D3.6: Methodology and Guidelines for MegaM@Rt2 Runtime Analysis
Methods and Tools

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

DISSEMINATION LEVEL: PUBLIC
HANDLE: http://hdl.handle.net/20.500.12004/1/P/MMART2/D3.6
LAST EDITED: 29/11/2019
Executive Summary

The main objective of WP3 is to provide Runtime Analysis methods and tools for creating and managing large-scale models at runtime in order to support efficient verification and testing activities including means for traceability of requirements. To achieve this objective, WP3 work was guided by the project case study challenges in three development phases (initial, intermediate, and final). During these phases, the main focus of WP3 activities was on investigating and developing relevant and efficient tool-supported solutions for automating and generalizing the conceptual runtime approaches.

Deliverable D3.6 provides methodologies and guidelines for the final version of the Runtime Tool Set presented in D3.5. The inputs to deliverable D3.6 are deliverables D3.1-3.5 and deliverable D5.5. Moreover, task T3.5 provided a set of guidelines for applying the methods and tool prototypes developed in WP3 for using models at runtime. The guidelines were taken into use by case study providers in order to apply the methods and tools to targeted application domains. Feedback from the case study providers was analyzed and used for iterating on and improving the outcomes of WP3.
# Table of Contents

Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>3</td>
</tr>
<tr>
<td>Acronyms</td>
<td>6</td>
</tr>
<tr>
<td>Introduction</td>
<td>10</td>
</tr>
<tr>
<td>Methodology and Guidelines</td>
<td>12</td>
</tr>
<tr>
<td>CompleteTest</td>
<td>12</td>
</tr>
<tr>
<td>Methodology for CompleteTest</td>
<td>12</td>
</tr>
<tr>
<td>Guidelines for CompleteTest</td>
<td>13</td>
</tr>
<tr>
<td>S3D</td>
<td>15</td>
</tr>
<tr>
<td>Methodology for S3D</td>
<td>15</td>
</tr>
<tr>
<td>Definition</td>
<td>15</td>
</tr>
<tr>
<td>Generation</td>
<td>16</td>
</tr>
<tr>
<td>Analysis</td>
<td>16</td>
</tr>
<tr>
<td>Guidelines for S3D</td>
<td>17</td>
</tr>
<tr>
<td>Traces definition</td>
<td>17</td>
</tr>
<tr>
<td>Traces Generation</td>
<td>17</td>
</tr>
<tr>
<td>Trace Analysis</td>
<td>17</td>
</tr>
<tr>
<td>PADRE</td>
<td>18</td>
</tr>
<tr>
<td>Methodology for PADRE</td>
<td>18</td>
</tr>
<tr>
<td>Guidelines for PADRE</td>
<td>20</td>
</tr>
<tr>
<td>CHESS</td>
<td>23</td>
</tr>
<tr>
<td>Methodology for CHESS</td>
<td>23</td>
</tr>
<tr>
<td>Guidelines for CHESS</td>
<td>23</td>
</tr>
<tr>
