D2.6: MegaMart Design Tool Set Guidelines

MILESTONE: 30/11/2019 (M32)
STATUS: FINAL
COORDINATOR: UNICAN
CONTRIBUTORS: ABO, ARM, ATOS, CON, FTS, INT, MDH, RO, SICS, SMA, SOFT, SSF, UAQ, UCAN, UOC, UPAU, VTT
REVIEWERS: ATOS, ARM

DISSEMINATION LEVEL: PUBLIC
HANDLE: HTTP://hdl.handle.net/20.500.12004/1/MMART2/D2.6
LAST EDITED: 30/11/2019
Executive summary

Info: This deliverable provides a description of the methods and guidelines for final version of the MegaMart Integrated Tool Set described in D2.5.

Participants: SOFT, SMA, ARM, UPAU, ATOS, UCAN, UOC, FTS, UAQ, INT, RO, ABO, SSF, VTT, CON, MDH, SICS, BUT

Input: D2.5

Description: Task 2.4 defined the guidelines to apply the system design approach envisaged in MegaM@Rt. Deliverable D2.6 aims to describe the methodologies and guidelines for the System Engineering tools developed in WP2, including the conceptual aspects and the process development steps for model driven development. In this document, the technology providers deliver the guidelines and provide specific support to industrial partners involved in the development of the use cases. They will receive back the feedback on tools usage, in terms of fixes and improvements that may be necessary within the lifetime of the project. Technology providers will analyze such requests for tool consolidation and will deliver an improved and completed version of the MegaM@Rt tool chain before the end of the project.

Coordinator: UCAN, Eugenio Villar (villar@teisa.unican.es)

History of the Document:

<table>
<thead>
<tr>
<th>version</th>
<th>date</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>October 23, 2019</td>
<td>Initial ToC</td>
</tr>
<tr>
<td>0.2</td>
<td>November 5, 2019</td>
<td>Initial draft</td>
</tr>
<tr>
<td>0.3</td>
<td>November 22, 2019</td>
<td>Comments from reviewers</td>
</tr>
<tr>
<td>1</td>
<td>November 30, 2019</td>
<td>Final document</td>
</tr>
</tbody>
</table>
# Table of Contents

## Acronyms
- Methodology for Modelio Variability Designer
- Guidelines for Modelio Variability Designer
- System Modeling Methodology improvements
- Verification Methodology improvements
- Google Test
- Google Remote Procedure Calls (gRPC)
- Interfaces adaptation
- Verification methodology
- Testing process
- Guidelines for S3D
- Component restrictions and properties
- PDM (Platform Description Model) Views
- Methodology for PADRE
- Guidelines for PADRE:
- Methodology for CHESS
- Guidelines for CHESS
- Collaboro Methodology: Making DSML Development Collaborative
  - Usage guidelines for Collaboro: Using Collaboro to build a DSML collaboratively
- EMFtoCSP
  - EMFtoCSP methodology
  - Usage guidelines for EMFtoCSP
- Xamber
  - Methodology for Xamber:
  - Guidelines for Xamber:
- XPM
  - Methodology for XPM:
  - Guidelines for XPM:
Papyrus extension for AOM 60
  Methodology for Papyrus extension 60
  Guidelines for Papyrus extension 60
Moka extension for logging 63
  Methodology for Moka extension for logging 63
  Guidelines for Moka extension: 63
VeriATL 65
  Methodology for VeriATL 65
  Guidelines for VeriATL: 67
HepsyCode 70
  Methodology for Hepsycode: 70
  Guidelines for Hepsycode: 71
JTL 76
  Methodology 76
  Guidelines 77
PauWare 78
  Methodology 78
  Guidelines 80
CMA 81
  Methodology for CMA 81
  Guidelines for CMA 82
MATERA2 82
  Methodology for MATERA2 82
  Guidelines for MATERA2 83
RCRS 83
  Methodology for RCRS: 83
  Guidelines for RCRS: 84
References 85
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>AL</td>
<td>Architectural Language</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APL</td>
<td>Apache Public License</td>
</tr>
<tr>
<td>ATL</td>
<td>Atlas Transformation Languages</td>
</tr>
<tr>
<td>BMM</td>
<td>Business Motivation Model</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Model and Notation</td>
</tr>
<tr>
<td>BVR</td>
<td>Better Variability Result</td>
</tr>
<tr>
<td>CBSE</td>
<td>Component-Based Software Engineering</td>
</tr>
<tr>
<td>CPS</td>
<td>Cyber-physical Systems</td>
</tr>
<tr>
<td>CSR</td>
<td>Case Study Requirement</td>
</tr>
<tr>
<td>CTS</td>
<td>Conceptual Tool Set</td>
</tr>
<tr>
<td>DDS</td>
<td>Data Distribution Service</td>
</tr>
<tr>
<td>DoDAF</td>
<td>U.S. Department of Defense Architecture Framework</td>
</tr>
<tr>
<td>DSE</td>
<td>Design Space Exploration</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>DSML</td>
<td>Domain-Specific Modelling Language</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>EPL</td>
<td>Eclipse Public License</td>
</tr>
<tr>
<td>FOSS</td>
<td>Free Open Source Software</td>
</tr>
<tr>
<td>GPL</td>
<td>GNU Public License</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HPV</td>
<td>Hypervisor</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>INCOSE</td>
<td>International Council on Systems Engineering</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardisation</td>
</tr>
<tr>
<td>LGPL</td>
<td>Lesser GNU Public License</td>
</tr>
<tr>
<td>M2C</td>
<td>Model To Code</td>
</tr>
<tr>
<td>M2M</td>
<td>Model To Model</td>
</tr>
<tr>
<td>M2T</td>
<td>Model To Text</td>
</tr>
<tr>
<td>MARTE</td>
<td>Modeling and Analysis of Real-Time Embedded Systems</td>
</tr>
<tr>
<td>MAST</td>
<td>Modeling and Analysis Suite for Real-Time Applications</td>
</tr>
<tr>
<td>MBD</td>
<td>Model-Based Development</td>
</tr>
<tr>
<td>MBSE</td>
<td>Model-Based System Engineering</td>
</tr>
<tr>
<td>MBT</td>
<td>Model-Based Testing</td>
</tr>
<tr>
<td>MDE</td>
<td>Model Driven Engineering</td>
</tr>
<tr>
<td>MDSD</td>
<td>Model Driven Software Development</td>
</tr>
<tr>
<td>MDT</td>
<td>Eclipse Modeling Development Tools</td>
</tr>
<tr>
<td>Acronym</td>
<td>Full Form</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>MFR</td>
<td>MMRT Framework Requirement</td>
</tr>
<tr>
<td>ML</td>
<td>Modelling Language</td>
</tr>
<tr>
<td>MODAF</td>
<td>British Ministry of Defence Architecture Framework</td>
</tr>
<tr>
<td>NFP</td>
<td>Non Functional Property</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OCRA</td>
<td>Othello Contracts Refinement Analysis</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PDM</td>
<td>Platform Description Model</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>RTES</td>
<td>Real-Time Embedded Systems</td>
</tr>
<tr>
<td>S3D</td>
<td>Single Source System Design</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-Oriented Architecture</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>SysML</td>
<td>System Modelling Language</td>
</tr>
<tr>
<td>TCP</td>
<td>Tool Component Purpose</td>
</tr>
<tr>
<td>TP</td>
<td>Tool/Method Provider</td>
</tr>
<tr>
<td>TS-MM</td>
<td>MegaModelling Tool Set</td>
</tr>
<tr>
<td>TS-RT</td>
<td>RunTime Tool Set</td>
</tr>
<tr>
<td>TS-SE</td>
<td>System Engineering Tool Set</td>
</tr>
<tr>
<td>TSC</td>
<td>Tool Set Component</td>
</tr>
<tr>
<td>UC</td>
<td>Use Case</td>
</tr>
<tr>
<td>UCP</td>
<td>Use Case Provider</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UTP</td>
<td>UML Test Profile</td>
</tr>
<tr>
<td>UPDM</td>
<td>Unified Profile for DoDAF/MODAF</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>Verification and Validation</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst Case Execution Time</td>
</tr>
<tr>
<td>XAL</td>
<td>XtratuM Abstraction Layer</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
1. Introduction

In this document, the technology providers deliver the guidelines and provide specific support to industrial partners involved in the development of the use cases. They will receive back the feedback on tools usage, in terms of fixes and improvements that may be necessary within the lifetime of the project. Technology providers will analyze such requests for tool consolidation and will deliver an improved and completed version of the MegaM@Rt tool chain before the end of the project.

The different tools cover specific aspects of concrete Model-Driven Design flows. All together build the MegaM@RT Design Framework. In order to facilitate the UC providers inside the project but also any potential interested external adopter, a global system engineering conceptual framework is presented in this document. Each tool from the MegaM@Rt2 Design tool set is positioned inside this conceptual framework, thus allowing a better understanding of its role in the design flow and its interaction with the rest of tools in the Megamart Framework.

2. System Engineering Tool Set methodological view

The architecture of the System Engineering Conceptual Framework has been initially defined and presented in deliverable D1.2. Refinements and updates have been provided in deliverable D2.3 based on the WP tasks progress. Further refinements may be required in the context of the WP5 work to support the framework implementation and the integration requirements. Thus they will be reported in the corresponding WP5 deliverables.

The implementation plan of the System Engineering Tool Set is detailed in D2.2 through the "roadmap" tables for tools and for the framework. These tables define the availability plan of the tools purposes related to each framework requirement. Thus, the roadmap tables are used as a basic instrument to track the evolution of the framework and the tool set capabilities, and to compare the improvements between the previous, current and following milestones.

The high level System Engineering activities, depicted in fig.2-1, are related to the main SE Tool Set architectural components, namely the Requirement Modelling, System Architecture Modelling, the Detail Design Modelling and Model Verification & Validation. The relationships between these macro activities are characterized by: i) the exchanged artefacts and ii) any sequencing and iterations related to specific methodological steps (e.g. to provide output artefact required as input to the following activity)

The artifacts provided by such macro-activities are the System Requirements (specialised in Functional and Non-Functional Requirements) and the System Models (specialised in Structural and Behavioral Models). These artefacts represent both the data exchanged between System Engineering tools and the data provided to the other components of the MegaM@Rt framework, i.e. tools coming from the Runtime Analysis Tool Set and the Model & Traceability Management Tool Set.

The flow that drive the design steps sequencing and iterations, is a peculiarity of the applied methodology. MegaM@Rt aims to support and tailor a wide range of methodological approach to comply with the current and improved industrial practices. The diagram exploit the BPMN formalism
just to show that no constraints are foreseen executing a specific activity based on the available artefacts.

Fig.2-1: System Engineering high level methodological description.

The following diagrams detail each macro activities. So, **fig.2-2** describes the requirements specification and traceability design tasks. System functional and non-functional requirements are specified and traced all along the design process. In **fig.2-3**, the architectural design activities are presented. After that, the architectural modeling, functional analysis, reuse & variability modeling and architectural component modeling can be carried out. The HW/SW co-design is described in **fig.2-4**. Finally, **fig.2-5** relates all the analysis, verification and validation tasks together. This includes:

- Requirement coverage analysis,
- Dependability analysis,
- Schedulability analysis,
- Performance analysis,
- System simulation,
- Formal methods, and
- Model-based testing.
In section 3, all the tools developed or improved in WP2 are described.

Fig.2-2: Requirement Specification activities

Fig.2-3: Architectural Design activities
Fig.2-4 Detail Design activities

Fig.2-5 Model Verification & Validation activities
3. MegaMart Tool Set

3.1. Modelio

3.1.1. Methodology for Modelio Variability Designer

In the context of the MegaM@Rt2 project, Softeam developed a Variability Designer to cover the requirements by the VCE case study.

![Diagram of Modelio instantiation of the System Engineering methodology](image)

Figure 3.1-1 Modelio instantiation of the System Engineering methodology

The process starts with creation of 150% system model which is overcharged with the functionalities. This activity may eventually take as inputs many related system models to be aggregated in a single overcharged system model. The activity can be complemented with adding specific constraints that affect variants creation. When specific functionalities are specified, a variant system model can be generated.

3.1.2. Guidelines for Modelio Variability Designer

**Installation**

Add the module to the catalog. Download the *.jmdac* module file joined to this project.

Open the Configuration > Modules catalog… menu. Click Add a module to the catalog… and add the file downloaded before.

The Variability Designer module can now be deployed in any Modelio project.

**Deploy the module in the project**

If not already done, add the module to the module catalog (See previous section). Open the Configuration > Modules… menu. Click on Add and select the Variability Designer module in the list.
**Definitions**

- **150% model**: model containing all possible assets for your product line
- **Feature model**: model handled in a Variant Management Tool, high level representation of your product line
- **Variation point**: group of variation
- **Variant management tool**: tool providing a feature model management and a way to generate new variants of a system from the features (Ex: pure::variants)

**Purpose of the module**

The goal of this module is to provide a way to create variants of a model, using a variant management tool, to use Modelio with a Product Line architecture. The general usage is to design a so-called 150% model containing all assets of all variants, annotated with constraint specifying the variability of the system. This module is not here to create and manage a feature model, which is done in other tools. Such variant management tools can be used to design new products by selecting a new set of features. The model of the new variant can then be created in Modelio from the 150% model.

**Variability Exchange Language**

This module uses the VEL (Variability Exchange Language) to communicate with other tools.

You can find more detailed information on the dedicated website. The VEL format has two modes: Description and Configuration.
A description outlines all variability point of the model, with all possible variations. Each variation is linked to model elements concerned by it. The file is used to transfer information about the model to the variant management tools.

