implemented by the scheduler to distribute the assigned processing capacity among the schedulable resources associated to the same scheduler.
  - <Referenced> **Schedulable_Resource_List** : Schedulable_Resource [1 .. *] → The list of schedulable resources served by the scheduler.

Class **Primary_Scheduler**

- Semantics. It represents the base system scheduler for its associated processing resource.
- Ancestors hierarchy: *Model_Element.Scheduler*
- Attributes:
  - **name** : Identifier [1]→ Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Referenced> **Schedulable_Resource_List** : Schedulable_Resource [1 .. *] → Inherited from *Scheduler*.
  - <Aggregated> **Policy** : Scheduling_Policy [1] → Inherited from *Scheduler*.
  - <Referenced> **Host** : Processing_Resource [1] → It references the processing resource that supplies the processing capacity to be scheduled.

Class **Secondary_Scheduler**

- Semantics. It is allowed to deliver to its schedulable resources just the processing capacity that it receives from its associated schedulable resource, scheduled in turn by some other scheduler.
- Ancestors hierarchy: *Model_Element.Scheduler*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Referenced> **Schedulable_Resource_List** : Schedulable_Resource [1 .. *] → Inherited from *Scheduler*.
  - <Aggregated> **Policy** : Scheduling_Policy [1] → Inherited from *Scheduler*.
  - <Referenced> **Host** : Schedulable_Resource [1] → It references the schedulable resource that supplies the processing capacity to be scheduled.

4.3.1 SCHEDULING POLICIES

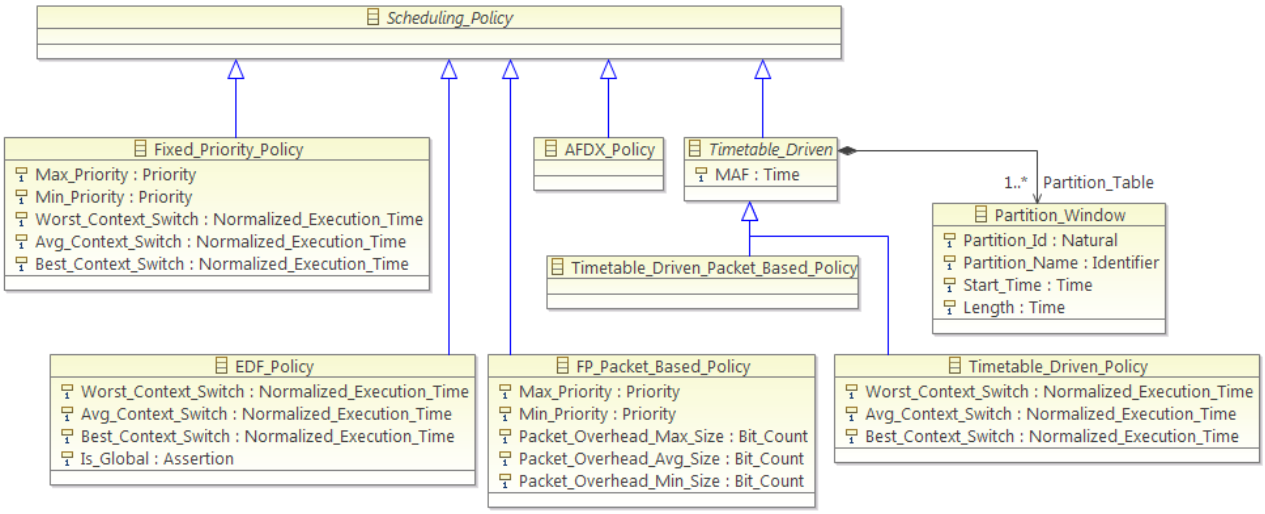

Figure 13 – Hierarchy of scheduling policies

<<abstract>> class **Scheduling_Policy**

- Semantics. A scheduling policy represents the basic strategy that is implemented in a scheduler to deliver the assigned processing capacity to those schedulable resources associated with the same scheduler. Each of these schedulable resources has a Scheduling_Parameters object that describes the parameters used by the scheduler.
- Ancestors hierarchy: -
- Direct subclasses:
  - Fixed_Priority_Policy
  - EDF_Policy
  - FP_Packet_Based_Policy
  - AFDX_Policy
  - Timetable_Driven
- Attributes: -
- References: -

Class **Fixed_Priority_Policy**

- Semantics. It represents a fixed-priority policy. It can only be assigned to a scheduler that has a Regular_Processor as its host.
- Ancestors hierarchy: *Scheduling_Policy*
- Direct subclasses: -

- Attributes:
  - **Max_Priority** : Priority [1] = MAXIMUM → It defines the maximum priority that is valid for normal operations on schedulable resources scheduled with this policy.
  - **Min_Priority** : Priority [1] = MINIMUM → It defines the minimum priority that is valid for normal operations on schedulable resources scheduled with this policy.
  - **Worst_Context_Switch** : Normalized_Execution_Time [1] = 0.0 → Maximum overhead for a context switch between schedulable resources.
  - **Avg_Context_Switch** : Normalized_Execution_Time [1] = 0.0 → Average overhead for a context switch between schedulable resources.
  - **Best_Context_Switch** : Normalized_Execution_Time = 0.0 → Minimum overhead for a context switch between schedulable resources.
- References: -

Class **EDF_Policy**

- Semantics. It represents an Earliest Deadline First policy. In the EDF policy each schedulable resource is assigned a relative deadline, and is scheduled with an absolute deadline that is equal to the relative deadline plus the generation time of the event instance being executed. The generation time can be accounted for in two different ways, leading to the so called Global EDF or Local EDF variants. In Global EDF the event generation time corresponds to the generation time of the workload event that is being processed. This time is meaningful only for steps in the same processing resource where the workload event was generated, or in processing resources that have their system timers synchronized with it. However, it is not meaningful in processing resources that are not synchronized with the processing resource where the workload event was generated. For these non-synchronized resources local EDF is assumed, implying that the generation time of the event is the time at which the internal event being processed arrived in the processing resource. The EDF Policy can only be assigned to a scheduler that has a Regular_Processor as its host.
- Ancestors hierarchy: *Scheduling_Policy*
- Direct subclasses: -
- Attributes:
  - **Worst_Context_Switch** : Normalized_Execution_Time [1] = 0.0 → Maximum overhead for a context switch between schedulable resources.
  - **Avg_Context_Switch** : Normalized_Execution_Time [1] = 0.0 → Average overhead for a context switch between schedulable resources.
  - **Best_Context_Switch** : Normalized_Execution_Time [1] = 0.0 → Minimum overhead for a context switch between schedulable resources.

- **Is_Global** : Assertion [1] = Yes → If this parameter is set to the value "Yes", the scheduler manages global deadlines. Otherwise, it manages local deadlines

Class **FP_Packet_Based_Policy**

- Semantics. It represents a fixed priority policy used in a packet oriented communication network. Network packets are assumed to be non preemptible. It can only be assigned to a scheduler that has a Packet_Based_Network as its host.
- Ancestors hierarchy: *Scheduling_Policy*
- Direct subclasses: -
- Attributes:
  - **Max_Priority** : Priority [1] = MAXIMUM → It defines the maximum priority valid for messages to be sent using this policy.
  - **Min_Priority** : Priority [1] = MINIMUM → It defines the minimum priority valid for messages to be sent using this policy.
  - **Packet_Overhead_Max_Size** : Bit_Count [1] = 0 → Maximum overhead associated with sending each packet, because of the protocol messages or headers that need to be sent before or after each packet. It specifies the overheads using a bit count (which is then translated to time using the throughput attribute of the network associated to the scheduler that uses this policy).
  - **Packet_Overhead_Avg_Size** : Bit_Count [1] = 0 → Average overhead associated with sending each packet, because of the protocol messages or headers that need to be sent before or after each packet. It specifies the overheads using a bit count (which is then translated to time using the throughput attribute of the network associated to the scheduler that uses this policy).
  - **Packet_Overhead_Min_Size** : Bit_Count [1] = 0 → Minimum overhead associated with sending each packet, because of the protocol messages or headers that need to be sent before or after each packet. It specifies the overheads using a bit count (which is then translated to time using the throughput attribute of the network associated to the scheduler that uses this policy).
- References: -

Class **Partition_Window**

- Semantics. Data required to define each execution window inside a partition in the Timetable_Driven policy.
- Ancestors hierarchy: -
- Direct subclasses: -

- Attributes:
  - **Partition_Id** : Natural [1] → Natural integer that is used as identifier of the partition to which this window belongs. Multiple windows may belong to the same partition.
  - **Partition_Name** : Identifier [1] = null → Optional identifier that is used as the partition name, for expressiveness purposes. Each partition name must correspond to a unique partition Id.
  - **Start_Time** : Time [1] → Start time of the window inside the partition (relative to the start of the major frame (MAF) used as the origin of time for the partitions).
  - **Length** : Time [1] → Duration of the window.
- References: -

<<abstract>> class **Timetable_Driven**

- Semantics. It represents a timetable-driven policy. A scheduler using this policy schedules its associated Schedulable Resources by assigning them to the partitions specified in the corresponding Timetable_Driven_Params. Each partition contains one or more time windows which are the only intervals during which schedulable resources assigned to the partition will be allowed to execute. The partition windows operate during an interval called the major frame (MAF). This schedule is then repeated in a cyclic manner. When more than one schedulable resource are assigned to the same partition a secondary scheduler may be needed to schedule the available time inside the partition.
- Ancestors hierarchy: *Scheduling_Policy*
- Direct subclasses:
  - Timetable_Driven_Policy
  - Timetable_Driven_Packet_Based_Policy
- Attributes:
  - **MAF** : Time [1] = MAXIMUM → Time interval in which the execution of the partition windows is scheduled. The schedule is repeated in a cyclic manner. All the Timetable_Driven_Policy objects that are synchronized through their associated processing resources should have the same MAF.
- References:
  - <Aggregated> **Partition_Table** : Partition_Window [1 .. *] → List of partition windows to be scheduled in each MAF.

Class **Timetable_Driven_Policy**

- Semantics. It represents a concrete Timetable Driven policy for processors. It only can be assigned to a scheduler that has a Regular_Processor as host.

- Ancestors hierarchy: *Scheduling_Policy. Timetable_Driven*
- Direct subclasses: -
- Attributes:
  - **MAF** : Time [1] = MAXIMUM → Inherited from *Timetable_Driven*.
  - **Worst_Context_Switch :** Normalized_Execution_Time [1] = 0.0 → Worst overhead for a context switch between schedulable resources.
  - **Avg_Context_Switch :** Normalized_Execution_Time [1] = 0.0 → Average overhead for a context switch between schedulable resources.
  - **Best_Context_Switch :** Normalized_Execution_Time [1] = 0.0 → Minimum overhead for a context switch between schedulable resources.
- References:
  - <Aggregated> **Partition_Table** : Partition_Window [1 .. *] → Inherited from *Timetable_Driven*.

Class **Timetable_Driven_Packet_Based_Policy**

- Semantics. It represents a Timetable Driven policy assigned to a scheduler that has a Packet_Based_Network as its host.
- Ancestors hierarchy: *Scheduling_Policy.Timetable_Driven*
- Direct subclasses: -
- Attributes:
  - **MAF** : Time [1] = MAXIMUM → Inherited from *Timetable_Driven*.
- References:
  - <Aggregated> **Partition_Table** : Partition_Window [1 .. *] → Inherited from *Timetable_Driven*.

Class **AFDX_Policy**

- Semantics. It represents the scheduling policy used in an AFDX network in which messages are scheduled through virtual links when they are originated at an end-system (a processor). Messages are scheduled in FIFO order when they are originated at an AFDX switch. The AFDX policy may only be assigned to a scheduler that has an AFDX_Link as its host.
- Ancestors hierarchy: *Scheduling_Policy*
- Direct subclasses: -
- Attributes: -

- References: -

4.4 SCHEDULABLE RESOURCES

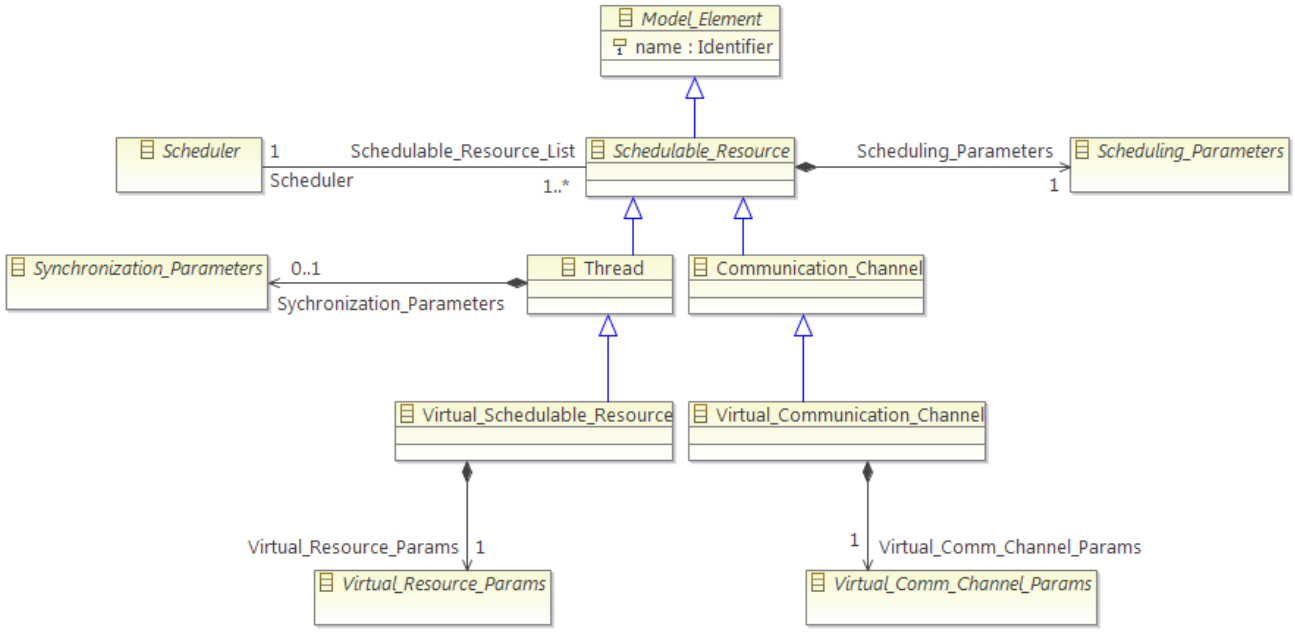

Figure 14 – Hierarchy of schedulable resources

<<abstract>> class **Schedulable_Resource**

- Semantics. It models a schedulable entity in a processing resource. It represents a unit of concurrent execution in a processor (task, single-threaded process, thread, interrupt service routine ...) or network (communication channel, communication session ...).
- Ancestors hierarchy: *Model_Element*
- Direct subclasses:
  - Thread
  - Communication_Channel
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Scheduling_Parameters** : Scheduling_Parameters [0 .. 1] → Object with the parameters that are needed by the scheduler to apply the corresponding scheduling policy. The scheduling parameters of a schedulable resource must be compatible with the scheduling policy of the corresponding scheduler. It can be null only in the case of a Virtual_Schedulable_Resource.

