# Modeling and Analysis Suite for Real-Time Applications (MAST 2.0)

# Description of the MAST Model

| | | |
|---|---|---|
| By: | 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 |

## 1. Introduction

In this document we describe the basic characteristics of Version 2 of MAST, a Modeling and Analysis Suite for Real-Time Applications.

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.

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.

The main aspects of MAST are the following:

- MAST allows modelling both single-processor and complex distributed systems.
- A very rich model of the real time system is used. It is an event-driven model in which complex dependence patterns among the different tasks can be established. For example, tasks may be activated with the arrival of several events, or may generate several events

at their output. This makes it ideal for analysing real-time systems that have been designed using UML or similar design tools, which have event driven models of the system.

- The system model is independent of the actual analysis techniques. The toolset provides different techniques that can be used for comparison purposes.
- Different scheduling techniques are supported, such as fixed priorities, dynamic priorities with earliest deadline first (EDF) policy, time-partitioned scheduling, and mixed systems.
- The analysis tools include automatic calculation of blocking times caused by the use of mutual exclusion resources, sensitivity analysis, and automatic assignment of priorities and other scheduling parameters.
- The system model contains independent views for describing the execution platform, the software modules and data exchanged through the networks, the concurrent architecture, and the workload and flow of events for a particular configuration of the application.. This independence among the various elements of the model is ideal for building a full model through the composition of partial models developed independently.
- Schedulers may be composed in a hierarchical way.
- The toolset is open source and fully extensible. That means that other teams may provide enhancements.

## 2. The MAST Toolset

The MAST toolset includes the tools that appear in Figure 1:

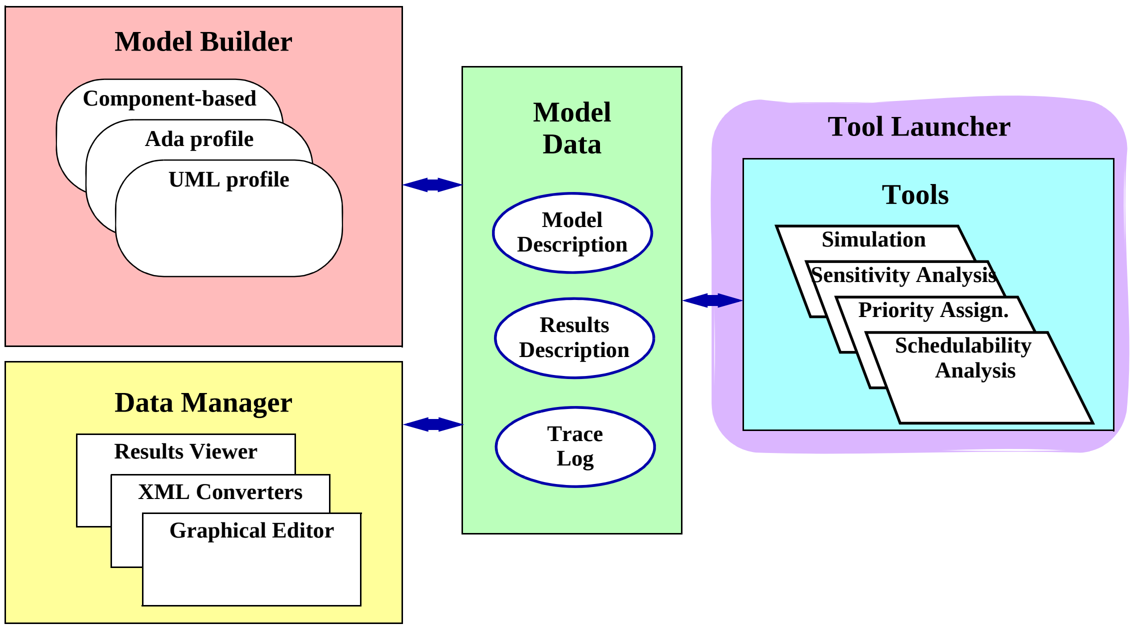


**Figure 1. MAST toolset environment**

- *Schedulability analysis* in single-processor and distributed systems. This is the main tool, and it checks that the worst-case behaviour always meets the hard real-time requirements. Figure 2 represents some internal details of the MAST schedulability analysis tool The system model is defined in a description time and a parser converts it

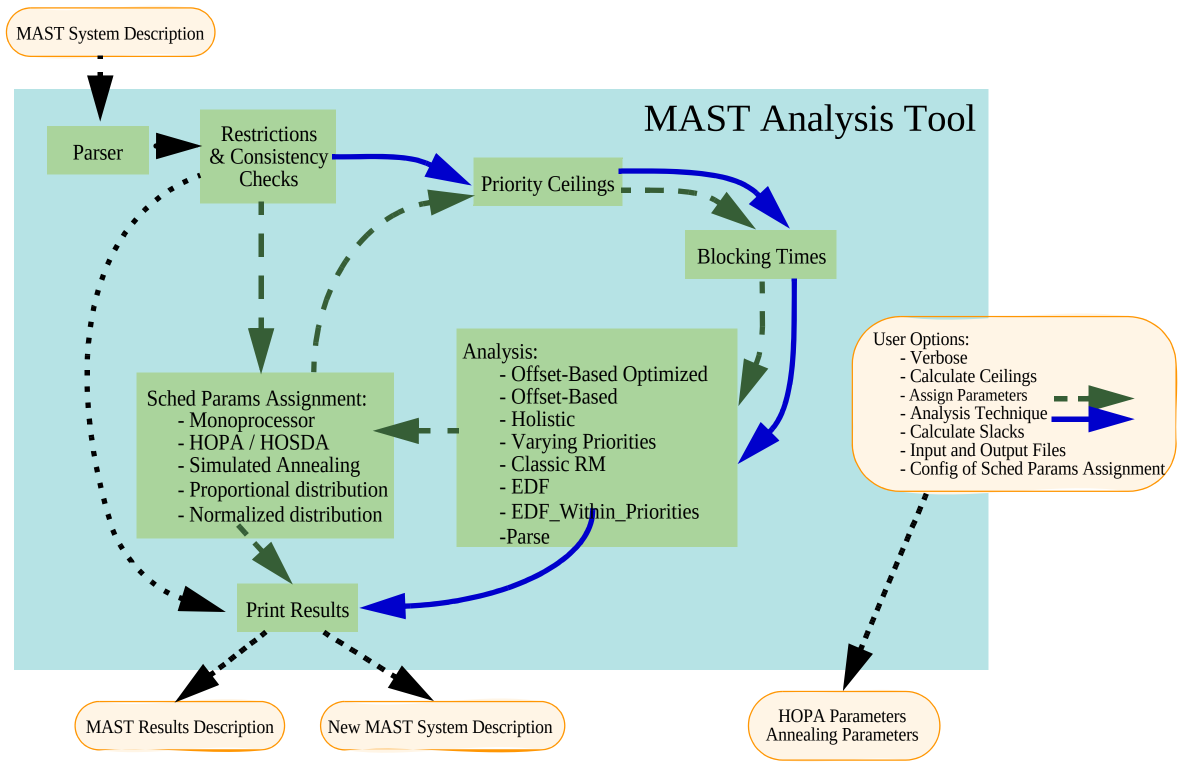


**Figure 2. MAST Analysis tools**

into a data structure that is used by the tools. A module is also available to convert the data structure back into the chosen model description.

The MAST analysis tools produce several output files:

- *Console output*: Describes the work carried out by the tools, and any possible errors, in free format. If the verbose option is set, the tools provide a more detailed output. The last line in the file contains the string "`Final analysis status: code`", where code is a single word that is either "DONE", or some error indication.
- *Source destination file*: Describes the source of the MAST model of the analysed system, including any elements introduced by the analysis tools into the system such as priorities, or priority ceilings. It follows the file format used for the MAST model. This file is only produced if the corresponding option is set.
- *Results file*: Describes the results of the analysis tools. If a filename is not provided for the results, they are written to the standard output, together with the Console Output. See Section 8 for a description of its format.

• *Automatic assignment of scheduling parameters*. This tool provides the user with capabilities to automatically calculate the following system parameters:

- optimum priorities for fixed priority scheduling, and scheduling deadlines for EDF scheduling. The automatic assignment uses optimum methods when available, and heuristics or optimization techniques when the optimum assignment is not available.

  - priority ceilings and preemption levels for mutual exclusion mutual exclusion resources

- *Sensitivity analysis*, through the calculation of slack values for the whole system, particular processing resources, end-to-end flows, or particular operations. A slack value is the percentage by which the associated element (for instance the processor speed, the execution time of an operation, or the execution time of an end-to-end flow or of the entire system) may be increased while keeping the system schedulable, or in case of negative values, the percentage by which the associated element has to be reduced to reach schedulability. These slack values are very useful in providing insight into which parts of the system must be modified to reach schedulability, or how much space there is for growth or for new steps in the system. The slack calculation tools repeat the analysis in a binary search algorithm in which execution times are successively increased or decreased.

- A *discrete-event simulator* tool obtains accurate data about average results and performance features. Likewise, the simulator lets us estimate worst and best case response parameters when the particular features of the model do not obey the restrictions of the worst-case schedulability analysis techniques. The simulation tools are able to simulate the behaviour of the system to check soft timing requirements and generate temporal traces of the simulated execution.

- A *graphical editor* generates the system description.

- A *results viewer* is available to view the analysis results in a convenient way.

- *XML converter.* The model and the results are specified through an ASCII description that serves as the input and output of the analysis tools. Two ASCII formats have been defined: a text special-purpose format and an XML-based format. The XML format provides the designer with capabilities to use free standard XML tools to validate, parse, analyse, and display the model files. The converter tool converts from one format to the other or back.

Using a standard UML tool, it is possible to describe the real-time behaviour of the system by means of a set of appropriate UML modeling primitives that are defined in different profiles in accordance with the technology used to design the real-time system (object oriented, Ada language, component based, etc.). Then, an automatic tool is used to compile the UML real-time view and to build the MAST real-time model description. No special framework is needed with this approach, but the designer must incorporate the real-time view into the UML description. Refer to the UML-MAST project page[1] for more information of this issue.

The capabilities of the currently implemented schedulability analysis tools are represented in the following tables. The tools with dark shading are still under development.

**Table 1. Fixed-priority schedulability analysis tools**

| **Technique** | **Single-Processor** | **Multi-Processor** | **Simple Transact.** | **Linear Transact.** | **Multiple Event T.** |
|---|---|---|---|---|---|
| Classic Rate Monotonic | ☑ |  | ☑ |  |  |
| Varying Priorities | ☑ |  | ☑ | ☑ |  |

1. http://mast.unican.es/umlmast/

**Table 1. Fixed-priority schedulability analysis tools**

| Technique | Single-Processor | Multi-Processor | Simple Transact. | Linear Transact. | Multiple Event T. |
|---|---|---|---|---|---|
| Holistic | ☑ | ☑ | ☑ | ☑ | |
| Offset Based | ☑ | ☑ | ☑ | ☑ | |
| Offset Based Optimized | ☑ | ☑ | ☑ | ☑ | |
| Multiple Event | ☑ | ☑ | ☑ | ☑ | ☑ |

**Table 2. EDF schedulability analysis tools**

| Technique | Single-Processor | Multi-Processor | Simple Transact. | Linear Transact. | Multiple Event T. |
|---|---|---|---|---|---|
| Single Processor | ☑ | | ☑ | | |
| EDF_Within_Priorities | ☑ | | ☑ | | |
| Holistic_Local | ☑ | ☑ | ☑ | ☑ | |
| Holistic_Global | ☑ | ☑ | ☑ | ☑ | |
| Offset Based | ☑ | ☑ | ☑ | ☑ | |

## 3. Real-Time System Model

A real-time situation, representing a particular model of operation of a real-time system, is modelled as a set of concurrent end-to-end flows (previously called transactions) that compete for the resources offered by the platform. Each end-to-end flow is activated from one or more workload events, and represents a set of steps or steps that are executed in the system. When a step finishes its execution it generates an event that is internal to the end-to-end flow, and that may in turn activate other steps. Special event handling structures exist in the model to handle events in special ways. Internal events may have observers associated with them, and representing timing requirements or other requirements that must be observed.

Figure 3 shows an example of a system with one of its end-to-end flows highlighted. End-to-end flows are represented through graphs showing the event flow. This particular end-to-end flow is activated by only one workload event. After two steps have been executed, a multicast event handling object is used to generate two events that activate the last two steps in parallel.

The “boxes” that are included in the end-to-end flow are called *Event Handlers*. As we have mentioned, there are event handlers that just manipulate events, like the *Fork* event handler in Figure 3. Another very important event handler is the *Step*, which represents the execution of an operation, i.e., a procedure or function in a processor, or a message transmission in a network.

The elements that define a step are described in Figure 4. We can see that each step is activated by one *input event*, and generates an *output event* when completed. If intermediate events need to be generated, the step would be partitioned into the appropriate parts. Each step executes an *Operation*, which represents a piece of code (to be executed on a processor), or a message (to

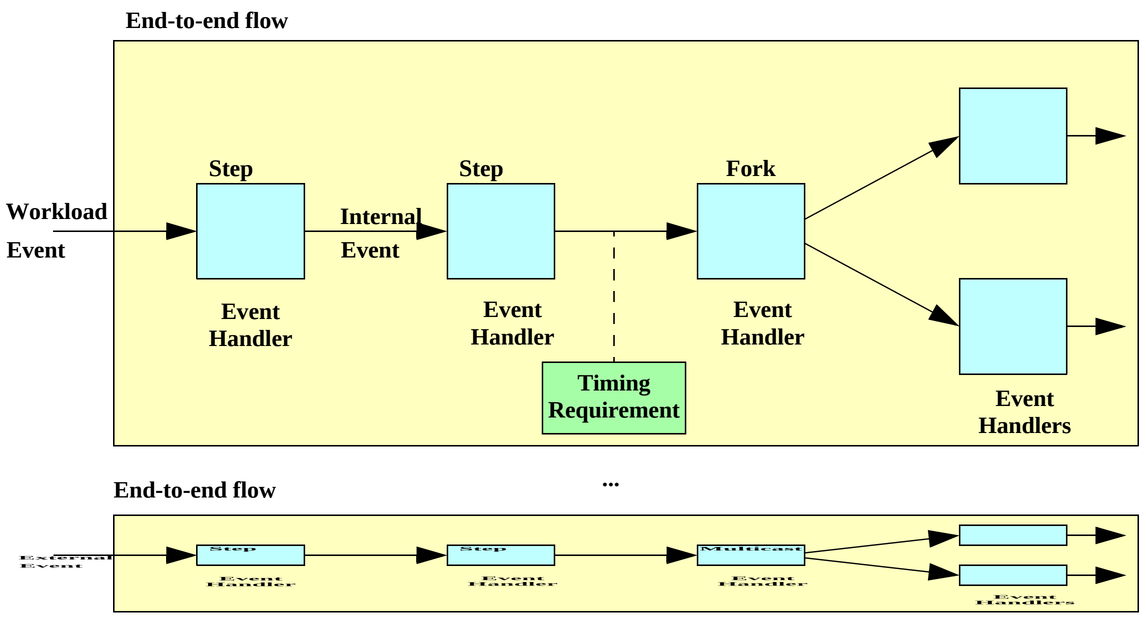


**Figure 3. Real-Time System composed of transactions**

be sent through a network). An operation may have a list of *Mutual Exclusion Resources* that it needs to use in a mutually exclusive way.

The step is executed by a *Schedulable Resource*, which represents a schedulable entity in the *Scheduler* to which it is assigned. This scheduler belongs to a *Processor* or a *Network*, although we will see that when hierarchical scheduling is modelled the situation is somehow more complex. For example, the model for a schedulable resource in a processor is a task or thread. A thread may be responsible of executing several steps (procedures). The schedulable resource is assigned a *Scheduling Parameters* object that contains the information on the scheduling policy and parameters used.