A configuration describes a specific variant for the system. The file itself is actually similar to the description with the addition of selected tags on some variations. This file is kind of a modified version of the description, generated by the variant management tool.

**Project setup**

Create a package containing the 150% model

Add stereotype 150% model root:

(Optional) Create a variability diagram:
Add the constraints to setup your variability model. If you are using a variability diagram, select the tool for the constraint you want to apply.

Click on the model elements you want to include in the variation. Fill the constraints fields. Otherwise you can manually create a Constraint on the element to include in the variation, then add the variation stereotype and fill the constraints fields.
**Constraints:**

To represent the variability of a 150% model, we apply stereotyped constraints on model elements. There are two types of variation constraints with two sub-types each:

- Structural
- Optional
- Alternative
- Parameter
- Calculated
- Alternative

These constraints types are derived from the variation types defined in VEL.

### Structural

Structural variations are applied to the model elements we intend to remove (or not) depending on the chosen variant.

- Variation name: identifier for this specific variation
- Variation point: identifier for this group
- Type: fill with the type of the following condition
- Condition: the expression here will trigger the variation if fulfilled, the format of the condition depends on the type selected before

### Alternative

The alternative type means that exactly one variation can be chosen from the variation point. For example, if a car needs to have an electric motor or a classic motor, both model element can have an alternative variation related to the same variation point. It will ensure that only one of them is selected in the variant creation process.

### Optional

An optional variation is one where the model element will be removed depending on one condition, there is no restriction to how many variations under an Optional variation point can be selected.
Parameter variations are related to the value in a model element. For example, we can change the multiplicity max of an association, or the default value of an attribute.

- Variation name: identifier for this specific variation
- Variation point: identifier for this group
- Applied on: name of the attribute to be changed, it depends on the constrained element
- Both sub-types have their own specific attributes

Alternative

Each alternative parameter variation represents a possible value for the concerned variation point.

- Type: fill with the type of the following condition (See possible values here)
- Condition: the expression here will trigger the variation if fulfilled, the format of the condition depends on the type selected before
- Value: expression here will replace the selected attribute

Calculated

This variation contains an expression that will be evaluated to change the value of selected attribute.

- Type: fill with the type of the following expression (See possible values here, only “-expression” types are relevant here)
- Expression: the formula to determine the new value of the attribute

Possible values

The following are the possible values for the Type:

- single-feature-condition
- and-feature-condition
- or-feature-condition
- pvscl-expression
- ocl-expression
**Commands**

- **Export VEL description**

  This command will create a VEL description

  ![VEL description](image)

- **Create variant**

  Import a VEL configuration to create a new variant

  ![Create variant](image)

  The command will copy the 150% model, without every unselected model elements and variability constraints.
  The new package has to be renamed, it can also be imported by another Modelio if required.
  For more detailed tutorial please refer to the following video:
  https://www.youtube.com/watch?v=6plSzt_ikIs

3.2. **S3D**

3.2.1. **System Modeling Methodology improvements**

The improvements to S3D in WP2 regarding System Engineering affect the Platform Description Modeling (PDM). PDM captures all the information required to describe the HW/SW platform of computing resources used to execute the system functionality described in the PIM (Platform Independent Model). PDM fundamental elements are the following:

- **Network nodes**: In order to deal with the modeling of very complex systems of systems (SoS), the complete system should be partitioned in parts which should be partitioned again hierarchically until the detailed computing platform can be described by its computing architecture of HW devices. These hierarchical parts are nodes connected to each other through a network infrastructure.

- **Memory spaces**: Depending on the characteristics of the computing platform, the application mapped on it may be implemented in only one process (an executable) or several. Each executable process will share the computing resources with the other processes but in its own
3.2.2.1. Software platform: An essential element in any computing platform is the Operating System (OS), eventually, several of them when the computing platform is complex and heterogeneous enough. In some cases, when a system or a subsystem has real-time constraints, a Real-Time Operating System (RTOS) is required.

- Hardware resources: MARTE supports the modeling of HW providing a functional classification of hardware entities such as processors, memories, buses, peripherals, etc. They are grouped in the HW modeling package.
- Silicon implementation: MARTE ‘HW_Physical’ model represents hardware resources as physical components with details on their shape, size, position within platform, power consumption, heat dissipation, and many other physical properties. Based on this information, S3D will generate automatically all the information needed to feed the corresponding design flows.

Using these architectural modeling artifacts, with all the performance details they include, it is possible to obtain a more detailed simulation and related performance analysis, as shown in Figure 2.5. These modeling artifacts are being assessed in the Avionics use case by Thales TRT.

A second improvement refers to the modeling of design requirements and observed performance properties. Interfaces integrate the services provided/required by a component and define its characteristics. A Generic Component is declared with Generic Interfaces, and they should not be annotated with properties limiting its architectural applicability. When the component is instantiated in a specific application, its function interfaces and/or main functions can be annotated with properties.

Properties are defined making use of two different MARTE stereotypes:

- ResourceUsage stereotype assigned to the operation. Here, observed execution times are annotated in its execTime section, along with energy consumption in its energy section.
- RtSpecification stereotype assigned to a comment of the operation. Here, timing properties are annotated, such as period (occKind), deadline (relDl) and best/worse case execution times (rdTime).

3.2.2. Verification Methodology improvements

From the verification point of view, the system is called 'Device Under Test' (DUT). In practice, this DUT is running in a certain platform, but this does not necessarily imply that the test bench is hosted in the same platform too. If the DUT and the test bench are running in different platforms (even physically separated, in other places), it is necessary to communicate one with the other, implementing micro-services using remote procedure calls, provided by Google libraries (gRPC).

3.2.2.1. Google Test

The Google C++ Testing Framework or Google Test is a unit test library that eases the writing of C++ tests. The key concept at the time of describing a test is assertions, which are statements that check if a condition is true. As a result of that, the result could be “success”, “nonfatal failure” or “fatal failure”. Success means that the condition is verified correctly, and nonfatal failure means the opposite. In
either case, the execution of the test continues, the only difference is that the test passed or not. On the other hand, fatal failure cuts the execution of the test [1].

3.2.2.2. Google Remote Procedure Calls (gRPC)

gRPC is an open source remote procedure call developed by Google, which uses protocol buffers as its Interface Definition Language (IDL) so as its underlying message interchange format. The key of gRPC is that the definition of micro services is easy, and it is based on a client-server model [2].

3.2.2.3. Interfaces adaptation

The verification methodology is based on the component division of the DUT. Each component has a certain functionality and a series of interfaces that allows to exchange information and communicate with the other components. There are two types of interfaces: provided and required.

Starting from this point, the main goal is verifying components or higher-level structures, such as subsystems or even the complete system. To achieve this objective a testbench is needed: from a simpler point of view, another unit (testbench) sends information to the DUT and collect the results, or vice versa.

Required interfaces (DUT perspective) demand data from the testbench to operate. This information can be provided by a group of libraries specifically created with that purpose, gmock libraries. Using these libraries demands to transform the component interfaces in order to invoke gmock procedures.

This verification methodology adapts the original interfaces into new ones. Required interfaces become mock interfaces (I_example_prov MOCK_ I_example_prov) and provided interfaces become call interfaces (I_example_req CALL_ I_example_req).

Also the components, defined in C++ as classes, have functions apart from interfaces, and they required to be adapted to. Those functions are grouped in a new class, called own class (internal functions of the component class OWN_component_functions).
3.2.3. Verification methodology

The Thales Use Case was modelled with a hierarchy, considering the component as the basic functional unit. Each component can be connected with other, conforming a subsystem. There is no critical difference between a component and a sub-system, since both of them have a certain functionality (which could be simple or complex) and have their own interfaces. Precisely, the idea of interface is one of the keys in the verification methodology.

![Thales FMS functional architecture](image)

Fig.3.2-1: Thales FMS functional architecture.

This hierarchy idea is shown in Fig. 3.2-1. The biggest box corresponds with the biggest component, the hole system. It has certain interfaces, some of them provided, some of them required.

Inside the system there are subsystems and components interconnected and also linked with the system interfaces. As an example, the module SENS_C1 is a component whereas LocGroup is a subsystem. To keep showing the hierarchy, LocGroup is divided in a sort of components, each one with its own functionality:

- Loc_C1
- Loc_C2
- Loc_C3
- Loc_C4
As it happened in a higher abstraction level (system level), the bigger unit contains lower functional units, these units are connected with other elements in the same abstraction level through interfaces.

All the components are modelled with two related packages: the C/C++ codes that brings the functionality to the component and the TestBench which includes the Google Test codes to verify the correct functioning of the component.

As said before, the main goal of the test is being able to send and receive data through the component interfaces. The test and the component are not going to be in the same physical platform, so a communication procedure is needed, which is provided by Google Remote Procedure Calls (gRPC).

Messages and services are described in the proto file:

- **Messages.** Messages have to follow certain rules to be understood by Protocol Buffers:
  
  - Field types: It is mandatory to specify the name of the variable and its type. Here resides the potential of gRPC: enumerations and other messages could be defined inside a message. Applied to the Thales case of use, if a component has an interface (class) which contains not only basic types but enumeration or other classes, they could be defined as nested messages.
  - As an example: class 1 contains class 2 and class 3. The only requirement that imposes Protocol Buffer is that class 2 and 3 are defined before than class 1 in the proto file, so when the compiler tries to compile class 1; classes 2 and 3 are not unknown to him.
  - Field numbers: each field in the message definition has a unique number, which is used to identify the fields in the message binary format and should not be changed
  - Field rules: The message fields could be required, optional or repeated.
  - Scalar value types. Depending on the language the proto file is going to be compiled for, there has to be an equivalence among proto types and concrete language types.
As in MegaM@rt2 the language used is C++, only proto-C++ equivalence is going to be shown:

<table>
<thead>
<tr>
<th>Proto type</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>Float</td>
</tr>
<tr>
<td>Int32</td>
<td>Int32</td>
</tr>
<tr>
<td>Int64</td>
<td>Int64</td>
</tr>
<tr>
<td>Uint32</td>
<td>Uint32</td>
</tr>
<tr>
<td>Uint64</td>
<td>Uint64</td>
</tr>
<tr>
<td>Sint32</td>
<td>Int32</td>
</tr>
<tr>
<td>Sint64</td>
<td>Int64</td>
</tr>
<tr>
<td>Fixed32</td>
<td>Uint32</td>
</tr>
<tr>
<td>Fixed64</td>
<td>Uint64</td>
</tr>
<tr>
<td>Bool</td>
<td>Bool</td>
</tr>
<tr>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>Bytes</td>
<td>String</td>
</tr>
</tbody>
</table>

- Enumerations: useful when a field has just a predefined list of values. Each enumeration is declared in the same way as a message, but each element has to be numbered beginning with 0.

- Services. In order to use the message types with an RPC system, a service has to be defined in the proto file, so the Protocol Buffer will generate service interface code and stubs in the chosen language.

By default, the compiler will generate an abstract interface called ServiceName and a corresponding stub implementation. Once the proto file is generated, a server and a client must be created in order to be the actors in the information exchange process. They follow a slave-master philosophy: depending if the service is required or provided, one takes the role of master and the other is the slave. In concrete, the two sides of the communication are the component and the test. If:

1. The component requires a service, the component is the master and the test acts as a slave, providing (mocking) the information needed.
2. The component provides a service, the component acts as a slave and the test acts as the master, invoking (calling) the component to get a certain functionality.
3.2.4. Testing process

Since the main objective is to verify the correct functionality of the components, it has been devised a logical way of ordering the infrastructure needed in the test. gRPC is needed at this point to exchange data, but the communication process adds more elements to the schema: the client, the server and the protocol buffer/gRPC infrastructure.

This second approach could be valid though there is a certain incompatibility. The component uses interfaces, and the (complex) data types used by them are defined as classes. These classes were described in the proto file and re-generated again after the Protocol Buffer compilation, with the aim of adapting them to the communication infrastructure. Furthermore, the component interfaces are not prepared to bear calls and mocks.

For example, required interfaces (DUT perspective) demand data from the testbench to operate. This information is provided by gmock libraries. The use of these libraries demand to transform the component interfaces in order to invoke gmock procedures.

These test interfaces or new interfaces are classes that inherit from the original interfaces and add the modifications needed to do calls and mocks. There are three types of new interfaces:

- Call interfaces, which are the test variation of the interfaces used in provided services.
- Mock interfaces, which are the test version of the interfaces used in required services.
- A third type, called own interfaces. Those interfaces arise from the need of testing functions inside the component. To access them, a new interface is created.

Once all the interfaces are adapted to be tested, it is needed to create a new version of the component: the test component:
3.2.5. Guidelines for S3D

3.2.5.1. Component restrictions and properties

When the component is instantiated in a specific application, its function interfaces and/or its main functions can be annotated with properties. These properties include period, energy consumption, timing properties, etc. To specify them, component’s functions should be decorated with the \texttt{<<ResourceUsage>>} stereotype, where best, mean and worst observed execution times can be indicated in the field ‘execTime’ in the form “BOET/MOET/WOET = value; unit = unit”, and estimated energy consumption.