- <Referenced> **Scheduler** : Scheduler [1] → Reference to the scheduler that serves it. It can be null only in the case of a Virtual_Schedulable_Resource.

Class **Thread**

- Semantics. It is a specialized Schedulable_Resource for the execution of code in a Computing_Resource. It represents a conceptual flow of execution that is implemented as a thread, or a task, or a single-threaded process, or even an interrupt service routine.
- Ancestors hierarchy: *Model_Element.Schedulable_Resource*
- Direct subclasses:
  - Virtual_Schedulable_Resource
- Attributes:
  - **name** : Identifier → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Scheduling_Parameters** : Scheduling_Parameters [0 .. 1] → Inherited from *Schedulable_Resource*.
  - <Referenced> **Scheduler** : Scheduler [0 .. 1] → Inherited from *Schedulable_Resource*.
  - <Aggregated> **Synchronization_Parameters** : Synchronization_Parameters [0 .. 1] → Object with the parameters that are needed by the scheduler to apply the corresponding synchronization protocol in the mutually exclusive access to shared resources using a mutual exclusion resource. It should be present whenever the synchronization protocols used by the thread need more information than the one available in the scheduling parameters (for instance, for the SRP protocol).

Class **Virtual_Schedulable_Resource**

- Semantics. It is a specialized Schedulable_Resource that is used to schedule the execution of activities independently of the processing resource in which they are physically executed and the workload of that processing resource. It plays a double role with two viewpoints. When seen from the application viewpoint, the virtual schedulable resource represents a portion of a schedulable resource with guaranteed bandwidth and responsiveness. When seen from the viewpoint of the processing resource it represents a concrete workload that has to be scheduled.

- From the point of view of the applications, the virtual schedulable resource represents a resource reservation contract that is referenced through the Resource_Reservation_Params association. The reservation is independent of the underlying implementation of the virtual resource, which is captured in the referenced scheduler (Scheduler association) and the scheduling parameters assigned (Scheduling_Parameters association); these do not affect the

reservation guaranteed to the application, except that they are used to know the Speed_Factor of the underlying Processing_Resource and the mutual exclusion operations that may influence the scheduling due to being executed on it.

- From the point of view of the execution platform, the virtual schedulable resource represents a concrete workload with the timing requirements that are defined by the resource reservation contract, which must be scheduled in an actual scheduler (Scheduler association) with the actual scheduling parameters characterized by the Scheduling_Parameters association.

This double viewpoint of the virtual schedulable resource makes it possible to make two kinds of analysis : on the one hand, it is possible to analyze an application running on top of a virtual resource independently of the rest of the system. On the second hand, it is possible to analyze the schedulability of a set of virtual schedulable resources running in a particular physical platform, independently of the actual applications that will run on top of them. This kind of double analysis makes it possible to independently develop application components that run in virtual resources, and, later, analyze the feasibility of a particular composition of these application components, without the need to know about their internal details.

- Ancestors hierarchy: *Model_Element.Schedulable_Resource*.Thread
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Scheduling_Parameters** : Scheduling_Parameters [0 .. 1] → Inherited from *Schedulable_Resource*.
  - <Referenced> **Scheduler** : Scheduler [0 .. 1] → Inherited from *Schedulable_Resource*.
  - <Aggregated> **Synchronization_Parameters** : Synchronization_Parameters [0 .. 1] → Inherited from Thread.
  - <Aggregated> **Virtual_Resource_Params** : Virtual_Resource_Params [1] → Scheduling parameters of the resource reservation contract associated to the Virtual_Schedulable_Resource.

Class **Communication_Channel**

- Semantics. It is a specialized Schedulable_Resource for message transmission in a network.
- Ancestors hierarchy: *Model_Element.Schedulable_Resource*
- Direct subclasses:
  - Virtual_Communication_Channel
- Attributes:

- **name** : Identifier [1] → Inherited from *Model_Element*.

- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Scheduling_Parameters** : Scheduling_Parameters [0 .. 1] → Inherited from *Schedulable_Resource*.
  - <Referenced> **Scheduler** : Scheduler [0 .. 1] → Inherited from *Schedulable_Resource*.

Class **Virtual_Comm_Channel**

- Semantics. It is a specialized Communication_Channel that is used to schedule the submission of a set of messages independently of the physical network through which they are sent and the traffic that it supports. It plays a double role :

  From the point of view of the applications it represents the scheduling of the messages in the resource reservation contract referenced through the Resource_Reservation_Params association. The reservation is independent of the underlying implementation of the virtual resource, which is captured in the referenced scheduler (Scheduler association) and the scheduling parameters assigned (Scheduling_Parameters association); these do not affect the reservation guaranteed to the application, except that they are used to know the Speed_Factor of the underlying Processing_Resource.

  From the point of view of the execution platform, it represents the scheduling of the messages with the timing requirements defined in the resource reservation contract, which must be scheduled by the real scheduler (referenced through the Scheduler association) and with the scheduling parameters referenced through the Scheduling_Parameters association.
- Ancestors hierarchy: *Model_Element.Schedulable_Resource*.Communication_Channel
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Scheduling_Parameters** : Scheduling_Parameters [0.1] → Inherited from Schedulable_Resource.
  - <Referenced> **Scheduler** : Scheduler [0 .. 1] → Inherited from *Schedulable_Resource*.
  - <Aggregated> **Virtual_Comm_Channel_Params** : Virtual_Comm_Channel_Params → Scheduling parameters that define the resource reservation contract associated to the Virtual_Communication_Channel.

Object **DEFAULT_COMMUNICATION_CHANNEL : Communication_Channel** → Instance of Communication_Channel that represents the default communication channel used for communications through AFDX links when the message is originated at an AFDX switch. This instance is predefined for each AFDX Link for convenience, because no attributes are needed. The implicit scheduling policy is FIFO ordering for the messages.

4.4.1 SCHEDULING PARAMETERS

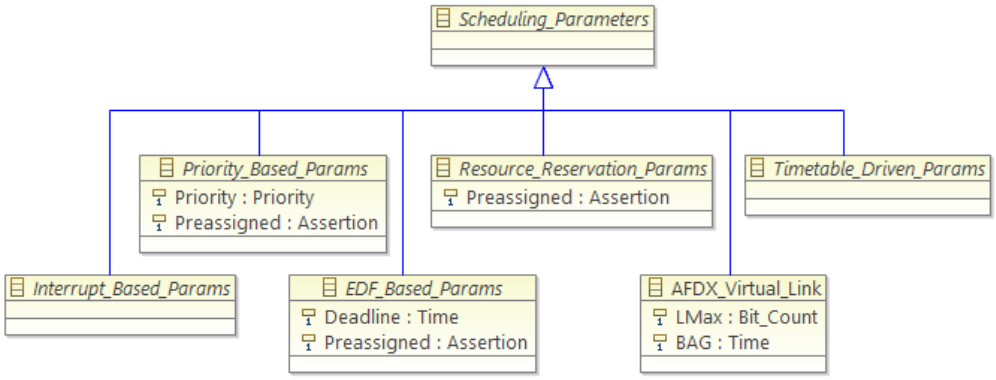

Figure 15 – Hierarchy of scheduling parameters

<<abstract>> class **Scheduling_Parameters**

- Semantics. This class represents the parameters that are attached to a schedulable resource. They are used by its scheduler to make its scheduling decisions when the schedulable resource is competing with other ones. Some scheduling policies allow several compatible scheduling behaviours to coexist in the system. For example, the fixed priority scheduling policy allows both preemptive and non preemptive activities. These different scheduling behaviours are determined by the scheduling parameters type and are also called the per-server policy parameters. There are several classes of scheduling parameters. There are some restrictions on the compatibility between scheduling parameters and the scheduling policy of the associated scheduler.
- Ancestors hierarchy:
- Direct subclasses:
  - Interrupt_Based_Params
  - Priority_Based_Params
  - EDF_Based_Params
  - Resource_Reservation_Based_Params
  - Timetable_Driven_Based_Params
  - AFDX_Virtual_Link
- Attributes: -
- References: -

4.4.1.1 PRIORITY-BASED SCHEDULING PARAMETERS

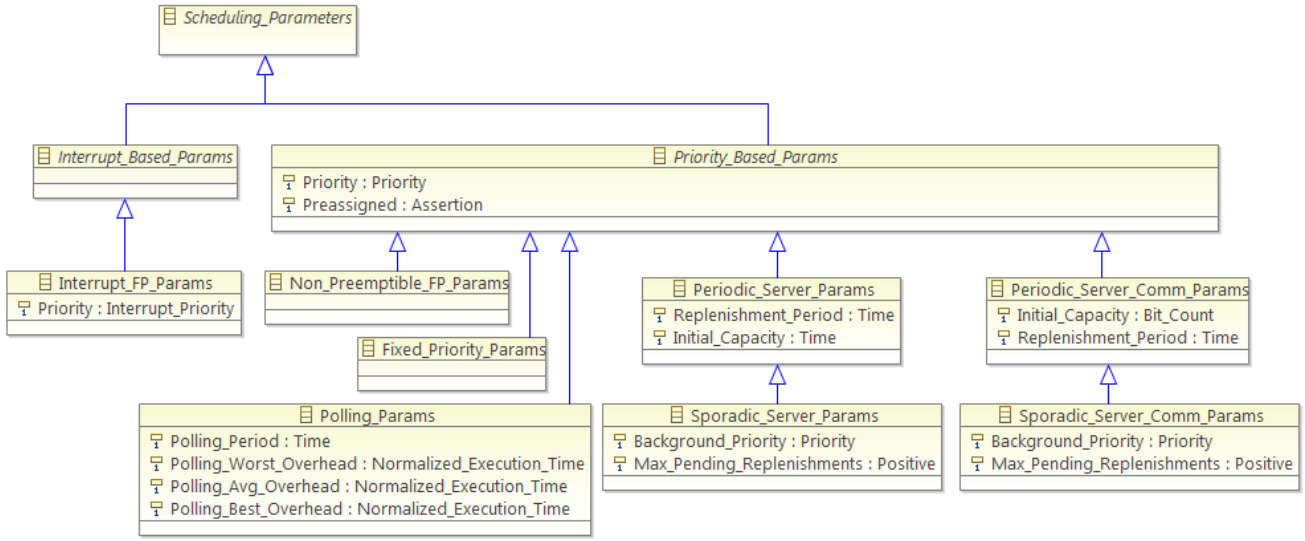

Figure 16 – Hierarchy of priority-based scheduling parameters

<<abstract>> class **Interrupt_Based_Params**

- Semantics. It represents an interrupt service routine. It has no additional attributes. It is applicable to any application scheduler, because interrupt service routines are handled directly by the Computing Resource.
- Ancestors hierarchy: *Scheduling_Parameters*
- Direct subclasses:
  - Interrupt_FP_Params
- Attributes: -
- References: -

Class **Interrupt_FP_Params**

- Semantics. It represents the parameters of an interrupt service routine. This kind of activity cannot be preempted by application activities, but can be preempted by interrupt activities with higher priority.
- Ancestors hierarchy: *Scheduling_Parameters.Interrupt_Based_Params*
- Direct subclasses: -
- Attributes:
  - **Priority** : Interrupt_Priority [1] = MINIMUM → A positive number that represents the interrupt priority. It must be within the valid ranges for the associated computing resource.