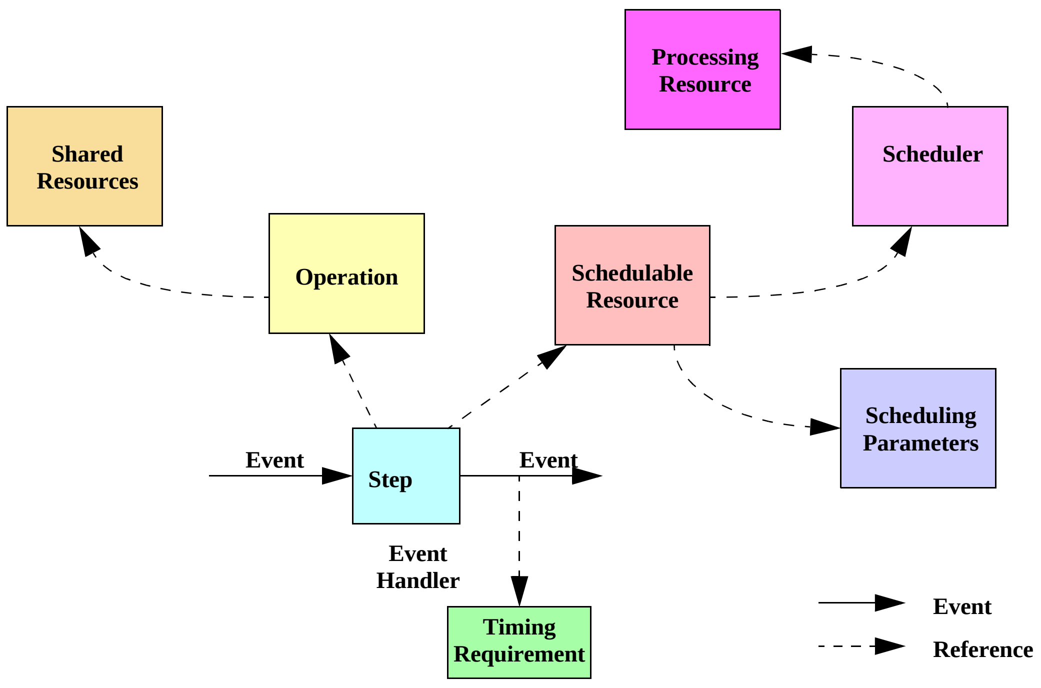


**Figure 4. Elements that define a step**

The MAST model is developed to be able to handle most real-time systems built using commercial standard operating systems and languages (e.g., POSIX, ARINC-664, Ada, ...). This implies that there is support for fixed priority scheduled systems, EDF, time-partitioned scheduling, as well as hierarchical schedulers. Among fixed priorities, different scheduling strategies are allowed:

- preemptive and non preemptive
- interrupt service routines
- sporadic servers
- polling

The emphasis of the MAST model is on event-driven systems in which each task may conditionally generate multiple events at its completion. A task may also be activated by a conditional combination of one or more events, through the following event handlers:

- merge
- join
- branch (with decision made by sender)
- queried_branch (branch with request from receiver)
- fork

The workload events arriving at the system should be of different kinds:

- periodic
- unbounded aperiodic
- sporadic
- bursty
- singular (arriving only once)

The system model is rich enough to facilitate the independent description of overhead parameters such as:

- Processor overheads
- Timer overheads
- Network overheads
- Network driver overheads

Timing requirements are allowed to be both hard and soft. Deadlines as well as maximum output jitter requirements are allowed.

A metamodel of the MAST model can be found in the document named “MAST Metamodel”.

## 4. Type definitions

The following types are used in the definitions of the components of the MAST File and the MAST Results File:

- *Identifier*. String of characters following the rules described in the following section.
- *Priority*. Positive integer of model-defined range that must be within [1…$2^{15}$-1], defining the scheduling priority of tasks.
- *Preemption_Level*. Positive integer of model-defined range that must be within [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 mutual exclusion resources.
- *Interrupt_Priority*. Positive integer of implementation defined range, defining the scheduling priority of interrupt service routines.
- *Any_Priority*. Positive integer that is either in the *Priority* range or in the *Interrupt_Priority* range.
- *Normalized_Execution_Time*. 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. It is obtained by multiplying the real execution time by the processing resource's speed factor.
- *Bit_Count*. Floating point number that represents the length 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.
- *Throughput*. Floating point number that represents the transmission bandwidth of a communication network in bits per time unit.
- *Time*. Floating point number that represents a time interval in unspecified time units.
- *Absolute_Time*. Floating point number that represents an absolute time measured from and arbitrary time origin, in unspecified units.
- *Float*. It represents any float type.
- *Positive*. Integer positive number (excluding zero).
- *Natural*. Integer number that is greater than or equal to zero.
- *Percentage*. 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.
- *"Text"*: String of arbitrary characters, excluding the double quote character, and delimited within double quotes.
- *Date-Time*: 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).
- *Pathname*: String representing a pathname of a file.

## 5. Writing the MAST File with the special-purpose format

The MAST Model can be specified using a special-purpose text format or an XML file. This section defines the special-purpose text format. In Appendix A, we describe the syntax and rules for writing the Mast files with the XML-Mast format.

The rules for writing the file with a real-time system according to the defined real-time system model are the following:

- Each object has the format:
  object_name (arguments);
- Most objects have a type and/or a name argument. In those cases, they are mandatory arguments, and they have to be defined as the first and second argument, respectively. All other arguments can go in any order, and are optional, except when marked.
- Blank spaces, tabs and new lines are ignored.
- Identifiers or names follow the Ada rules for composite identifiers: begin with a letter, followed by letters, digits, underscores ('_') or periods ('.').
- Identifiers or names can be expressed with or without quotes. A quoted name can be the same as one of the MAST-reserved words (appearing in bold face below).
- Float types without fractional part can be expressed without the decimal point.
- Comments are like in Ada: they begin with two dashes ("--"), anywhere in a line, and end at the end of the line.
- The description is not case-sensitive.

It is not necessary to define an identifier before it is used.

## 6. Elements of the MAST model

In this section we review in detail the particular classes and attributes of the different elements of the MAST model. The elements that we will review are:

- Overall system model
- Processing Resources
- Timers
- Network Drivers
- Schedulers (primary scheduler, secondary schedulers,....)
- Scheduling Policies (fixed priorities, EDF,...)
- Scheduling parameters (priorities, deadlines,...)
- Synchronization parameters (preemption levels,...)
- Schedulable Resources (tasks, processes, threads,...)
- Mutual exclusion resources (for mutually exclusive access)

- Operations (procedures, functions, messages,...)
- Events
- Observers
- Event Handlers
- End-to-end flows

## 6.1 Overall Model

A Real-time situation represents the overall MAST model of a real-time situation that a particular system may have, and that needs to be analysed. Global information about the real-time situation and the underlying implementation is described in the Model object, which contains the following attributes:

- *Model name*: an identifier containing the name of the modeled real-time situation.
- *Model date*: the date in which the real-time situation model was created.
- *System PiP Behaviour*: the behaviour 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. 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. The default value is STRICT.

```
Model (
     Model_Name                        => Identifier,
     Model_Date                        => YYYY-MM-DDThh:mm:ss,
     System_PiP_Behaviour[1]           => STRICT | POSIX);
```

## 6.2 Processing Resources

They model the processing capacity of a hardware component that executes some of the modelled system steps, which are generally pieces of code to be executed, operations carried out directly by hardware, or messages to be transferred.

Common attributes:

- *Name*. A string.
- *Speed factor*. 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.

1. The alternative spelling **`System_PiP_Behavior`** is also supported

- *Synchronization_Source*: A reference to 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 (i.e., not present) 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

Classes of Processing Resources: there is an abstract class, called *Computing_Resource*, that models a device capable of executing pieces of application code or operations carried out directly by hardware; another abstract class, called *Network*, models a communication system specialized in the transmission of messages; a third abstract class called *Network_Switch* models a communication subsystem capable of exchanging messages among different networks.

### 6.2.1 Computing resources

One concrete class is defined as a computing resource:

- *Regular Processor.* It represents a physical processor or a device; it contains attributes that describe the basic overheads associated with its hardware interrupt services. 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 modelled as an overhead called "ISR_Switch" and is modelled by three normalized execution times that define the worst, average, or best case overheads. It has the following additional attributes:
  - *Max Interrupt priority* and *Min Interrupt priority.* They define the range of priorities valid for steps executed as interrupt service routines. Their default values are the maximum and minimum values of the *Any_Priority* type, respectively.
  - *ISR Switch Overheads* (Worst, Average, Best execution time of an interrupt switch).
  - *Timer_List*. List of references to the hardware timers used. They influence the activation time of the timed activities 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. One of the timers in the list may be a system timer. If none of the timers is a system timer an implicit system timer is assumed, with zero overheads. The system timer is used as the time reference for the timetable-driven and global EDF policies.

```
Processing_Resource (
    Type                            => Regular_Processor,
    Name                            => Identifier,
    Speed_Factor                    => Float,
    Worst_ISR_Switch                => Normalized_Execution_Time,
    Avg_ISR_Switch                  => Normalized_Execution_Time,
    Best_ISR_Switch                 => Normalized_Execution_Time,
    Max_Interrupt_Priority          => Interrupt_Priority,
    Min_Interrupt_Priority          => Interrupt_Priority,
    Timer_List                      => (Identifier, Identifier, ...));
```

### 6.2.2 Networks

A network is an abstract class that models a communication system specialized in the transmission of messages among processors or network switches. It has the following attributes in addition to the Name and Speed_Factor:

- *Throughput*: Normalized network bandwidth in bits per time unit. The actual network throughput is affected by the *Speed Factor.* The default value is 0 bits per time unit.
- *List of Drivers*. A list of references to network drivers, that contain the processor overhead model associated with the transmission of messages through the network. See the description of the drivers below. The default is an empty list.

The network has two direct subclasses: Packet_Based_Network and AFDX_Link.

- *Packet Based Network*. It represents 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). A Packet_Based_Network has the following additional attributes:
  - *Transmission kind*: *Simplex, Half Duplex*, of *Full Duplex.* The default value is *Half_Duplex*.
    - 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.
  - *Max Blocking*. 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* and *Min Packet Size*. They describe the 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. 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. Their default values are the maximum value of the *Bit_Count* type.

```
Processing_Resource (
    Type                           => Packet_Based_Network,
    Name                           => Identifier,
    Speed_Factor                   => Float,
    Throughput                     => Float,
    Transmission                   => Simplex | Half_Duplex | Full_Duplex,
    Max_Blocking                   => Normalized_Execution_Time,
    Max_Packet_Size                  => Bit_Count,
    Min_Packet_Size                  => Bit_Count,
    List_of_Drivers                  => (
                                     Driver 1,
                                     Driver 2,
                                     ...));
```

- *RTEP_Network*. It is a specialization of the Packet_Based_Network that 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. The Transmission type must be HALF_DUPLEX. A RTEP_network has the following additional attributes:
    - *Number_Of_Stations*: Positive = 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. The default value is .
    - *Token_Delay*: Time. 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. 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. Maximum number of retransmissions that we allow for each packet containing a protocol token. The default value is 0.
    - *Packet_Transmission_Retries*: Natural = 0 => Maximum number of retransmissions that we allow for each packet with user information. The default value is 0.
    - *Arbitration_Token_Size*: Bit_Count. Number of bits of the message used for priority arbitration. The default value is 0.
    - *Transmit_Token_Size*: Bit_Count. Number of bits of the message used for appointing the next transmit node. The default value is 0.
    - *Transmit_Info_Size*: Bit_Count. Number of bits of the protocol information that is sent in a packet containing user information The default value is 0.

```
Processing_Resource (
    Type                             => RTEP_Network,
    Name                             => Identifier,
    Speed_Factor                     => Float,
    Throughput                       => Float,
    Transmission                     => Half_Duplex,
    Max_Blocking                     => Normalized_Execution_Time,
    Max_Packet_Size                  => Bit_Count,
    Min_Packet_Size                  => Bit_Count,
    Number_Of_Stations               => Integer,
    Token_Delay                         => Time,
    Failure_Timeout                     => Time,
    Token_Transmission_Retries          => Integer,
    Packet_Transmission_Retries         => Integer,
    Arbitration_Token_Size              => Bit_Count,
    Transmit_Info_Size                  => Bit_Count,
    Arbitration_Token_Size              => Bit_Count,
    List_of_Drivers                     => (
                                     Driver 1,
                                     Driver 2,
                                     ...));
```

- *AFDX_LINK*. 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. An AFDX_Link has the following additional attributes:
  - *Max Packet Size* and *Min Packet Size*. They describe the 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. Their default values are the maximum value of the *Bit_Count* type.
  - *Max_HW_Tx_Latency*: Time. Maximum technological latency in transmission in the AFDX hardware. The default value is a large time value.
  - *Min_HW_Tx_Latency*: Time. Minimum technological latency in transmission in the AFDX hardware. The default value is a large time value.
  - *Avg_HW_Tx_Latency*: Time. Average technological latency in transmission in the AFDX hardware. The default value is a large time value.
  - *Max_HW_Rx_Latency*: Time. Maximum technological latency in reception in the AFDX hardware. The default value is a large time value.
  - *Min_HW_Rx_Latency*: Time. Minimum technological latency in reception in the AFDX hardware. The default value is a large time value.
  - *Avg_HW_Rx_Latency*: Time. Average technological latency in reception in the AFDX hardware. The default value is a large time value.
  - *Ethernet_Overhead*: Bit_Count . 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. 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.

```
     Processing_Resource (
          Type                                => AFDX_Link,
          Name                                => Identifier,
          Speed_Factor                        => Float,
          Throughput                          => Float,
          Max_Packet_Size                     => Bit_Count,
          Min_Packet_Size                     => Bit_Count,
          Max_HW_Tx_Latency                => Time,
          Min_HW_Tx_Latency                => Time,
          Avg_HW_Tx_Latency                => Time,
          Max_HW_Rx_Latency                => Time,
          Min_HW_Rx_Latency                => Time,
          Avg_HW_Rx_Latency                => Time,
          Ethernet_Overhead                => 160, -- bit count
          Ethernet_Overhead                => 376, -- bit count
          List_of_Drivers                  => (
                                           Driver 1,
                                           Driver 2,
                                           ...));
```

### 6.2.3 Network switches and network routers

The abstract class Network_Switch models a communication system specialized in the transfer of messages among networks, and it has a subclass called Network_Router that is able to make routing decisions and deliver messages to one output network chosen out of a set of possible outputs.

- *Network_Switch*: 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. The attributes of a network switch are, in addition to the Name and Speed Factor:
  - *Max_Fixed_Fork_Latency*: Normalized_Execution_Time. 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. The default value is zero.
  - *Avg_Fixed_Fork_Latency*: Normalized_Execution_Time. 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. The default value is zero.
  - *Min_Fixed_Fork_Latency*: Normalized_Execution_Time. 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. The default value is zero.

- *Max_Variable_Fork_Latency*: Normalized_Execution_Time. 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. The default value is zero.
- *Avg_Variable_Fork_Latency*: Normalized_Execution_Time. 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. The default value is zero.
- *Min_Variable_Fork_Latency*: Normalized_Execution_Time. 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 . The default value is zero.
- *Max_Delivery_Latency*: Normalized_Execution_Time. Maximum Latency introduced by the switch for each port-to-port message delivery. The default value is zero.
- *Avg_Delivery_Latency*: Normalized_Execution_Time. Average Latency introduced by the switch for each port-to-port message delivery. The default value is zero.
- *Min_Delivery_Latency*: Normalized_Execution_Time. Minimum latency introduced by the switch for each port-to-port message delivery. The default value is zero.