To indicate temporal properties as deadlines, period, best or worst-case execution times, a new comment is created in the operation stereotyped as \texttt{<<RtSpecification>>}, and referencing the operation owning this comment with the property ‘\texttt{annotatedElement}’. Possible properties that can be specified in the \texttt{RtSpecification} stereotype used by S3D are shown in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Field</th>
<th>Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline</td>
<td>relDl</td>
<td>Deadline = 1000; unit = ms</td>
</tr>
<tr>
<td>Best / Worst case execution time</td>
<td>rdTime</td>
<td>BCET = 1; WCET = 10; unit = ms</td>
</tr>
<tr>
<td>Periodicity</td>
<td>occKind</td>
<td>Periodic (Period = 1000; unit = ms)</td>
</tr>
</tbody>
</table>
3.2.5.2. **PDM (Platform Description Model) Views**

PDM captures all the information required to describe the HW/SW platform of computing resources that is used to execute the system functionality described in the PIM, and is described using different views:

- **Memory Spaces view:** The Memory Spaces package is created in the main model and stereotyped as <<MemorySpaceView>>. Then, create the memory partitions needed in your application as new components, and decorate them with the <<MemoryPartition>> stereotype. You will notice that the symbol of the component changes.
- **Software Platform view:** Create the SW Platform package and adorn it with the <<SWPlatformView>> stereotype. Proceeding on the same way as in the memory spaces view, create one component per software used in your application and assign the <<OS>> stereotype, indicating that it is an operative system.
- **Hardware Resources view:** Create the HW resources view and decorate it with the <<HWResourcesView>> stereotype. Then instantiate, if required, the network nodes of the application as components, and inside declare each HW component that shape the node with a Property, adorning the properties with different HW stereotypes (HwProcessor, HwRAM, HwBus…). Finally, Composite Structure diagrams are created to represent this HW architecture.
- **Silicon implementation:** It is defined within the Architectural view, and it defines how HW components (RAMs, ASICs, FPGAs...) are mapped into specific physical components (HwPLD, HwASIC…).

3.3. **PADRE**

3.3.1. **Methodology for PADRE**

![PADRE Methodology Diagram]

**Fig.3.3-1: PADRE methodology**

**Fig.3.3-1** instantiates the PADRE framework within this WP concerns. In particular, PADRE is aimed at introducing automation in software model refactoring, driven by performance issues. PADRE provides a very valid support to drive UML model refactoring towards models that satisfy performance requirements. Without such support, software designers and performance experts should rely only on their own experience to find performance problems and related solutions. In particular, PADRE uses
both the architectural and functional models in order to detect performance flaws with respect to Non-Functional Requirements.

Fig.3.3-2 illustrates a typical Software Performance Engineering (SPE) process based on UML, in which we have plugged the Performance Anti-pattern Detection and Model Refactoring Framework (PADRE).

**Fig.3.3-2**: PADRE’s refactoring framework plugged in a Software Performance Engineering process.

Usually, a software model does not contain performance attributes and/or indices. However, consolidated techniques exist in the literature for transforming software models into performance models, estimating performance attributes like demand vectors and workload, and obtaining indices like throughput and response time from performance model solution. This is the goal of the Performance-oriented Model Integration component in Fig.3.3-2, namely to get performance indices and set the tagged values of MARTE stereotypes that we assume as their containers in the UML model, thus obtaining a Performance-oriented Software Model.

The latter is then given as input to our EPSILON-based Performance Antipattern Detection & Model Refactoring Framework, which includes three engines based on different EPSILON languages (i.e., EPL, EVL, and EWL) depicted in Fig.3.3-3, Fig.3.3-4 and Fig.3.3-5 respectively. Each engine accomplishes the task of: (i) detecting bad design practices that degrade application performance (i.e. performance antipatterns), by verifying codified detection rules, and (ii) removing such bad design by applying codified refactoring actions. Although none of EPSILON languages was conceived for addressing performance-driven model refactoring, we have identified different kinds of model refactoring support that they can provide due to their different execution semantics, as summarized in the following.
**Batch refactoring sessions:** Based on EPL, they allow to sequentially execute a set of antipattern detection rules and refactoring actions. The process can be repeated once (i.e., standard mode) or until no more antipattern occurrences are found (i.e., iterative mode) as depicted in.

![EPL refactoring engine](image)

**User-driven multiple refactoring sessions:** Based on EVL, they allow to execute interactive antipattern detection and refactoring sessions. In fact, after the list of detected antipattern occurrences in the performance-oriented software model is presented to the user, the EVL engine enables a number of available refactoring actions (i.e., fixes) as applicable. Each refactoring is applied to a current temporary version of the software model and, when the user stops the session, the current version is finalized and represents the session output (see [Fig.3.3-3](image)).

![EVL refactoring engine](image)

**User-driven single refactoring sessions:** Based on EWL, as an element is selected in the modeling environment, antipattern occurrences are immediately detected with respect to the selected element type. Then, the EWL engine enables the antipattern solutions, among which the user can select the one to apply to the software model, thus producing a refactored model, which represents the session output.
A subsequent element selection would trigger a new refactoring session. Due to the strong need for graphical support, these types of sessions are directly integrated with Eclipse-based Graphical Modeling Frameworks, e.g. Papyrus\(^1\) (see Fig.3.3-4).

### 3.3.2. Guidelines for PADRE:

The framework architecture is illustrated in Fig.3.3-6, which basically shows the main actors that it involves, together with its workflow.

![Fig.3.3-6: Epsilon-based approach for model refactoring](image)

The framework is centered on three performance anti-patterns detection and solution engines that provide different interactive support to the designer, and that are respectively based on EPL, EVL, and EWL. The designer selects the engine to use in order to perform refactoring sessions starting from an initial software model, i.e. \(M_0\), which conforms to UML+MARTE metamodel and is given as input to the selected engine. During a refactoring session, a number of new refactored models, i.e., \(M_1, \ldots, M_n\), can be created until a software model that satisfies performance requirements is obtained, i.e. \(M_n\).

Our framework deals with UML models made of the following diagrams: a Component Diagram that describes the software components and their Interfaces/Operations; a Deployment Diagram that describes the allocation of artifacts, corresponding to components, on platform nodes; a Use Case Diagram that describes the actors and the use cases that they can execute; a number of Sequence Diagrams, one for each use case, that describe the system behavior in terms of interactions among components.

Hence, each model \(M_i\) in Figure is required to be a UML model, as specified above, annotated with MARTE stereotypes and tags that represent the information needed to execute the Performance-oriented Model Integration step of Figure 3.3-6, that is: (i) the performance parameters required to generate and solve a performance model, and (ii) the performance indices required by antipattern detection and refactoring, and that are filled in with the outputs of the performance model solution.

\(^1\) [https://www.eclipse.org/papyrus/](https://www.eclipse.org/papyrus/)
A performance expert has to build the basic knowledge $K_{\text{UML+MARTE}}$, which is the UML+MARTE representation of performance antipatterns detection rules and refactoring actions that can be applied to remove them. Such knowledge is then used to produce detection and refactoring code in one of the considered EPSILON languages.

PADRE deals with UML+MARTE models. All the views of the system are embedded in a single model file, as it is customary among UML graphical design tools. Concerning the performance analysis of a software model, PADRE needs a multi-views representation since a performance anti-pattern involves different aspects of the system under analysis. An example of a model that PADRE can analyze is depicted in Figure 3.3-7, where the UML Component Diagram (Figure 3.3-7 (a)) shows an excerpt of the static view of the application.

The UML Deployment Diagram (Figure 3.3-7 (b)) depicts the deployment view of the system, i.e., hardware nodes and how the application artifacts are allocated on them, while Figure 3.3-7 (c) shows a sequence diagram that shall represent the scenario considered during the performance analysis.

Fig.3.3-7: An excerpt of a UML Software Model used in PADRE
3.4. CHESS

3.4.1. Methodology for CHESS

CHESS is a model driven methodology for the design, verification and implementation of high-integrity, hard real-time, safety critical systems adopting the “correctness by construction” concept. CHESS supporting toolset is implemented on top of Eclipse Papyrus UML editor and is made available as open source Eclipse-Polarsys project [3].

The figure 3.4-1 provides a summary view of the CHESS organisation referred to the WP general workflow. Actually, code generation solution, belonging to WP3, and traceability, belonging to WP4, are also shown for completeness. The CHESS methodological approach is widely described in documents available from the project site [3][4][5]. Fundamental characteristics and the main improvements, provided in the context of WP2 MegaM@Rt project, are briefly described heredown. When applicable, the instantiations on relevant use cases is also described.

Figure 3.4-1: CHESS workflow

Metamodel

CHESS comes with its own component model and modelling language, the latter implemented as UML, MARTE and SysML profile, called CHESSML. CHESSML profile restricts the set of UML entities that can be created, for consistency, avoiding redundancy and fixing semantic variation points, and provides: i) a set of stereotypes and profiles to support component-based design; ii) some MARTE stereotypes extensions to allow the specification of computation-independent real-time properties; iii) and a new set of stereotypes to support the dependability modelling.

Additional extensions and integration with other tools (e.g. OCRA, OpenCert, BVR, etc.) are provided to support the contracts modelling and analysis and to support variability modelling.
Safety critical systems: contract-based design and model checking approach

The capability to address modelling of safety critical systems, via contract-based approach, is one of the main achievements of the CHESS tool, but the MegaM@Rt’s Use Cases do not explicitly focus on this area, thus not resulting in a concrete application of the tool.

The figure 3.4-2 shows the methodological steps required to proceed with the contract-based design of a safety-critical system. Three macro activities are foreseen: the “system definition” and the “early validation” that can be iteratively executed until a viable model is designed; then the “safety analysis” provides amean to check the system behaviour, e.g. injecting faults or through fault tree technique.

Figure 3.4-2: CHESS Safety-critical System design

Requirement specification

CHESS comes with a dedicated view/package called “RequirementView”, where the SysML Requirement Diagram is applied to model the requirements. In addition, Papyrus, which is the tool on which CHESS is based, allows importing requirements from external sources [6] like excel, csv files, or
by using the ReqIF [7] (requirement interchange format), the latter supported by most of the commonly used requirement management tools, like DOORS [8].

**Component definition and requirement traceability**

The definition of the system architecture foresees the design of components in isolation, i.e. components designed out of any particular context (to exploit their reuse), or the identification (reuse) of the components from existing libraries/models. The modelling of the component is comprehensive of its boundary identification (in terms of the input/output ports) and of additional information about its possible (functional and extra-functional) behaviours and the expected behaviours of the context.

Requirements available in the CHESS/Papyrus model can then be linked to the defined components by using the standard SysML support, e.g. by allocating requirements to components using the “satisfy” relationships. In addition, the usage of Capra traceability tool is foreseen to manage traceability links between information stored in the CHESS model and information available in other artefacts. E.g. Capra allows to directly trace requirements available in external model/repository (like DOORS or ReqIF) to the CHESS architectural elements. CHESS can also exploit Capra flexibility to link a safety requirements to an assurance cases that may be defined using the OpenCert tool. This approach enables reasoning and provide assurance about the system capability to satisfy requirements. When using Capra, the traces (e.g. the satisfy relationships between system components and requirement) are stored in a dedicated traceability model.

**Requirements formalization and contract definition**

This activity has the goal of translating textual requirements allocated to system components into formal properties. Well-formed formal properties are textual expressions specified using a restricted grammar (Linear Temporal Logic) so that their semantics is formally defined in rigorous mathematical terms.

Formal properties can be further structured into contracts by assigning them to the assumption and guarantee fields of the contract.

**Early Validation**

Different strategies are conceived to support the designer evaluating the correctness and consistency of the system model. The Contracts with formal properties expressed in Linear Temporal Logic (or some expressively equivalent Bounded First-Order extension) can be automatically analysed checking their consistency. The validation is done by checking if a specific guarantee of the contract satisfies the assumption of another contract.

**Architecture and requirement refinement in CHESS**

The architecture refinement details how the different parts of the system are composed and interact in order to fulfil the system requirements. To refine the architecture, the sub requirements and subcomponents of the system must be identified. The allocation of requirements/sub-requirements to subcomponents must be managed, as well as the formalization of sub-requirements.

Information about requirements refinement is supported in CHESS by using the SysML standard modelling support, in particular by using the DeriveReqt relationships from the child to the parent requirement.
Contract refinement in CHESS

Contracts refinement must follow the architecture and requirements refinement. Indeed, a contract is explicitly linked to a system component and implicitly linked to the formalized requirements (via its assumption and guarantee properties); so, the requirements referred by contracts appearing in a refinement relationship should also have a corresponding DeriveReqt refinement traced.

Nominal and faulty behaviour definition and verification

The definition of a component can be enriched with behavioural models by means of state machine diagrams. In CHESS, state machine diagrams can be used to model the nominal and the faulty behaviour of the component. By using a subset of the standard UML support for state machine modelling, it is also possible to use model transformation to automatically generate the representation of the modelled nominal state machines, together with the components structure definition, in the SMV input language of nuXmv, and perform the automatic generation of fault tree.

The faulty behaviour is modelled in CHESS by using a dedicated dependability profile. The CHESS dependability profile enables dependability architects to model dependability information necessary to conduct dependability analysis, basically by allowing the modelling of failure modes and criticality, failure behaviours for a component in isolation (i.e. how events can lead to components failure modes) and failure modes propagation between connected components (via components ports).

CHESS can verify that the behavioural specification of the system is compliant with the contract specification. This is currently restricted to behaviours specified by state machines.

Besides the verification of contracts, other properties on the state machine can be verified. The specifications to be checked on the FSM can be expressed in temporal logics like Computation Tree Logic (CTL), Linear Temporal Logic (LTL) extended with Past Operators, and Property Specification Language (PSL) that includes CTL and LTL with Sequential Extended Regular Expressions (SERE), a variant of classical regular expressions. It is also possible to analyse quantitative characteristics of the FSM by specifying real-time CTL specifications.

Model-based and contract-based safety analysis

Specific stereotypes are defined in the CHESS dependability profile to identify the faulty behaviour for a component provided through a state machine and the error state tagged with such stereotypes.