<td>XPM</td>
<td>23</td>
</tr>
<tr>
<td>Methodology for XPM</td>
<td>23</td>
</tr>
<tr>
<td>Guidelines for XPM</td>
<td>25</td>
</tr>
<tr>
<td>Papyrus Extension for AOM</td>
<td>25</td>
</tr>
<tr>
<td>Methodology for Papyrus Extension for AOM</td>
<td>25</td>
</tr>
<tr>
<td>Guidelines for Papyrus Extensions</td>
<td>26</td>
</tr>
<tr>
<td>AIPHS</td>
<td>28</td>
</tr>
<tr>
<td>Methodology for AIPHS</td>
<td>28</td>
</tr>
<tr>
<td>Guidelines for AIPHS</td>
<td>29</td>
</tr>
<tr>
<td>JTL</td>
<td>30</td>
</tr>
<tr>
<td>Methodology for JTL</td>
<td>30</td>
</tr>
<tr>
<td>Guidelines for JTL</td>
<td>32</td>
</tr>
<tr>
<td>CertifyIt</td>
<td>33</td>
</tr>
<tr>
<td>Methodology for CertifyIt</td>
<td>33</td>
</tr>
<tr>
<td>Guidelines for CertifyIt</td>
<td>34</td>
</tr>
<tr>
<td>MBeetle</td>
<td>35</td>
</tr>
<tr>
<td>Methodology for MBeetle</td>
<td>35</td>
</tr>
<tr>
<td>Guidelines for MBeetle</td>
<td>37</td>
</tr>
<tr>
<td>Convex Hull</td>
<td>38</td>
</tr>
</tbody>
</table>
Methodology for Convex Hull
Guidelines for Convex Hull
Clusterability
Methodology for Clusterability
Guidelines for Clusterability
Moka Extension for Logging
Methodology for Moka Extension for Logging
Guidelines for Moka Extension for Logging
MATERA2-MBMÅÅ
Methodology for MATERA2-MBMÅÅ
Guidelines for MATERA2-MBMÅÅ
Tool chain dependencies
SPREAD
UPPAAL TRON
Dtron
Test Tracker
Adapter
Steps to run the tool chain
Example
MATERA2-MBPeT
Methodology for MATERA2-MBPeT
Guidelines for MATERA2-MBPeT
User Models
Adapter
Test configurations
Running MATERA2-MBPeT
Test Report
MATERA2-PerfXGA
Methodology for MATERA2-PerfXGA
Guidelines for MATERA2-PerfXGA
MATERA2-iPerfXRL
Methodology for MATERA2-iPerfXRL
Guidelines for MATERA2-iPerfXRL
MATERA2-ADCT
Methodology for MATERA2-ADCT
Guidelines for MATERA2-ADCT
MATERA2-AlfTester
Methodology for MATERA2-AlfTester
Guidelines for MATERA2-AlfTester
Modelio
Methodology for Modelio
Guidelines for Modelio
Installation
Add the module to Modelio
Deploy the module in the project
Usage
Create System Model with DDS profile 70
Command: Include in IDL files 70
Command: Create Topic / Add topic stereotype 70
IDL modelling details 70
   IDL naming conventions 71
Code generation customization 71
VeriATL 71
   Methodology for VeriATL 71
   Guidelines for VeriATL 72
LIME TestBench 72
   Methodology for LIME TestBench 72
   Guidelines for LIME TestBench 72
RCRS 73
   Methodology for RCRS 73
   Guidelines for RCRS 73
PauWare 73
   Methodology for PauWare 73
   Guidelines for PauWare 76
CMA 76
   Methodology for CMA 76
   Guidelines for CMA 77