The Network Switch has a direct subclass called Regular_Switch

- *Regular_Switch*: 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.

```
Processing_Resource (
     Type                                => Regular_Switch,
     Name                                => Identifier,
     Speed_Factor                        => Float,
     Max_Fixed_Fork_Latency              => Normalized_Execution_Time,
     Avg_Fixed_Fork_Latency              => Normalized_Execution_Time,
     Min_Fixed_Fork_Latency              => Normalized_Execution_Time,
     Max_Variable_Fork_Latency           => Normalized_Execution_Time,
     Avg_Variable_Fork_Latency           => Normalized_Execution_Time,
     Min_Variable_Fork_Latency           => Normalized_Execution_Time,
     Max_Delivery_Latency                => Normalized_Execution_Time,
     Avg_Delivery_Latency                => Normalized_Execution_Time,
     Min_Delivery_Latency                => Normalized_Execution_Time);
```

The Regular-Switch class has a direct subclass called AFDX_Switch

- *AFDX_Switch*: Concrete and simple switch working according to the AFDX specification.

```
Processing_Resource (
     Type                           => AFDX_Switch,
     Name                           => Identifier,
     Speed_Factor                   => Float,
     Max_Fixed_Fork_Latency         => Normalized_Execution_Time,
     Avg_Fixed_Fork_Latency         => Normalized_Execution_Time,
     Min_Fixed_Fork_Latency         => Normalized_Execution_Time,
     Max_Variable_Fork_Latency      => Normalized_Execution_Time,
     Avg_Variable_Fork_Latency      => Normalized_Execution_Time,
     Min_Variable_Fork_Latency      => Normalized_Execution_Time,
     Max_Delivery_Latency           => Normalized_Execution_Time,
     Avg_Delivery_Latency           => Normalized_Execution_Time,
     Min_Delivery_Latency           => Normalized_Execution_Time);
```

The Network_Switch has another direct abstract subclass that represents network routers

- *Network_Router*: 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)). The additional attributes are:
  - *Max_Fixed_Branch_Latency*: Normalized_Execution_Time. This attribute represents 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. The default value is zero.
  - *Avg_Fixed_Branch_Latency*: Normalized_Execution_Time. This attribute represents 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. The default value is zero.
  - *Min_Fixed_Branch_Latency*: Normalized_Execution_Time. This attribute represents 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. The default value is zero.
  - *Max_Variable_Branch_Latency*: Normalized_Execution_Time. 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. The default value is zero.

- *Avg_Variable_Branch_Latency*: Normalized_Execution_Time. 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. The default value is zero.
- *Min_Variable_Branch_Latency*: Normalized_Execution_Time. 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. The default value is zero.

The Network-Router class has a direct concrete subclass called Regular_Router

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

```
Processing_Resource (
     Type                             => Regular_Router,
     Name                             => Identifier,
     Speed_Factor                     => Float,
     Max_Fixed_Fork_Latency           => Normalized_Execution_Time,
     Avg_Fixed_Fork_Latency           => Normalized_Execution_Time,
     Min_Fixed_Fork_Latency           => Normalized_Execution_Time,
     Max_Variable_Fork_Latency        => Normalized_Execution_Time,
     Avg_Variable_Fork_Latency        => Normalized_Execution_Time,
     Min_Variable_Fork_Latency        => Normalized_Execution_Time,
     Max_Delivery_Latency             => Normalized_Execution_Time,
     Avg_Delivery_Latency             => Normalized_Execution_Time,
     Min_Delivery_Latency             => Normalized_Execution_Time,
     Max_Fixed_Branch_Latency         => Normalized_Execution_Time,
     Avg_Fixed_Branch_Latency         => Normalized_Execution_Time,
     Min_Fixed_Branch_Latency         => Normalized_Execution_Time,
     Max_Variable_Branch_Latency      => Normalized_Execution_Time,
     Avg_Variable_Branch_Latency      => Normalized_Execution_Time,
     Min_Variable_Branch_Latency      => Normalized_Execution_Time);
```

## 6.3 Timing_Objects

They represent the a mechanism for implementing timing services, including Timers, which implement a clock, and Clock Synchronization Objects which represent a synchronization mechanism among different clocks belonging to different processing resources. There are two classes of timing objects: Timers and Clock_Synchronization_Objects

### 6.3.1 Clock Synchronization Objects

- *Clock_Synchronization_Object*: 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. The attributes are:

- *Name*: Identifier.
- *Precision*: A time value that represents the 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.

```
Timing_Object (
    Type                          => Clock_Synchronization_Object
    Name                          => Identifier,
    Precision                     => Time);
```

### 6.3.2 Timers

- *Timer*: This is an abstract class that 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. A timer is a system timer if Is_System_Timer is Yes; in this case, 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. There are two concrete classes: Ticker and Alarm_Clock. The attributes that are common to all timers are:
  - *Name*: Identifier.
  - *Max_Overhead*: Normalized_Execution_Time. This is the maximum overhead of the timer interrupt, which is assumed to execute at the highest interrupt priority. The default value is 0.0.
  - *Avg_Overhead*: Normalized_Execution_Time. This is the average overhead of the timer interrupt, which is assumed to execute at the highest interrupt priority. The default value is 0.0.
  - *Min_Overhead*: Normalized_Execution_Time. This is the minimum overhead of the timer interrupt, which is assumed to execute at the highest interrupt priority. The default value is 0.0.
  - *Is_System_Timer*: Assertion. A "Yes|No" value that determines whether or not the timer is a system timer. Only one timer per processor may be a system timer. The system timer is used in the timetable-driven and global EDF scheduling policies. The default value is “Yes”.
  - Is_Locally_Synchronized: Assertion. 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. The default value is “No”.
  - Precision: Time. 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.

The two concrete Timers are:

- *Alarm Clock*. 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. It has no additional attributes.

```
Timing_Object (
     Type                              => Alarm Clock
     Worst_Overhead                    => Normalized_Execution_Time,
     Avg_Overhead                      => Normalized_Execution_Time,
     Best_Overhead                     => Normalized_Execution_Time,
     Is_System_Timer                   => Yes|No,
     Is_Locally_Synchronized           => Yes|No,
     Precision                         => Time);
```

- *Ticker*. This represents 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. The additional attribute is:
  - *Period*. Period of the ticker interrupt. Its default value is a very large time.

```
Timing_Object (
     Type                              => Ticker
     Worst_Overhead                    => Normalized_Execution_Time,
     Avg_Overhead                      => Normalized_Execution_Time,
     Best_Overhead                     => Normalized_Execution_Time,
     Is_System_Timer                   => Yes|No,
     Is_Locally_Synchronized           => Yes|No,
     Precision                         => Time,
     Period                            => Time);
```

## 6.4 Network Drivers

They represent operations executed in a processor as a consequence of the transmission or reception of a message or a message packet through a network. We define three classes:

- *Packet Driver*. It represents a driver that is activated at each message packet transmission or reception. Its attributes are:
  - *Packet server*: The identifier of the thread 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).
  - *Packet Send Operation*. The identifier of the operation that is executed by the *packet server* each time a packet is sent.
  - *Packet Receive Operation*. The identifier of the operation that is executed by the *packet server* each time a packet is received.

- *Message Partitioning*. 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. The default value is “Yes”.
- *High_Utilization_Mode*. A “Yes|No” value that 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 modelled 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 modelled 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.

```
Driver = (
    Type                              => Packet_Driver,
    Packet_Server                     => Identifier,
    Packet_Send_Operation             => Identifier,
    Packet_Receive_Operation          => Identifier,
    Message_Partitioning              => Yes | No,
    High_Utilization_Mode             => Yes | No)
```

- *Character Packet Driver*. 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. Its attributes are those of a packet driver plus the following:
    - *Character server*: The identifier of the thread 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).
    - *Character Send Operation*. The identifier of the operation that is executed by the *character server* each time a character is sent.
    - *Character Receive Operation*. The identifier of the operation that is executed by the *character server* each time a character is received.
    - *Character Transmission Time*. Time of character transmission. It determines the period used for the overhead models of the character send and character receive operations.

```
Driver = (
    Type                              => Character_Packet_Driver,
    Packet_Server                     => Identifier,
    Packet_Send_Operation             => Identifier,
    Packet_Receive_Operation          => Identifier,
    Message_Partitioning              => Yes | No,
    High_Utilization_Mode             => Yes | No,
    Character_Server                  => Identifier,
    Character_Send_Operation          => Identifier,
    Character_Receive_Operation       => Identifier,
    Character_Transmission_Time      => Time)
```

- *RT-EP Packet Driver*. It is a specialization of a packet driver that characterizes the Real-Time Ethernet Protocol, RTEP[1]. 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 end-to-end flows 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. The attributes of the *RT-EP Packet Driver* are those of a *Packet Driver* plus the following:
    - *Packet interrupt server*: The identifier of the schedulable resource (of the thread type) 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.
    - *Packet ISR operation*: The identifier of the operation executed by the packet interrupt server for each incoming packet, independently of whether it is a packet with information or a protocol token.
    - *Token check operation*: The identifier of the operation executed by the *packet server* to receive and check a token packet.
    - *Token manage operation*: The identifier of the operation executed by the *packet server* to send a token.
    - *Packet discard operation*: The identifier of the operation executed by the *packet server* when a packet is received that is addressed to another processing node.
    - *Token retransmission operation*: The operation executed by the *packet server* when a lost token is retransmitted, or a reference to it.
    - *Packet retransmission operation*: The identifier of the operation executed by the *packet server* when a packet with information that was lost is retransmitted.

```
Driver = (
    Type                             => RTEP_Packet_Driver,
    Packet_Server                    => Identifier,
    Packet_Send_Operation            => Identifier,
    Packet_Receive_Operation         => Identifier,
    Message_Partitioning             => Yes | No,
    High_Utilization_Mode            => Yes | No,
    Packet_Interrupt_Server          => Identifier,
    Packet_ISR_Operation             => Identifier,
    Token_Check_Operation            => Identifier,
```

1. See: J.M. Martínez, M. González Harbour, and J.J. Gutiérrez. “RT-EP: Real-Time Ethernet Protocol for Analyzable Distributed Applications on a Minimum Real-Time POSIX Kernel”. Proceedings of the 2nd International Workshop on Real-Time LANs in the Internet Age, RTLIA 2003, Porto (Portugal), July 2003.

```
    Token_Manage_Operation          => Identifier,
    Packet_Discard_Operation        => Identifier,
    Token_Retransmission_Operation  => Identifier,
    Packet_Retransmission_Operation => Identifier)
```

## 6.5 Schedulers

They represent 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, like the one shown in Figure 5. 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.

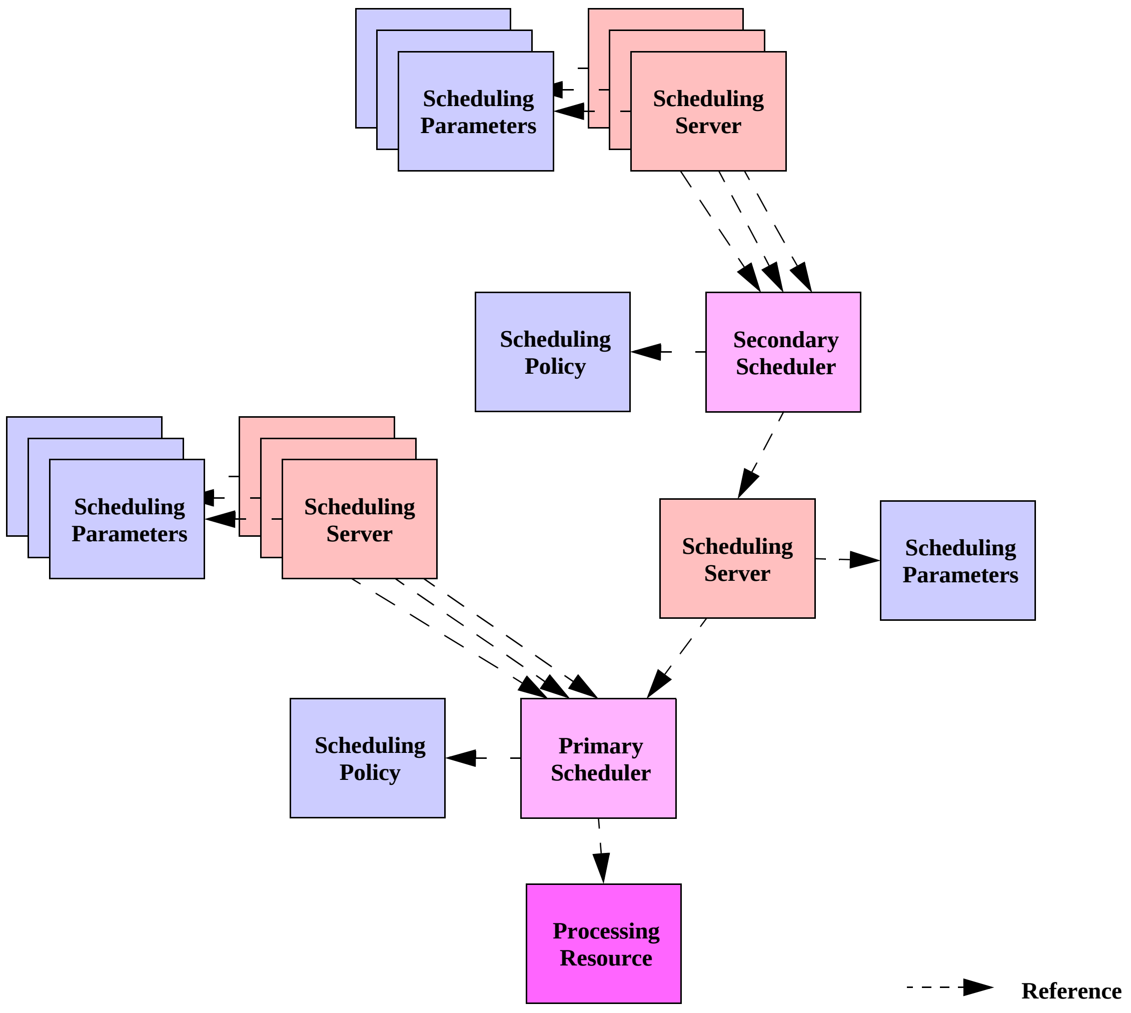


**Figure 5. Hierarchical scheduler structure**

Common attributes:

- *Name*. A string.

- *Policy*. It defines the scheduling policy that is implemented by the scheduler to distribute the assigned processing capacity among the schedulable resources associated to the same scheduler.

There are two classes of schedulers, according to the source of processing capacity:

- *Primary Scheduler*. It represents the base system scheduler for its associated processing resource. It has the following additional attribute:
  - *Host*. A reference to the processing resource that supplies the processing capacity to be scheduled.

```
Scheduler (
     Type                                => Primary_Scheduler,
     Name                                => Identifier,
     Policy                              => Scheduling_Policy,
     Host                                => Identifier); --Processing_Resource
```

- *Secondary Scheduler*. It represents a scheduler that is able to handle only a certain fraction of processing capacity, which in turn is served by a particular schedulable resource through another scheduler associated to it. It has the following additional attribute:
  - *Host*. A reference to the schedulable resource that supplies the processing capacity to be scheduled.

```
Scheduler (
     Type                                => Secondary_Scheduler,
     Name                                => Identifier,
     Policy                              => Scheduling_Policy,
     Host                                => Identifier); --Schedulable_Resource
```

## 6.6 Scheduling policies

### 6.6.1 Policies

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.