The interrupt priority is always preassigned (i.e., it is not computed by the priority assignment tools)

<<abstract>> class **Priority_Based_Params**

- Semantics. This kind of scheduling parameters is only applicable to schedulers with the fixed priority policy or the Packet_Based_FP policy.
- Ancestors hierarchy: *Scheduling_Parameters*
- Direct subclasses:
  - Non_Preemptible_FP_Params
  - Fixed_Priority_Params
  - Periodic_Server_Params
  - Periodic_Server_Comm_Params
  - Polling_Params
- Attributes:
  - **Priority** : Priority [1] = MINIMUM → A positive number that represents the scheduling priority. It must be within the valid ranges for the associated scheduler.
  - **Preassigned** : Assertion [1] = No → If this parameter is set to the value "No", the priority may be assigned by one of the priority assignment tools. Otherwise, the priority is fixed and cannot be changed by those tools.
- References: -

Class **Non_Preemptible_FP_Params**

- Semantics. Activities scheduled with these parameters are scheduled under non preemptive fixed priorities.
- Ancestors hierarchy: *Scheduling_Params.Priority_Based_Params*
- Direct subclasses: -
- Attributes:
  - **Priority** : Priority [1] = MINIMUM → Inherited from *Priority_Based_Params*.
  - **Preassigned** : Assertion [1] = No → Inherited from *Priority_Based_Params*.
- References: -

Class **Fixed_Priority_Params**

- Semantics. Activities scheduled with these parameters are scheduled under preemptive fixed priorities.
- Ancestors hierarchy: *Scheduling_Parameters.Priority_Based_Params*

- Direct subclasses: -
- Attributes:
  - **Priority** : Priority [1] = MINIMUM → Inherited from *Priority_Based_Params*.
  - **Preassigned** : Assertion [1] = No → Inherited from *Priority_Based_Params*.
- References: -

Class **Polling_Params**

- Semantics. Activities scheduled with these parameters use a scheduling mechanism by which there is a periodic fixed priority task that polls for the arrival of its input event. Thus, execution of the event may be delayed until the next period. The polling action has an overhead even if the event has not arrived.
- Ancestors hierarchy: *Scheduling_Parameters.Priority_Based_Params*
- Direct subclasses: -
- Attributes:
  - **Priority** : Priority [1] = MINIMUM → Inherited from *Priority_Based_Params*.
  - **Preassigned** : Assertion [1] = No → Inherited from *Priority_Based_Params*.
  - **Polling_Period** : Time [1] = 0.0 → Period of the polling task.
  - **Polling_Worst_Overhead** : Normalized_Execution_Time [1] = 0.0 → Maximum overhead of the polling task.
  - **Polling_Avg_Overhead** : Normalized_Execution_Time [1] = 0.0 → Average overhead of the polling task.
  - **Polling_Best_Overhead** : Normalized_Execution_Time [1] = 0.0 → Minimum overhead of the polling task.
- References: -

Class **Periodic_Server_Params**

- Semantics. Activities with these parameters are scheduled under the periodic server scheduling algorithm. Following this algorithm, for each time interval equal to the Replenishment_Period, the task can be scheduled only during a time equal to Initial_Capacity. The capacity is available at the beginning of the replenishment period and is not preserved if the activity is not ready of terminates. The available capacity is consumed according to the preemptive fixed priority scheduling rules.
- Ancestors hierarchy: *Scheduling_Parameters.Priority_Based_Params*
- Direct subclasses:

  - Sporadic_Server_Params
- Attributes:
  - **Priority** : Priority [1] = MINIMUM → Inherited from *Priority_Based_Params*.
  - **Preassigned** : Assertion [1] = No → Inherited from *Priority_Based_Params*.
  - **Initial_Capacity** : Time [|] = 0.0 → It is the initial value of the execution capacity. Its default value is zero.
  - **Replenishment_Period** : Time [1] = 0.0 → It is the period after which the capacity consumed is replenished.

Class **Sporadic_Server_Params**

- Semantics. Activities with these parameters are scheduled under the sporadic server scheduling algorithm. The task is assigned an initial capacity that is preserved until it is consumed. Portions of capacity that are consumed are replenished after one replenishment period. When the task has execution capacity available it is scheduled with its normal priority. Once the capacity becomes zero the priority is reduced to the Background_Priority, and restored back to the normal priority when more capacity becomes replenished.
- Ancestors hierarchy: *Scheduling_Parameters.Priority_Based_Params*.Periodic_Server_Params
- Direct subclasses: -
- Attributes:
  - **Priority** : Priority [1] = MINIMUM → Inherited from *Priority_Based_Params*.
  - **Preassigned** : Assertion [1] = No → Inherited from *Priority_Based_Params*.
  - **Initial_Capacity** : Time [1] = 0.0 → Inherited from *Periodic_Server_Params*.
  - **Replenishment_Period** : Time [1] = 0.0 → Inherited from Periodic_Server_Params.
  - **Background_Priority** : Priority [1] = MINIMUM → The priority at which the task executes when there is no available execution capacity.
  - **Max_Pending_Replenishments** : Positive [1] = 1 → It is the maximum number of simultaneously pending replenishment operations.
- References: -

Class **Periodic_Server_Comm_Params**

- Semantics. Communication channels with these parameters transmit their messages under the periodic server scheduling algorithm. When sending a message, only a number of bits equal to Initial_Capacity are sent for each time interval equal to Replenishment_Period. This capacity is

not preserved if there is no message ready or is the messages available get transmitted before the capacity is consumed.

- Ancestors hierarchy: *Scheduling_Params.Priority_Based_Params*
- Direct subclasses:
  - Sporadic_Server_Comm_Params.
- Attributes:
  - **Priority** : Priority [1] = MINIMUM → Inherited from *Priority_Based_Params*.
  - **Preassigned** : Assertion [1] = No → Inherited from *Priority_Based_Params.*
  - **Initial_Capacity** : Bit_Count [1] = 0 → It is the initial value of the number of bits that can be sent.
  - **Replenishment_Period** : Time [1] = 0.0 → It is the period after which the capacity spent is replenished.
- References: -

Class **Sporadic_Server_Comm_Params**

- Semantics. Communication channels with these parameters transmit their messages under the sporadic server scheduling algorithm. The channel is assigned an initial capacity that is preserved until it is consumed. Portions of capacity that are consumed are replenished after one replenishment period. When the channel has execution capacity available it is scheduled with its normal priority. Once the capacity becomes zero the priority is reduced to the Background_Priority, and restored back to the normal priority when more capacity becomes replenished.
- Ancestors hierarchy: *Scheduling_Params.Priority_Based_Params*.Periodic_Server_Comm_Params
- Direct subclasses: -
- Attributes:
  - **Priority** : Priority [1] = MINIMUM → Inherited from *Priority_Based_Params*.
  - **Preassigned** : Assertion [1] = No → Inherited from *Priority_Based_Params*.
  - **Initial_Capacity** : Bit_Count [1] = 0 → Inherited from Periodic_Server_Comm_Params.
  - **Replenishment_Period** : Time [1] = 0.0 → Inherited from Periodic_Server_Params.
  - **Background_Priority** : Priority [1] = MINIMUM → The priority at which the channel sends its messages when there is no available capacity.
  - **Max_Pending_Replenishments** : Positive [1] = 1 → It is the maximum number of simultaneously pending replenishment operations.

- References: -

4.4.1.2 EDF SCHEDULING PARAMETERS

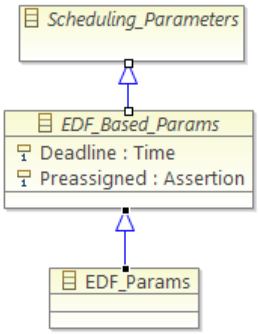

Figure 17 – Hierarchy of EDF scheduling parameters

<<abstract>> class **EDF_Based_Params**

- Semantics. These scheduling parameters are only applicable to schedulers with EDF policy
- Ancestors hierarchy: *Scheduling_Parameters*
- Direct subclasses:
  - EDF_Params
- Attributes:
  - **Deadline** : Time [1] = MAXIMUM → Relative deadline of the associated schedulable resource.
  - **Preassigned** : Assertion [1] = No → If this parameter is set to the value "No", the deadline may be assigned by one of the deadline assignment tools. Otherwise, the deadline is fixed and cannot be changed by those tools.
- References: -

Class **EDF_Params**

- Semantics. Concrete EDF scheduling parameters.
- Ancestors hierarchy: *Scheduling_Parameters.EDF_Based_Params*
- Direct subclasses: -
- Attributes:
  - **Deadline** : Time [1] = MAXIMUM → Inherited from *EDF_Based_Params*.
  - **Preassigned** : Assertion [1] = No → Inherited from *EDF_Based_Params*.
- References: -

4.4.1.3 RESOURCE RESERVATION SCHEDULING PARAMETERS

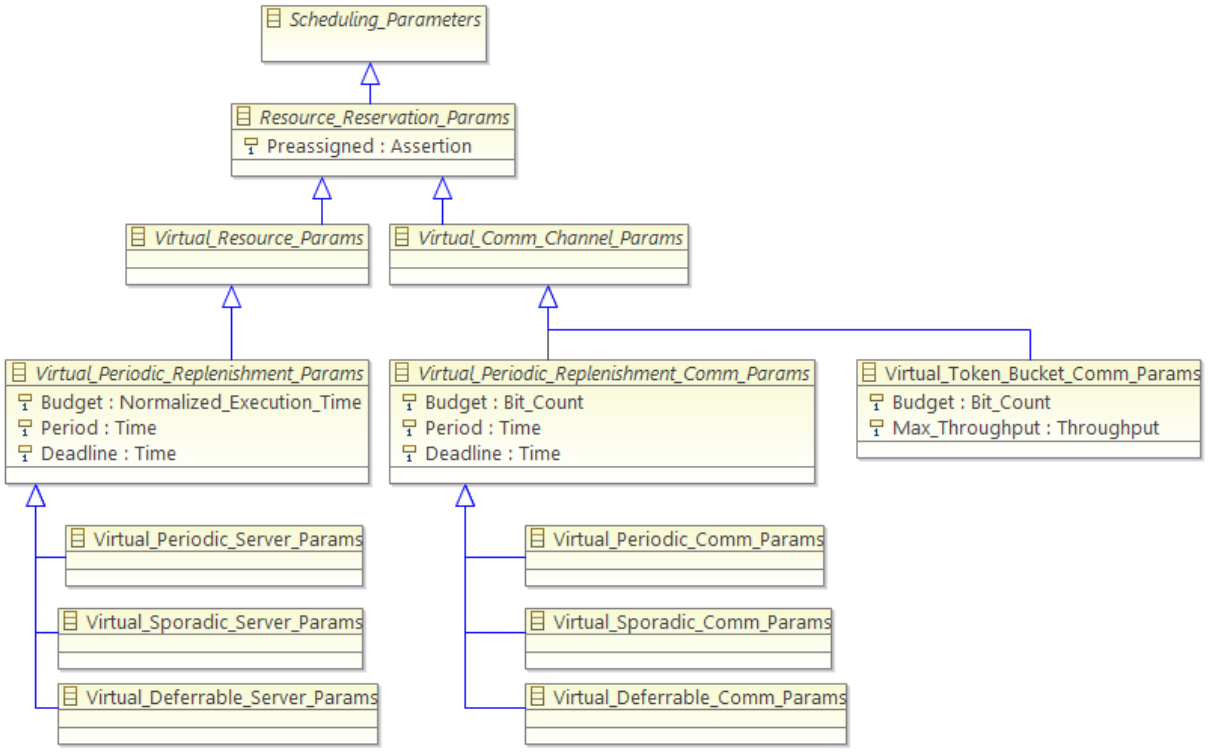

Figure 18- Hierarchy of resource reservation params

<<abstract>> class **Resource_Reservation_Params**

- Semantics. The scheduling parameters are only applicable to virtual schedulable resources (Virtual_Communication_Channel or Virtual_ Schedulable_Resource) as their Resource_Reservation_Params association.
- Ancestors hierarchy: *Scheduling_Parameters*
- Direct subclasses:
  - *Virtual_Resource_Params*
  - *Virtual_Comm_Channel_Params*
- Attributes:
  - **Preassigned** : Assertion [1] = No → If this parameter is set to the value "No", the scheduling parameters may be assigned by one of the scheduling design tools. Otherwise, the parameters are fixed and cannot be changed by those tools.
- References: -

<<abstract>> class **Virtual_Resource_Params**

- Semantics. Resource reservation contract for a Resource Reservation policy. It represents a service contract that contains the information related to the application minimum resource requirements.
- Ancestors hierarchy: *Scheduling_Parameters.Resource_Reservation_Params*
- Direct subclasses:
  - *Virtual_Periodic_Replenishment_Params*
- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from *Resource_Reservation_Params*.
- References: -

<<abstract>> class **Virtual_Periodic_Replenishment_Params**

- Semantics. The server executes any ready tasks until its capacity is exhausted. The server budget will be filled up a policy defined in the concrete classes, depending on a period attribute. The server guarantees that a piece of work of size less than or equal to the minimum budget and requested for a server with full capacity will be completed by the server's deadline.
- Ancestors hierarchy:
  Scheduling_Parameters.Resource_Reservation_Params.Virtual_Resource_Params
- Direct subclasses:
  - Virtual_Periodic_Server_Params
  - Virtual_Deferrable_Server_Params
  - Virtual_Sporadic_Server_Params
- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from *Resource_Reservation_Params*
  - **Budget** : Normalized_Execution_Time [1] = 0.0 → Minimum execution capacity per server period.
  - **Deadline** : Time [1] = MAXIMUM → The server guarantees that a piece of work of size less than or equal to the minimum budget and requested for a server with full capacity will be completed by the server's deadline.
  - **Period** : Time [1] = MAXIMUM → The period of the replenishment mechanism. The virtual resource will guarantee that every period, the part of the application running on it will get, if requested at the start of the period, at least the minimum budget on the processing resource on which the associated schedulable resource is running.