The contract-based safety analysis identifies the component failures as the failure of its implementation in satisfying the contract. When the component is composite, its failure can be caused by the failure of one or more subcomponents and/or the failure of the subcomponent environment to satisfy the assumption.

As a result, this analysis produces a fault tree where each intermediate event represents the failure of a component or its environment and is linked to a Boolean combination of other nodes; the top-level event is the failure of the system component, while the basic events are the failures of the leaf components and the failure of the system environment.
Fault-tree generation

Considering the components definition together with the nominal and error model behaviour, the fault tree generation can be obtained by invoking the xSAP symbolic model checker through the CHESS environment. Along with the fault-tree generation, it is possible to generate the FMEA table.

Fault tree/FMEA generation starting from the designed components and associated contracts is also supported.

OpenCert integration and assurance case

When a given analysis has been executed, and the analysis results have been inspected, the artefact(s) containing the analysis results can be collected as evidence. The integration with OpenCert evidence editor allows to create evidence entities linked to the artefact(s) owning the analysis result.

Evidences can be used to support the assurance cases, for instance, to demonstrate that a given process step required by the process model has been executed, and can be referred directly in the assurance case editor.

System variability and management of product lines

Reuse of available components and management of different versions of a set of component is a challenge in industrial practice aiming to reduce the development time and maintenance costs.

Since CHESSML does not natively offer support for variability, the CHESS tool has been seamlessly integrated with the BVR component offering a complete process for variability management. The CHESS-BVR integration is a result of the AMASS project (terminated recently in March 2019), provided by the Mälardalen University with the support of Intecs. Now, the CHESS-BVR variability feature has been made available and fully integrated in the MegaM@Rt framework, thus exploitable by UC providers. The Scenario 1 of the VCE Use Case, focusing on variability modelling, is an example of possible application. The current VCE developments and experiments are based on a different solution using the Orthogonal Variability Modeling (OVM) tool already adopted in VCE. However, the CHESS-BVR solution has been shown to VCE during the last Santander Hackathon for its possible exploitation, at least, in a small application example.

The product related reuse strives for building a component once using CHESS methodology, then re-use it in different applications or products based on the feature diagrams provided by BVR. Three editors are available: i) a feature diagram associated to a component model is modelled in the VSpec editor, ii) the component configuration is performed in the Resolution editor, iii) and the placements and replacements are specified in the Realization editor. Fragment substitution removes elements within the placement and substitutes them with the elements in replacement. Therefore, the links between VSpec features and fragment substitutions need to be established. The inclusion/exclusion of particular choices for the configuration/resolution is therefore considered. In order to derive the configuration, the execute option in particular Resolution is selected. This generates the desired models that are automatically exported back to the CHESS project.

When adopting the contract-based approach, the reuse of a safe component requires the evaluation and validation of its assumption/guarantee contracts applied on the relevant Resolved Models.
3.4.2. Guidelines for CHESS

CHESS is a tool realised on top of Papyrus modeller, thus inheriting the main modelling and editing characteristics. A comprehensive CHESS use and reference guide is available at [5].

The CHESS toolset implements support for views implementation, where the views are defined by the CHESS methodology. The following figure 3.4-4 summarizes the design flow addressing the different available views.

In this section, the Scenario 1 of the TRT UC is used to present a concrete example of some relevant CHESS capabilities related to timing properties, platform definition and schedulability analysis.

The Figure 3.4-5 shows the diagrams used to model the sensor group’s component types together with the relevant interfaces and data types, created based on the provided TRT UC’s documentation.

Among the basic modelling capabilities:

- the figure 3.4-5-a) provides an example of the “Requirement View” where requirements can be modelled and can be represented using the SysML requirement diagram;
- the system view provided in figure 3.4-5-b) shows the possibility to create a SysML “Block Diagram” for a high level description of the main system characteristics;
- the Component View allows to model components, interfaces and data types using class diagrams and composite structure diagrams, as shown in figure 3.4-5-c) and 3.4-5-d).

One of the peculiarities of the CHESS component model is the distinction between the <<ComponentType>>, that describe a component by means of its ports, properties and methods, and the <<ComponentImplementation>> that represent the “realisation” of the component itself. The <<ComponentImplementation >> inherits, through the “realise” association, all the relevant entities (i.e. ports, properties and methods) from the correspondent << ComponentType >> (see figure 3.4-5-d).
CHESS provides specific profiles for domain specific modelling, among other, the space, avionic and automotive are foreseen.

To rely with avionic architecture it is necessary to take care of the “Integrated Modular Avionics” concepts (IMA), i.e. time and space partitioning. Specific stereotypes are available to characterise: partitions, processes and functions entities. MARTE profiles are also available to characterise properties like: timing aspects (e.g. execution time, rate), priorities, precedence relations. By timing point of view, two-level scheduling are foreseen: “periodic” fixed at partition level and “priority” fixed at process level.

To proceed with avionic schedulability analysis, the following actions are required:

- Select “avionics” as domain for the model (CHESS tab in the model properties) to enable “ARINC” profiles
- It can be noted how the ports can be stereotyped as <<ClientServerPort>> and characterised, using MARTE profile, as “required” or “provided” ports, associated to their relevant interface.
- For each relevant object (i.e. class in the class diagram) add one <<ARINCProcess>> operation and stereotype the other relevant operations as <<ARINCFunction>>
  - Complete “OperationGroup” field for <<ARINCProcess>> listing the operation (i.e. ARINCfunctions) belonging to the process
  - Complete “RateDivider” and “Followed-by” fields for each <<ARINCFunction>>, e.g. defining precedence properties among functions using the “followed-by” field
- Create the functional partition by means of:
- Create a SW System component (i.e. FlyanceSystem) stereotyped as <<CHGaResourcePlatform>>
- Create a partition component for each relevant partition (e.g. DAL_B_Partition, DAL_C_Partition) stereotyped as <<FunctionalPartition>>

- Instantiate the SW system components (instance view) by invoking the “Build Instance” command. The actual SwSystem Instance Specification is created with its defined functional partitions.

Additional real time properties, relevant for the following timing analysis, can be set for the ARINCProcess and the ARINCFuncton operations for a given ARINComponentImpl component instance, according to the CHESS profile for ARINC; in particular real time properties are provided through the CHRtSpecification stereotype. For TRT UC the relevant annotations has been created on the instance view for each process and function. The <<CHRtSpecification>> stereotype allows to define, among other, the Worst Case Execution Time (WCET), the priority, the periodicity and the phase of the activation.

- Create the HW platform deployment
  - Create a HW System component (i.e. HFlyanceSystem) stereotyped as <<CHGaResourcePlatform>>
Create the HW components (e.g., one or multiple CPUs with single or multiple cores). <<CH_HwProcessor>>, <<CH_HwBus>> and <<CH_HwComputingResource>> stereotypes are available to tune the HW component characterisation.

- Instantiate the HW components by invoking the “Build Instance” command on the HW resource platform.
  - IMA Partition Support facilities
    - Allocate the SW components instances (i.e., the components stereotyped as <<ComponentImplementation>> instantiated by the Build Instance command) to the relevant partition
    - Assign partitions to relevant CPU’s cores
    - Finally, generate partition schedule in order to compute the MIF, MAF, Activation windows for all cores and each partition

If the partition schedule completes successfully, the “schedulability analysis” can be invoked from CHESS menu. The “Analysis context” is returned in the form of a table (see figure 3.4-6) that reports, for each process and for each related function: the given timing parameters, the resulting response time and a flag YES/NO that confirm if the function is schedulable.

![Figure 3.4-6: Analysis context table](image)

As briefly shown in this example, CHESS provides several facilities to support, and when possible, to automate the designing steps. It is an open tool, still evolving, that allows to easily extend its capabilities by integrating new functionalities or interoperating with external tools (like Capra, OpenCert, etc.).

### 3.5. Collabro

Collabro is an approach to make language development processes more participative, meaning that both developers and users of the language can collaborate together to create and evolve it. Collabro supports both the collaborative definition of the abstract (i.e., metamodel) and concrete (i.e., notation) syntaxes for your DSL by providing a collaborative environment enabling the discussion. With Collabro, anyone has the chance to request changes, propose solutions and give an opinion (and vote) about those from others. This discussion enriches the language definition significantly and ensures that the end result satisfies as much as possible the expectations of the end-users. Moreover, since all the discussions are recorded, we can always trace back any language element to the people and motivations that originated it in the first place.
3.5.1. Collaboro Methodology: Making DSML Development Collaborative

We propose a collaborative approach to develop DSMLs following the process summarized in Figs. Figure 3.5-1 and Figure 3.5-2. Figure 3.5-1 summarizes the process of generating the initial DSML that will serve as the basis for the application of Collaboro. Roughly speaking, the process is as follows. Once there is an agreement to create the language, developers get the requirements from the end-users to create a preliminary version of the language to kickstart the actual collaboration process. This first version should include at least a partial abstract syntax but typically also includes a first concrete syntax draft. An initial set of sample models are also defined by the developers to facilitate an example-based discussion, usually easier for non-technical users. Sample models are rendered according to the current concrete syntax definition. It is worth noting that the rendering is done on-the-fly without the burden of generating the DSML tooling since we are just showing the snapshots of the models to discuss the notation, not actually providing at this point a full modeling environment.

![Collaboro Methodology Diagram](image)

**Figure 3.5-1: Simplified process of creating a new DSML**

Now the community starts working together in order to shape the language (Figure 3.5-2). Community members can propose ideas or changes to the DSML, e.g., they can ask for modifications on how some concepts should be represented (both at the abstract and concrete syntax levels). These change proposals are shared in the community, who can also suggest and discuss how to improve the change proposals themselves. All community members can also suggest solutions for the requested changes and give their opinion on the solutions presented by others. At any time, rendering the sample models with the latest proposals helps members to have an idea of how a given proposal will evolve the language (if accepted). During this step, a recommender system also checks the current DSML definition to spot possible issues according to quality metrics for DSMLs. If the recommender system detects possible improvements, it will create new proposals that should also be discussed by the community. All these proposals and solutions are eventually accepted or rejected. Acceptance/rejection depends on whether the community reaches an agreement regarding the proposal/solution. For that, community members can vote. A decision engine then takes these votes into account to calculate which collaborations are accepted/rejected by the community. The engine could follow an automatic process but a specific role of community manager could also be assigned to a member/s to consolidate the proposals and get a consensus on conflicting opinions (e.g., when there is no agreement between technical and business considerations). Once an agreement is reached, the contents of the solution are incorporated into the language, thus creating a new version. The process keeps iterating until no more changes are proposed. Note that these changes on the language may also have an impact on the model examples which may need to be updated to comply with the new language definition.
At the end of the collaboration, the final DSML definition is used as a starting point to implement a full-fledged DSML tooling with the confidence that it has been validated by the community (e.g., transforming/importing the DSML definition into language workbenches like Xtext or Graphical Modeling Framework (GMF)). Note that even when the language does not comply with commonly applied quality patterns, developers can be sure that it at least fulfills the end-users’ needs. Moreover, all aspects of the collaboration are recorded, thus keeping track of every interaction and change performed in the language. Thus, at any moment, this traceability information can be queried (e.g., using standard Object Constraint Language (OCL) expressions) to discover the rationale behind the elements of the language (e.g., the argumentation provided for its acceptance).

Usage guidelines for Collaboro: Using Collaboro to build a DSML collaboratively

Inputs:
- An .ecore model with the abstract syntax of a DSML

Outputs:
- An updated .ecore model, with the new abstract syntax of a DSML
- A .notation model, with a mockup of the abstract syntax of a DSML
- A .history model, capturing all the interactions among the involved stakeholders.

Running example: the Baggage Claim DSML
As a running example, imagine the development of a DSML to facilitate the planning of the baggage claim service in airports. Let's assume that the airport baggage service needs to specify every morning the full daily assignment of flights to baggage claim conveyors so that operators can know well in advance how to configure the actual baggage system. For that, developers and domain experts (i.e., baggage managers) collaborate to define a DSML that serves this purpose.

Typically, domain experts are only involved at the very beginning and the very end of the DSML development process. Assuming this is also the case for our example, during the analysis phase, developers would study the domain with the help of the baggage managers and decide that the DSML should include concepts such as Flight, Bag and Conveyors to organize the baggage delivery as shown in Fig.3.5-3.

![Fig.3.5-3: Abstract syntax of the Baggage Claim DSML](image)

At this point, developers may want to involve end users in the development process by using Collaboro. For example, an end user (End-User 1) may decide that a textual syntax is required (Fig.3.5-3).

Developers would then propose a possible textual syntax (see Fig.3.5-4, right), and as a result, Collaboro provides a sample of what that possible textual syntax would look like (Fig.3.5-4, left). Based on the samples, a voting phase is started to determine if the proposal is accepted or not. The steps are summarized in Fig.3.5-5, first, each stakeholder uses his/her own interface to vote, and the votes appear in the Collaboration View (A); second, the decision engine is launched through the Collaboro menu (B); the decision engine analyses the votes, and according the quorum criteria defined, determines which solutions are accepted (C); and finally, the accepted proposals are marked as such using a green mark in the Version View (D).
Fig. 3.5-4: Change proposal to create a textual concrete syntax

Fig. 3.5-5: Proposal for a textual concrete syntax
Following this process, the first round of collaboration would create the abstract and concrete syntax of the DSML, thus coming up with a textual DSML like, for instance, the one shown in Fig. 3.5.4.

As aforementioned, typically a development process would end here: after the initial requirements elicitation and once the tooling has been defined, no further improvements are possible (or their inclusion would be difficult). However, to better illustrate how Collaboro would support further changes to the DSML, we will explain the process shown in Fig. 3.5.6. We also provide screenshots of how each step in Fig. 3.5.6 would look like in Collaboro.