Classes of Scheduling Policies:

- *Fixed Priority*. It represents a fixed-priority policy. It can only be assigned to a scheduler that has a Regular_Processor as its host. It has the following attributes:
  - *Context Switch Overheads* (Worst, Average, Best). Overhead for a context switch between schedulable resources. Their default values are zero.
  - *Max Priority* and *Min Priority*. They define the range of priorities that are valid for normal operations on schedulable resources scheduled with this policy. Special operations (such as interrupt service routines in processors) may have other priority ranges. The default values are respectively the maximum and minimum values of the *Any_Priority* type.

```
Scheduling_Policy (
     Type                             => Fixed_Priority,
     Worst_Context_Switch             => Normalized_Execution_Time,
     Avg_Context_Switch               => Normalized_Execution_Time,
     Best_Context_Switch              => Normalized_Execution_Time,
     Max_Priority                     => Priority,
     Min_Priority                     => Priority);
```

- *EDF*. It represents an Earliest Deadline First policy. It can only be assigned to a scheduler that has a Regular_Processor as its host. It has the following attributes:
  - *Context Switch Overheads* (Worst, Average, Best). Overhead for a context switch between schedulable resources. Their default values are zero.

```
Scheduling_Policy (
     Type                             => EDF,
     Worst_Context_Switch             => Normalized_Execution_Time,
     Avg_Context_Switch               => Normalized_Execution_Time,
     Best_Context_Switch              => Normalized_Execution_Time);
```

- *Fixed Priority Packet Based*. 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. It has the following attributes:
  - *Packet Overhead* (Worst, Average, Best). This is the overhead associated with sending each packet, because of the protocol messages or headers that need to be sent before or after each packet. The overheads are specified 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. The default values for these attributes are zero.
  - *Max Priority* and *Min Priority*. They define the range of priorities valid for messages to be sent using this policy. The default values are respectively the maximum and minimum values of the *Any_Priority* type.

```
Scheduling_Policy (
     Type                             => FP_Packet_Based,
     Packet_Overhead_Max_Size         => Bit_Count,
     Packet_Overhead_Avg_Size         => Bit_Count,
     Packet_Overhead_Min_Size         => Bit_Count,
     Max_Priority                     => Priority,
     Min_Priority                     => Priority);
```

- *Timetable_Driven*. 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 resources are assigned to the same partition a secondary scheduler may be needed to schedule the available time inside the partition. It only can be assigned to a scheduler that has a Regular_Processor as host. It has the following attributes:

- *MAF*. Time interval in which the execution of the partition windows is scheduled. The schedule is repeated in a cyclic manner. All the Timetable_Driven or Timetable_Driven_Packet_Based objects that are synchronized through their associated processing resources should have the same MAF. The default value is a large time value.
- *Context Switch Overheads* (Worst, Average, Best). Overhead for a context switch between schedulable resources. Their default values are zero.
- *Partition_Table*: A list of partition windows to be scheduled in each MAF (see the definition of a partition window below).

```
Scheduling_Policy (
     Type                               => Timetable_Driven,
     MAF                                => Time,
     Worst_Context_Switch               => Normalized_Execution_Time,
     Avg_Context_Switch                 => Normalized_Execution_Time,
     Best_Context_Switch                => Normalized_Execution_Time,
     Partition_Table                    => (
                                                 Partition_Window,
                                                 Partition_Window,
                                                 ...
);
```

• *Timetable_Driven_Packet_Based*. It represents a Timetable Driven policy assigned to a scheduler that has a Packet_Based_Network as its host.

- *MAF*. Time interval in which the execution of the partition windows is scheduled. The schedule is repeated in a cyclic manner. All the Timetable_Driven or Timetable_Driven_Packet_Based objects that are synchronized through their associated processing resources should have the same MAF. The default value is a large time value.
- *Partition_Table*: A list of partition windows to be scheduled in each MAF (see the definition of a partition window below).

```
Scheduling_Policy (
     Type                               => Timetable_Driven,
     MAF                                => Time,
     Partition_Table                    => (
                                                 Partition_Window,
                                                 Partition_Window,
                                                 ...
);
```

• *AFDX*: 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.

### 6.6.2 Partition Windows

For the Timetable_Driven policies it is necessary to define a *Partition_Window*, which is the data required to define each execution window inside a partition in the Timetable_Driven policy. The attributes of a *Partition_Window* are:

- *Partition_Id*: Natural. Natural integer that is used as the partition identifier. Multiple windows may belong to the same partition.
- *Partition_Name*: Identifier. Optional identifier that is used as the partition name, for expressiveness purposes. Each partition name mush correspond to a unique partition Id.
- *Start_Time*: Time. 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. Duration of the window.

```
Partition_Window= (
     Partition_Id                      => Integer,
     Partition_Name                    => Identifier,
     Start_Time                        => Time,
     Length                            => Time);
```

## 6.7 Scheduling parameters

These parameters are attached to a schedulable resource and are used by its scheduler to make its scheduling decisions when the schedulable resource is competing with other schedulable resources. 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 steps. These different scheduling behaviours are determined by the scheduling parameters type and are also called the per-schedulable resource 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:

- *Interrupt based*: Used to represent an interrupt service routine. It is applicable to any application scheduler, because interrupts are handled directly by the Computing_Resource.
- *Priority based*: This kind of scheduling parameters is only applicable to schedulers with fixed priority policy.
- *EDF based*: This kind of scheduling parameters is only applicable to schedulers with EDF policy.
- *Resource reservation based*: This kind of scheduling parameters is only applicable to virtual schedulable resources (Virtual_Schedulable_Resource or Virtual_Communication_Channel) as their Resource_Reservation_Params association.
- *Timetable driven*: This kind of scheduling parameters is only applicable to schedulers with the Timetable_Driven_Policy.

- *AFDX_Virtual_Link*: This kind of scheduling parameters is only applicable to schedulers with the AFDX_Policy.

### 6.7.1 Interrupt-based scheduling parameters

A schedulable resource with this kind of parameters represents an interrupt service routine. The interrupt priorities are always pre-assigned. It is applicable to any application scheduler, because interrupts are handled directly by the Computing_Resource.

There is one class of Interrupt-based scheduling parameters:

- *Interrupt Fixed Priority Policy*. Represents an interrupt service routine. No additional attributes. The "*Preassigned*" field cannot be set to "*No*", because interrupt priorities are always preassigned. Is only attribute is:
- Priority: Interrupt_Priority. A natural 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). The default value is the minimum interrupt priority.

```
Sched_Parameters = (
     Type                               => Interrupt_FP_Params,
     The_Priority                       => Interrupt_Priority)
```

### 6.7.2 Priority-based scheduling parameters

This kind of scheduling parameters is only applicable to schedulers with fixed priority policy. All the fixed priority scheduling parameters have the following set of common attributes:

- *The_Priority*. A natural number that represents the scheduling priority. It must be within the valid ranges for the associated scheduler. Its default value is the minimum value of the *Any_Priority* type.
- *Preassigned*. If this parameter is set to the value "*No*", the priority may be assigned by one of the scheduling parameters assignment tools. Otherwise, the priority is fixed and cannot be changed by those tools. Its default value is "*No*" if no priority field appears, and "*Yes*" if a priority field appears in the MAST description.

The classes defined are:

- *Non Preemptible Fixed Priority Parameters*. Activities scheduled with these parameters are scheduled under non preemptive fixed priorities. It has no additional attributes.

```
Sched_Parameters = (
     Type                               => Non_Preemptible_FP_Params,
     The_Priority                       => Priority,
     Preassigned                        => Yes | No)
```

- *Fixed Priority Parameters*. Activities scheduled with these parameters are scheduled under preemptive fixed priorities. It has no additional attributes.

```
Sched_Parameters = (
     Type                               => Fixed_Priority_Params,
     The_Priority                       => Priority,
     Preassigned                       => Yes | No)
```

- *Polling Parameters*. 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. Its additional attributes are:
  - *Polling Period*. Period of the polling task. Its default value is zero.
  - *Polling Overhead* (Worst, Average, Best). Overhead of the polling task. Their default values are zero.

```
Sched_Parameters = (
     Type                              => Polling_Params,
     The_Priority                      => Priority,
     Preassigned                       => Yes | No,
     Polling_Period                    => Time,
     Polling_Worst_Overhead            => Normalized_Execution_Time,
     Polling_Avg_Overhead              => Normalized_Execution_Time,
     Polling_Best_Overhead             => Normalized_Execution_Time)
```

- *Periodic Server Parameters*. 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. Its additional attributes are:
  - *Initial Capacity*. Its the initial value of the execution capacity. Its default value is zero.
  - *Replenishment Period*. It is the period after which the capacity consumed is replenished. Its default value is zero.

```
Sched_Parameters = (
     Type                              => Periodic_Server_Params,
     Normal_Priority                   => Priority,
     Preassigned                       => Yes | No,
     Initial_Capacity                  => Time,
     Replenishment_Period              => Time)
```

- *Sporadic Server Parameters*. 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. Its additional attributes are:
  - *Background Priority*. The priority at which the task executes when there is no available execution capacity. Its default value is the minimum value of the *Priority* type.

- *Initial Capacity*. Its the initial value of the execution capacity. Its default value is zero.
- *Replenishment Period*. It is the period after which a portion of consumed execution capacity is replenished. Its default value is zero.
- *Max Pending replenishments*. It is the maximum number of simultaneously pending replenishment operations. Its default value is one.

```
Sched_Parameters = (
     Type                              => Sporadic_Server_Params,
     Normal_Priority                   => Priority,
     Preassigned                       => Yes | No,
     Background_Priority               => Priority,
     Initial_Capacity                  => Time,
     Replenishment_Period              => Time,
     Max_Pending_Replenishments        => Positive)
```

- *Periodic Server Communications Parameters*. 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.. Its additional attributes are:
  - *Initial Capacity*. Its the initial value of the number of bits that can be sent. Its default value is zero.
  - *Replenishment Period*. It is the period after which the capacity spent is replenished. Its default value is zero.

```
Sched_Parameters = (
     Type                              => Periodic_Server_Comm_Params,
     Normal_Priority                   => Priority,
     Preassigned                       => Yes | No,
     Initial_Capacity                  => Bit_Count,
     Replenishment_Period              => Time)
```

- *Sporadic Server Communications Parameters*. 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. Its additional attributes are:
  - *Background Priority*. The priority at which the channel sends its messages when there is no available execution capacity. Its default value is the minimum value of the *Priority* type.
  - *Initial Capacity*. Its the initial value of the number of bits that can be sent. Its default value is zero.

- *Replenishment Period*. It is the period after which the capacity spent is replenished. Its default value is zero.
- *Max Pending replenishments*. It is the maximum number of simultaneously pending replenishment operations. Its default value is one.

```
Sched_Parameters = (
     Type                          => Sporadic_Server_Comm_Params,
     Normal_Priority               => Priority,
     Preassigned                   => Yes | No,
     Background_Priority           => Priority,
     Initial_Capacity              => Bit_Count,
     Replenishment_Period          => Time,
     Max_Pending_Replenishments    => Positive)
```

### 6.7.3 Overridden scheduling parameters

Part of the fixed priority scheduling parameters may also be overridden for a particular operation, by including an overridden scheduling parameters object in its definition.

In the Overriden_Fixed_Priority parameters the change of priority is in effect only until the operation is completed.

```
Overridden_Sched_Parameters = (
     Type                          => Overridden_Fixed_Priority,
     The_Priority                  => Any_Priority)
```

In the Overridden_Permanent_FP parameters 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).

```
Overridden_Sched_Parameters = (
     Type                          => Overridden_Permanent_FP,
     The_Priority                  => Any_Priority)
```

### 6.7.4 EDF Scheduling Parameters

These scheduling parameters are only applicable to schedulers with EDF policy. All the EDF scheduling parameters have the following set of common attributes:

- *Deadline*. Relative deadline of the associated schedulable resource. Its default value is a Large Time.
- *Preassigned*. If this parameter is set to the value “*No*”, the deadline may be assigned by one of the deadline assignment tools. Otherwise, it is fixed and cannot be changed by those tools. Its default value is “*No*” if no deadline field appears in the MAST description, and “*Yes*” if any of them is present.

There is only one class of EDF scheduling parameters defined at the moment:

- *Earliest Deadline First Parameters*. It contains the parameters associated with the “earliest deadline first” policy. It has no additional attributes.

```
Sched_Parameters = (
     Type                             => EDF_Params,
     Deadline                         => Time,
     Preassigned                      => Yes | No)
```

### 6.7.5 Resource Reservation Scheduling Parameters

The Resource Reservation Scheduling Parameters represent a service contract that contains the information related to the application minimum resource requirements. They are only applicable to virtual schedulable resources (Virtual_Communication_Channel or Virtual_Schedulable_Resource) as their Resource_Reservation_Params association. The attributes that are common to all resource reservation parameters are:

- *Preassigned*: Assertion. 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. The default value is “No”.

**Virtual Computing Resource Parameters.** The main resource reservation parameters that can be used in Computing Resources are represented in the abstract class *Virtual_Periodic_Replenishment_Server*, which represent the parameters of a policy in which a 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. The direct subclasses are: Virtual_Periodic_Server, Virtual_Deferrable_Server and Virtual_Sporadic_Server. The common attributes are:

- *Budget*: Normalized_Execution_Time. Minimum execution capacity per server period. The default value is zero.
- *Deadline*: Time. 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. The default value is a large time.
- *Period*: Time. 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. The default value is a large time.

• *Virtual_Periodic_Server*: 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 preempted 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. It has no additional attributes.

```
Sched_Parameters = (
     Type                              => Virtual_Periodic_Server,
     Preassigned                       => Yes | No,
     Budget                            => Time,
     Deadline                          => Time,
     Period                            => Time)
```

- *Virtual_Deferrable_Server*: 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. It has no additional attributes.

```
Sched_Parameters = (
     Type                              => Virtual_Deferrable_Server,
     Preassigned                       => Yes | No,
     Budget                            => Time,
     Deadline                          => Time,
     Period                            => Time)
```

- *Virtual_Sporadic_Server*: 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. It has no additional attributes.

```
Sched_Parameters = (
     Type                              => Virtual_Sporadic_Server,
     Preassigned                       => Yes | No,
     Budget                            => Time,
     Deadline                          => Time,
     Period                            => Time)
```

**Virtual Network Resource Parameters.** The main resource reservation parameters that can be used in Networks are represented in the abstract class *Virtual_Comm_Channel_Params*, which represents a reservation of a communication channel  contract for a network with Reservation Resource policy.  It represents a service contract that contains the information related to the application minimum resource requirements. Its direct subclasses are Virtual_Periodic_Replenishment_Comm_Channel and Virtual_Token_Bucket_Comm_Channel.

- *Virtual_Token_Bucket_Comm_Channel*: 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.

Its attributes are:

- *Budget*: Bit_Count. The maximum number of tokens in the bucket. The default value is zero.
- *Max_Throughput*: Throughput. The rate of token flow in the bucket, or the average flow of bits in the channel. The default value is the maximum possible value.

```
Sched_Parameters = (
     Type                            => Virtual_Token_Bucket_Comm_Channel,
     Budget                          => Time,
     Max_Throughput                  => Throughput)
```

The abstract class *Virtual_Periodic_Replenisment_Comm_Channel* defines the parameters of a communication 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. Its direct subclasses are Virtual_Periodic_Comm_Channel, Virtual_Deferreable_Comm_Channel and Virtual_Sporadic_Comm_Channel. The common attributes are:

- *Budget*: Bit_Count. Minimum transmission capacity per server period. The default value is zero.
- *Deadline*: Time. => 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. The default value is a large time.
- *Period*: Time. 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. The default value is a large time.