- References: -

Class **Virtual_Periodic_Server_Params**

- Semantics. The Periodic Server is invoked with a fixed period and executes any ready tasks until its capacity is exhausted. Note each application is assumed to contain an idle task that continuously consumes the available capacity and so on, therefore the server's capacity is fully consumed during each period. Once the server's capacity is exhausted, the server suspends execution until its capacity is replenished at the start of its next period. If a task arrives before the server's capacity has been exhausted then it will be serviced. Execution of the server may be delayed and or pre-empted by the execution of other servers at a higher priority. The server guarantees that a piece of work of size less than or equal to the minimum budget and requested for a server with full capacity at the start of the period will be completed by the server's deadline. The release jitter of the Periodic Server is assumed to be zero.
- Ancestors hierarchy:
  Scheduling_Parameters.Resource_Reservation_Params.Virtual_Resource_Params.Virtual_Periodic_Replenishment_Params
- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from *Resource_Reservation_Params*
  - **Budget** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Virtual_Periodic_Replenishment_Params*
  - **Deadline** : Time [1] = MAXIMUM → Inherited from *Virtual_Periodic_Replenishment_Params*
  - **Period** : Time [1] = MAXIMUM → Inherited from *Virtual_Periodic_Replenishment_Params*
- References: -

Class **Virtual_Deferrable_Server_Params**

- Semantics. The Virtual_Deferrable_Server allows any of its clients to use its resource any time within the period until its budget is exhausted. The server budget will be filled up periodically with the specified period. The budget cannot be saved for future use, which means that any unclaimed budget left from the previous replenishment is always thrown away at the next replenishment.
- Ancestors hierarchy:
  Scheduling_Parameters.Resource_Reservation_Params.Virtual_Resource_Params.Virtual_Periodic_Replenishment_Params
- Direct subclasses: -

- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from *Resource_Reservation_Params*
  - **Budget** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Virtual_Periodic_Replenishment_Params*
  - **Deadline** : Time [1] = MAXIMUM → Inherited from *Virtual_Periodic_Replenishment_Params*
  - **Period** : Time [1] = MAXIMUM → Inherited from *Virtual_Periodic_Replenishment_Params*
- References: -

Class **Virtual_Sporadic_Server_Params**

- Semantics. The sporadic server allows any of its clients to use its resource any time until its budget is exhausted. The sporadic server replenishes each portion of consumed budget one period after the associated task was ready.
- Ancestors hierarchy: *Scheduling_Parameters.Resource_Reservation_Params.Virtual_Resource_Params.Virtual_Periodic_Replenishment_Params*
- Direct subclasses: -
- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from *Resource_Reservation_Params*
  - **Budget** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Virtual_Periodic_Replenishment_Params*
  - **Deadline** : Time [1] = MAXIMUM → Inherited from *Virtual_Periodic_Replenishment_Params*
  - **Period** : Time [1] = MAXIMUM → Inherited from *Virtual_Periodic_Replenishment_Params*
- References: -

<<abstract>> class **Virtual_Comm_Channel_Params**

- Semantics. Reservation of a communication channel contract for a network with Resource Reservation policy. It represents a service contract that contains the information related to the application minimum resource requirements.
- Ancestors hierarchy: *Scheduling_Parameters.Resource_Reservation_Params*
- Direct subclasses:
  - *Virtual_Periodic_Replenishment_Comm_Params*
  - Virtual_Tocken_Bucket_Comm_Channel

- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from *Resource_Reservation_Params*
- References: -

<<abstract>> class **Virtual_Periodic_Replenisment_Comm_Params**

- Semantics. The Virtual_Periodic_Reprenishment_Comm_Channel sends any message until its capacity is exhausted. The server budget will be filled up according to a policy that depends on the concrete class and depends on the period attribute. The Virtual communication channel guarantees that a piece of message of size less than or equal to the minimum budget and requested for a channel with full capacity at the start of the period will be completed by the channel's deadline.
- Ancestors hierarchy:
  Scheduling_Parameters.Resource_Reservation_Params.*Virtual_Comm_Channel_Params*
- Direct subclasses:
  - Virtual_Periodic_Comm_Params
  - Virtual_Deferrable_Comm_Params
  - Virtual_Sporadic_Comm_Params
- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from Resource_Reservation_Params**.**
  - **Budget** : Bit_Count [1] = 0 → Minimum transmission capacity per server period.
  - **Deadline** : Time [1] = MAXIMUM → The server guarantees that a piece of message of size less than or equal to the budget and requested for a server with full capacity at the start of the period will be completed by the server's deadline.
  - **Period** : Time [1] = MAXIMUM → The period of the replenishment mechanism. The virtual channel will guarantee that every period, the messages being sent through it will get, if they are available at the start of the period, at least the minimum budget on the network to which the associated schedulable resource is connected.

Class **Virtual_Periodic_Comm_Params**

- Semantics. The Virtual_Periodic_Comm_Channel sends any available message until its capacity is exhausted. Once the server's capacity is exhausted, the virtual communication channel suspends transmission until its capacity is replenished at the start of its next period. Transmission on the server may be delayed and or pre-empted by the transmission of other channels at higher priorities. The Virtual communication channel guarantees that a piece of message of size less than or equal to the minimum budget and requested for a channel with full capacity at the start of the period will be completed by the channel's deadline. The release jitter of the Virtual_Periodic_Comm_Channel is assumed to be zero.

- Ancestors hierarchy:
  Scheduling_Parameters.Resource_Reservation_Params.Virtual_Comm_Channel_Params.Virtual_Periodic_Replenishment_Comm_Params
- Direct subclasses: -
- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from *Resource_Reservation_Params*
  - **Budget** : Normalized_Execution_Time = 0.0 → Inherited from *Virtual_Periodic_Replenishment_Comm_Params*
  - **Deadline** : Time = MAXIMUM → Inherited from *Virtual_Periodic_Replenishment_Comm_Params*.
  - **Period** : Time = MAXIMUM → Inherited from *Virtual_Periodic_Replenishment_Comm_Params*

Class **Virtual_Deferrable_Comm_Params**

- Semantics. The Virtual_Deferrable_Comm_Channel allows any of its clients to transmit any bits within the period, until its budget is exhausted. The channel budget will be filled up periodically with the specified period. The budget cannot be saved for future use, which means that any unclaimed budget left from the previous replenishment is always thrown away at the next replenishment.
- Ancestors hierarchy:
  Scheduling_Parameters.Resource_Reservation_Params..Virtual_Comm_Channel_Params.Virtual_Periodic_Replenishment_Comm_Params
- Direct subclasses: -
- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from *Resource_Reservation_Params*
  - **Budget** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Virtual_Periodic_Replenishment_Comm_Params*.
  - **Deadline** : Time [1] = MAXIMUM → Inherited from *Virtual_Periodic_Replenishment_Comm_Params*.
  - **Period** : Time [1] = MAXIMUM → Inherited from *Virtual_Periodic_Replenishment_Comm_Params*.

Class **Virtual_Sporadic_Comm_Params**

- Semantics. The sporadic server allows any of its clients to use its resource any time until its budget is exhausted. The sporadic server replenishes each portion of consumed budget one

period after the associated task was ready. The Virtual_Sporadic_Comm_Server transmits available messages until its budget is exhausted. The sporadic channel replenishes each portion of consumed budget one period after the associated messages were available for transmission.

- Ancestors hierarchy:
  Scheduling_Parameters.Resource_Reservation_Params.Virtual_Comm_Channel_Params.Virtual_Periodic_Replenishment_Comm_Params
- Direct subclasses: -
- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from *Resource_Reservation_Params*.
  - **Budget** : Normalized_Execution_Time [1] = 0.0 → Inherited from *Virtual_Periodic_Replenishment_Comm_Params.*
  - **Deadline** : Time = MAXIMUM → Inherited from Virtual_Periodic_Replenishment_Comm_Params.
  - **Period** : Time = MAXIMUM → Inherited from Virtual_Periodic_Replenishment_Comm_Params.
- References: -

Class **Virtual_Token_Bucket_Comm_Params**

- Semantics. The token bucket is a control mechanism that dictates when traffic can be transmitted, based on the presence of tokens in the bucket. The algorithm can be conceptually understood as follows :
  - A token is added to the bucket every 1 / Max_Throughput seconds.
  - The bucket can hold at the most budget tokens. If a token arrives when the bucket is full, it is discarded.
  - When a packet of n bits arrives, n tokens are removed from the bucket, and the packet is sent to the network.
  - If fewer than n tokens are available, no tokens are removed from the bucket, and the packet is enqueued for subsequent transmission when sufficient tokens have accumulated in the bucket.
- Ancestors hierarchy:
  Scheduling_Parameters.Resource_Reservation_Params.Virtual_Comm_Channel_Params
- Direct subclasses: -
- Attributes:
  - **Preassigned** : Assertion [1] = No → Inherited from *Resource_Reservation_Params*
  - **Budget** : Bit_Count [1] = 0 → The maximum number of tokens in the bucket.

- **Max_Throughput** : Throughput [1] = MAXIMUM → The rate of token flow in the bucket, or the average flow of bits in the channel.

4.4.1.4 TIMETABLE DRIVEN SCHEDULING PARAMETERS

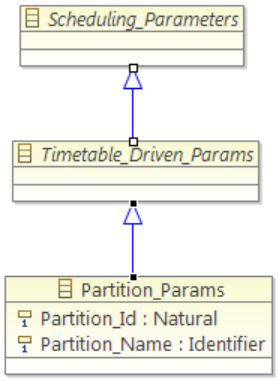

Figure 19 – Hierarchy of timetable-driven scheduling parameters

<<abstract>> class **Timetable_Driven_Params**

- Semantics. These scheduling parameters are only applicable to schedulers with the Timetable_Driven_Policy.
- Ancestors hierarchy: *Scheduling_Parameters*
- Direct subclasses:
  - Partition_Params
- Attributes: -
- References: -

Class **Partition_Params**

- Semantics. Concrete scheduling parameters for a Timetable_Driven policy
- Ancestors hierarchy: *Scheduling_Params.Timetable_Driven_Params*
- Direct subclasses: -
- Attributes:
  - **Partition_Id** : Natural [1] → Optimal identifier of the assigned partition.
  - **Partition_Name** : Identifier [1] = null → ~~Optional~~ identifier that is used as the partition name, for expressiveness purposes. Each partition name must correspond to a unique partition Id.
- References: -

4.4.1.5 AFDX VIRTUAL LINK SCHEDULING PARAMETERS

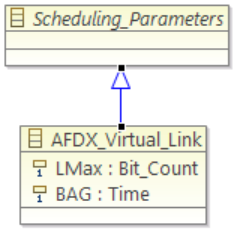

Figure 20 – AFDX scheduling parameters

Class **AFDX_Virtual_Link**

- Semantics. Concrete scheduling parameters for the AFDX_Policy
- Ancestors hierarchy: *Scheduling_Parameters*
- Direct subclasses: -
- Attributes:
  - **LMax** : Bit_Count [1] = 0 → Maximum length of the packets sent through this virtual link.
  - **BAG** : Time [1] = MAXIMUM → Bandwidth allocation gap. This is the minimum time that must elapse between the transmission of two packets from this virtual link. It is constrained to values representing intervals in milliseconds equal to a power of 2 value in the range {1,128].
- References: -

4.4.2 SYNCHRONIZATION PARAMETERS

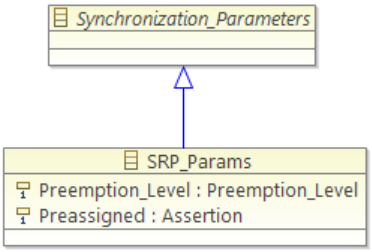

Figure 21 – Hierarchy of synchronization parameters

<<abstract>> class **Synchronization_Parameters**

- Semantics. These parameters are attached to a schedulable resource to specify the parameters used by that resource when performing a mutually exclusive access to shared resources using a mutual exclusion resource. The synchronization parameters should be specified whenever the scheduling policy is such that the given Scheduling Parameters do not have enough information

for the synchronization protocols used. For example, no synchronization parameters are needed for the priority inheritance or immediate priority ceiling protocols, because the only information they require from the server is its priority.

- Ancestors hierarchy:
- Direct subclasses:
  - SRP_Params
- Attributes: -
- References: -

Class **SRP_Params**

- Semantics. It is associated with the synchronization protocol named Stack Resource Protocol.
- Ancestors hierarchy: *Synchronization_Parameters*
- Direct subclasses: -
- Attributes:
  - **Preemption_Level** : Preemption_Level [1] = MINIMUM → A positive number that represents the level of preemptability of the schedulable resource in relation to the mutual exclusion resource. It must be within the valid ranges for the implementation
  - **Preassigned** : Assertion [1] = No → If this parameter is set to the value "No", the value of the preemption level may be assigned by any of the specific design tools. Otherwise, it is fixed and cannot be changed by those tools.
- References: -

4.5 MUTUAL EXCLUSION RESOURCES

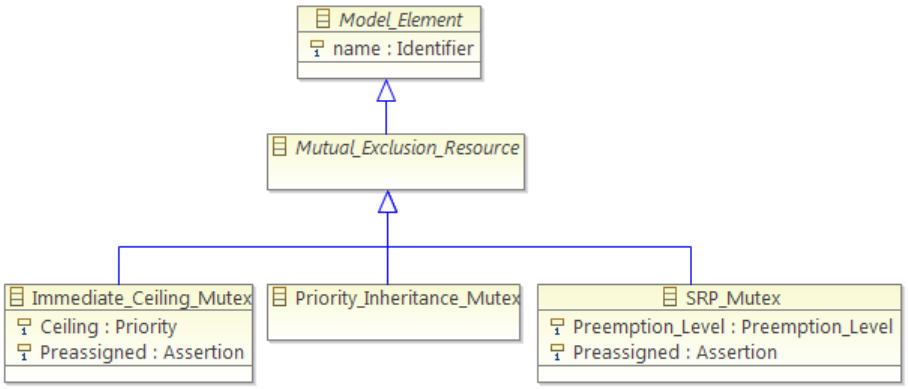

Figure 22 – Hierarchy of mutual exclusion resources

<<abstract>> class **Mutual_Exclusion_Resource**

- Semantics. It models a resource that is shared among different threads or tasks, and that must be used in a mutually exclusive way. Only protocols that avoid unbounded priority inversion are allowed.
- Ancestors hierarchy: *Model_Element*
- Direct subclasses:
  - Immediate_Ceiling_Mutex
  - Priority_Inheritance_Mutex
  - SRP_Mutex
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

Class **Immediate_Ceiling_Mutex**

- Semantics. Uses the immediate priority ceiling resource protocol. This is equivalent to Ada's Priority Ceiling, or the POSIX priority protect protocol.
- Ancestors hierarchy: *Model_Element.Mutual_Exclusion_Resource*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.