After developers completed a first version of the language, the collaboration continues with a community member (End-User 1) who detects the need of expressing the capacity of the conveyors. At this point, the community has the chance to discuss the best way to adapt the language to support this new information. The member that identified the problem would create a change proposal with that aim, and if the change is deemed as important by the community, other members could propose a solution/s to adapt the language. As an example, Figure 3.5.7 graphically depicts a possible collaboration scenario assuming a small community of one end-user and two developers. Each collaboration is represented as a bubble, and each step has been numbered.

In Figure 3.5.7, End-User 1 proposes a language change (Figure 3.5.7 step 1 and Fig. 3.5.8 A), which is accepted by the community (Figure 3.5.7 step 2, and Fig. 3.5.8 B and C), and then Developer 1 specifies a solution (Figure 3.5.7 step 3, Fig. 3.5.9). The solution is rejected by End-User 1 (Fig. 3.6.10), including also the explanation of the rejection (Figure 3.5.7 step 4, Fig. 3.6.11). As the
rejection is accepted (Figure 3.5-7 step 5), the Developer 1 redefines the solution (Fig. 3.6-12), which is eventually voted and accepted (Figure 3.5-7 step 6, Fig. 3.6-13) and the changes are then incorporated into the language (Figs. 3.6-14 and Fig. 3.6-15).

Figure 3.5-7: Refinement of the Baggage Claim DSML

Fig. 3.5-8: Change proposal for representing capacity
Fig. 3.6-9: Possible solution to represent the capacity

Fig. 3.6-10: Rejection of the first possible solution
Fig. 3.6-11: Rationale behind the solution rejection.

The capacity of a conveyor should be indicated in terms of the number of pieces it can support.

Votes Agree: Developer 1, Developer 2
Votes Disagree:

Fig.3.6-12: New solution proposal for the conveyor capacity
Fig. 3.6-13: Voting accepting the second solution proposal

Fig. 3.6-14: Report of the modifications done on the abstract syntax of the DSML

Fig. 3.6-15: Result of the modifications in the abstract syntax of the DSML
Fig. 3.6-16: Proposal for updating the concrete syntax of the DSML

3.6. **EMFtoCSP**

3.6.1. **EMFtoCSP methodology**

![Diagram of EMFtoCSP Verification Process]

Figure 3.6-1: EMFtoCSP Verification Process
EMFtoCSP is a lightweight tool for the verification and validation of static models. Input models can either be EMF or UML class diagrams, annotated with integrity constraints written in the Object Constraint Language.

The tool can be used to solve different tasks concerning the correctness of these models:

- Detect if all the integrity constraints in the model can be satisfied
- Detect if there are any redundant integrity constraints
- Detect if there are contradictory integrity constraints
- Compute sample valid instances of the model, e.g. for validation or testing
- Compute sample instances satisfying a given property

EMFtoCSP uses the hidden formal methods paradigm: the user does not need any knowledge about the reasoning procedure. Internally, the tool transforms the entire model, the integrity constraints and the property being checked into a Constraint Satisfaction Problem (CSP). Each solution of the CSP is an instance of the model that constitutes an example (or counterexample) of the property under analysis. For the analysis EMFtoCSP relies on the ECLiPSe\(^2\) toolkit, a constraint logic programming solver, to define and solve CSPs, and provides solutions graphically as an object diagram.

For pragmatic purposes, EMFtoCSP applies a bounded verification approach: solutions are only considered within a finite search space that is defined by the user. Although bounded verification methods are not complete, they rely on an observation called “small scope hypothesis”, claiming that a majority of defects can be identified by small examples / counterexamples. For this reason, users need to define the finite search space that will be used in the analysis and to understand that this analysis is inconclusive outside of the bounds they have defined.

The complete description of how verification problems on models can be formalized as CSPs can be found in the paper “On the verification of UML/OCL class diagrams using constraint programming” [9]. A discussion of how the performance of this analysis can be improved (without loss of precision) by automatically reducing the size of the search space is provided in “Smart bound selection for the Verification of UML/OCL class diagrams” [10].

Figure 3.6-1 summarizes a typical session using EMFtoCSP. First, the designer needs to define the model under analysis (including its OCL constraints) and define the search space in terms of the domains of attributes (the list of accepted values) and the cardinalities of classes (maximum number of instances) and associations (maximum number of links). The designer can also select the property that needs to be checked, e.g. satisfiability of the model or detecting whether a given constraint is redundant. After that, the process is completely automatic: the tool transforms the problem into a CSP in the Prolog-based notation used by ECLiPSe (.ecl) and executes the solver to find a solution; if there are solutions, they are presented graphically to the designer as an object diagram. See next subsection for an instantiation of this process.

\(^2\) [http://eclipseclp.org/](http://eclipseclp.org/)
Figure 3.6-2: Configuring EMFtoCSP

Usage guidelines for EMFtoCSP

Inputs:

Either an .ecore or a .uml model, plus an (optional) .ocl file.

Bounds of the search space.

Set of properties to check.

Outputs:

Either an example valid instance, or the .ecl code file used by the solver.

We detail below all the steps a user must follow in order to exercise the tool.

EMFToCSP Configuration

Before start using EMFToCSP the tool has to be configured by filling the fields in the EMFToCSP preference page.

The preferences are accessible from the Windows > Preferences menu, under the EMFToCSP category on the left pane. The user must provide the ECLiPSe constraint solver and GraphViz paths, which needed to run EMToCSP. Once configured, the window can be closed.
Using EMFtoCSP

Launching EMFtoCSP is pretty straightforward. From the Package Explorer window, the validation can be launched by right clicking on the model to validate. EMFtoCSP is launched by choosing the option "Validate model..." from the popup menu.

Setting OCL constraints

In the first window of the wizard, the user can select an .ocl file with the model OCL constraints. If the user does not want to provide an external OCL file or the OCL constraints are embedded in the model,
this step can be skipped by clicking on the *Skip selection* checkbox. Once this step has been done (either by selecting a file or by checking the skip selection checkbox) finished, the user can click *Next*.

![Figure 3.6-4: Running EMFtoCSP, step 1](image)

**Setting domains and cardinalities**

Once you the user has finished with the OCL settings, she must provide the domains and cardinalities for the model's elements. This way, the user will set the boundaries of the search space where the solver will try to find a valid instance of the selected model. Once the boundaries have been set, she can proceed by clicking *Next*. 
### Selection of the properties you want to check

Once the search space has been defined, the user must select the properties she wishes to validate among the ones offered by the tool. Once the user has finished, she can click Next.

![Figure 3.6-5: Running EMFtoCSP, step 2.](image1)

![Figure 3.6-6: Running EMFtoCSP, step 3](image2)
Setting where to store the validation's results

The last step before launching the validation process is to select where the results will be stored. The user has to choose a folder and then click "Finish".

![Figure 3.6-7: Running EMFtoCSP, step 4](image)

The verification process will take place after clicking on the *Finish* button. If the verification process finds an instance of the model that satisfies all the restrictions (i.e., a valid instance), the tool will show the following message:

![Figure 3.6-8: Message shown when EMFtoCSP finds a valid instance for the input model](image)

The user will find an image with the description of the instance along with the generated ECL code file used by the solver in the result location.
If the verification process yields no results, then the tool will show the message shown in Figure 3.6-10 and it will store the generated ECL code file used by the solver in the result location.

3.7. Xamber

3.7.1. Methodology for Xamber:

Xamber is used in Scenario 2 of 01_TRT - Flight Management System (FMS) use case. Xamber generates the scheduling plan and performance verification. This task has been done traditionally by hand. However, there are temporal models impossible to generate and verify by hand. Xamber automates this process using existing schedulability algorithms adapted to the temporal model of a partition system. To do this, a set of transformation rules have been established to fill the semantic gap between the design model and the performance verification model. The requirements and contribution to KPIs can be found in D5.5.

The performance verification of FMS implies both the scheduling analysis of the temporal model and the estimation of the worst case latencies of functional chains.

Functional chains are not part of the Xamber model so they have been added in the tool (called data flows). A data flow is a list of steps with precedence relations. Once the functional chain is defined, Xamber calculates the response latency of each activation of the data flow. This information is important to know if the functional chain meet the temporal requirements defined.
3.7.2. Guidelines for Xamber:

Steps:

1. Time4Sys: All performance relevant data and properties are extracted from the Capella design model to generate a corresponding Time4Sys model

2. Xamber model generated from Time4Sys model

3. Xamber generates:
   a. Scheduling plan: Verification of periodic activities
   b. Calculation of functional chain latencies: New feature implemented in Xamber
As output, Xamber generates a correct configuration file for XtratuM [11]. This implies a schedulable temporal plan and, as a new feature in the framework of this project, the latencies of the functional chains of the model.

3.8. XPM

3.8.1. Methodology for XPM:

XPM (XtratuM Project Manager) is an application to automatically generate the deployment for a target partitioned system based on the XtratuM hypervisor. The main goal is to provide an efficient way to develop real-time and embedded applications with minimal intrusion on the target system.

The deployment generated from a XtratuM configuration comprises the required files (makefiles, customisation files, etc.) for building the whole system according to the execution environments (XAL, PaRTiKle [12] or LithOS [13]) of the partitions.

The workflow assumes that the model is initially created in Xamber [14], by specifying all the available information to create a partial configuration file. This partial config file is the input to XPM and is used to create a template with all the tree structure of the XM project and the part of the source code related to the creation of partitions and communication channels. See D3.6 for an explanation about how this code is auto generated.

3.8.2. Guidelines for XPM:

Inputs: XtratuM configuration file (XMCF) generated by Xamber. This file may not be complete. In this case, the missing information will be populated by XPM.
Manual steps:

- If the XtratuM configuration file is complete, then the user will have to select the path of XtratuM and the execution environments used as well as the toolchain.
- Otherwise, the XMCF will have to be completed.

Outputs: The tool produces the deployment of the system which consists of:

- A directory for each partition with the following files:
  - Source code in C (partially generated by XPM)
  - Configuration file (for LithOS execution environment)
  - Makefile (of the partition)
- XtratuM configuration file (final version if not completed in Xamber)
- Makefile (general)

3.9. Papyrus extension for AOM

3.9.1. Methodology for Papyrus extension

This papyrus extension supports the modeling of cross-cutting concerns (such as logging, verification, etc.) as aspects within a model representation of a target system. In this way, these concerns are modeled in isolation from the target system model so that they are not entangled with it.

This Papyrus extension has been applied to one of the Ikerlan use cases, for injecting logging and message verification of the IOT Gateway component of their Skyline system. This tool satisfies the IKER_04 requirement. This tool contributes to KPI1.1 and KPI1.3.

In the context of MegaM@RT2 we have developed a Papyrus extension to support the modeling of aspects that manage the injection of cross-cutting advices within the pointcuts defined in the target system model. This Papyrus extension contains a UML profile for Aspect Oriented Modeling (AOM) that supports the representation of AspectJ directives.

In this context, we facilitate the modeling of Aspect Oriented Programming (AOP) based systems, using the normative UML metamodel.

The implemented AOM profile for UML is integrated within the Papyrus support for UML profiling.

3.9.2. Guidelines for Papyrus extension

Papyrus support for AOM requires, as input, an existing UML model of the target system.

The way this Papyrus extension for AOM works is the following:

1. The user opens the existing UML model of the target system using Papyrus; he/she also opens the Papyrus perspective.
2. Then, the user creates a class diagram for defining his/her aspects.
3. Next, the Papyrus profile for AOM needs to be applied to the model. For that, the user clicks on the model background, and opens the Profile tab of the properties view.

   a. The user clicks on the icon to apply a registered Profile and select the AspectJ profile from the list in the popping dialog (Figure 3.9-1).

![Figure 3.9-1: Selection of AspecJ profile](image)

4. Then, the user can define aspects as UML classes, following the standard Papyrus procedure to model UML classes. Similarly it can create class properties to define pointcuts and operations to define advices. Once created, aspect classes needs to be annotated with the `AOM::Aspect stereotype`. Similarly, pointcut properties must be annotated with any stereotype that subclasses the `AOM::Pointcut stereotype`. Advices must be annotated with the `AOM::Advice`.

   a. To apply an AOM profile stereotype, the user must select the target model element (e.g. a class, a class property, a class operation) in the Papyrus model canvas, and then open the Profile type in the Properties view.

   b. Then, the user clicks on the button, and select an applicable stereotype from the leftmost panel in the popping wizard (Figure 3.9-2), by clicking on the right-pointing arrow (this sends the selected stereotype to the rightmost panel)
c. Additionally, the user may need to fill in some of the slots of the applied stereotype (Figure 3.9-4)

Figure 3.9-4 Slots edition for Advice stereotype

5. Once the aspect model is complete, the user should save it into the file system. An example of AOM model for the Ikerlan IoT Gateway for the Skyline system is shown in Figure 3.9-5.

After using the Papyrus profile for AOM with AspectJ, the user produces an aspect model, which, in turn, can be used to generate the associated AspectJ aspect classes (serialized in the filesystem) using MegaM@Rt2 Papyrus extension for AOM code generation (see [D3.6] for more details).

Figure 3.9-5: AOM model for the Ikerlan IoT Gateway component
3.10. Moka extension for logging

3.10.1. Methodology for Moka extension for logging

Generating fUML compliant models for Moka simulation is a cumbersome task because the fUML specification is quite complex, so that designing fUML executable models, free of bugs, is a hard task. Moreover, current Papyrus/Moka validation support is quite limited; there are many situations were fUML models are failing when executed, but the previous validation does not detect such issues.