Conclusions 78
References 79
<table>
<thead>
<tr>
<th>Acronyms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AADL</td>
<td>ARCHITECTURE ANALYSIS AND DESCRIPTION LANGUAGE</td>
</tr>
<tr>
<td>ADL</td>
<td>ARCHITECTURE DESCRIPTION LANGUAGE</td>
</tr>
<tr>
<td>AL</td>
<td>ARCHITECTURAL LANGUAGE</td>
</tr>
<tr>
<td>ALEX</td>
<td>AUTOMATA LEARNING EXPERIENCE</td>
</tr>
<tr>
<td>ALF</td>
<td>ACTION LANGUAGE FOR FOUNDATIONAL UML</td>
</tr>
<tr>
<td>API</td>
<td>APPLICATION PROGRAMMING INTERFACE</td>
</tr>
<tr>
<td>APL</td>
<td>APACHE PUBLIC LICENSE</td>
</tr>
<tr>
<td>ARINC</td>
<td>AERONAUTICAL RADIO INCORPORATED</td>
</tr>
<tr>
<td>ASCET</td>
<td>ADVANCED SIMULATION/SOFTWARE AND CONTROL ENGINEERING TOOL</td>
</tr>
<tr>
<td>ASIL</td>
<td>AUTOMOTIVE SAFETY INTEGRITY LEVEL</td>
</tr>
<tr>
<td>ASL</td>
<td>ACTION SPECIFICATION LANGUAGE</td>
</tr>
<tr>
<td>ATL</td>
<td>ATLAS TRANSFORMATION LANGUAGE</td>
</tr>
<tr>
<td>AUTOSAR</td>
<td>AUTOMOTIVE OPEN SYSTEM ARCHITECTURE</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>CBSE</td>
<td>COMPONENT-BASED SOFTWARE ENGINEERING</td>
</tr>
<tr>
<td>CDO</td>
<td>CONNECTED DATA OBJECTS</td>
</tr>
<tr>
<td>CP</td>
<td>CONSTRAINT PROGRAMMING</td>
</tr>
<tr>
<td>CPS</td>
<td>CYBER-PHYSICAL SYSTEMS</td>
</tr>
<tr>
<td>CSP</td>
<td>CONSTRAINT SOLVING PROBLEM</td>
</tr>
<tr>
<td>CSR</td>
<td>CASE STUDY REQUIREMENT</td>
</tr>
<tr>
<td>CTF</td>
<td>COMMON TRACE FORMAT</td>
</tr>
<tr>
<td>CTS</td>
<td>CONCEPTUAL TOOL SET</td>
</tr>
<tr>
<td>DDS</td>
<td>DATA DISTRIBUTION SERVICE</td>
</tr>
<tr>
<td>DL</td>
<td>DESCRIPTION LOGICS</td>
</tr>
<tr>
<td>DMA</td>
<td>DIRECT MEMORY ACCESS</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>EAST-EEA</td>
<td>ELECTRONICS ARCHITECTURE &amp; SOFTWARE TECHNOLOGIES - EMBEDDED ELECTRONIC ARCHITECTURE</td>
</tr>
<tr>
<td>EMF</td>
<td>ECLIPSE MODELING FRAMEWORK</td>
</tr>
<tr>
<td>EMOF</td>
<td>ESSENTIAL MOF</td>
</tr>
<tr>
<td>EPL</td>
<td>ECLIPSE PUBLIC LICENSE</td>
</tr>
<tr>
<td>FBD</td>
<td>FUNCTION BLOCK DIAGRAM</td>
</tr>
<tr>
<td>FMI</td>
<td>FUNCTIONAL MOCK-UP INTERFACE</td>
</tr>
<tr>
<td>FOSS</td>
<td>FREE OPEN SOURCE SOFTWARE</td>
</tr>
<tr>
<td>F UML</td>
<td>SEMANTICS OF A FOUNDATIONAL SUBSET FOR EXECUTABLE UML MODEL</td>
</tr>
<tr>
<td>GMF</td>
<td>GRAPHICAL MODELING FRAMEWORK</td>
</tr>
<tr>
<td>GPL</td>
<td>GENERAL-PURPOSE (MODELLING) LANGUAGES</td>
</tr>
<tr>
<td>GPL</td>
<td>GNU PUBLIC LICENSE</td>
</tr>
<tr>
<td>GUI</td>
<td>GRAPHICAL USER INTERFACE</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TCS</td>
<td>Helsinki University of Technology - Laboratory for Theoretical Computer Science</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>ITEA</td>
<td>Information Technology for European Advancement</td>
</tr>
<tr>
<td>LBT</td>
<td>Learning-Based Testing</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>MAENAD</td>
<td>Model-based Analysis &amp; Engineering of Novel Architectures for Dependable Electric Vehicles</td>
</tr>
<tr>
<td>MAF</td>
<td>Major Frame</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>MFR</td>
<td>MMRT Framework Requirement</td>
</tr>
<tr>
<td>ML</td>
<td>Modelling Language</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