- **Ceiling** : Priority [1] = MAXIMUM → Priority ceiling used for the resource. May be computed automatically by the tool, upon request.
- **Preassigned** : Assertion [1] = No → If this parameter is set to the value "No", the priority ceiling may be assigned by the "Calculate Ceilings" tool. Otherwise, the priority ceiling is fixed and cannot be changed by those tools.

Class **Priority_Inheritance_Mutex**

- Semantics. Uses the basic priority inheritance protocol.
- Ancestors hierarchy: *Model_Element.Mutual_Exclusion_Resource*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

Class **SRP_Mutex**

- Semantics. Uses the Stack Resource Protocol (SRP). This is similar to the immediate ceiling protocol but works for non-priority-based policies.
- Ancestors hierarchy: *Model_Element.Mutual_Exclusion_Resource*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
  - **Preemption_Level** : Preemption_Level [1] = MAXIMUM → Level of preemptability used for the resource. May be computed automatically by the MAST tools, upon request.
  - **Preassigned** : Assertion [1] = No → If this parameter is set to the value "No", the preemption level may be assigned by a tool. Otherwise, the preemption level is fixed and cannot be changed by any tool.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

4.6 OPERATIONS

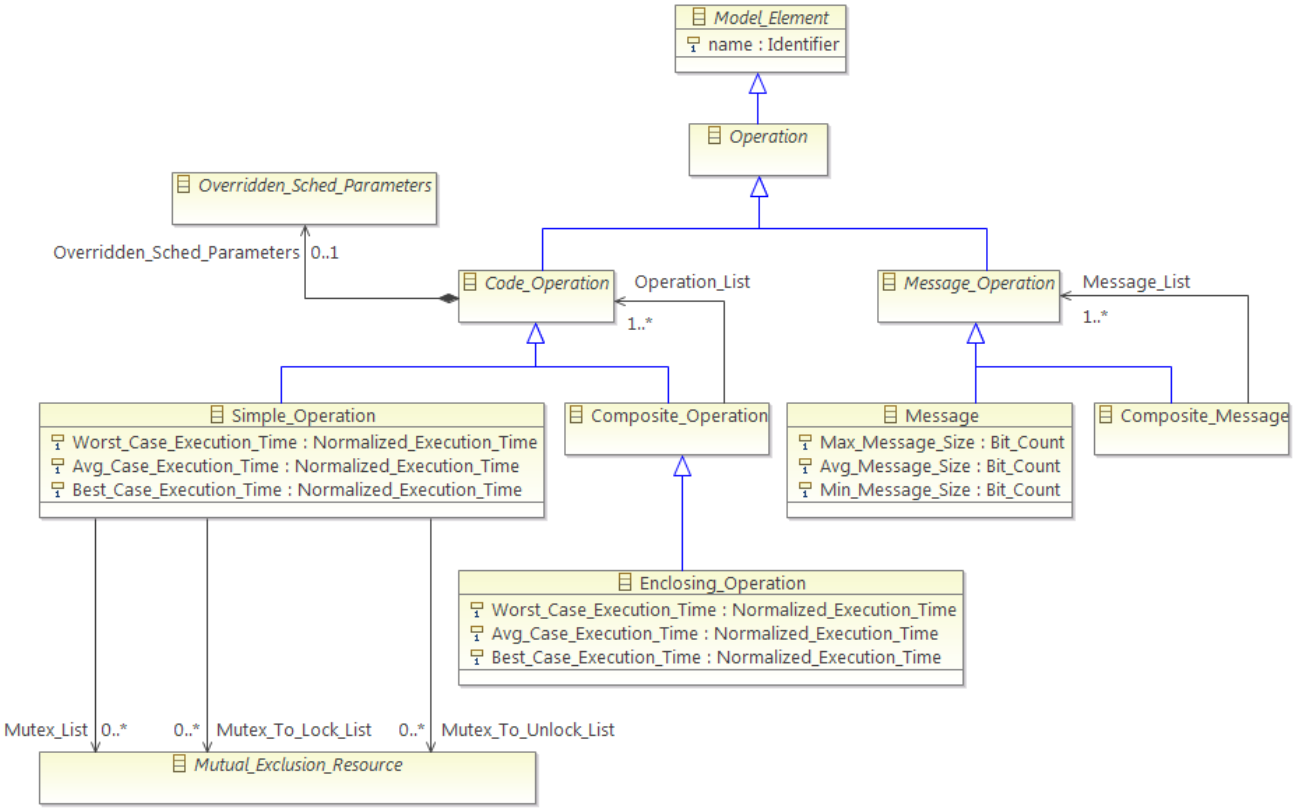

Figure 23 – Hierarchy of operations

<<abstract>> class **Operation**

- Semantics. It describes the processing capacity usage which is required for the execution of a piece of code or for the transmission of a message.
- Ancestors hierarchy: *Model_Element*
- Direct subclasses:
  - *Code_Operation*
  - *Message_Operation*
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

<<abstract>> class **Code_Operation**

- Semantics. It describes the processing capacity usage, measured in normalized execution time, which is required for the execution of a piece of code.
- Ancestors hierarchy: *Model_Element.Operation*
- Direct subclasses:
  - Simple_Operation
  - Composite_Operation
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Overridden_Sched_Parameters** : Overridden_Sched_Parameters [0 .. 1] → Optional overridden scheduling parameters

Class **Simple_Operation**

- Semantics. It represents a simple piece of sequential code execution.
- Ancestors hierarchy: *Model_Element.Operation.Code_Operation*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
  - **Worst_Case_Execution_Time** : Normalized_Execution_Time [1] = MAXIMUM → In normalized units, it represents the maximum execution time of the code.
  - **Avg_Case_Execution_Time** : Normalized_Execution_Time [1] = MAXIMUM → In normalized units, it represents the average execution time of the code.
  - **Best_Case_Execution_Time** : Normalized_Execution_Time [1] = 0.0 → In normalized units, it represents the minimum execution time of the code.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Overridden_Sched_Parameters** : Overridden_Sched_Parameters [0 .. 1] → Inherited from *Code_Operation*.
  - <Referenced> **Mutex_List** : Mutual_Exclusive_Resource [1 .. *] → List of references to the mutual exclusion resources that must be locked before executing the operation, and unlocked after executing the operation.

- <Referenced> **Mutex_To_Lock_List** : Mutual_Exclusive_Resource [1 .. *] → List of references to the mutual exclusion resources that must be locked before executing the operation.
- <Referenced> **Mutex_To_Unlock_List** : Mutual_Exclusive_Resource [1 .. *] → List of references to the mutual exclusion resources that must be unlocked after executing the operation.

Class **Composite_Operation**

- Semantics. It represents an operation composed of an ordered sequence of other operations, simple or composite. The execution time attribute of this class cannot be set, because it is the sum of the execution times of the comprised operations.
- Ancestors hierarchy: *Model_Element.Operation.Code_Operation*
- Direct subclasses:
  - Enclosing_Operation
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Overridden_Sched_Parameters** : Overridden_Sched_Parameters [0 .. 1] → Optional overridden scheduling parameters
  - <Referenced> **Operation_List** : Code_Operation [1 .. *] → List of references to other operations with which is built.

Class **Enclosing_Operation**

- Semantics. It represents an operation that contains other operations as part of its execution. The execution time is not the sum of execution times of the comprised operations, because other pieces of code may be executed in addition. The enclosed operations need to be considered for the purpose of calculating the blocking times associated with their mutual exclusion resource usage.
- Ancestors hierarchy: *Model_Element.Operation.Code_Operation*.Composite_Operation
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.

  - **Worst_Case_Execution_Time** : Normalized_Execution_Time [1] = MAXIMUM → In normalized units, it represents the maximum execution time of the code.
  - **Avg_Case_Execution_Time** : Normalized_Execution_Time [1] = MAXIMUM → In normalized units, it represents the average execution time of the code.
  - **Best_Case_Execution_Time** : Normalized_Execution_Time [1] = 0.0 → In normalized units, it represents the minimum execution time of the code.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Overridden_Sched_Parameters** : Overridden_Sched_Parameters [0 .. 1] → Inherited from Operation. Overridden parameters in enclosing operations take precedence over the respective overridden parameters of the contained operations.
  - <Referenced> **Operation_List** : Code_Operation [1 .. *] → Inherited from Composite_Operation

<<abstract>> Class **Message_Operation**

- Semantics. It describes the processing capacity usage which is required for the transmission of a message.
- Ancestors hierarchy: *Model_Element.Operation*
- Direct subclasses:
  - Message
  - Composite_Message
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

Class **Message**

- Semantics. Represents a message to be transmitted through a network.
- Ancestors hierarchy: *Model_Element.Operation.Message_Operation*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.

- **Max_Message_Size** : Bit_Count [1] = MAXIMUM → Maximum length of the message in Bit_Count units. The transmission time is obtained by dividing the size by the Throughput and by the Speed_Factor of the corresponding Network.
- **Avg_Message_Size** : Bit_Count [1] = MAXIMUM → Average length of the message in Bit_Count units. The transmission time is obtained by dividing the size by the Throughput and by the Speed_Factor of the corresponding Network.
- **Min_Message_Size** : Bit_Count [1] = 0 → Minimum length of the message in Bit_Count units. The transmission time is obtained by dividing the size by the Throughput and by the Speed_Factor of the corresponding Network.

- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*

Class **Composite_Message**

- Semantics. It represents a message composed of other messages, simple or composite, to be transmitted through a network
- Ancestors hierarchy: *Model_Element.Operation.Message_Operation*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Referenced> **Message_List** : Message_Operation [1 .. *] → List of references to other messages with which is built.

4.6.1 Overriden scheduling parameters

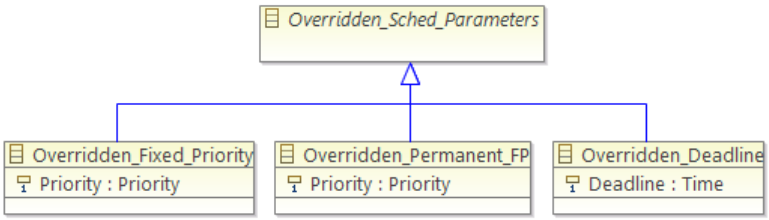

Figure 24 – Hierarchy of overriden scheduling parameters

<<abstract>> class **Overridden_Sched_Parameters**

- Semantics. It models scheduling parameters defined in the code, which should be used by the schedulable resource when executing the associated operation. Overridden parameters in composite or enclosing operations take precedence over the respective overridden parameters of the contained operations. The overridden parameters are set in the Schedulable_Resource before it locks the Mutual_Exclusion_Resources referenced by the operation.
- Ancestors hierarchy:
- Direct subclasses:
  - Overridden_Fixed_Priority
  - Overridden_Permanent_FP
  - Overridden_Deadline
- Attributes: -
- References: -

Class **Overridden_Fixed_Priority**

- Semantics. It represents a priority level that overrides the normal priority level at which the operation would execute. The change of priority is in effect only until the operation is completed.
- Ancestors hierarchy: *Overridden_Sched_Parameters*
- Direct subclasses: -
- Attributes:
  - **Priority**: Priority [1] = MAXIMUM → The overridden priority value.
- References: -

Class **Overridden_Permanent_FP**

- Semantics. It represents a priority level that overrides the normal priority level at which the operation would execute. The change of priority is in effect until another operation with permanent overridden priority starts, or until the end of the segment of activities (a segment is a contiguous sequence of operations executed by the same schedulable resource).
- Ancestors hierarchy: *Overridden_Sched_Parameters*
- Direct subclasses: -
- Attributes:
  - **Priority** : Priority [1] = MAXIMUM → The overridden priority value**.**
- References: -

Class **Overridden_EDF**

- Semantics. It represents a relative deadline that overrides the normal scheduling deadline at which the operation would execute. The change of scheduling deadline is in effect only until the operation is completed.
- Ancestors hierarchy: *Overridden_Sched_Parameters*
- Direct subclasses: -
- Attributes:
  - **Deadline** : Time [1] = MAXIMUM → The overridden relative deadline value.
- References: -

4.7 END-TO-END FLOWS

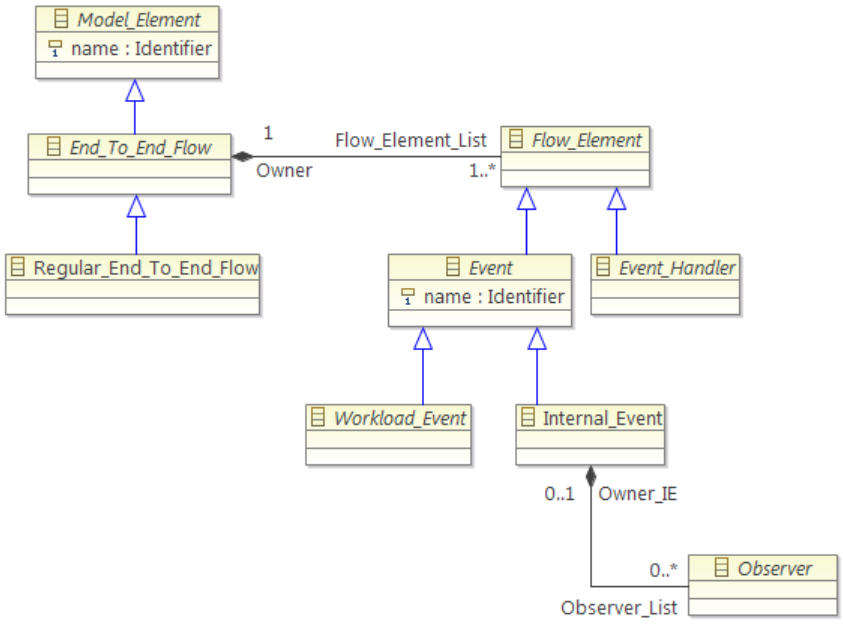

Figure 25 – End-to-end flow transaction model

<<abstract>> class **End_To_End_Flow**

- Semantics. It represents the set of activities and actions executed in the system in response to a workload event or an event arrival pattern. It is described as a graph with three different element types :

  ~~A list of workload events,~~

  ~~A list of internal events, with their timing requirements if any,~~

  ~~A list of event handlers.~~
- Ancestors hierarchy: *Model_Element*
- Direct subclasses:
  - Regular_End_To_End_Flow
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Flow_Element_List** : Flow_Element [1 .. *] → List of instances that comprise an end-to-end flow transaction, i.e. events (external, generated by the environment or by a timer as well as internal ones) and event handlers.