Extended Moka fUML validation have been used for the verification of the Ikerlan model for their IoT Gateway component of their Skyline system.

This tool satisfies the IKER_04 requirement. This tool contributes to the KPI1.1 and K1.4.

The existing EMF validation support for UML, provided by the UML2 tools (integrated within Papyrus) and by Moka, has been extended by creating additional OCL constraints. These new constraints have been registered within the EMF validation system. New OCL constraints could be defined for additional verification by extending the existing ones in the moka.ocl file included within the plugin.

3.10.2. Guidelines for Moka extension:

This Moka extended verification framework can be applied to any existing fUML compliant model.

The process for validating a fUML compliant model is as follows:

1. Select the created project in the Project Explorer. Open the model into the Papyrus editor by double-clicking on it. In the Papyrus view, welcome tab, click on an existing model diagram to open it.
2. Right click on the Papyrus model graphical view. Select Validation/Validate Model (Figure 18.1).
3. The results of the Moka validation are shown in the *Model Validation View*. New validation constraints added by MegaM@Rt are prefixed with "Moka_" (Figure 18.2)

4. Additionally, verification issues are reported within the visual diagram, connected to the affected UML element (Figure 18.3)
VeriATL is a tool for allowing the verification of several kinds of functional requirements (e.g. syntactic/semantic correctness) over existing model-to-model transformations defined using the Eclipse ATL model transformation language. VeriATL provides support for (transformation) Model Verification and Validation activities based on the use of Formal Methods, as shown in the process excerpt displayed right after.

Unfortunately, at the end of the project, the tool has not been used in practice in the context of any use case. This is mostly due to the fact that a really few use case scenarios implied the specification and deployment of model-to-model transformations written in the ATL language, and that none of them actually required to formally verify these ATL transformations. However, we still believe VeriATL proposes interesting formal verification capabilities to be added to the MegaM@Rt2 toolbox: it could
be potentially used in the future (by partners both inside and outside the MegaM@Rt2 consortium) in any scenario and/or technical solution where ATL transformations are involved and need to be verified.

In the contract-based development of model transformations, continuous deductive verification helps the transformation developer in early bug detection. However, because of the execution performance of current verification systems, re-verifying from scratch after a change has been made would introduce impractical delays. VeriATL addresses this problem by proposing a fine-grained incremental verification approach and applying it to the ATL model-transformation language. As it can be seen in the figure hereafter, VeriATL takes as inputs an ATL model-to-model transformation (as well as the corresponding metamodels) and a set of OCL pre/post conditions. The VeriATL approach is based on decomposing each contract into subgoals, and caching the subgoal verification results. At each change, the semantics of relational model transformation is exploited to determine whether a cached verification result may be impacted. Consequently, less postconditions/sub-goals need to be re-verified. When a change forces the re-verification of a postcondition, the cached verification results of sub-goals are used to construct a simplified version of the postcondition to verify. As outputs, and in case the input ATL transformation is not finally verified as correct, problematic transformation scenarios are provided back to the transformation developers. These can be used for further transformation analysis and fixing.

Figure 3.11-1: Overview of the VeriATL approach and process

The soundness of the VeriATL approach has been proven and has shown its effectiveness by continuous and extensive evaluations:

- “Slicing ATL Model Transformations for Scalable Deductive Verification and Fault Localization” - [https://hal.inria.fr/hal-01763410](https://hal.inria.fr/hal-01763410)
- “A Deductive Approach for Fault Localization in ATL Model Transformations” - [https://hal.archives-ouvertes.fr/hal-01435977](https://hal.archives-ouvertes.fr/hal-01435977)
- “Incremental Deductive Verification for Relational Model Transformations” - [https://hal.archives-ouvertes.fr/hal-01435974](https://hal.archives-ouvertes.fr/hal-01435974)

In practice, VeriATL is built on top of a state-of-the-art incremental verification tool for imperative languages, named Boogie. Through evaluation, it has been shown that:

- VeriATL speeds up re-verifications by at least 70%.
- Because of using fine-grained verification (notably enabling program slicing and impact analysis), VeriATL is consistently faster than the state-of-the-art incremental verification tool by 16% to 45%.
- Using fine-grained verification allows VeriATL to scale to verify large model transformations.
3.11.2. Guidelines for VeriATL:

As introduced before, VeriATL takes as inputs an ATL model-to-model transformation (already existing or under development) as well as the corresponding source and target metamodel(s).

To give a simple practical example of usage of VeriATL (and also as it has not been used in the contest of a particular MegaM@Rt2 use case), we consider here a model-to-model transformation that transforms hierarchical state machine (HSM) models to flattened state machine (FSM) models, namely the HSM2FSM transformation. Both models conform to the same simplified state machine metamodel, cf. the Figure right after.

![Figure 3.11-2: The hierarchical and flattened state machine metamodel](image)

For clarity, classifiers in the two metamodels are distinguished by the HSM and FSM prefix. In detail, a named StateMachine contains a set of labelled Transitions and named AbstractStates. Each AbstractState has a concrete type, which is either RegularState, InitialState or CompositeState. A Transition links a source to a target AbstractState. Moreover, CompositeStates are only allowed in the models of HSM, and optionally contain a set of AbstractStates.

![Figure 3.11-3: Examples of HSM (left) and FSM (right, obtained by flattening the HSM). In both cases, abstract syntax (top) and concrete graphical syntax (bottom)](image)

The figure right before depicts a HSM model that includes a composite state and demonstrates how the HSM2FSM transformation is expected to flatten it into an FSM: (a) composite states need to be removed, the initial state within needs to become a regular state, and all the other states need to be
preserved; (b) transitions targeting a composite state need to redirect to the initial state of such composite state, transitions outgoing from a composite state need to be duplicated for the states within such composite state, and all the other transitions need to be preserved.

In addition, a set of related OCL pre and post conditions must also be provided to the tool. In some cases, these can be already available from the concerned transformation and its source and target metamodels. In other cases, they have to be specified/added manually to the transformation and metamodels, e.g. by the transformation developers.

**Specifying OCL contracts.** For this simple example, we consider a contract-based development scenario where the developer first specifies correctness conditions for the to-be-developed ATL transformation by using OCL contracts. For example, let us consider the contract shown in the following listing.

```oclausage
context HSM!Transition inv Pre1:
  HSM!Transition.allInstances()->forAll(t | not t.source.oclIsUndefined())

context FSM!Transition inv Post1:
  FSM!Transition.allInstances()->forAll(t | not t.source.oclIsUndefined())
```

**Listing 3.11.4: The OCL contracts for HSM and FSM**

The precondition `Pre1` specifies that in the input model, each `Transition` has at least one `source`. The postcondition `Post1` specifies that in the output model, each `Transition` has at least one `source`. While these pre-/post-conditions are generic well-formedness properties for state machines, the user could specify transformation-specific properties in the same way. For instance, the complete version of this use case also contains the following transformation-specific contract: if states have unique names within any source model, states will have unique names also in the generated target model. In general, there are no restrictions on what kind of correctness conditions could be expressed, as long as they are expressed in the subset of OCL currently supported by VeriATL (cf. the tool documentation).

```oclausage
module HSM2FSM;
create OUT : FSM from IN : HSM;

rule SM2SM {
  from sm1 : HSM!StateMachine
  to sm2 : FSM!StateMachine
  ( name <- sm1.name )
}

rule RS2RS {
  from rs1 : HSM!RegularState
  to rs2 : FSM!RegularState
  ( stateMachine <- rs1.stateMachine,
  name <- rs1.name )
}

rule IS2RS {
  from is1 : HSM!InitialState
  ( not is1.compositeStateMachine.ocIsUndefined() )
  to rs : FSM!RegularState
  ( stateMachine <- is1.stateMachine,
  name <- is1.name )
}

22  -- mapping each transition between two non-composite states
23  rule T2TA { ... }
24  
25  -- mapping each transition whose source is a composite state
26  rule T2TB { ... }
27  
28  -- mapping each transition whose target is a composite state
29  rule T2TC {
30  from tl : HSM!Transition,
31  src : HSM!AbstractStateMachine,
32  trg : HSM!CompositeStateMachine,
33  c : HSM!InitialState
34  ( tl.source = src and tl.target = trg
35  and c.compositeStateMachine = trg
36  and not src.ocIsTypeOf(HSM!CompositeStateMachine))
37  to t2 : FSM!Transition
38  ( label <- tl.label,
39  stateMachine <- tl.stateMachine,
40  source <- src,
41  target <- c)
```

3

https://github.com/veriatl/VeriATL/#fine-grained-incrementality-for-deductive-verification-of-model-transformations-online
Developing/Refining the ATL Transformation. Then, the developer implements/refines the ATL transformation HSM2FSM (a snippet is shown in the listing right before). The transformation is defined via a list of ATL matched rules in a mapping style. The first rule maps each StateMachine element to the output model (SM2SM). Then, we have two rules to transform AbstractStates: regular states are preserved (RS2RS), initial states are transformed into regular states when they are within a composite state (IS2RS). Notice here that initial states are deliberately transformed partially to demonstrate our problem, i.e., we miss a rule that specifies how to transform initial states when they are not within a composite state. The remaining three rules are responsible for mapping the Transitions of the input state machine.

As outputs, and in case the input ATL transformation is not finally verified as correct, problematic transformation scenarios are automatically provided back to the transformation developers. These can be used for further transformation analysis and fixing in order to ultimately provide a correct version of the ATL transformation. More details are given in what follows.

Formally verifying the ATL Transformation by VeriATL. The source and target EMF metamodels and OCL contracts combined with the developed ATL transformation form a verification condition (VC) which can be used to verify the correctness of the ATL transformation for all possible inputs. The VC semantically means that, assuming the axiomatic semantics of the involved EMF metamodels (MM) and OCL preconditions (Pre), by executing the developed ATL transformation (Exec), the specified OCL postcondition has to hold (Post). Specifically, the VeriATL system describes in Boogie what correctness means for the ATL language in terms of structural VCs. Then, VeriATL delegates the task of interacting with Z3 for proving these VCs to Boogie. In particular, VeriATL encodes: 1) MM using axiomatized Boogie constants to capture the semantics of metamodel classifiers and structural features, 2) Pre and Post using first-order logic Boogie assumption and assertion statements respectively to capture the pre-/post-conditions of MTs, 3) Exec using Boogie procedures to capture the matching and applying semantics of ATL MTs.

Listing 3.11-6: The problematic transformation scenario of the HSM2FSM transformation w.r.t. Post1

```
context HSM!Transition inv Pre1: ...
3 rule RS2RS { ... }
4 rule IS2RS { ... }
5 rule T2TC { ... }

context FSM!Transition inv Post1_sub:
8 *hypothesis* var t0
9 *hypothesis* FSM!Transition.allInstances() ->includes(t0)
10 *hypothesis* genBy(t0,T2TC)
11 *hypothesis* t0.source.oclIsUndefined()
12 *hypothesis* not (genBy(t0.source,RS2RS) or genBy(t0.source,IS2RS))
13 *goal* false
```

Debugging. In our example, VeriATL successfully reports that the OCL postcondition Post1 is not verified by the model-to-model transformation from the previous listing. This means that the transformation does not guarantee that each Transition has at least one source in the output model. Without any capability of fault localization, the developer would need to manually inspect the full transformation and contracts to understand that the transformation is incorrect because of the absence
of an ATL rule to transform InitialStates that are not within a CompositeState. To address this, VeriATL proposes a fault localization approach that automatically presents users with the information in the listing shown right before. The output includes: (a) the slice of the MT code containing the bug (that in this case involves only three rules), (b) a set of debugging clues, deduced from the original postcondition (in this case pointing to the fact that T2TC can generate transitions without source). This information is a valuable help in identifying the cause of the bug.

**Scalability.** While for illustrative purposes we consider here a very small transformation, it is not difficult to extend it to a realistically sized scenario. For instance we can imagine this transformation excerpt to be part (up to renaming) of a refactoring transformation for the full UML metamodel (e.g. including statecharts, but also class diagrams, sequence diagrams, activity diagrams etc.). Since the UML v2.5 metamodel contains 194 concrete classifiers (plus 70 abstract classifiers), even the basic task of simply copying all the elements not involved in the refactoring would require at least 194 rules. Such large transformation would need to be verified against the full set of UML invariants, that describe the well-formedness of UML artifacts according to the specification. While the initial version of VeriATL had been successfully used for contract-based development of smaller transformations, we showed that it would need hours to verify a refactoring on the full UML against 50 invariants. To address this issue, we designed an upgraded scalable version of VeriATL aiming at 1) reducing the verification complexity/time of each postcondition and 2) grouping postconditions that have high probability of sharing proofs when verified in a single verification task. Thanks to these techniques the verification time of our use case in UML refactoring is reduced by about 79%.

### 3.12. HepsyCode

#### 3.12.1. Methodology for Hepsycode:

Hepsycode (acronym of HW/SW CO-DEsign of HEterogeneous PARallel dedicated SYstems) is a methodology, framework and tool able to help designers during the whole design flow of embedded parallel dedicated systems. Hepsycode defines different level of abstraction, fully integrated inside the MegaM@Rt framework. At System Engineering level, starting from System Requirements (both Functional and Non-Functional), it is possible to define System Behavior and Structural Models in order to refine Architectural Design and Detail Design and produce Verification and Validation Models (as shown in Fig. 2.1).

Respect to Architectural Design activities (Fig. 2.3), Hepsycode offers Functional Analysis using SystemC code, and introduces Architectural Component Modeling activities in terms of possible technology alternatives (called Technology Library) that are used as input for Design Space Exploration (DSE) steps.

