**Tutorial: Schedulability analysis of a communication processing system with MARTE**

Ansgar Radermacher, based on work from Shuai Li (CEA LIST)

In this tutorial we will model a communication processing system of a rover, then analyze its schedulability. The communication processing system reads incoming errors and commands. Raw errors and commands are first processed into a readable format. Then they are handled and some data are sent over the air. UML and MARTE will be used for modeling.

|  |  |
| --- | --- |
| **Pre-lab** |  |

**Objectives**:

* Install Papyrus SW designer
* Import projects
* Apply MARTE profile
* Prioritize labs

**Exercises:**

1. Use the Papyrus SW designer update site  
   (see <https://wiki.eclipse.org/Papyrus/Designer/getting-started>)
2. Open Papyrus.
3. Import the sched\_rover project that you downloaded
4. Open the sched\_rover\_student.di Papyrus model
5. Verify that the MARTE profile is applied on the model. If not, apply it: Click model root > Properties > Profile > Apply registered profile (button next to red cross) > MARTE > Select all sub-profile
6. In the following, some of the exercises may already have been done for you. Before you start modeling, **verify well** if the demanded model elements are already present (do not re-model them).
7. The student model of Lab 1 is complete enough for you to continue the rest of the labs. Since Lab 1 reviews many concepts you have already seen in previous tutorials, we suggest to only verify the existing elements in Lab 1, and to go directly to Lab 2. We suggest to come back to Lab 1 after you have finished at least Lab 4.

|  |  |
| --- | --- |
| **Lab 1 – Functional model** |  |

**Objectives**:

* ***Note that Lab1 is already done in the student model, just verify the concepts***
* Model the functional classes
* Model the internal structure of a Communication Processing system
* Use MARTE stereotypes to specify communication protocols between internal parts

**Exercises:**

1. Model the following functional classes in the **Design::Functional::Functions** package, with the **Functions** diagram in the same package:
   * A **Communication Processing** class
   * An **Error Processing** class
   * A **Command Processing** class
   * A **Command And Error Handler** class
2. In the **Functional::FunctionalArchitecture** package, use the pre-filled **FunctionalArchitecture** diagram to model the internal structure of the **Communication Processing** class, by considering the specifications below.

* The **Communication Processing** class has three parts: **errorProcessing** part, **commandProcessing** part, and **commandAndErrorHandler** part.
* **errorProcessing** part and **commandProcessing** part interact with the **commandAndErrorHandler** part. All parts interact with the external environment of the **CommunicationProcessing** class.
* Interaction is done through connected ports.

1. Based on the previous two exercises, model the behaviors below as owned behaviors of the functional classes. You can model an owned behavior of a class as an *Activity* (but do not actually describe the activity)!
   * **Error Processing Behavior**
   * **Command Processing Behavior**
   * **Error Handling Behavior**
   * **Check Command Behavior**
2. Type the ports of the **CommandAndErrorHandler** part with types in the **Design::Functional::Functional::Datatypes** package. Check how other ports are typed, and make sure to stay consistent with how you connected ports previously.
3. For ports of **CommandAndErrorHandler** that are not stereotyped, stereotype them as flow ports and specify the direction of the data flow.

**Stereotypes that may be used:**

* <<FlowPort>>

|  |  |
| --- | --- |
| **Lab 2 – Platform model** |  |

**Objectives**:

* Model a generic platform
* Model instances of the platform elements

**Exercises:**

1. In the **Platform** package, use the **Abstract Platform** to model types of a generic platform. Generic platform types to model :
   * Task
   * CPU
2. In the **SchedAnalysis** package, use the “Platform instance” diagram (within the PlatformInstance class) to model a reference instance platform containing task instances and processor instances. Instances to model:

* There are two tasks: **errorTask** and **commandTask**. The former has a priority of 10, the latter a priority of 20.
* The system is uniprocessor with a **mainCpu**. The *frequency* of the processor is 1Ghz.
* Don’t forget to type instances with the types of the abstract platform you modeled previously.