• *Virtual_Periodic_Comm_Channel*: It represents the parameters of a communication channel that 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 preempted 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. It has no additional attributes.

```
Sched_Parameters = (
     Type                            => Virtual_Periodic_Comm_Channel,
     Budget                          => Time,
     Deadline                        => Time,
     Period                          => Time)
```

- *Virtual_Deferrable_Comm_Channel*: It represents the parameters of a communication channel that 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. It has no additional attributes.

```
Sched_Parameters = (
        Type                            => Virtual_Deferrable_Comm_Channel,
        Budget                          => Time,
        Deadline                        => Time,
        Period                          => Time)
```

- *Virtual_Sporadic_Comm_Channel*: 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. It has no additional attributes.

```
Sched_Parameters = (
        Type                            => Virtual_Sporadic_Comm_Channel,
        Budget                          => Time,
        Deadline                        => Time,
        Period                          => Time)
```

### 6.7.6 Timetable-Driven Scheduling Parameters

These scheduling parameters are only applicable to schedulers with the Timetable_Driven_Policy.

- *Partition_Params*: Concrete scheduling parameters for a Timetable_Driven policy. Its attributes are:
    - *Partition_Id*: Natural. Optional identifier of the assigned partition.
    - *Partition_Name*: Identifier. Optional identifier that is used as the partition name, for expressiveness purposes. Each partition name mush correspond to a unique partition Id. The default value is null.

```
Sched_Parameters = (
        Type                            => Partition_Params,
        Partition_Id                    => Natural,
        Partition_Name                  => Identifier)
```

### 6.7.7 AFDX Virtual Link Scheduling Parameters

- AFDX_Virtual_Link: Concrete scheduling parameters for the AFDX_Policy. Its attributes are:
  - *Lmax*: Bits . Maximum length of the packets sent through this virtual link. The default value is zero.
  - *BAG*: Time. 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]. The default value is a large time value.

```
Sched_Parameters = (
     Type                                  => AFDX_Virtual_Link,
     Lmax                                  => Bit_Count,
     BAG                                   => Time)
```

## 6.8 Synchronization Parameters

These parameters are attached to a schedulable resource to specify the parameters used by that resource when performing a mutually exclusive access to mutual exclusion resources.

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 schedulable resource is its priority.

The only class defined for the synchronization parameters is named *Stack Resource Protocol Parameters* because it is associated with that synchronization protocol. Its attributes are:

- *Preemption Level*. A natural 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. Its default value is the minimum value of the *Preemption_Level* type.
- *Preassigned*. 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. Its default value is "*No*" if no preemption level field appears in the MAST description, and "*Yes*" if it is present.

```
Synch_Parameters = (
     Type                                  => SRP_Params,
     Preemption_Level                      => Preemption_Level,
     Preassigned                           => Yes | No)
```

## 6.9 Schedulable Resources

They represent schedulable entities in a processing resource. They are units of concurrent execution in a processor (task, single-threaded process, thread, interrupt service routine ...) or network (communication channel, communication session ...).

Their common attributes are:

- *Name*
- *Scheduling Parameters*. 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 not present only in the case of a Virtual_Schedulable_Resource.
- *Scheduler*. Identifier of the scheduler that serves it. It can be not present only in the case of a Virtual_Schedulable_Resource.

There are two kinds of schedulable resources: *Thread*, for use in a computing resource, and *Virtual_Communication_Channel*, for use in a network.

- *Thread*: 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.. Its additional attributes are:
  - *Synchronization_Parameters*: Synchronization_Parameters. 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).

```
schedulable_resource (
    Type                               => Thread,
    Name                               => Identifier,
    Scheduling_Parameters              => Sched_Parameters,
    Synchronization_Parameters         => Synch_Parameters,
    Scheduler                          => Identifier);
```

- *Virtual_Schedulable_Resource*: It is a specialized Scheduling 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 analyse an application running on top of a virtual resource independently of the rest of the system. On the second hand, it is possible to analyse 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, analyse the feasibility of a particular composition of these application components, without the need to know about their internal details. It the following additional attributes.

- *Synchronization_Parameters*: Synchronization_Parameters. 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).
- *Resource_Reservation_Params*: Virtual_Resource_Params. Scheduling parameters of the resource reservation contract associated to the Virtual_Schedulable_Resource.

```
schedulable_resource (
     Type                             => Virtual_Schedulable_Resource,
     Name                             => Identifier,
     Scheduling_Parameters            => Sched_Parameters,
     Synchronization_Parameters       => Synch_Parameters,
     Scheduler                        => Identifier,
     Resource_Reservation_Params      => Resource_Reservation_Sched_Params);
```

- *Communication_Channel*: It is a specialized Schedulable_Resource for message transmission in a network. It has no additional attributes.

```
schedulable_resource (
     Type                             => Communication_Channel,
     Name                             => Identifier,
     Scheduling_Parameters            => Sched_Parameters,
     Scheduler                        => Identifier);
```

- *Virtual_Communication_Channel*: It is a specialized Communication Channel 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.

- 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.

Its additional attributes are:

- *Resource_Reservation_Params*: Virtual_Communication_Channel_Params. Scheduling parameters that define the resource reservation contract associated to the Virtual_Communication_Channel.

```
schedulable_resource (
     Type                                 => Virtual_Communication_Channel,
     Name                                 => Identifier,
     Scheduling_Parameters                => Sched_Parameters,
     Scheduler                            => Identifier,
     Resource_Reservation_Params          => Virtual_Comm_Channel_Params);
```

There is a predefined object of the class Communication_Channel called *DEFAULT_COMMUNICATION_CHANNEL*. It is an instance of Virtual_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.

## 6.10 Mutual Exclusion Resources

They represent resources that are 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. There are three classes, depending on the protocol:

- *Immediate Ceiling Mutex*. Uses the immediate priority ceiling resource protocol. This is equivalent to Ada's *Priority Ceiling*, or the POSIX *priority protect* protocol. Its attributes are:
  - *Name*.
  - *Ceiling*. Priority ceiling used for the resource. May be computed automatically by the tool, upon request. Its default value is the maximum value of the *Any_Priority* type.
  - *Preassigned*. 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. Its default value is "*No*" if no ceiling field appears, and "*Yes*" if a ceiling field appears.

```
Shared_Resource (
     Type                                 => Immediate_Ceiling_Mutex,
     Name                                 => Identifier,
     Ceiling                              => Any_Priority,
     Preassigned                          => Yes | No);
```

- *Priority Inheritance Mutex*. Uses the basic priority inheritance protocol. Its attributes are:
    - *Name*.

```
Shared_Resource (
     Type                                => Priority_Inheritance_Mutex,
     Name                                => Identifier);
```

- *Stack Based Mutex*. Uses the Stack Resource Protocol (SRP). This is similar to the immediate ceiling protocol but works for non-priority-based policies. Its attributes are:
    - *Name*.
    - *Preemption Level*. Level of preemptability used for the resource. May be computed automatically by the MAST tools, upon request. Its default value is the maximum value of the *Preemption_Level* type.
    - *Preassigned*. If this parameter is set to the value "*No*", the preemption level may be assigned by a tool. Otherwise, the priority ceiling is fixed and cannot be changed by any tool. Its default value is "*No*" if no preemption level field appears in the MAST description, and "*Yes*" if a preemption level field appears.

```
Shared_Resource (
     Type                                => SRP_Mutex,
     Name                                => Identifier,
     Preemption_Level                    => Preemption_Level,
     Preassigned                         => Yes | No);
```

## 6.11 Operations

They represent a piece of code, or a message. They describe the processing capacity usage that is required for the execution of a piece of code or for the transmission of a message. They all have the following common attributes:

- *Name*. The identifier of the operation
- *Overridden Scheduling Parameters*. It models scheduling parameters defined in the operation itself, which should be used by the schedulable resource when executing the operation. For instance, it can represent a priority level above the normal priority level that at which the operation would execute:
    - For a regular overridden priority (*Overridden_Fixed_Priority*), the change of priority is in effect only until the operation is completed.
    - For a permanent overridden priority (*Overridden_Permanent_FP*), the change of priority is in effect until another permanent overridden priority, or until the end of the segment of steps, i.e., a set of consecutive steps (consecutive in the end-to-end flow graph) executed by the same schedulable resource.

The following classes of operations are defined:

- *Code_Operation:* It describes the processing capacity usage, measured in normalized execution time, that is required for the execution of a piece of code. It is an abstract class with two direct subclasses: Simple_Operation and Composite_Operation.

- *Simple Operation*. It represents a simple piece of sequential code execution. Additional attributes are:
    - *Execution Time* (Worst, Average and Best). In normalized units, it represents the execution time of the code. The default values are very large time values for the worst and average, and zero for the best execution time.
    - *Mutexes to lock*. List of references to the mutual exclusion resources that must be locked before executing the operation.
    - *Mutexes to unlock*. List of references to the mutual exclusion resources that must be unlocked after executing the operation.

```
Operation (
     Type                                => Simple,
     Name                                => Identifier,
     Overridden_Sched_Parameters         => Overridden_Sched_Parameters,
     Worst_Case_Execution_Time           => Normalized_Execution_Time,
     Avg_Case_Execution_Time             => Normalized_Execution_Time,
     Best_Case_Execution_Time            => Normalized_Execution_Time,
     Mutexes_To_Lock                     => (
                                         Identifier,
                                         Identifier,
                                         ...),
     Mutexes_To_Unlock                   => (
                                         Identifier,
                                         Identifier,
                                         ...));
```

- *Composite Operation*. 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. Its additional attributes are:
    - *Operation List*: List of references to other operations

```
Operation (
     Type                                => Composite,
     Name                                => Identifier,
     Overridden_Sched_Parameters         => Overridden_Sched_Parameters,
     Operation_List                      => (
                                         Identifier,
                                         Identifier,
                                         ...));
```

- *Enclosing*. It is an extension of the composite operation, and 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. Its additional attributes are:

- *Execution Time* (Worst, Average and Best). In normalized units. The default values are very large time values for the worst and average, and zero for the best execution time.
- *Operation List*: List of references to other operations

```
Operation (
     Type                               => Enclosing,
     Name                               => Identifier,
     Overridden_Sched_Parameters        => Overridden_Sched_Parameters,
     Worst_Case_Execution_Time          => Normalized_Execution_Time,
     Avg_Case_Execution_Time            => Normalized_Execution_Time,
     Best_Case_Execution_Time           => Normalized_Execution_Time,
     Operation_List                     => (
                                        Identifier,
                                        Identifier,
                                        ...));
```

- *Message_Transmission*. Represents a message to be transmitted through a network. Its additional attributes are:
  - *Message Size* (Worst, Average and Best). 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. The default values are very large *Bit_Count* values for the worst and average, and zero for the best message size.

```
Operation (
     Type                               => Message_Transmission,
     Name                               => Identifier,
     Overridden_Sched_Parameters        => Overridden_Sched_Parameters,
     Max_Message_Size                   => Bit_Count,
     Avg_Message_Size                   => Bit_Count,
     Min_Message_Size                   => Bit_Count);
```

<tbd: will add a Composite_Message>

## 6.12 Events

Events represent event streams containing an 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.

- *Internal events*. They are generated by an event handler. Their attributes are:
  - *Name*.
  - *Observer*. Requirements imposed on the generation of the event, which must be observed by the analysis tools. See the description of the observers below

```
Internal_Event = (
     Type                               => Regular,
     Event                              => Identifier)
```

```
    Observer                          => Observer)
```

Workload Events represent 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 a timer. For the workload events, the following classes are defined:

- *Periodic*. It is a timed event that represents a stream of events that are generated periodically. It has the following attributes:
    - *Name*. Identifier of he event
    - *Timer*. The identifier of the timer that generates the event. If not present, the event is generated by the environment though an implicit timer with no overhead.
    - *Period*. Event period. Its default value is zero.
    - *Max Jitter*. 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. Its default value is zero.
    - *Phase*. It is the instant of the first activation, if it had no jitter. After that time, the following events are periodic (possibly with jitter). Its default value is zero.
    - *Synchronized*: Assertion. 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. The default value is No.

```
workload_event = (
    Type                              => Periodic,
    Name                              => Identifier,
    Timer                             => Identifier,
    Period                            => Time,
    Max_Jitter                        => Maximum jitter of Periodic event,
    Phase                             => Absolute_Time
    Synchronized                      => Yes | Nob);
```

- *Singular*. Represents an event that is generated only once. It has the following attributes:
    - *Name*. Identifier of he event
    - *Timer*. The identifier of the timer that generates the event. If not present, the event is generated by the environment though an implicit timer with no overhead.
    - *Phase*. It is the instant of the first activation. Its default value is zero.

```
workload_event = (
    Type                              => Singular,
    Name                              => Identifier,
    Timer                             => Identifier,
    Phase                             => Absolute_Time);
```

- *Sporadic*. It represents a stream of aperiodic events that have a minimum interarrival time. It has the following attributes:

- *Name*.
- *Min Interarrival*. Minimum time between the generation of two events. Its default value is zero.
- *Average Interarrival*. Average interarrival time. Its default value is zero.
- *Distribution*. It represents the distribution function of the aperiodic events. It can be *Uniform* (default value) or *Poisson*.

```
workload_event = (
    Type                              => Sporadic,
    Name                              => Identifier,
    Avg_Interarrival                  => Time,
    Distribution                      => Uniform|Poisson,
    Min_Interarrival                  => Time);
```

- *Unbounded*. 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. It has the following attributes:
  - *Name*.
  - *Average Interarrival*. Average interarrival time. Its default value is zero.
  - *Distribution*. It represents the distribution function of the aperiodic events. It can be *Uniform* (default value) or *Poisson*.

```
workload_event = (
    Type                              => Unbounded,
    Name                              => Identifier,
    Avg_Interarrival                  => Time,
    Distribution                      => Uniform|Poisson);
```

- *Bursty*. 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). It has the following attributes:
  - *Name*.
  - *Bound_Interval*. Interval for which the amount of event arrivals is bounded. Its default value is zero.
  - *Max_Arrivals*. Maximum number of events that may arrive in the *Bound_Interval*. Its default value is one.
  - *Average Interarrival*. Average interarrival time. Its default value is zero.
  - *Distribution*. It represents the distribution function of the aperiodic events. It can be *Uniform* (default value) or *Poisson*.

```
workload_event = (
    Type                              => Bursty,
    Name                              => Identifier,
    Avg_Interarrival                  => Time,
    Distribution                      => Uniform|Poisson,
    Bound_Interval                   => Time,
    Max_Arrivals                     => Positive);
```

## 6.13 Observers

Observers represent non-functional requirements associated to an internal event. The analysis tools will observe whether the requirement is satisfied or not. There are different kinds of requirements: Timing Requirements and Queue Size Requirements. It is also possible to specify several of these requirements in a Composite Observer

### 6.13.1 Timing Requirements

- *Deadlines*. They represent a maximum time value allowed for the generation of the associated event. They are expressed as a relative time interval, relative to the generation of another event. There are two different kinds:
  - *Local Deadlines*: they appear associated with the output event of a step; the deadline is relative to the arrival of the event that activated that step.
  - *Global deadlines*: the deadline is relative to the arrival of a *Referenced Event*, that is an attribute of the deadline, and which must be a workload event.

  In addition, deadlines may be hard or soft:
  - *Hard Deadlines*: they must be met in all cases, including the worst case
  - *Soft Deadlines*: they must be met on average.