</tr>
<tr>
<td>M&amp;S</td>
<td>Modelling and Simulation</td>
</tr>
<tr>
<td>NFP</td>
<td>Non Functional Property</td>
</tr>
<tr>
<td>OAL</td>
<td>Object Action Language</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>ODM</td>
<td>Ontology Definition Metamodel</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OMT</td>
<td>Object-Modeling Technique</td>
</tr>
<tr>
<td>OOA</td>
<td>Object-Oriented Analysis</td>
</tr>
<tr>
<td>OOM</td>
<td>Object-Oriented Modelling</td>
</tr>
<tr>
<td>OOSA</td>
<td>Object-Oriented Systems Analysis</td>
</tr>
<tr>
<td>OOSE</td>
<td>Object-Oriented Software Engineering</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>PAL</td>
<td>Platform-independent Action Language</td>
</tr>
<tr>
<td>PDM</td>
<td>Platform Description Model</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable Logic Controllers</td>
</tr>
<tr>
<td>PSCS</td>
<td>Precise Semantics of UML Composite Structures</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>PSSM</td>
<td>Precise Semantics of UML State Machines</td>
</tr>
<tr>
<td>PTA</td>
<td>Priced Timed Automata</td>
</tr>
<tr>
<td>QVT</td>
<td>Query/View/Transformation</td>
</tr>
<tr>
<td>RC</td>
<td>Resource-Constrained</td>
</tr>
<tr>
<td>REAL</td>
<td>Requirement Enforcement Analysis Language</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>SAT</td>
<td>Propositional Satisfiability Problem</td>
</tr>
<tr>
<td>SBSE</td>
<td>Search-Based Software Engineering</td>
</tr>
<tr>
<td>SCADE</td>
<td>Safety-Critical Application Development Environment</td>
</tr>
<tr>
<td>SCROLL</td>
<td>Starr's Concise Relational Action Language</td>
</tr>
<tr>
<td>SWRL</td>
<td>Semantic Web Rule Language</td>
</tr>
<tr>
<td>SMALL</td>
<td>Shlaer-Mellor Action Language</td>
</tr>
<tr>
<td>SMC</td>
<td>Statistical Model Checking</td>
</tr>
<tr>
<td>SMM</td>
<td>Structured Metrics Metamodel</td>
</tr>
<tr>
<td>SMT</td>
<td>SAT Modulo Theories</td>
</tr>
<tr>
<td>SMMUML</td>
<td>Symbolic Methods for UML Behavioural Diagrams</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>SWRL</td>
<td>Semantic Web Rule Language</td>
</tr>
<tr>
<td>SysML</td>
<td>System Modelling Language</td>
</tr>
<tr>
<td>TADL</td>
<td>Timing Augmented Description Language</td>
</tr>
<tr>
<td>TCP</td>
<td>Tool Component Purpose</td>
</tr>
<tr>
<td>TCTL</td>
<td>Timed Computation Tree Logic</td>
</tr>
<tr>
<td>TIMMO</td>
<td>TIMing MOdel</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>TSP</td>
<td>Time and Space Partitioning</td>
</tr>
<tr>
<td>T&amp;E</td>
<td>Test and Evaluation</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UCAN</td>
<td>Unconventional Computer Architecture and Networks</td>
</tr>
<tr>
<td>UTA</td>
<td>UPPAAL Timed Automata</td>
</tr>
<tr>
<td>UTP</td>
<td>UML Testing Profile</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>XMCF</td>
<td>XtratruM Configuration File</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>Language</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition</td>
</tr>
<tr>
<td>xtUML</td>
<td>Executable Translatable UML</td>
</tr>
<tr>
<td>xUML</td>
<td>Executable UML</td>
</tr>
</tbody>
</table>
1. Introduction