Respect to Detail Design activities (Fig. 2.4), Hepsycode offers an HW/SW Co-Design methodology, a framework and corresponding tool that are described in more details in several MegaM@Rt deliverables (i.e., D2.3, D2.4, D2.5), conference and journal papers [15, 16, 17], and official website (www.hepsycode.com).

Finally, in the Model Verification & Validation activities (Fig. 2.4) Hepsycode offers Simulation, Performance Analysis and Schedulability Analysis (by means of SystemC co-simulators), and takes as input Functional Model, Architectural Model (results of DSE steps) as well as Functional/Non-Functional Requirements.

Hepsycode is thought to be applied to UC 01_TRT. HEPSYCODE_10 overlap the TRT_05 UC requirement (The obtained timing performance results should be integrated in the model of the Flight
Management System application of the TRT case study). More details have been explained in D2.5 (Section §4.2.1 and §4.2.2), and will be reported in the WP5 – D5.3 (Integration framework approach, considering the possible tool interoperability). Hepsycode contributed to KPI 1.1 (Reduction of design time/design effort in the range of by design artefacts reuse), KPI 1.2 (Improvement of the time required for identification of design problems), KPI 1.3 (Reduction of time/efforts for requirements validation) and to the Academic KPI group.

Hepsycode solves the historical HW/SW Co-Design partitioning problem offering tools features, functionalities and methods able to drive designer from first input to the final implementation, avoiding system deviation or design errors. All the Hepsycode functions have been implemented using Eclipse MDE technologies, the SystemC custom simulator and evolutionary algorithms for partitioning activities, all integrated into an automatic framework. More details will be explained in the next paragraph, while it is possible to access to several online documentations, using the main Hepsycode and the MegaM@Rt2 website.

3.12.2. Guidelines for Hepsycode:

The design activity in HEPSYCODE starts with a high abstraction level modeling language (called HEPSYCODE Modeling Language, HML). The initial HML model is then transformed into an executable SystemC model based on Communicating Sequential Processes (CSP) Model of Computation (MoC). HEPSYCODE defines two system-level models, the System Behavior Model (SBM) and the Technology Library (TL). The former is realized from the initial HML. The latter includes all necessary and low-level hardware architectural details (w.r.t. processing units PUs, memory units, MUs, communication units, CUs). Such models are supported by some C++ libraries that extend the standard SystemC library in order to implement CSP.

The Hepsycode System-Level Modeling Language describes application as a Process Network connected via synchronous channels. Our reference languages is the SystemC, a C++ class library able to capture and define system specifications. The SBM is implemented by SystemC modules and threads.

The Technology Library (TL) defines the basic HW components available to build the final HW platform based on the selected target template architecture. The final HW platform is composed of several basic HW components. These components are collected into the TL, that can be considered as a generic “database” that provides the characterization of the available processor technologies. In this section a reference use case example, called Fir-Fir-GCD, is presented. This reference use case has been used in several MegaM@Rt2 activities in order to exploit Hepsycode functionalities. Fir-Fir-GCD is a synthetic application that takes in input two values (triggered by Stimulus), makes two filtering actions (FIR8 and FIR16) and then makes the greatest common divisor (GCD) and displays the result.

Fig. 3.12-1 shows the data flow (data path) model associated to the reference application, that represents the data path from input (Data Input) to the final output (Data Output). Two data input signals arrive to two subsystems, and the results have been elaborated by a GCD component, in order to release output signal as final computational value. It is worth noting that the application can be split into 3 main macro sub-systems: the FIR8 (FIR 8-bit filtering step), the FIR16 (FIR 16-bit filtering step), and the GCD component. The GCD data flow model has been derived from traditional Euclidean algorithm. Starting from this reference dataflow model (that represents the input system specification). It is possible to realize the FIR-FIR-GCD HML model in Fig. 3.12-2, where the application is composed of eight processes and twelve channels. Two more processes (Stimulus and display) and three more channels are then used to describe and connect the testbench (represented by 2 input channel $i_1$ and $i_2$ and 1 output channel $o_1$).
$G = \{PS, CH\}$ is the graph of the Fir-Fir-GCD specification, where the graph nodes are the processes and the graph edges are the data transfer. The application specification is listed below.

$PS = \{ps_1, ps_2, ps_3, ps_4, ps_5, ps_6, ps_7, ps_8\}$

$ps_1 = \{Fir8, c, LP, 0, DT_1\}$, $DT_1 = \{dt_{1,1}, dt_{1,2}, dt_{1,3}, dt_{1,4}\}$

$ps_2 = \{Fir8Eval, c, LP, 0, DT_2\}$, $DT_2 = \{dt_{2,1}, dt_{2,2}\}$

$ps_3 = \{Fir8Shift, c, LP, 0, DT_3\}$, $DT_3 = \{dt_{3,1}, dt_{3,2}\}$
support.

Considering the architectural models, in this example a fixed number of Basic Block (basic HW component composing the HW Architectural specification, the Technology library), some of them with Hypervisor support. The CC4CS [18] values for each data type (int8, int16, int32 and float) are related to the minimum, maximum and average, respectively. This is the reference timing metric used to simulate system behavior and extract as-much-as-possible system information.

The technology library is composed of these components:

- **PU = {pu₁, pu₂, pu₃, pu₄, pu₅}, where:**
  - pu₁: GPP 16 MHz 8-bit 8051 CISC core, CC4CS int8 = [59, 375, 117], CC4CS int16 = [82, 493, 162], CC4CS int32 = [106, 473, 223], CC4CS float = [4, 1322, 526];
  - pu₂: ASP 16 MHz 16-bit PIC24 core, CC4CS int8 = [59, 375, 117], CC4CS int16 = [82, 493, 162], CC4CS int32 = [[106, 473, 223], CC4CS float = [4, 1322, 526];

\[\text{pu} \]
o pu_1: GPP 75 MHz 32-bit LEON3 soft-processor, CC4CS int8 = [11, 2197, 193], CC4CS int16 = [11, 2194, 291], CC4CS int32 = [23, 2194, 437], CC4CS float = [28, 2200, 440];
  o pu_2: SPP 50 MHz Spartan3an, CC4CS = [0.038, 5.813, 1.227];
  o pu_3: SPP 250 MHz Virtex-7, CC4CS = [0.048, 3.886, 0.928];

- EIL = \{e_i, e_i, e_i, e_i\}, where:
  - e_i: Point-to-Point GPIO with 9600 bit/s max bandwidth, number of connectable processing units equal to 2 and 8 bit physical width, unit cost 1;
  - e_i: Point-to-Point UART with 38400 bit/s max bandwidth, number of connectable processing units equal to 2 and 8 bit physical width, unit cost 10;
  - e_i: BUS I2C with 100000 bit/s max bandwidth, number of connectable processing units equal to 10 and 8 bit physical width, unit cost 10;
  - e_i: BUS SPI with 30000000 bit/s max bandwidth, number of connectable processing units equal to 20 and 8 bit physical width, unit cost 15;

- MU = \{m_1, m_2\} = \{V RAM, NV ROM\};

For the sake of simplicity, only a fixed value of CC4CS was chosen for timing simulation activities. The available BBs are:

- b_i_1: \{pu_i, m_i(128byte), m_i(4KB), e_i, e_i\}, CC4CS = 260, cost = 10;
- b_i_2: \{pu_i, m_i(1KB), m_i(14KB), e_i, e_i\}, CC4CS = 260, cost = 20;
- b_i_3: \{pu_i, m_i(256MB), m_i(16KB), NULL, e_i\}, CC4CS = 345, cost = 100, Hypervisor support (e.g., Xtratum);
- b_i_4: \{pu_i, NULL, NULL, 700K System Gates, e_i, e_i\}, CC4CS = 6, cost = 400;
- b_i_5: \{pu_i, NULL, NULL, 12M System Gates, e_i, e_i, e_i\}, CC4CS = 3, cost = 900;

Considering Architectural Constraints (AC), the maximum number of instances for each bb is 2 (i.e., the total number of BBs instances is 10) and the maximum number of instances of bb, considered into the DSE is equal to the number of processes. This is an example of Hepsycode input artifacts, that can be extracted from UML/MARTE (Papyrus) models (S3D or CHESS, for example, see D2.3, D2.4, and D2.5), or from other custom meta-models (Java Xamber models, see D2.3, D2.4, and D2.5). More details about HML specification can be found in [19].

Since the whole Co-Design framework has been developed around Eclipse technologies, the system behavioral model, the processes and channels parameters, and the system requirements and specification (in terms of F/NF requirements) have to be described inside the Hepsycode tool, as shown in Fig. 3.12-3.
It can be noted that all the process and the channels respect the Model of Computation Communicating Sequential Process (CSP) according to the HEPSYCODE approach. All processes will be given additional information saved in XML files (that will be annotated during the design flow. In this Hepsycode version, the GUI allows to create four different elements:

- Packages: a set of process rely by channels
- Channels: allows the communication between the processes and SubProcesses
- SubProcess: allows to create processes inside a package.
- Process: describe a part of the entire system (without system behavior specification, that will be introduced using SystemC specification language, now these are just black box processes).

Using M2M transformation activities, the initial process network representing the reference application will be translated into SystemC functional simulable code, and the designer could introduce behavior inside the several processes, and refine the model, using the CSP rendezvous blocking channels. The different system-level design flow activities are well described in different MegaM@Rt2 deliverables, while other additional information can be found in research papers or official Hepsycode website and repository.

The final Hepsycode output artifacts are related to the so called Deployment view, a set of processes, channels, allocation and mapping between processes and Basic HW components (the BBs), and also possible virtualization technologies (i.e., Hypervisors) on the top of processors that support them. As an example, Fig. 3.12-4 shows a possible Hepsycode artifact result generated by the tool and framework design flow activities.

![Fig. 3.12-4: Hepsycode Deployment View.](image-url)
3.13. JTL

3.13.1. Methodology

JTL is an Eclipse EMF-based tool, realized to design and manipulate models, maintain consistency and synchronize software artifacts, and keep traceability during design. Its constraint-based and relational model transformation engine is specifically tailored to support bidirectionality, change propagation and traceability.

JTL is an EMF-based tool that is therefore able to support languages based on EMF Ecore and it is integrated with other Eclipse-based modeling tools. This enables the tool to support both standard and domain specific modelling languages and profiles. The JTL bidirectional model transformation engine is able to guarantee compliance among the involved metamodels. Finally, the JTL bidirectional mechanism allows to provide the automatic generation of the performance analysis model from the UML/MARTE model and vice versa.

JTL has been successfully applied to the use case provided by CSY, concerning the log analysis of a safety-critical system. Specifically, JTL has been able to generate traceability information between logs models, B models (CSY_02), and design models (CSY_01). Such information has been represented as traceability models (in EMF) and provided to EMFViews to perform a log analysis (CSY_03).

The JTL transformation mechanism provides relational semantics that relies on Answer Set Programming (ASP). Given a change to one source, JTL uses the DLV constraint solver to find a consistent choice for the other source. JTL has been proposed for dealing with non-deterministic transformations, thus it allows to specify non-bijective correspondences between source and target models (both in terms of consistency relations and traceability links). Furthermore, more than one solution models can be generated as output of the bidirectional transformation according to the non-deterministic specification.

The process underlying the generation of traceability information is outlined in Figure 3-13.2. Within JTL, the tracing information mechanism stores relevant details about the linkage between source and target model elements at execution-time (including the applied transformation rules). The traceability
mechanism is an intrinsic characteristic of the ASP-based engine. Trace links are extrapolated during the transformation execution and made explicit by the framework. Thus, trace models are explicit and maintained as models conforms to the JTL Trace Metamodel, as defined in its Ecore format within EMF. Trace models can be stored, viewed and manipulated (if needed) from the designer. Within JTL, trace models are re-used during the transformation execution. In particular, trace model, can be given as input of the transformation in order to (re-) establish consistency, manage ambiguities and guarantee the correctness of the transformation.

The JTL tool components are:
- Bidirectional Model Transformation Engine: that allows executing model transformation in forward and backward direction according to the relational specification.
- Traceability Engine: that allows to generate trace links that relate source and target model elements, for each transformation executions and maintain them in dedicated models.

The JTL framework has been implemented within the Eclipse EMF framework and has been realized as a set of plugins that exploiting Eclipse RCP.

3.13.2. Guidelines

When executing a bidirectional transformation, JTL requires in input:
- $\text{MM}_{\text{source}}$: a source metamodel in EMF format
- $M_{\text{source}}$: a source model in EMF format
- $\text{MM}_{\text{target}}$: a target metamodel in EMF format
- Bidirectional transformation: a specification of relations between metamodels, defined using the JTL syntax within the JTL editor provided in Eclipse

In this case, JTL will produce as output $M_{\text{target}}$, a target model in EMF format and conforming to $\text{MM}_{\text{target}}$, as well as a traceability model.
JTL can also be used to just generate traceability information by providing the M_target model as additional input. In this case, the JTL traceability engine will only generate a traceability model containing link that relates elements in M_source and elements in M_target.

JTL can generate target models as well as traceability models. As an example, Figure 3.13-3 shows a log model on the left-hand side, a traceability model in the middle, and a model of a B specification on the right-hand side. The traceability model links elements from both sides using traceability links. Such links are generated by the JTL traceability engine using a correspondences specification.

![Figure 3.13.1 Traceability model (middle) between logs (left-hand side) and the B specification (right-hand side)](image)

### 3.14. PauWare

#### 3.14.1. Methodology

PauWare tools aim at developing software based on executable models. An executable model reifies the behavior of the system. For instance, an elevator system can reach a given floor, open and close its doors, etc.: there are the business actions of the system. The behavior of the system can be defined with a finite state machine: based on the events generated by the user when pushing a button, the state machine defines what to do and controls the business actions of the elevator system.