  This gives way to four kinds of deadlines:
  - *Hard Global Deadline*. Attributes are the value of the *Deadline* (default=0), and a reference to the *Referenced Event*.
  - *Soft Global Deadline*. Attributes are the value of the *Deadline* (default=0), and a reference to the *Referenced Event*.
  - *Hard Local Deadline*. The only attribute is the value of the *Deadline* (default=0).
  - *Soft Local Deadline*. The only attribute is the value of the *Deadline* (default=0).

```
Observer = (
     Type                             => Hard_Global_Deadline,
     Deadline                         => Time,
     Referenced_Event                 => Identifier)

Observer = (
     Type                             => Hard_Local_Deadline,
     Deadline                         => Time)

Observer = (
     Type                             => Soft_Global_Deadline,
     Deadline                         => Time,
     Referenced_Event                 => Identifier)

Observer = (
     Type                              => Soft_Local_Deadline,
     Deadline                          => Time)
```

- *Max Output Jitter Requirement*: 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. Consequently, the attributes are:
  - *Max Output Jitter*. Time value (default=0).
  - *Referenced Event*. Workload event used as the time reference for the deadline.

```
Observer = (
     Type                              => Max_Output_Jitter_Req,
     Max_Output_Jitter                 => Time,
     Referenced_Event                  => Identifier)
```

- *Max Miss Ratio*: It represents a kind of soft deadline in which the deadline cannot be missed more often than a specified ratio. Its attributes are
  - *Deadline*. Time Value (default=0).
  - *Ratio*. Percentage representing the maximum ratio of missed deadlines (default=5%).

  There are two kinds of Max Miss Ratio requirements: global or local:
  - *Local Max Miss Ratio*. The deadline is relative to the activation of the step to which the timing requirement is attached. It has no additional attributes.
  - *Global Max Miss Ratio*. The deadline is relative to a *Referenced Event*, which is an additional attribute of this class.

```
Observer = (
     Type                              => Global_Max_Miss_Ratio,
     Deadline                          => Time,
     Ratio                             => Percentage,
     Referenced_Event                  => Identifier)

Observer = (
     Type                              => Local_Max_Miss_Ratio,
     Deadline                          => Time,
     Ratio                             => Percentage)
```

### 6.13.2 Queue Size Requirements

- Queue_Size_Req: 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. Its attributes are:
  - *Max_Events*: Natural. Maximum number of queued event instances that are admitted. The default value is zero

```
Observer = (
    Type                              => Queue_Size_Req,
    Max_Events                        => Natural)
```

### 6.13.3 Composite Observer

- *Composite*: An event may have several observers imposed at the same time, which are expressed via a composite observer. It is just a list of simple observers.

```
Observer = (
    Type                              => Composite,
    Requirements_List                 => (
                                      Observer 1,
                                      Observer 2,
                                      ...))
```

## 6.14 Event Handlers

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. Any overhead associated with their implementation is charged to the associated steps. Figure 6 shows the different classes of events.

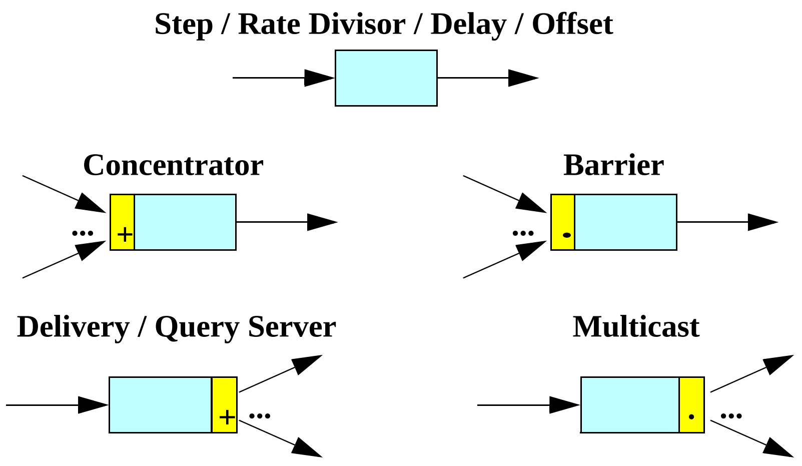


**Figure 6. Classes of Event Handlers**

### 6.14.1 Step Event Handlers

- *Step*. It represents the execution of an operation by a schedulable resource in a processing resource, with some given scheduling parameters. Its attributes are:
    - *Input event*. Reference to the event that activates the execution of the step.
    - *Output event*. Reference to the internal event that is generated when the step is finished.
    - *Step Operation*. Reference to the operation

- *Step Schedulable_Resource*. Reference to the schedulable resource (which in turn contains references to the scheduling parameters and the processing resource).
- *Lock_Schedulable_Resource*: Assertion. If this attribute is set, the thread used by the step is locked at the start of the execution of the step operation for usage by the current end-to-end flow, until released. This attribute only has effect for schedulable resources of the Thread class. The default value is Yes.
- *Unlock_Schedulable_Resource*: Assertion. If this attribute is set, the thread used by the step is released at the end of the execution of the step operation. This attribute only has effect for schedulable resources of the Thread class. The default value is Yes.

```
Event_Handler = (
     Type                              => Step,
     Input_Event                       => Identifier,
     Output_Event                      => Identifier,
     Step_Operation                    => Identifier,
     Step_Server                       => Identifier,
     Lock_Schedulable_Resource         => Yes|No,
     Unlock_Schedulable_Resource       => Yes|No)
```

### 6.14.2 Timed Event Handlers

- *Delay*. It is an event handler that generates its output event after a time interval has elapsed from the arrival of the input event. The delay may be handled by a Timer or by the environment (when the Timer reference is not present). Its attributes are:
  - *Input event*. Reference to the input event
  - *Output event*. Reference to the output event
  - *Delay Max Interval*. Longest time interval used to generate the output event. Its default value is zero.
  - *Delay Avg Interval*. Average time interval used to generate the output event. Its default value is zero.
  - *Delay Min Interval*. Shortest time interval used to generate the output event. Its default value is zero.
  - *Timer*. Reference to the timer, used when the delay has an overhead that must be modelled. If not present, the overhead associated to the delay operation is negligible.

```
Event_Handler = (
     Type                              => Delay,
     Input_Event                       => Identifier,
     Output_Event                      => Identifier,
     Delay_Max_Interval                => Time,
     Delay_Min_Interval                => Time,
     Timer                             => Identifier)
```

- *Offset*. 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. Its attributes are the same as for the *Delay* event handler, plus the following:
    - *Referenced Event*: Reference to the workload event that is used as temporal reference.

```
Event_Handler = (
     Type                              => Offset,
     Input_Event                       => Identifier,
     Output_Event                      => Identifier,
     Delay_Max_Interval                => Time,
     Delay_Min_Interval                => Time,
     Timer                             => Identifier,
     Referenced_Event                  => Identifier)
```

### 6.14.3 Flow Control event Handlers

- *Merge*. It is an event handler that generates its output event every time one of its input events arrives. Its attributes are:
    - *Input events*. References to the input events
    - *Output event*. Reference to the output event

```
Event_Handler = (
     Type                              => Merge,
     Output_Event                      => Identifier,
     Input_Events_List                 => (
                                       Identifier,
                                       Identifier,
                                       ...))
```

- *Join*. 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 (see below). Its attributes are:
    - *Input events*. References to the input events
    - *Output event*. Reference to the output event

```
Event_Handler = (
     Type                              => Join,
     Output_Event                      => Identifier,
     Input_Events_List                 => (
                                       Identifier,
                                       Identifier,
                                       ...))
```

- *Branch*. 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. Its attributes are:
  - *Input event*. Reference to the input event
  - *Output events*. References to the output events
  - *Delivery Policy*. Is the policy used to determine the output path. It may be *Scan* (the output path is chosen in a cyclic fashion) or *Random* (default value).
  - *Output_Weights*. List of weights for each output.For a RANDOM policy, if the weights are $w_1$, $w_2$, …, $w_i$, the probability of choosing a particular output $j$ is $w_j/w_T$, where $w_T$ is the total weight, $w_T=w_1+w_2+\ldots+w_i$. If policy is SCAN, the weight must be integer; if the weights for this case are $w_1$, $w_2$, …, $w_i$, the first $w_1$ events will be directed through the first output, the next $w_2$ events through the second output, and so on.

```
Event_Handler = (
     Type                              => Branch,
     Delivery_Policy                   => Scan|Random,
     Input_Event                       => Identifier,
     Output_Weights                    => (Number, Number, ...),
     Output_Events_List                => (
                                       Identifier,
                                       Identifier,
                                       ...))
```

- *Queried Branch*. 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. Its attributes are:
  - *Input event*. Reference to the input event
  - *Output events*. References to the output events
  - *Request Policy*. Is the policy used to determine the output path when there are several pending requests from the connected steps. It may be *Scan* (the output path is chosen in a cyclic fashion), *Random* (the output path is chosen randomly), *FIFO* or *LIFO*. The default value is *Scan*.

```
Event_Handler = (
     Type                              => Queried_Branch,
     Request_Policy                    => Random|FIFO|LIFO|Scan,
     Input_Event                       => Identifier,
     Output_Events_List                => (
                                       Identifier,
                                       Identifier,
                                       ...))
```

- *Fork*. It is an event handler that generates one event in every one of its outputs each time an input event arrives. Its attributes are:
  - *Input event*. Reference to the input event

- *Output events*. References to the output events

```
Event_Handler = (
     Type                             => Fork,
     Input_Event                      => Identifier,
     Output_Events_List               => (
                                      Identifier,
                                      Identifier,
                                      ...))
```

- *Rate Divisor*. It is an event handler that generates one output event when a number of input events equal to the *Rate Factor* have arrived. Its attributes are:
  - *Input event*. Reference to the input event
  - *Output event*. Reference to the output event
  - *Rate Factor*. Number of events that must arrive to generate an output event. Its default value is one.

```
Event_Handler = (
     Type                             => Rate_Divisor,
     Input_Event                      => Identifier,
     Output_Event                     => Identifier,
     Rate_Factor                      => Positive)
```

### 6.14.4 Message Event Handlers

Message event handlers are used to handle the messages in switches and routers. There are three kinds: Message_Fork, Message_Delivery, and Message_Branch. They all share the following attribute:

- *Switch*: Network_Switch. Reference to the switch in which the transfer is executed.

- *Message_Fork*: An input message is transmitted through several output networks; it is a multicast operation. Its additional attributes are:
  - *Input_Event*. Reference to the input event
  - *Output_Events_List*. List of references to the output events.

```
Event_Handler = (
     Type                             => Message_Fork,
     Switch                           => Identifier,
     Input_Event                      => Identifier,
     Output_Events_List               => (
                                      Identifier,
                                      Identifier,
                                      ...))
```

- *Message_Delivery*: It represents an input message that is delivered to one output network. Its additional attributes are:
  - *Input_Event*. Reference to the input event
  - *Output_Event*. Reference to the output event.

```
Event_Handler = (
     Type                              => Message_Delivery,
     Switch                            => Identifier,
     Input_Event                       => Identifier,
     Output_Event                      => Identifier)
```

- *Message_Branch*: 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. Its attributes are:
  - *Delivery_Policy*: Delivery_Policy_Type. The policy used to determine the output path. It may be Scan (the output path is chosen in a cyclic fashion) or Random
  - *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.
  - *Output_Weights*. List of weights for each output.For a RANDOM policy, if the weights are $w_1$, $w_2$, …, $w_i$, the probability of choosing a particular output $j$ is $w_j/w_T$, where $w_T$ is the total weight, $w_T$=$w_1$+$w_2$+…+$w_i$. If policy is SCAN, the weight must be integer; if the weights for this case are $w_1$, $w_2$, …, $w_i$, the first $w_1$ events will be directed through the first output, the next $w_2$ events through the second output, and so on.
  - *Input_Event*. Reference to the input event.
  - *Output_Events_List*. List of references to the output events.

```
Event_Handler = (
     Type                              => Message_Branch,
     Delivery_Policy                   => Scan|Random,
     Input_Event                       => Identifier,
     Output_Weights                    => (Number, Number, ...),
     Output_Events_List                => (
                                       Identifier,
                                       Identifier,
                                       ...))
```

## 6.15 End-to-end flows

The end-to-end flow represents a 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 or observers if any
- A list of event handlers

In addition, each end-to-end flow has a *Name* attribute. There is only one class of end-to-end flow defined, called a *Regular* end-to-end flow.

```
End_To_End_Flow (
Type                             => Regular,
Name                             => Identifier,
Workload_Events                  => (
                                 Workload_Event 1,
                                 Workload_Event 2,
                                 ...),
Internal_Events                  => (
                                 Internal_Event 1,
                                 Internal_Event 2,
                                 ...),
Event_Handlers                   => (
                                 Event_Handler 1,
                                 Event_Handler 2,
                                 ...));
```

# 7. Templates for the MAST File

In the text special-purpose format, the structure of a Mast model file is the following:

```
<tbd>
```

# 8. Results File Format

<tbd>

The results of the analysis are stored in the *results file* and are attached to different elements of the MAST model:

- the overall system:
  - slacks
  - traces
- end-to-end flows:
  - timing results: for each output event global response times (worst, best average) and maximum output jitter
  - end-to-end flow-specific slack
- processing resources:
  - slack
  - utilization
  - scheduler queue size
- operations:
  - slack
- schedulable resources:
  - priorities

- mutual exclusion resources:
  - priority ceilings
  - queue size

The text special-purpose format of the results file is described next, and appendix A describes the corresponding XML-Mast Format.

In the text format, the *results file* follows the same rules as the MAST model file (see Section 6, “Writing the MAST file”). The *results file* contains objects of the following types, without any particular ordering imposed:

## 8.1 Real-Time Situation

The overall system results are relative to a real-time situation that has been analysed, and contain a set of results (described below) and the following attributes:

- *Model_Name*: Name of the analysed real-time situation model.
- *Model_Date*: Date of last modification of the analyses real-time situation model, in the ISO 8601 format *YYYY-MM-DDThh:mm:ss*.
- *Generation_Tool*: Quoted text representing the name of the tool that generated the results.
- *Generation_Profile*: Quoted text representing the command and options used to invoke the tool for the generation of the results.
- *Generation_Date*: Date of generation of results, in the ISO 8601 format *YYYY-MM-DDThh:mm:ss*.

```
Real_Time_Situation (
    Model_Name                  => Identifier,
    Model_Date                  => YYYY-MM-DDThh:mm:ss,
    Generator_Tool              => "Text",
    Generation_Profile          => "Text",
    Generation_Date             => YYYY-MM-DDThh:mm:ss,
    Results                     => (
                                Result 1,
                                Result 2,
                                ...));
```

The specific results that may refer to a real-time situation are:

- *Slack*: If positive, it is the percentage by which all the execution times of all the operations in the real-time situation may be increased while still keeping the system schedulable. If negative, it is the percentage by which all the execution times of all the operations in the real-time situation have to be decreased to make the system schedulable. If zero, it means that the system is just schedulable.

```
Result = (
    Type                        => Slack,
    Value                       => Percentage)
```

- *Trace*: It describes the name of a file where trace information on the simulation of a MAST real-time situation can be found.

```
Result = (
     Type                                 => Trace,
     Pathname                             => Pathname)
```

## 8.2 End-to-end flow

The end-to-end flow results are relative to an end-to-end flow in the system that has been analysed, and contain the name of the end-to-end flow and a set of results (described below), using the following format:

```
Transaction (
     Name                                 => Identifier,
     Results                              => (
                                          Result 1,
                                          Result 2,
                                          ...));
```

The specific results that may refer to a real-time situation are:

- *Slack*: If positive, it is the percentage by which all the execution times of all the operations used by the end-to-end flow may be increased while still keeping the system schedulable. If negative, it is the percentage by which all the execution times of all the operations used by the end-to-end flow have to be decreased to make the system schedulable. If zero, it means that the end-to-end flow is just schedulable.

```
Result = (
     Type                                 => Slack,
     Value                                => Percentage)
```

- *Timing_Result*: Represents the timing results of a relevant event of the end-to-end flow and obtainable by a schedulability analysis tool. Its attributes are:
  - *Event_Name*: Name of event. The timing results always corresponds to the step or steps that generated the event represented by this name.
  - *Worst_Local_Response_Time*: Worst local response time, measured as the worst difference between the activation and completion times of the step that generated the event with this result.
  - *Best_Local_Response_Time*: Best local response time, measured as the best difference between the activation and completion times of the step that generated the event with this result.
  - *Worst_Blocking_Time*: Worst-case delay caused by the used of mutual exclusion resources. It represents the blocking time for the segment of steps preceding the referenced event. A segment of steps is a set of consecutive steps (consecutive in the end-to-end flow graph) that are run by the same schedulable resource.
  - *Num_Of_Suspensions*: Maximum number of suspensions caused by mutual exclusion resources, for the segment of steps preceding the referenced event.