Class **Regular_End_To_End_Flow**

- Semantics. Concrete end-to-end-flow.
- Ancestors hierarchy: *Model_Element.End_To_End_Flow*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Model_Element*.
- References:
  - <Referenced> **Container** : Mast_Container [1] → Inherited from *Model_Element*
  - <Aggregated> **Flow_Element_List** : Flow_Element [1 .. *] → Inherited from *End_To_End_Flow*.

<<abstract>> class **Flow_Element**

- Semantics. It is the common ancestor of the classes which instances comprise an end-to-end flow transaction, i.e. events and event handlers.
- Ancestors hierarchy: -
- Direct subclasses:
  - *Event*
  - *Event_Handler*
- Attributes: -
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → The end-to-end flow transaction to which the element belongs to.

<<abstract>> class **Event_Handler**

- Semantics. Event handlers represent actions that are activated by the arrival of one or more events, and that in turn generate one or more events at their output. There are two fundamental classes of event handlers. The Steps represent the execution of an operation by a schedulable resource, in a processing resource, and with some given scheduling parameters. The other event handlers are just a mechanism for handling events, with no runtime effects.
- Ancestors hierarchy: *Flow_Element*
- Direct subclasses:
  - Step
  - Delay
  - Offset

  - Rate_Divisor
  - Fork
  - Join
  - Merge
  - Branch
  - Queried_Branch
  - *Message_Event_Handler*
- Attributes: -
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*

<<abstract>> class **Event**

- Semantics. It represents an event stream containing a potentially infinite number of individual event instances. Each event instance activates an instance of a step, or influences the behaviour of the event handler to which it is directed. Events may be internal or workload events generated by a clock or by the environment.
- Ancestors hierarchy: *Flow_Element*
- Direct subclasses:
  - Internal_Event
  - *Workload_Event*
- Attributes:
  - **name** : Identifier [1] → Name of the event
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*

Class **Internal_Event**

- Semantics. They are generated by an event handler.
- Ancestors hierarchy: *Flow_Element.Event*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Event*.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*

- <Aggregated> **Observer_List** : Observer [*] → List of requirements associated to the event, to be observed.

<<abstract>> class **Workload_Event**

- Semantics. It represents a stream of events that are generated by the environment or by a system clock or a timer. The Timed Events have a reference to the timer.
- Ancestors hierarchy: *Flow_Element.Event*
- Direct subclasses:
  - *Timed_Event*
  - Sporadic_Event
  - Unbounded_Event
  - Bursty_Event
- Attributes:
  - **name** : Identifier [1] → Inherited from *Event*
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*

<<abstract>> class **Observer**

- Semantics. It represents non-functional requirements associated to an internal event. The analysis tools will observe whether the requirement is satisfied or not.
- Ancestors hierarchy:
- Direct subclasses:
  - *Timing_Requirement*
  - Queue_Size_Req
  - Composite_Observer
- Attributes: -
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → A reference to the internal event to which the observer is assigned, in case it is not included within a composite observer.
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → A reference to the parent observer in which the observer is included.

4.7.1 WORKLOAD EVENTS

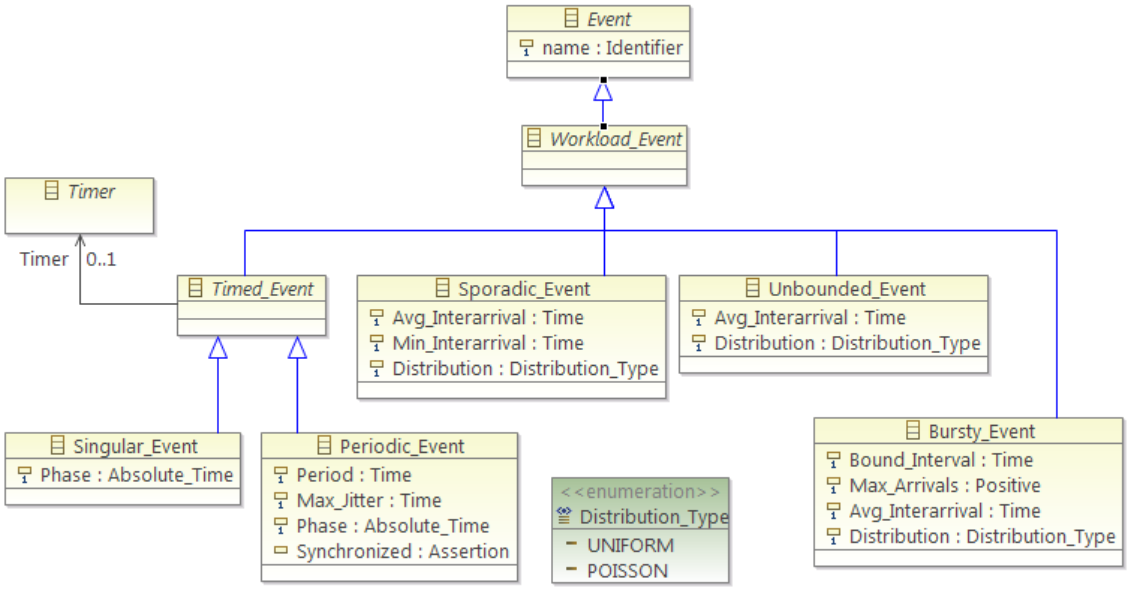

Figure 26 – Hierarchy of workload events

<<abstract>> class **Timed_Event**

- Semantics. Events generated by a timer, either implicit or explicit.
- Ancestors hierarchy: *Flow_Element.Event.Workload_Event*
- Direct subclasses:
  - Singular_Event
  - Periodic_Event
- Attributes:
  - **name** : Identifier [1] → Inherited from *Event*.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Timer** : Timer [0 .. 1] → It references the timer that generates the event. If it is null the event is generated by the environment though an implicit timer with no overhead.

Class **Singular_Event**

- Semantics. It represents an event that is generated only once.
- Ancestors hierarchy: *Flow_Element.Event.Workload_Event.Timed_Event*
- Direct subclasses: -

- Attributes:
  - **name** : Identifier [1] → Inherited from *Event*.
  - **Phase** : Absolute_Time [1] = 0.0 → It is the instant of the activation.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Timer** : Timer [0 .. 1] → Inherited from *Timed_Event*.

Class **Periodic_Event**

- Semantics. It represents a stream of events that are generated periodically.
- Ancestors hierarchy: *Flow_Element.Event.Workload_Even.Timed_Event*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Event*.
  - **Period** : Time [1] = 0.0 → Event period.
  - **Phase** : Absolute_Time [1] = 0.0 → It is the instant of the activation, if it had no jitter. After that time, the following events are periodic (possibly with jitter).
  - **Max_Jitter** : Time [1] = 0.0 → The event jitter is an amount of time that may be added to the activation time of each event instance, and is bounded by the maximum jitter attribute. It influences the schedulability of the system.
  - **Synchronized** : Assertion [0 .. 1] = NO → It describes whether or not the event is synchronized with the partition table (in case of Timetable-driven policies) or with the virtual resource activation (in resource reservation policies). In Timetable-driven policies synchronized events are restricted to being exact multiples or submultiples of the MAF whereas in resource reservation policies they are restricted to being exact multiples or submultiples of the virtual resource period.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Timer** : Timer [0 .. 1] → Inherited from *Timed_Event*.

Enumeration **Distribution_Type**

- Semantics. It represents the distribution function of aperiodic events. It can be UNIFORM or POISSON.
- Values :

- **UNIFORM** → The distribution function is uniform.
- **POISSON** → The distribution function is of the Poisson type.

Class **Sporadic_Event**

- Semantics. It represents a stream of aperiodic events that have a minimum interarrival time.
- Ancestors hierarchy: *Flow_Element.Event.Workload_Event*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Event*.
  - **Avg_Interarrival** : Time [1] = 0.0 → Average interarrival time.
  - **Min_Interarrival** : Time [1] = 0.0 → Minimum time between the generation of two events.
  - **Distribution** : Distribution_Type [1] = UNIFORM → It represents the distribution function of the aperiodic events.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*

Class **Unbounded_Event**

- Semantics. It represents a stream of aperiodic events for which it is not possible to establish an upper bound on the number of events that may arrive in a given interval.
- Ancestors hierarchy: *Flow_Element.Event.Workload_Event*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Event*.
  - **Avg_Interarrival** : Time [1] = 0.0 → Average interarrival time.
  - **Distribution** : Distribution_Type [1] = UNIFORM → It represents the distribution function of the aperiodic events.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*

Class **Bursty_Event**

- Semantics. It represents a stream of aperiodic events that have an upper bound on the number of events that may arrive in a given interval. Within this interval, events may arrive with an arbitrarily low distance among them (perhaps as a burst of events).
- Ancestors hierarchy: *Flow_Element.Event.Workload_Event*
- Direct subclasses: -
- Attributes:
  - **name** : Identifier [1] → Inherited from *Event*.
  - **Bound_Interval** : Time [1] = 0.0 → Interval for which the amount of event arrivals is bounded.
  - **Max_Arrivals** : Positive [1] = 1 → Maximum number of events that may arrive in the Bound_Interval.
  - **Avg_Interarrival** :Time [1] = 0.0 → Average interarrival time.
  - **Distribution** : Distribution_Type [1] = UNIFORM → It represents the distribution function of the aperiodic events.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*

4.7.2 OBSERVERS

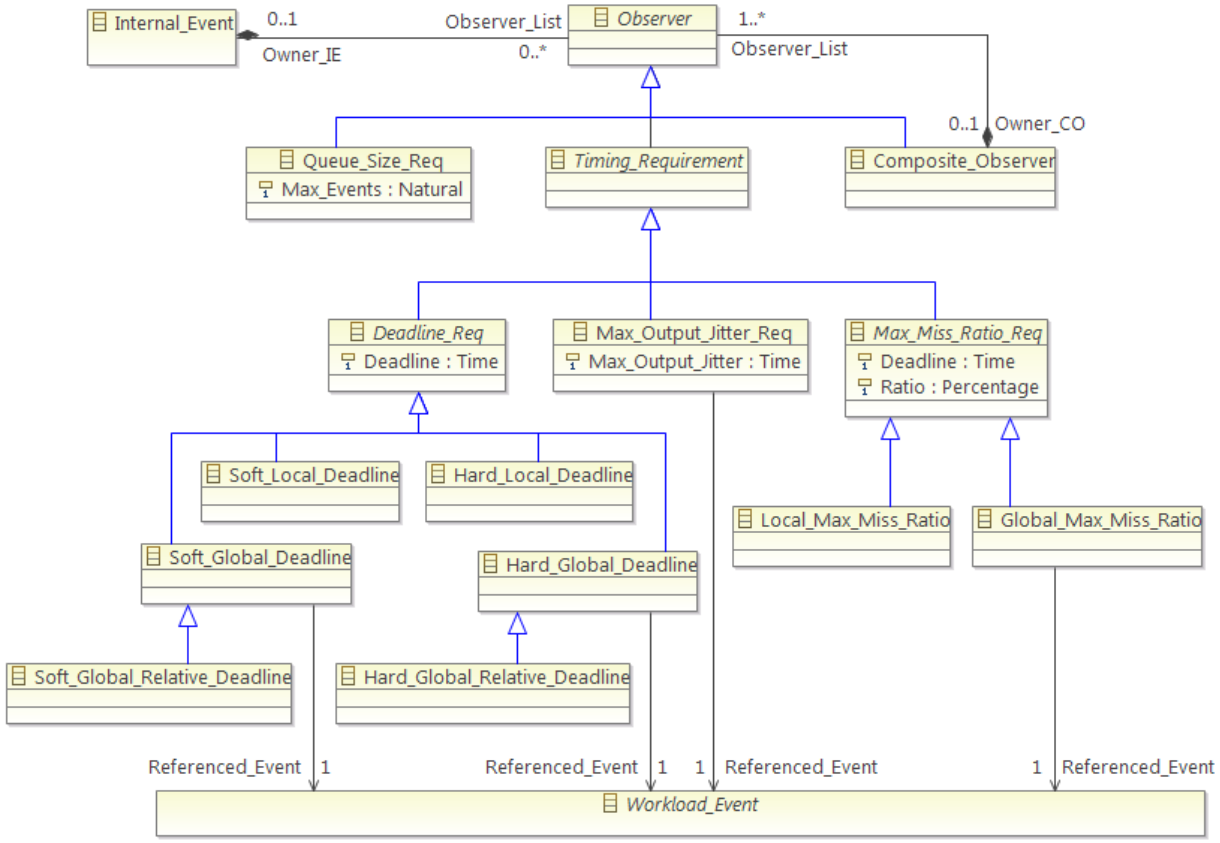

Figure 27 – Hierarchy of observers

Class **Queue_Size_Req**

- Semantics. It represents a requirement imposed on the maximum number of event instances that may be queued. It is used for internal events that activate a Step and, therefore, it represents the maximum number of pending activations of that Step.
- Ancestors hierarchy: *Observer*
- Direct subclasses: -
- Attributes:
  - **Max_Events** : Natural [1] = 0 → Maximum number of queued event instances that are admitted.
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*

<<abstract>> class **Timing_Requirement**

- Semantics. It represents requirements imposed on the instant of generation of the associated internal event.
- Ancestors hierarchy: *Observer*
- Direct subclasses:
  - *Deadline_Req*
  - Max_Output_Jitter_Req
  - *Max_Miss_Ratio_Req*
- Attributes: -
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*

<<abstract>> class **Deadline_Req**

- Semantics. It represents a requirement imposed on the maximum time value allowed for the generation of the associated event. It is expressed as a relative time interval relative to the generation of another event.
- Ancestors hierarchy: *Observer.Timing_Requirement*
- Direct subclasses:
  - Hard_Global_Deadline
  - Hard_Local_Deadline
  - Soft_Global_Deadline
  - Soft_Local_Deadline
- Attributes:
  - **Deadline** : Time [1] = 0.0 → The deadline value.
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*

Class **Hard_Global_Deadline**

- Semantics. As global, the deadline is relative to the workload event that activates the current execution. Specifically the deadline is relative to the exact generation time of the workload event (without taking into account the jitter). As hard, it must be met in all cases, including the worst case.