PauWare mainly focuses on UML state machines but executable DSL can also be defined with Xmodeling Studio. The main problem with executable models is how to weave business operations into the behavioral part reified in the executable model.

For Xmodeling Studio, user guide and examples can be found at [https://pauware.univ-pau.fr/xmodeling/index.html](https://pauware.univ-pau.fr/xmodeling/index.html) and in “Eric Cariou, Olivier Le Goaer, Léa Brunschwig and Franck Barbier, A generic solution for weaving business code into executable models, 4th International Workshop on Executable Modeling at MoDELS (EXE 2018), CEUR Workshop Proceedings vol. 2245, October 2018”
The rest of this section will describe the whole process when using PauWare for implementing a reactive-based software system. This part will be common between D2.6 and D3.6. The guideline section of each deliverable will describe the particular tasks at design or runtime level.

Pauware enables to:
- Implement an application using a UML state machine as the reification of its behavior
- Weave business code written in plain Java with the elements (states, transitions…) of the UML state machine
- Monitor the execution or the simulation of the state machine and of its business operations to ensure the right execution of the system or generate execution traces that can be afterwards analyzed and verified

PauWare is fully in line with a continuous development approach:
- The state machine defined at design is exactly the same that is executed at runtime in the running system
- The simulation of the state machine in a verification purpose at design and its execution at runtime is made with the same engine. Thus, the semantics of the behavior of the system and the semantics of its execution are the same at design that at runtime. A property verified at design is then by principle also ensured at runtime.

PauWare is related to UC 03_IKER and KPI 1.1, KPI 1.2, KP 2.1 and KPI 3.1.

The figure shows the development of a Java-based software using PauWare and its associated tools. The yellow boxes corresponds to tasks realized by an engineer whereas the green ones are automatically executed.

First of all, the designer defines a UML state machine with its favorite UML modeler, and possibly a UML class diagram for defining business method signatures. Then, the code corresponding of this state machine is generated for the Java API of PauWare with the PauWare Generator tool. This API enables to define states with business operations, transitions with guards and to use all the features of the UML specification for building a state machine in Java.

At this stage, the behavior of the system to build or to simulate is implemented thanks to this code generation. The PauWare state machine is already executable. However, it lacks the operations associated with the state machines: business operations of states and transitions, guards of transitions
and possibly invariants of the state machine. All these operations have been implemented aside in plain Java. The links between them and the state machine is usually straightforward as the signatures of these operations are already generated and associated with the elements of the state machine. For instance, the following two lines of code has been automatically generated from UML diagrams:

```java
AbstractStatechart baking = new Statechart("Baking");
baking.set_entryAction(mwb, "heat");
```

It defines a state called “Baking” and associates an entry action with this state: the call of a method “heat” on an object mwb. The code integration consists simply here in instantiating this mwb object that is implementing a method called “heat” (without parameters).

Once the integration between the PauWare code of the state machine and the Java business code done, a complete running application is available. Once launched, the state machine is made evolving by processing events. An event can make the state machine going from one active state to another one and automatically executes the associated business operations.

If one wants to control or verify the execution of the state machine and/or its business operations, it requires to implement observers. An observer is a Java class which implements the interface Observable:

```java
public interface Observable {
    void onStateEntry(AbstractStatechart state);
    void onStateExit(AbstractStatechart state);
    void onEntryAction(AbstractStatechart state, AbstractAction action);
    void onExitAction(AbstractStatechart state, AbstractAction action);
    void onDoActivity(AbstractStatechart state, AbstractAction action);
    void onAllowedEvent(AbstractStatechart state, AbstractAction action);
    void onViolatedInvariant(AbstractStatechart state, AbstractAction invariant);
    void onVerifiedInvariant(AbstractStatechart state, AbstractAction invariant);
    void onStateMachineStart(AbstractStatechart stateMachine);
    void onStateMachineStop(AbstractStatechart stateMachine);
    void onGuardValid(Transition transition, AbstractGuard guard);
    void onGuardUnvalid(Transition transition, AbstractGuard guard);
    void onTransitionOperation(Transition transition, AbstractAction action);
    void onFiredTransition(Transition transition);
    void startCompletionCycle(String event);
    void endCompletionCycle(String event);
    void onError(Statechart_exception err);
    void setStateMachine(AbstractStatechart_monitor sm);
}
```

This interface enables an observer to be informed of almost everything happening during the execution of a state machine: a transition is followed, a business operation is called, an invariant is violated, etc. With this strategy, it is possible to implement, for instance, an observer that generates an execution trace or another one that is monitoring at runtime the execution of the state machine and ensuring its correctness.
3.14.2. Guidelines

At design level, the engineer has to define its state machine with its favorite UML modeler and to generate the code of the state machine. The PauWare code generator is able to process models from most of the UML modelers (Modelio, Papyrus, Software Architect, Star UML…). The simulation of the state machine is simply made by executing the state machine by processing a sequence of events through the call of the “run_to_completion(event)” method on the state machine object. A run to completion cycle follows all the transitions from the active states associated with the given event and executes all the required operations. It is then easy to perform verification tasks on the state machine such as checking that at the end of a given event sequence, the expected state has been reached or by analysing a trace execution. If an error is found during the simulation, the UML model has to be modified and the PauWare code is regenerated for re-executing the simulations.

At this stage of simulation, usually business operations or other operations such as guards are still not implemented (such as the “heat” method of the above “Baking” state), the main goal is to verify the behavior of the state machine. However, if the PauWare engine is here used for simulation, it has been developed for executing a complete running systems and then the business operations associated with the states and transitions. This requires to implement these operations or at least empty operations for the expected signatures. A particular attention should be paid to the guards of transitions. Indeed, two transitions starting from the same state with the same event can exist if their guards return a different result. In this case, for testing, if concrete guards can still not be defined, it is required to define mock guards (concretely that returns “true” or “false” without any calculus but depending on what is expected). A simulation test case will then be defined by the sequence of processed events and the choices of the constant return values of the guards.

3.15. CMA

3.15.1. Methodology for CMA

CMA, acronym of Completeness Metric Analyzer, is a command-line tool that helps to reduce the manual effort required to transform Natural Language (NL) requirements into semi-formal boilerplate requirements and thus to improve the overall elicitation, verification and validation processes.

The term boilerplate refers to a textual requirement template which consists of a series of attributes and Fixed Syntax Elements (FSEs), such as:

- `<system>` may be `<state>`
  - attributes: system, state
  - FSE: may be
- if `<event>`
  - attributes: event
  - FSE: if
- at least `<quantity>` times per `<unit>`
  - attributes: quantity, unit
  - FSE: times per

Once the notions and the axioms contained in the Domain Ontology (DO) are identified, CMA provides command-line functionalities, which are used to link the NL concepts to boilerplates attributes (i.e. the
term "passenger" refers to the <user> boilerplate attribute) by means of a command-line helper that guides the user through the process.

3.15.2. Guidelines for CMA

CMA’s command-line helper shall guide the user throughout the process according to the following steps:

- **Import and export:** whilst it is possible to create new content by using the appropriate functionality, the user can also import and export existing requirements (XML, CSV, ModelBus CMM, RequisitePro RTP and Doors RTP extensions), Boilerplates (Builtin CESAR Boilerplates and XML extensions) and Ontology files (OWL, CSV and ModelBus OWL extensions).
- **Natural Language Processing (NLP):** the tool uses several NLP algorithms, it detects sentence beginning and ends, uses the identified DO concepts as a dictionary to correct typographic errors and transforms the listed requirements in such a way that matching boilerplates can be found based on user-defined substitution rules (i.e. replace occurrences such as "must", "should" or "will" with the word "shall").
- **Boilerplate, Requirements and Ontology editors:** these functionalities are used to add or modify the templates which are used by the tool to assign FSEs and boilerplates. It is possible to modify the substitution rules, insert new or modify existing requirements in either NL or directly into boilerplates and to tag them based on their Category (Functional, Non-Functional and Safety) and Type (Assumption, Goal and Requirement), add or modify the DO concepts to allow the tool to work with its NLP functionalities.
- **Requirements Analysis functionality:** the tool uses the transformed requirements, obtained after the matching phase with the boilerplates (in which FSEs are used to assign the text between them to attributes), to analyze the requirements and to output different qualitative and quantitative metrics. This feature allows the user to refine the requirements with more efficiency.

As stated, NLP based mechanisms are used to assess the completeness of requirements. To ensure and improve the consistency, completeness and correctness of requirements, formal languages represents an alternative to natural language (NL) requirement descriptions. The formalization of those requirements is a primarily manual task, which therefore is both cumbersome and error-prone. The transformation builds upon a domain ontology (DO) containing the knowledge of the problem domain and upon natural language processing techniques. The semi-automatic conversion process is based on six steps: Sentence Selection, Typographic Error Correction, Word Substitution, Boilerplate Matching, Attribute Value Splitting and Ranking.

3.16. MATERA2

3.16.1. Methodology for MATERA2

Åbo Akademi University (ABO)'s MATERA2 is a tool suite that comprises six components: (1) a model-based monitoring and online testing tool called MATERA2-MBMÅA, (2) a model-based performance and load testing tool called MATERA2-MBPeT, (3) a model-based testing tool for executable UML models called MATERA2-ATester, (4) a genetic algorithm based performance space exploration tool for web applications called MATERA2-PerfXGA (5) a reinforcement learning based
exploratory performance testing tool called MATERA2-iPerfXRL, and (6) a model-based conformance testing tool called MATERA2-ADCT.

In WP2, ABO has participated in UC 03_IKER and UC 05_NOK with two tools: MATERA2-AlfTester and MATERA2-ADCT. Moreover, MATERA2-MBMÅA has been applied to UC 06_BT in WP2. MATERA2-ADCT, MATERA2-AlfTester, and MATERA2-MBMÅA also satisfy WP3 requirements. The detailed methodologies for these three tools can be found in deliverable D3.6. In this section, we briefly present each one of these tools.

MATERA2-AlfTester provides a novel approach for exhaustive simulation and test generation from fUML ADs containing Alf code. It translates fUML ADs and associated Alf code into equivalent Java code and then automatically generates: (1) input data needed to cover or execute all paths in the executable fUML and Alf models and (2) a test suite comprising test cases and test oracle (expected output) for testing the actual implementation of the system under development.

MATERA2-ADCT utilizes specification models for conformance testing via introducing time properties in executable models and enabling execution engine to generate partial observable traces. The time properties enable the modeller to model time in executable models and generation of partially observable trace are used to conform the implementation under test with specifications.

MATERA2-MBMÅA provides a technique for monitoring of reactive real-time systems. The models of the system and its environment are created with UPPAAL timed automata. We use DTRON to passively execute the test cases and to verify the behavior of the system. During this process, we also compute the coverage level of the models with respect to the specified coverage criteria.

3.16.2. Guidelines for MATERA2

The guidelines for MATERA2-ADCT, MATERA2-AlfTester, and MATERA2-MBMÅA tools can be found in deliverable D3.6.

3.17. RCRS

3.17.1. Methodology for RCRS:

Refinement Calculus of Reactive Systems Toolset is an environment for compositional formal modeling and reasoning about reactive systems, built around Isabelle, Simulink, and Python. The toolset implements the Refinement Calculus of Reactive Systems (RCRS), a contract-based refinement framework inspired by the classic Refinement Calculus and interface theories. The toolset formalizes the entire RCRS theory in about 30000 lines of Isabelle code. The toolset also contains a translator of Simulink diagrams, and a formal analyzer implemented on top of Isabelle. The overview of the toll is presented in the next figure:
The RCRS Toolset consists of the following:

- A full formalization of the RCRS theory in the Isabelle proof assistant.
- A set of analysis procedures for RCRS models, implemented on top of Isabelle and collectively called the Analyzer.
- A Translator of Simulink diagrams into RCRS code.
- A library of basic RCRS components, including a set of basic Simulink blocks modeled in RCRS.

We used the the RCRS toolset in two hackathon challenges proposed by BT revealing problems in some Simulink models that were used as part of these challenges.

3.17.2. Guidelines for RCRS:

In one usage scenario, RCRS can be applied to a Simulink diagram, to obtain a formal representation of this diagram as an Isabelle theory. Next the theory can be analyzed in Isabelle to uncover possible inconsistencies or incompatibilities and also to simplify and flatten the computations. From within Isabelle, for compatible Simulink models, RCRS can generate Python or C/C++ code that can be used to simulate the model, or to integrate it in a larger project. Next summation Simulink diagram represents a discrete system which at each discrete time point outputs the sum of all previous inputs.

Running the RCRS translator on this model produces the following Isabelle representation of this model:

```isabelle
theory Summation imports ... begin ...
definition [basic_simps]: "Add = [- f, g ↝ f + g -]"
definition [basic_simps]: "UnitDelay = [- d, s ↝ s, d -]"
definition [basic_simps]: "Split = [- a ↝ a, a -]"
simplify_RCRS "summation =```

Page 84 of 86
feedback([- f, g, s \leadsto (f, g), s -]
  o (Add ** Id) o UnitDelay o (Split ** Id)
  o [- (f, h), s' \leadsto f, h, s'] -)
end

The simplify_RCRS keyword is used to invoke the Analyzer on this example. The Analyzer automatically simplifies the summation system to:

\[
\text{summation} = [- x, s \leadsto s, s + x -]
\]

RCRS toolset together with examples and papers describing its theory and usage are available from https://rcrs.gitlab.io.

References

  0requirements.pdf, https://www.youtube.com/watch?v=edHAXb8-1Io
[22] https://github.com/SINTEF-9012/bvr