- *Worst_Global_Response_Times*: List of global response times each representing the worst-case response time relative to a particular input event.
- *Best_Global_Response_Times*: List of global response times each representing the best-case response time relative to a particular input event.
- *Jitters*: List of maximum output jitter values, each representing the maximum jitter relative to a particular input event.

```
Result = (
     Type                               => Timing_Result,
     Event_Name                         => Identifier,
     Worst_Local_Response_Time          => Time,
     Best_Local_Response_Time           => Time,
     Worst_Blocking_Time                => Time,
     Num_Of_Suspensions                 => Natural,
     Worst_Global_Response_Times        => (
                                        Global_Response_Time 1,
                                        Global_Response_Time 2,
                                        ...),
     Best_Global_Response_Times         => (
                                        Global_Response_Time 1,
                                        Global_Response_Time 2,
                                        ...),
     Jitters                            => (
                                        Global_Response_Time 1,
                                        Global_Response_Time 2,
                                        ...));
```

- *Simulation_Timing_Result*: Represents the timing results of a relevant event of the end-to-end flow and obtained by a simulation tool. Its attributes are those of a *Timing_Result* plus the following:
  - *Avg_Local_Response_Time*: Average local response time, measured as the average difference between the activation and completion times of the step that generated the event with this result.
  - *Avg_Blocking_Time*: Average-case delay caused by the used of mutual exclusion resources. It represents the average blocking time for the segment of steps preceding the referenced event. A segment of steps is a set of consecutive steps (consecutive in the end-to-end flow graph) that are run by the same schedulable resource.
  - *Max_Preemption_Time*: Maximum time spent by the step preceding the event in the scheduler ready queue, while having been activated by a specific event instance. This is equivalent to the time the step is being preempted by higher priority steps.
  - *Suspension_Time*: Maximum time spent in the step input queue by the event that triggered the step preceding the event to which this result is attached. This time is larger than zero only if the triggering event arrives while the step is still busy processing a previous event.

- *Num_Of_Queued_Activations*: Maximum number of pending activations in the input queue of the step preceding the referenced event.
- *Avg_Global_Response_Times*: List of global response times each representing the average-case response time relative to a particular input event.
- *Local_Miss_Ratios*: List of local miss ratios, each representing the ratio of events that have missed a specific soft local deadline.
- *Global_Miss_Ratios*: List of global miss ratios, each representing the ratio of events generated at a specific input event channel, that have missed a specific soft global deadline.

```
Result = (
     Type                             => Simulation_Timing_Result,
     Event_Name                       => Identifier,
     Worst_Local_Response_Time        => Time,
     Avg_Local_Response_Time          => Time,
     Best_Local_Response_Time         => Time,
     Worst_Blocking_Time              => Time,
     Avg_Blocking_Time                => Time,
     Max_Preemption_Time              => Time,
     Suspension_Time                  => Time,
     Num_Of_Suspensions               => Natural,
     Num_Of_Queued_Activations        => Natural,
     Worst_Global_Response_Times      => (
                                      Global_Response_Time 1,
                                      Global_Response_Time 2,
                                      ...),
     Avg_Global_Response_TimeS        => (
                                      Global_Response_Time 1,
                                      Global_Response_Time 2,
                                      ...),
     Best_Global_Response_Times       => (
                                      Global_Response_Time 1,
                                      Global_Response_Time 2,
                                      ...),
     Jitters                          => (
                                      Global_Response_Time 1,
                                      Global_Response_Time 2,
                                      ...),
     Local_Miss_Ratios                => (
                                      Miss_Ratio 1,
                                      Miss_Ratio 2,
                                      ...),
     Global_Miss_Ratios               => (
                                      Global_Miss_Ratio 1,
                                      Global_Miss_Ratio 2,
                                      ...));
```

A *Global_Response_Time* contains the following attributes:

- *Referenced_Event*: Name of referenced input event, used for calculating the response time.
- *Time_Value*: Global response time, calculated as the difference between the arrival of the input referenced event and the generation of the event to which the result is attached, and adding the input jitter.

```
Global_Response_Time = (
     Referenced_Event                  => Identifier,
     Time_Value                        => Time),
```

A *Miss_Ratio* contains the following attributes:

- *Deadline*: Soft deadline against which the response time is compared to determine the ration of missed deadlines.
- *Ratio*: Percentage of events that have missed the soft deadline, relative to the total number of events.

```
Miss_Ratio = (
     Deadline                          => Time,
     Ratio                             => Percentage),
```

A *Global_Miss_Ratio* contains the following attributes:

- *Referenced_Event*: Name of referenced input event, used for calculating the response time.
- *Miss_Ratios*: List of miss ratios.

```
Global_Miss_Ratio = (
     Referenced_Event                  => Identifier,
     Miss_Ratios                       => (
                                       Miss_Ratio 1,
                                       Miss_Ratio 2,
                                       ...)),
```

## 8.3 Processing_Resource

The processing resource results are relative to a processing resource in the system that has been analysed, and contain the name of the resource and a set of results (described below), using the following format:

```
Processing_Resource(
     Name                              => Identifier,
     Results                           => (
                                       Result 1,
                                       Result 2,
                                       ...));
```

The specific results that may refer to a processing resource are:

- *Slack*: If positive, it is the percentage by which all the execution times of all the operations executed in the processing resource may be increased while still keeping the system schedulable. If negative, it is the percentage by which all the execution times of all the operations executed in the processing resource have to be decreased to make the system schedulable. If zero, it means that the processing resource is just schedulable.

```
Result = (
     Type                              => Slack,
     Value                             => Processing resource slack)
```

- *Utilization*: This result measures the relation, in percentage, between the time that the processing resource is being used to execute steps, and the total elapsed time. It may contain the following attributes:
    - *Total*: overall utilization in the processing result.

```
Result = (
     Type                              => Utilization,
     Total                             => percentage)
```

- *Detailed_Utilization*: This result measures the relation, in percentage, between the time that the processing resource is being used to execute steps, and the total elapsed time. This result can only be present if no “Utilization” result is present. It may contain the following attributes:
    - *Total*: overall utilization in the processing result.
    - *Application*: utilization of the processing resource by the application code, i.e., without the overhead elements included in the MAST model: context and interrupt switches, network drivers, and system timers.
    - *Context_Switch*: utilization of the processing resource by context and interrupt switch steps.
    - *Timer*: utilization of the processing resource by the system timer overhead.
    - *Driver*: utilization of the processing resource by the network drivers overhead.

```
Result = (
     Type                              => Detailed_Utilization,
     Total                             => percentage,
     Application                       => percentage,
     Context_Switch                    => percentage,
     Timer                             => percentage,
     Driver                            => percentage)
```

- *Ready_Queue_Size*: It contains the following attributes:
    - *Max_Num*: Maximum number of schedulable resources that are simultaneously ready in the processing resource.

```
Result = (
     Type                              => Ready_Queue_Size,
     Max_Num                           => Positive)
```

## 8.4 Operation

The operation results are relative to an operation in the system that has been analysed, and contain the name of the operation and a set of results (described below), using the following format:

```
Operation (
     Name                                   => Name of the operation,
     Results                                => (
                                            Result 1,
                                            Result 2,
                                            ...));
```

The specific results that may refer to an operation are:

- *Slack*: If positive, it is the percentage by which the execution times of the operation may be increased while still keeping the system schedulable. If negative, it is the percentage by which the execution times of the operation have to be decreased to make the system schedulable. If zero, it means that the system is just schedulable with regard to this operation.

```
Result = (
     Type                                   => Slack,
     Value                                  => Percentage)
```

## 8.5 Schedulable Resource

The schedulable resource results are relative to a schedulable resource in the system that has been analysed, and contain the name of the schedulable resource and a set of results (described below), using the following format:

```
schedulable_resource (
     Name                                   => Name of the schedulable resource,
     Results                                => (
                                            Result 1,
                                            Result 2,
                                            ...));
```

The specific results that may refer to a schedulable resource are:

- *Scheduling_Parameters*: The scheduling parameters that were used in the analysed system. Usually they are only written to the file if they were automatically calculated by the scheduling parameters assignment tools. See section on “Scheduling Parameters” for a description of their format.

```
Result = (
     Type                                   => Scheduling_Parameters,
     Server_Sched_Parameters                => Sched_Parameters)
```

- *Synchronization_Parameters*: The synchronization parameters that were used in the analysed system. Usually they are only written to the file if they were automatically calculated by the scheduling parameters assignment tools. See section on “Synchronization Parameters” for a description of their format.

```
Result = (
     Type                             => Synchronization_Parameters,
     Server_Synch_Parameters          => Synch_Parameters)
```

## 8.6 Mutual Exclusion Resource

The mutual exclusion resource results are relative to a mutual exclusion resource in the system that has been analysed, and contain the name of the mutual exclusion resource and a set of results (described below), using the following format:

```
Shared_Resource (
     Name                             => Name of the mutual exclusion
resource,
     Results                          => (
                                      Result 1,
                                      Result 2,
                                      ...));
```

The specific results that may refer to a mutual exclusion resource are:

- *Ceiling*: The priority ceiling automatically calculated by the MAST tool. Only mutual exclusion resources of the type *Immediate_Ceiling_Resource* may have this type of result.

```
Result = (
     Type                             => Priority_Ceiling,
     Ceiling                          => Any_Priority)
```

- *Preemption Level*: The preemption level automatically calculated by the MAST tool. Only mutual exclusion resources of the type *SRP_Resource* may have this type of result.

```
Result = (
     Type                             => Preemption_Level,
     Level                            => Preemption_Level)
```

- *Queue_Size*: Size of the waiting queue of the mutual exclusion resource. It contains the following attributes:
    - *Max_Num*: Maximum number of threads that were queued in the mutual exclusion resource, waiting to lock it.

```
Result = (
     Type                             => Queue_Size,
     Max_Num                          => Maximum number)
```

- *Utilization*: It measures the total time that the mutual exclusion resource has been locked during a simulation, relative to the total elapsed time

```
Result = (
     Type                                => Utilization,
     Total                               => percentage)
```

## 9. Traces File Format

The traces that are generated by some of the Mast tools are stored in the *traces file*. They contain four data block:

- the real-time situation description:
- Message types description.
- Message sources description.
- Messages list.

The overall system data are relative to a real-time situation that has been analysed, to the tools that has generate the trace and to the time range. It contains the following attributes:

- *Model_Name*: Name of the analysed real-time situation model.
- *Model_Date*: Date of last modification of the analyses real-time situation model, in the ISO 8601 format *YYYY-MM-DDThh:mm:ss*.
- *Generation_Tool*: Text representing the name of the tool that generated the traces.
- *Generation_Profile*: Text representing the command and options used to invoke the tool for the generation of the traces.
- *Generation_Date*: Date of generation of traces, in the ISO 8601 format *YYYY-MM-DDThh:mm:ss*.
- *Init-Time*: Start time of generation of traces, in the experiment scale time.
- *End-Time*: End time of generation of traces, in the experiment scale time

```
TRACE_FILE (
     Model_Name                          => Identifier,
     Model_Date                          => YYYY-MM-DDThh:mm:ss,
     Generator_Tool                      => "Text",
     Generation_Profile                  => "Text",
     Generation_Date                     => YYYY-MM-DDThh:mm:ss,
     Init_Time                           => Float,
     End_Time                            => Float,
     Msg_Type_List                       => (
                                         Msg_Type 1,
                                         Msg_Type 2,
                                         ...)
     Src_List                            => (
                                         Src 1,
                                         Src 2,
                                         ...)
     Msg_List                        => (
                                         Msg 1,
                                         Msg 2,
                                         ...));
```

The list included in a traces register of a real-time situation traces are:

- *Msg_Type_List*: List of types of messages referenced in the traces register. Each message type is described by:
  - *Mid*: Message type identification that is a natural number.
  - *Type*: Explanatory text of the message type (it is optional).

```
Msg_Type = (
    Mid                             => Natural,
    Type                            => String)
```

- *Src_List*: List of objects of the real-time situation model that are able to generate messages in the trace. Each source object is described by:
  - *Sid*: Source identification that is an integer number.
  - *Name*: Textual identifier of the source object in the model (it is optional).
  - *Type*: Type of the object in the model domain.

```
Src = (
    Sid                             => Integer,
    Name                            => Identifier,
    Type                            => String)
```

- *Msg_List*: List of timed message that build up the traces register. Each message item is described by:
  - *Time*: Time of message generation in the experiment scale time.
  - *Sid*: Identifier of source object that has generate the message.
  - *Mid*: Identifier of the type of the message.

```
Msj = (
    Time                            => Float,
    Sid                             => Integer,
    Mid                             => Natural)
```

## 10. Example of a Single-Processor System: CASEVA

CASEVA is a robot designed for automatic welding of junctions between pieces that don’t have axial symmetry. It has an embedded controller that uses a VME-bus based computer (an

HP 743rt) running HP-RT as its real-time operating system. The application software is concurrent, and written in Ada. The basic characteristics of its tasks are shown in Figure 7.

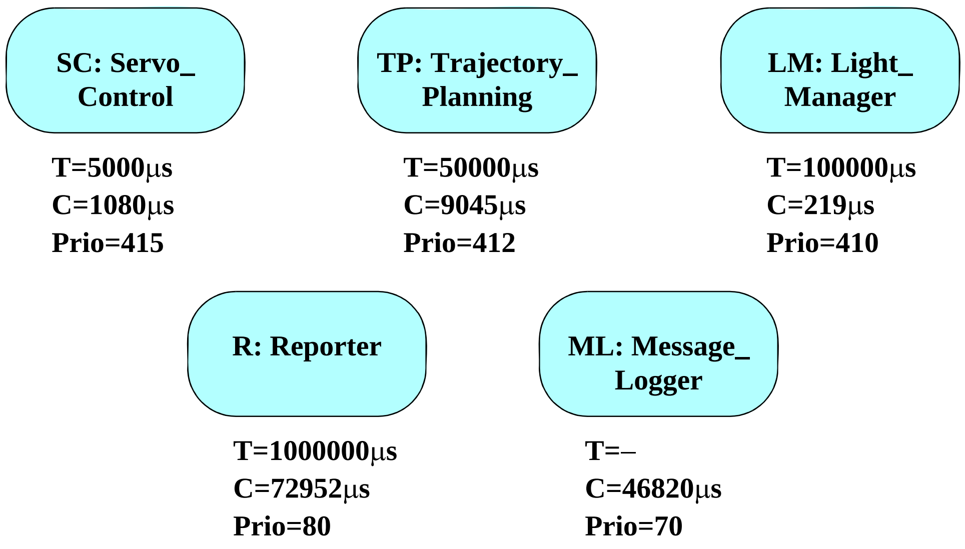


**Figure 7. Basic Characteristics of the tasks of the CASEVA controller**

Communication and synchronization between the different tasks is asynchronous, and based on mutual exclusion resources implemented using Ada's protected objects. In this document we present a simplified view of the mutual exclusion resources and associated protected operations, to make the description shorter. The following table shows the characteristics of the simplified protected objects and operations.

| **Mutual Exclusion Resource** | **Operation** | **WCET (μs)** | **Used by** |
|---|---|---|---|
| Servo_Data | Read_New_Point<br>New_Point | 87<br>54 | SC<br>TP |
| Arm | Read_Axis_Positions<br>Control_Servos | 135<br>99 | SC, R<br>SC |
| Lights | Turn_On<br>Turn_Off<br>Time_Lights | 74<br>71<br>119 | TP<br>TP<br>LM |
| Alarms | Read_All<br>Set | 78<br>59 | SC, TP, R<br>SC, TP |
| Error_Log | Notify_Error<br>Get_Error_From_Queue | 85<br>79 | TP<br>ML |

The MAST description of this system is shown next:

```
<tbd>
```

## 11. Example of linear end-to-end flows: RMT

The following example will show the aspect of the MAST file format that has been chosen to represent the timing behaviour of real-time applications. The example is a simplification of the control system of a teleoperated robot. This is a distributed system with two specialized nodes: a local robot controller, and a remote teleoperation station, where the operator manipulates the controls, and gets information about the system status. Figure 8 shows a diagram of the software architecture. The system has three end-to-end flows; one of them, the main control loop, implies execution in different processing resources, and has a global end-to-end deadline. Communication is through an ethernet network used in master-slave mode to achieve hard real-time behaviour.