**Stereotypes that may be used:**

* <<SwSchedulableResource>>
* <<HwProcessor>>

|  |  |
| --- | --- |
| **Lab 3 – Schedulability analysis model** |  |

**Objectives**:

* Model for real-time architecture candidate for schedulability analysis
  + System workload as end-to-end flows where work is done by tasks
  + Candidate instance platform
* Model allocations to relate system workload to candidate instance platform that executes it

**Exercises:**

1. Stereotype the **ThreadEnd2EndFlows** activity in the **SchedAnalysis** packagewith a MARTE stereotype to indicate that it is a “workload behavior”.
2. Open the **Thread End 2 End Flows** diagram in the **ThreadEnd2EndFlows** activity. Model the workload of an **ErrorTask**, as an *ActivityPartition* in the **ErrorProcessingFlow** activity partition, itself in the **ThreadEnd2EndFlows** activity. Model the following elements (most are contained by the **ErrorTask**):
   * We assume errors arrive at most every 100ms (i.e. periodically): use an *AcceptEventAction*, and specify it as a “workload event” thanks to MARTE.
   * The error is first processed in 10ms with the **ErrorProcessingBehavior** of the **ErrorProcessing** class in the **Functions** package: use a *CallBehaviorAction*, and specify it as a “step” thanks to MARTE.
   * Then the error is handled in 5ms, with the **ErrorHandlingBehavior** in the **CommandAndErrorHandler** class in the **Functions** package: use a *CallBehaviorAction*, and specify it as a “step” thanks to MARTE.
   * The WCET of the **ErrorTask** is thus the sum of execution times of both behaviors (15ms): use <<SaStep>> on the *ActivityPartition* called **ErrorTask** and fill the “hostDemand” property (currently can’t be derived automatically by tools so your manual input is necessary).
   * The **ErrorTask** is released by the periodic workload event it contains: use <<SaStep>> on the *ActivityPartition* called **ErrorTask** and fill the “cause” property (currently can’t be derived automatically by tools so your manual input is necessary).
   * The deadline of the **ErrorProcessingFlow** is 100ms: use MARTE to indicate this property of the “end-to-end flow”.
3. In the same way, model the workload of a **CommandTask**, as an *ActivityPartition* in the **CommandProcessingFlow** activity partition, itself in the **ThreadEnd2EndFlows** activity:
   * Commands arrive periodically every 50ms.
   * The command is first processed in 20ms.
   * Then the command is checked in 10ms.
   * The total WCET of the task is thus 30ms.
   * The task is released by the periodic workload event it contains.
   * The deadline of the end-to-end flow is 50ms.
4. Now that you have a complete model for schedulability analysis, you need to indicate that it can be analyzed:
   * Stereotype the **SchedAnalysis** package to indicate it is a “scheduling analysis context”.
   * Make sure the “optimization criterion” of the “scheduling analysis context” is to meet hard deadlines.

**Stereotypes that may be used:**

* <<GaWorkloadBehavior>>
* <<GaWorkloadEvent>>
* <<SaStep>>
* <<SaEndToEndFlow>>
* <<GaResourcesPlatform>>
* <<SaExecHost>>
* <<SaAnalysisContext>>

|  |  |
| --- | --- |
| **Appendix – Some useful MARTE stereotypes and how to use them** |  |

|  |  |
| --- | --- |
| **Stereotype and how to apply** | **Useful attributes** |
| <<FlowPort>> applied on Port | * direction: direction in which data transits |
| <<SwSchedulableResource>> applied on Class or Property | * schedParams: scheduling parameters in VSL, e.g. priority(10) |
| <<HwProcessor>> applied on Class or Property | * frequency: the frequency of the processor in VSL, e.g. {value=10, unit=Ghz} |
| <<SaAnalysisContext>> applied on a Package |  |
| <<GaResourcesPlatform>> applied on a Class | * Resources: the list of resources for analysis of this platform, e.g. list of properties stereotyped <<SasExecHost>> |
| <<SaExecHost>> applied on Property | * schedPolicy: the scheduling policy, e.g. FixedPriority |
| <<GaWorkloadBehavoir>> | NA |
| <<SaEndToEndFlow>> applied on Activity | * end2endD: the flow deadline in VSL, e.g. {value=50, unit=ms} |
| <<GaWorkloadEvent>> applied on AcceptEventAction | * pattern: the arrival pattern of the events in VSL, e.g. periodic({period={value=100.0,unit=ms}}) |
| <<SaStep>> applied on an ActivityPartition or CallBehaviorAction | * hostDemand: execution time of the step in VSL, e.g. {value=5, unit=ms} * cause: the <<GaWorkloadEvent>> that releases this step, e.g. an AcceptEventAction stereotyped <<GaWorkloadEvent>> |
| <<Allocate>> applied on Abstraction (relationship) | NA |