- Ancestors hierarchy: *Observer.Timing_Requirement.Deadline_Req*
- Direct subclasses:
  - Hard_Global_Relative_Deadline
- Attributes:
  - **Deadline** : Time [1] = 0.0 → Inherited from *Deadline_Req*
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Referenced_Event** : Workload_Event [1] → Workload event used as the time reference for the deadline.

Class **Hard_Global_Relative_Deadline**

- Semantics. As global, the deadline is relative to the workload event that activates the current execution. In this case the deadline is relative to the real generation time of the workload event (i.e. exact generation time + jitter). As hard, it must be met in all cases, including the worst case.
- Ancestors hierarchy: *Observer.Timing_Requirement.Deadline_Req*.Hard_Global_Deadline
- Attributes:
  - **Deadline** : Time [1] = 0.0 → Inherited from *Deadline_Req*.
- Direct subclasses: -
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Referenced_Event** : Workload_Event [1] → Inherited from Hard_Global_Deadline.

Class **Hard_Local_Deadline**

- Semantics. As local, it appears associated with the output event of a segment (see Section II.7.3 for the definition of segment) and the deadline is relative to the arrival of the event that activated that segment. As hard, it must be met in all cases, including the worst case.
- Ancestors hierarchy: *Observer.Timing_Requirement.Deadline_Req*
- Direct subclasses: -
- Attributes:

- **Deadline** : Time [1] = 0.0 → Inherited from *Deadline_Req*.

- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*

Class **Soft_Global_Deadline**

- Semantics. As global, the deadline is relative to a workload event that activates the current execution. As soft, it must be met on average.
- Ancestors hierarchy: *Observer.Timing_Requirement.Deadline_Req*
- Direct subclasses:
  - Soft_Global_Relative_Deadline
- Attributes:
  - **Deadline** : Time [1] = 0.0 → Inherited from *Deadline_Req*.
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Referenced_Event** : Workload_Event [1] → Workload event used as the time reference for the deadline.

Class **Soft_Global_Relative_Deadline**

- Semantics. As global, the deadline is relative to the workload event that activates the current execution. In this case the deadline is relative to the real generation time of the workload event (i.e. exact generation time + jitter). As soft, it must be met on average.
- Ancestors hierarchy: *Observer.Timing_Requirement.Deadline_Req.*Soft_Global_Deadline
- Direct subclasses: -
- Attributes:
  - **Deadline** : Time [1] = 0.0 → Inherited from *Deadline_Req*.
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Referenced_Event** : Workload_Event [1] → Inherited from Soft_Global_Deadline.

Class **Soft_Local_Deadline**

- Semantics. As local, it appears associated with the output event of a segment (see Section II.7.3 for the definition of segment); the deadline is relative to the arrival of the event that activated that segment. As soft, it must be met on average.
- Ancestors hierarchy: *Observer.Timing_Requirement.Deadline_Req*
- Direct subclasses: -
- Attributes:
  - **Deadline** : Time [1] = 0.0 → Inherited from *Deadline_Req*.
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*

Class **Max_Output_Jitter_Req**

- Semantics. It represents a requirement for limiting the output jitter with which the associated periodic internal event is generated. Output jitter is calculated as the difference between the worst-case response time and the best-case response time for the associated event, relative to a Referenced Event that is an attribute of this requirement.
- Ancestors hierarchy: *Observer.Timing_Requirement*
- Direct subclasses: -
- Attributes:
  - **Max_Output_Jitter** : Time [1] = 0.0 → Maximum jitter required.
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Referenced_Event** : Workload_Event [1] → Workload event used as the time reference for the deadline.

<<abstract>> class **Max_Miss_Ratio_Req**

- Semantics. It represents a kind of soft deadline in which the deadline cannot be missed more often than a specified ratio.
- Ancestors hierarchy: *Observer.Timing_Requirement*
- Direct subclasses:
  - Global_Max_Miss_Ratio

- Local_Max_Miss_Ratio

- Attributes:
  - **Deadline** : Time [1] = 0.0 → The deadline value.
  - **Ratio** : Percentage = 5.0 → Percentage representing the maximum ratio of missed deadlines.
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*

Class **Global_Max_Miss_Ratio**

- Semantics. Concrete Max_Miss_Ratio_Req, in which the deadline is relative to the Referenced Event.
- Ancestors hierarchy: *Observer.Timing_Requirement.Max_Miss_Ratio_Req*
- Direct subclasses: -
- Attributes:
  - **Deadline** : Time [1] = 0.0 → Inherited from *Max_Miss_Ratio_Req*.
  - **Ratio** : Percentage [1] = 5.0 → Inherited from *Max_Miss_Ratio_Req*.
- References:
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Referenced_Event** : Workload_Event [1] → Workload event that is used as the reference for the deadline.

Class **Local_Max_Miss_Ratio**

- Semantics. Concrete Max_Miss_Ratio_Req, in which the deadline is relative to the activation of the step to which the timing requirement is attached.
- Ancestors hierarchy: *Observer.Timing_Requirement.Max_Miss_Ratio_Req*
- Direct subclasses: -
- Attributes:
  - **Deadline** : Time [1] = 0.0 → Inherited from *Max_Miss_Ratio_Req*.
  - **Ratio** : Percentage [1] = 5.0 → Inherited from *Max_Miss_Ratio_Req*

- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - <Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*

Class **Composite_Observer**

- Semantics. An internal event may have several requirements imposed at the same time, which are expressed via a composite observer. It is just a list of simple observers, i.e., non-composite observers.
- Ancestors hierarchy: *Observer*.
- Direct subclasses: -
- Attributes: -
- References:
- References:
  - <Referenced> **Owner_IE** : Internal_Event [0 .. 1] → Inherited from *Observer*
  - < Referenced> **Owner_CO** : Observer [0 .. 1] → Inherited from *Observer*
  - <Aggregated> **Observer_List** : Observer [1 .. *] → List of simple observers.

4.7.3 STEP EVENT HANDLER

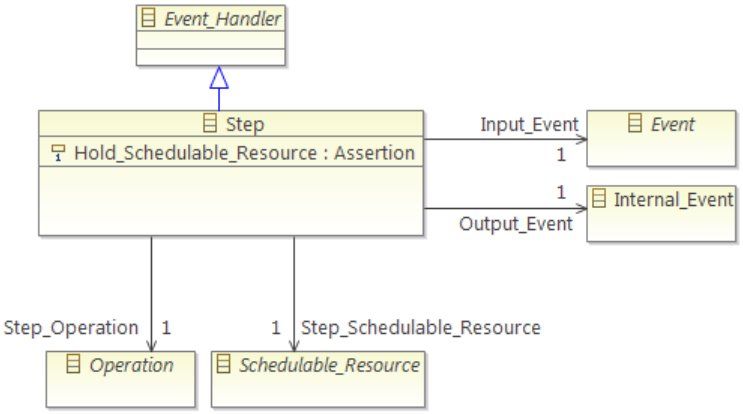

Figure 28 – The Step event handler

Class **Step**

- Semantics. It represents the execution of an operation by a schedulable resource, in a processing resource, and with some given scheduling parameters.
- Ancestors hierarchy: *Flow_Element.Event_Handler*

- Direct subclasses: -
- Attributes:
  - **Hold_Schedulable_Resource** : Assertion [1] = Yes → If this attribute is set, the thread used by the step is not released at the end of the execution of the step operation. The thread is held until the next step in which the attribute is set to "NO" or until the end-to-end flow is finished. This attribute only has effect for schedulable resources of the Thread class. The default value is NO.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Step_Schedulable_Resource** : Schedulable_Resource [1] → Reference to the schedulable resource (which in turn contains references to the scheduling parameters and the processing resource)
  - <Referenced> **Step_Operation** : Operation [1] → It references the operation.
  - <Referenced> **Input_Event** : Event [1] → It references the event that activates the execution of the step
  - <Referenced> **Output_Event** : Internal_Event [1] → It references the internal event that is generated when the step is finished.

Note : The concept of segment of step, used in several parts of the metamodel, is explained here. A segment is a contiguous set of steps executed by the same schedulable resource.

4.7.4 TIMED EVENT HANDLERS

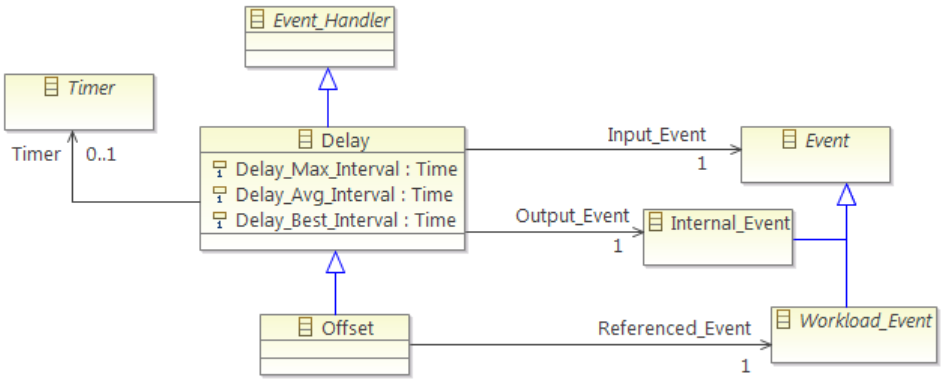

Figure 29 – The timed event handlers (Delay and Offset)

Class **Delay**

- Semantics. It is an event handler that generates its output event after a time interval has elapsed from the arrival of the input even. Note that the time intervals for different event instances are

independently computed, even if they overlap. The delay may be handled by a Timer or by the environment (when the Timer reference is null).

- Ancestors hierarchy: *Flow_Element.Event_Handler*
- Direct subclasses:
  - Offset
- Attributes:
  - **Delay_Max_Interval** : Time [1] = 0.0 → Longest time interval used to generate the output event.
  - **Delay_Avg_Interval** : Time [1] = 0.0 → Average time interval used to generate the output event.
  - **Delay_Min_Interval** : Time [1] = 0.0 → Shortest time interval used to generate the output event.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Input_Event** : Event [1] → It references the event that activates the step.
  - <Referenced> **Output_Event** : Internal_Event [1] → It references the internal event that is generated when the step is finished.
  - <Referenced> **Timer** : Timer [0 .. 1] → Reference to the timer, used when the delay has an overhead that must be modeled. The default is NULL, indicating that the overhead associated to the delay operation is negligible.

Class **Offset**

- Semantics. It is similar to the Delay event handler, except that the time interval is counted relative to the arrival of some (previous) workload event. If the time interval has already passed when the input event arrives, the output event is generated immediately.
- Ancestors hierarchy: *Flow_Element.Event_Handler*.Delay
- Direct subclasses: -
- Attributes:
  - **Delay_Max_Interval** : Time [1] = 0.0 → Inherited from Delay
  - **Delay_Avg_Interval** : Time [1] = 0.0 → Inherited from Delay
  - **Delay_Min_Interval** : Time [1] = 0.0 → Inherited from Delay
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*

- <Referenced> **Input_Event** : Event [1] → Inherited from Delay.
- <Referenced> **Output_Event** : Internal_Event [1] → Inherited from Delay
- <Referenced> **Timer** : Timer [0 .. 1] → Inherited from Delay.
- <Referenced> **Referenced_Event** :Workload_Event [1] → Reference to the workload event that is used as temporal reference.