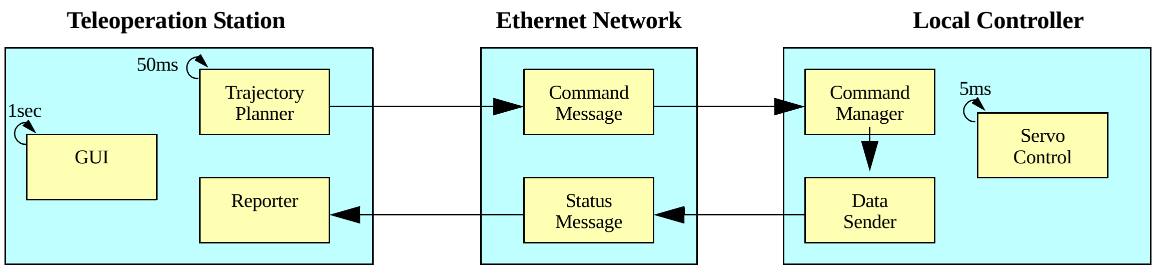


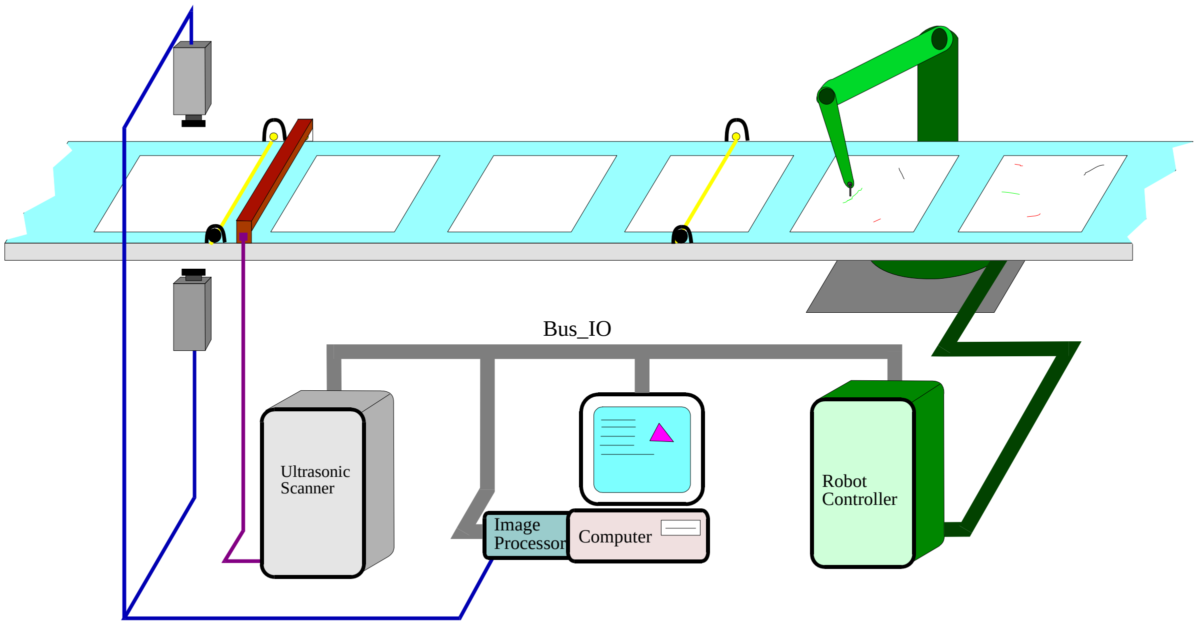
**Figure 8. Architecture of the teleoperated robot controller**

In the MAST description we can see that we declare, in this order, the processing resources, the schedulable resources, the mutual exclusion resources, the operations, and finally, the end-to-end flows. The timing requirements are embedded in the events described in the end-to-end flows. The timers (and also the network drivers) are embedded in the description of the processing resources. The scheduling parameters are embedded in the description of the schedulable resources. Finally, the events and event handlers are embedded in the description of the end-to-end flows. The description is shown next:

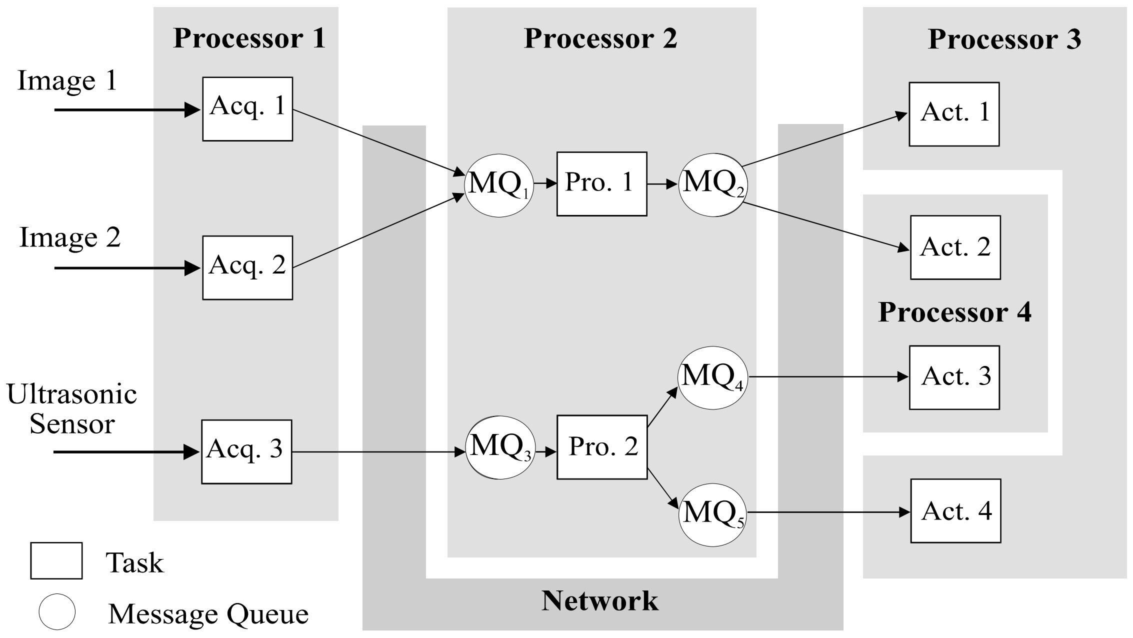
```
<tbd>
```

## 12. Example of multiple-event end-to-end flows

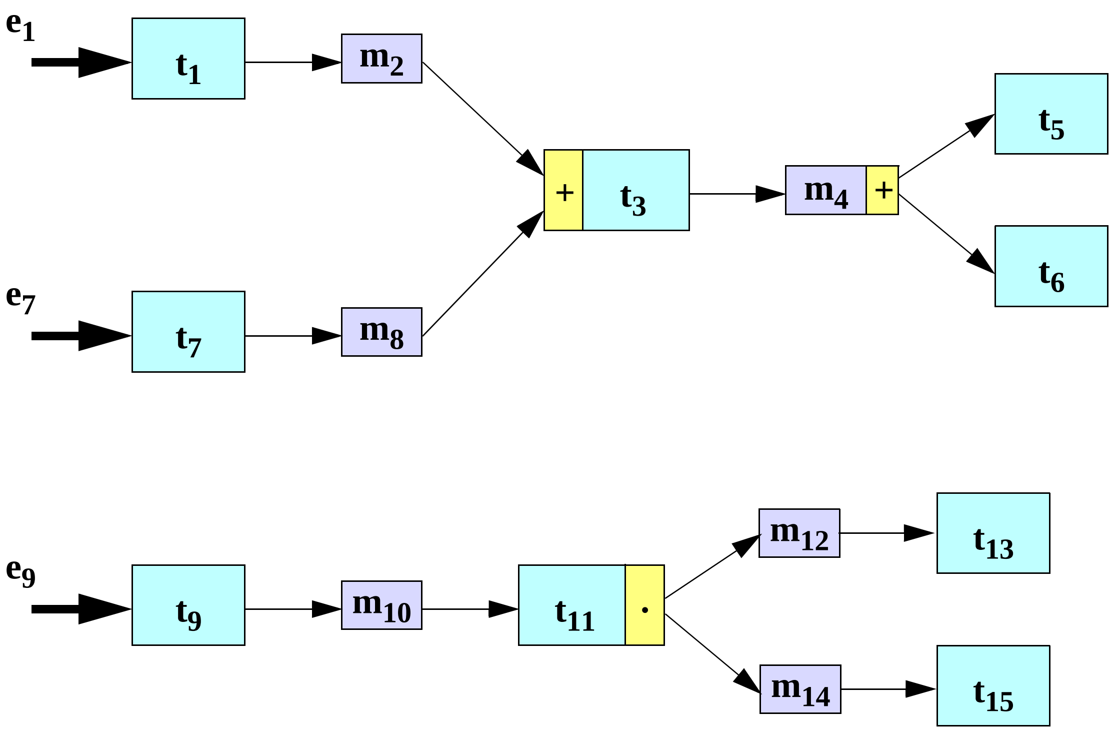
Example of steel bars inspection:



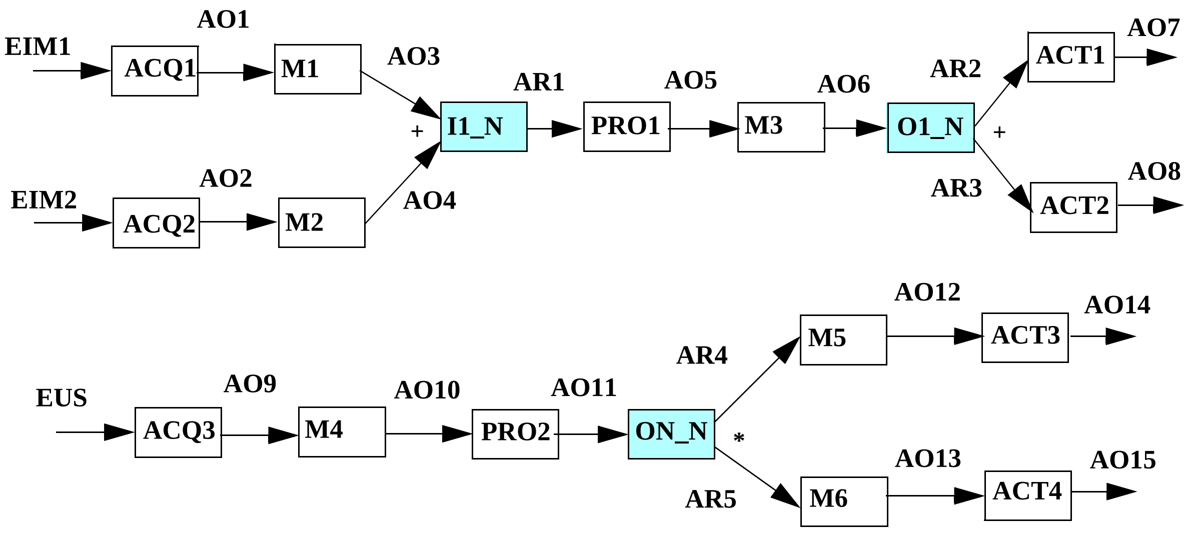
Software Architecture for this example:



Multiple event synchronization model for this example:



Graph for the example:



## Input File for the Multiple-Event Example

# APPENDIX A. XML MAST Format

In order to access the wide range of XML free tools that are available to validate, parse, analyse, and process an text file description formatted under XML rules, an XML format has been defined for the MAST model. It is formalized by means of a W3C-Schema.

The keywords of the special-purpose MAST description format have been used as tags in the XML format and therefore both have the same appearance.

## A.1. Writing the XML-Mast File

The rules for writing the Mast files with an XML-Mast format are the following:

- The tags used for delimiting the model element are the keywords that are defined as types in the special-purpose MAST description format. There are only a few exceptions, mentioned in Table 3.

**Table 3. Relation between MAST model format and XML tags**

| special-purpose format type | XML_Mast tag |
|---|---|
| Regular (schedulable_resource) | Regular_schedulable_resource |
| Fixed_Priority | Fiched_Priority_Scheduler |
| EDF (Scheduler) | EDF_Scheduler |
| FP_Packet_Based | FP_Packet_Based_Scheduler |
| Ticker (System_Timer) | Ticker_System_Timer |
| Alarm_Clock (System_Timer) | Alarm_Clock_System_Timer |
| Periodic (workload_event) | Periodic_workload_event |
| Sporadic (workload_event) | Sporadic_workload_event |
| Unbounded (workload_event) | Unbounded_workload_event |
| Bursty (workload_event) | Bursty_workload_event |
| Singular (workload_event) | Singular_workload_event |
| Regular (Event) | Regular_Event |
| Regular (end-to-end flow) | Regular_end-to-end flow |
| Composite (Timing req.) | Composite_Timing_Requirement |

- The name attributes are mandatory.
- Blank spaces, tabs and new lines are ignored.
- Identifiers or names follow the Ada rules for identifiers: they begin with a letter, followed by letters, digits, underscores ('_') or periods ('.').
- The identifiers are XML attributes and are always expressed with quotes. There are no reserved words.
- The order of declaration of the modelling elements is not relevant. A name may be referenced in the file before it is declared.
- Floating point types without fractional part can be expressed without the decimal point.
- Comments are like in XML: they begin with the four character pattern ("<!--") and end with the three character pattern ("-->").
- The description is case-sensitive, although the identifiers are not.

## A.2. W3C-Schema template for the XML Mast Model format

The XML-Mast format of the model file is defined in agreement with the terms and concepts that have been defined in the previous Section 6.

### A.2.1 W3C-Schema definition

See file mast_model_xsd.pdf

## A.3. W3C-Schema template for the XML Mast Results format

The XML-Mast format of the results file is defined in agreement with the terms and concepts that have been defined in Section 8.

### A.3.1 W3C-Schema definition

See file mast_results_xsd.pdf

## A.4. W3C-Schema template for the XML Mast Trace format

The XML-Mast format of the trace register file is defined in agreement with the terms and concepts that have been defined in Section 9.

### A.4.1 W3C-Schema definition.

See file mast_trace_xsd.pdf