4.7.5 FLOW CONTROL EVENT HANDLERS

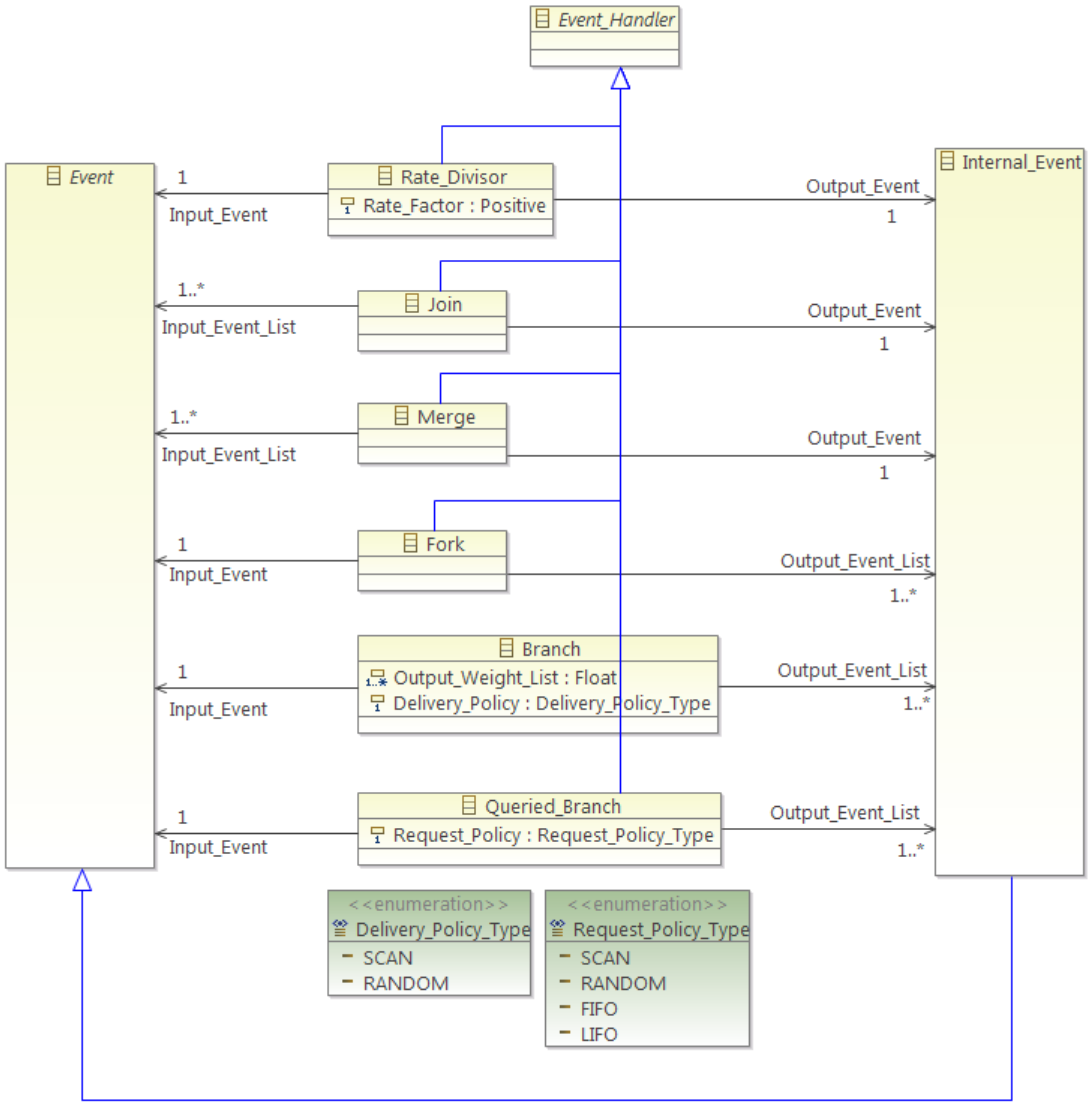

Figure 30 – The flow control event handlers

Class **Rate_Divisor**

- Semantics. It is an event handler that generates one output event when a number of input events equal to the Rate Factor have arrived.
- Ancestors hierarchy: *Flow_Element.Event_Handler*
- Direct subclasses: -
- Attributes:
  - **Rate_Factor** : Positive [1] = 1 → Number of events that must arrive to generate an output event.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Input_Event** : Event [1] → It references the input event for the event handler.
  - <Referenced> **Output_Event** : Internal_Event [1] → It references the internal event that is generated as output.

Class **Fork**

- Semantics. It is an event handler that generates one event in each of its outputs each time an input event arrives. If the step previous to the fork handler has a "YES" value assigned to its Hold_Schedulable_Resource attribute, only one of the output paths following the fork handler can have Steps assigned to the same Schedulable_Resource as that step. The schedulable resource is unlocked when one of those Steps has a "No" value assigned to its Hold_Schedulable_Resource attribute, or when the corresponding output path terminates.
- Ancestors hierarchy: *Flow_Element.Event_Handler*
- Direct subclasses: -
- Attributes: -
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Input_Event** : Event [1] → It references the input event the event handler.
  - <Referenced> **Output_Event_List** : Internal_Event [1 .. *] → List of references to the output events.

Class **Join**

- Semantics. It is an event handler that generates its output event when all of its input events have arrived. For worst-case analysis to be possible it is necessary that all the input events are periodic with the same periods. This usually represents no problem if the event handler is used to perform a "join" operation after a "fork" operation carried out with the Fork event handler. When the EDF

scheduling policy is being used, the generation time of the event used to obtain the absolute deadlines is set to the most recent of the generation times of the input event instances.

- Ancestors hierarchy: *Flow_Element.Event_Handler*
- Direct subclasses: -
- Attributes: -
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Input_Event_List** : Event [1 .. *] → List of references to the input events.
  - <Referenced> **Output_Event** : Internal_Event [1] → It references the event that is generated as output.

Class **Merge**

- Semantics. It is an event handler that generates its output event every time one of its input events arrives. When several pending event instances are queued in the Merge output, the policy assigned to the Event_Queueing_Policy attribute of the MAST_Model is used to determine which request must be processed first by the following step.
- Ancestors hierarchy: *Flow_Element.Event_Handler*
- Direct subclasses: -
- Attributes: -
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Input_Event_List** : Event [1 .. *] → List of references to input events.
  - <Referenced> **Output_Event** : Internal_Event [1] → It references the internal event that is generated as output.

Enumeration **Delivery_Policy_Type**

- Semantics. It is the policy used to determine the output path. It may be Scan (the output path is chosen in a cyclic fashion) or Random.
- Values:
  - **SCAN** → The output events are delivered to the outputs in a cyclic fashion.
  - **RANDOM** → The output events are delivered randomly.

Class **Branch**

- Semantics. It is an event handler that generates one event in only one of its outputs each time an input event arrives. The output path is chosen at the time of the event generation.
- Ancestors hierarchy: *Flow_Element.Event_Handler*
- Direct subclasses: -
- Attributes:
  - **Delivery_Policy :Delivery_Policy_Type** : RANDOM [1] → The policy used to determine the output path.
  - **Output_Weight_List** : Float [1 .. *] → List of weights for each output. For a RANDOM policy, if the weights are w1, w2, …, wi, the probability of choosing a particular output j is wj/wT, where wT is the total weight, wT = w1+w2+…+wi If policy is SCAN, the weight must be integer; if the weigths for this case are w1, w2, …, wi, the first w1 events will be directed through the first output, the next w2 events through the second output, and so on. The default value for the weight of each output is 1.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Input_Event** : Event [1] → Reference to the input event.
  - <Referenced> **Output_Event_List** : Internal_Event [1 .. *] → List of references to the output events.

Enumeration **Request_Policy_Type**

- Semantics. Is the policy used to determine the output path when there are several pending requests from the connected activities. It may be SCAN (the output path is chosen in a cyclic fashion), RANDOM, FIFO or LIFO.
- Values:
  - **SCAN** → The output events are delivered in a cyclic fashion.
  - **RANDOM** → The output events are delivered randomly.
  - **FIFO** → The output events are delivered with a FIFO policy.
  - **LIFO** → The output events are delivered with a LIFO policy.

Class **Queried_Branch**

- Semantics. It is an event handler that generates one event in only one of its outputs each time an input event arrives. The output path is chosen at the time of the event consumption by one of the Steps connected to an output event.

- Ancestors hierarchy: *Flow_Element.Event_Handler*
- Direct subclasses: -
- Attributes:
  - **Request_Policy :** Request_Policy_Type : SCAN → The policy used to determine the output path when there are several pending requests from the connected steps.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Input_Event** : Event [1] → Reference to the input event.
  - <Referenced> **Output_Event_List** : Event [1 .. *] → List of references to the output events

4.7.6 MESSAGE EVENT HANDLERS

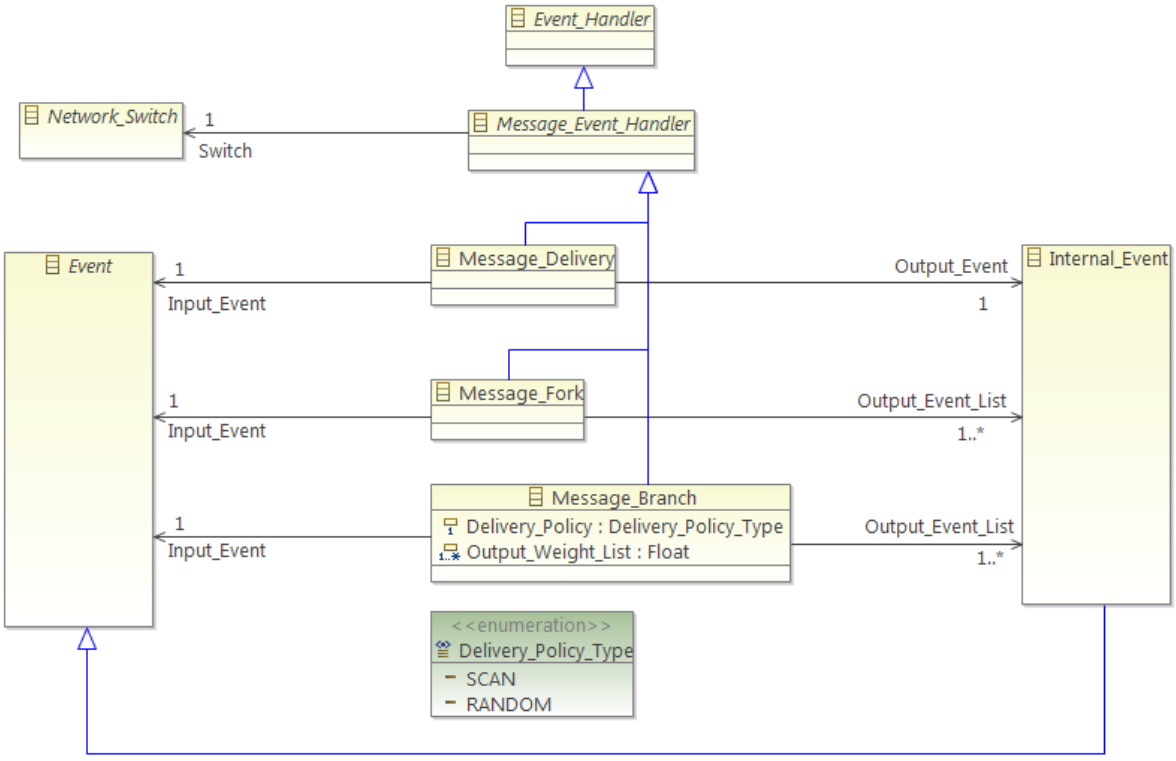

Figure 31 – The message flow event handlers

<<abstract>> class **Message_Event_Handler**

- Semantics. Event_Handler that handles the messages in switches and routers.
- Ancestors hierarchy: *Flow_Element.Event_Handler*
- Direct subclasses:

  - Message_Fork
  - Message_Delivery
  - Message_Branch
- Attributes: -
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Switch** : Network_Switch [1] → Reference to the switch in which the transfer is executed.

Class **Message_Fork**

- Semantics. An input message is transmitted through several output networks; it is a multicast operation.
- Ancestors hierarchy: *Flow_Element.Event_Handler.Message_Event_Handler*
- Direct subclasses: -
- Attributes: -
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Switch** : Switch_Network [1] → Inherited from *Message_Event_Handler*.
  - <Referenced> **Input_Event** : Event [1] → Reference to the input event
  - <Referenced> **Output_Event_List** : Internal_Event [1 .. *] → List of references to the output events.

Class **Message_Delivery**

- Semantics. It represents an input message that is delivered to one output network.
- Ancestors hierarchy: *Flow_Element.Event_Handler.Message_Event_Handler*
- Direct subclasses: -
- Attributes: -
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Switch** : Network_Switch [1] → Inherited from *Message_Event_Handler*.
  - <Referenced> **Input_Event** : Event [1] → Reference to the input event

- <Referenced> **Output_Event** : Internal_Event [1] → Reference to the output event.

Class **Message_Branch**

- Semantics. It is an event handler that models the management of messages in a router. It generates one event in only one of its outputs each time an input event arrives. The output path is chosen at the time of the event generation.
- Ancestors hierarchy: *Flow_Element.Event_Handler.Message_Event_Handler*
- Direct subclasses: -
- Attributes:
  - **Delivery_Policy** : Delivery_Policy_Type [1] = SCAN → The policy used to determine the output path.
  - **Output_Weight_List** : Float [1 .. *] → List of weights for each output. For a RANDOM policy, if the weights are w1, w2, …, wi, the probability of choosing a particular output j is wj/wT, where wT is the total weight, wT = w1+w2+…+wi. If policy is SCAN, the weight must be integer; if the weigths for this case are w1, w2, …, wi, the first w1 events will be directed through the first output, the next w2 events through the second output, and so on. The default value for the weight of each output is 1.
- References:
  - <Referenced> **Owner** : End_To_End_Flow [1] → Inherited from *Flow_Element*
  - <Referenced> **Switch** : Network_Router → It references the router in which the branching is executed. It is a specialized reference to the switch reference inherited from *Message_Event_Handler*.
  - <Referenced> **Input_Event** : Event [1] → Reference to the input event
  - <Referenced> **Output_Event_List** : Internal_Event [1 .. *] → List of references to the output events.