This deliverable presents methodologies and guidelines for the WP3 tools presented in D3.5. Each tool presented in this deliverable is positioned into the overall WP3 conceptual framework and runtime tool set architecture presented in deliverable D3.2 [27]. The WP3 conceptual framework requirements for runtime analysis are divided into five categories as follows:

1. Basic framework requirements
2. Analysis of execution logs requirements
3. Monitoring requirements
4. Code generation requirements
5. Testing requirements

The basic framework requirements were specified according to the expected tooling solutions provided by the MegaM@Rt2 partners to address the specific needs and expectations of the case study providers. The analysis of execution logs requirements specify the different techniques that the MegaM@Rt2 framework employs for the analysis of the execution logs. The monitoring requirements include different techniques for observing the internal operations of a system and its interactions with other external models and entities with the aim of monitoring both the runtime software and the abstract architectural constraints. The code generation requirements specify a set of tools and guidelines for automated code generation involving aspects weaving and model execution, continuous runtime validation including runtime verification, monitoring and online testing as well as trace analysis. Finally, the testing requirements specify online and offline testing capabilities included in the MegaM@Rt2 framework.

D3.2 [27] also presents detailed implementation plans for the runtime analysis tool set and a roadmap of how the above framework requirements are satisfied by WP3 tools. In this deliverable, we use the above categorization to position each of the WP3 tools and then present their methodologies and guidelines.

Figure 1 presents the overall conceptual framework and runtime tool set architecture for WP3 and the relationship of WP3 with WP1, WP2, and WP4. For clarity, all processes and artifacts external to WP3 are shaded with gray color. The WP3 processes and artifacts in Figure 1 are shaded based on the WP3 conceptual framework requirements and tasks. For example, all processes and artifacts related to the Code Generation requirements and tasks are shaded with blue color. For clarity, Table 1 presents the relationship between the five categories of WP3 conceptual framework requirements and the WP3 processes and artifacts presented in Figure 1.

It should be noted that the methodology and guidelines for the Conformiq Designer tool are not included in this deliverable because of Conformiq’s bankruptcy in 2019. Please refer to D2.5 and D3.5 for more details on Conformiq Designer.
Figure 1: Conceptual framework and runtime toolset architecture in WP3

Table 1: Relationship between the five categories of WP3 conceptual framework requirements and the WP3 processes and artifacts in Figure 1

<table>
<thead>
<tr>
<th>WP3 conceptual framework requirements</th>
<th>WP3 processes and artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic framework requirements</td>
<td>Test modeling for runtime specification</td>
</tr>
<tr>
<td>Analysis of execution logs requirements</td>
<td>Runtime trace analysis</td>
</tr>
<tr>
<td>Monitoring requirements</td>
<td>Monitor generation</td>
</tr>
<tr>
<td>Code generation requirements</td>
<td>Code generation</td>
</tr>
<tr>
<td>Testing requirements</td>
<td>Online test generation</td>
</tr>
<tr>
<td></td>
<td>Offline test generation</td>
</tr>
</tbody>
</table>
2. Methodology and Guidelines

In this section, we present methodology and guidelines of all WP3 tools. A tool methodology includes (1) a description of the problem that the tool addresses and (2) a brief description of the proposed solution, outlining the theoretical basis of the tool along with some important details on the method or process of the tool and the implementation of the tool features. In addition, for each tool that has been applied to a use case (UC), the tool methodology also includes a statement on how the tool is applied, which UC requirements as specified in deliverable D3.2 [27] are satisfied, and which key performance indicators (KPIs) as specified in deliverable D5.5 [30] are affected and improved. The guidelines for a tool include the tool inputs, outputs, and manual steps (if any) needed to use the tool.

2.1. CompleteTest

2.1.1. Methodology for CompleteTest

CompleteTest is a tool for automatic test generation based on the methodology shown in Figure 2. Code coverage criteria can be used by CompleteTest to generate test cases. The tool infrastructure provides a way for the user to select a program, generate tests for a selection of coverage criteria, visualize the generated test inputs, and determine the correctness of the result produced for each generated test by comparing the actual test output with the expected output. The tool is built from the following modules:

- an import editor used for validating the structure of a provided test model file in the FBD format (https://plcopen.org/iec-61131-10).
- a translation plugin that creates a format accepted by CompleteTest
- trace parser that collects a diagnostic trace from the model checker and outputs an executable test suite containing inputs, outputs and timing information (i.e., the time parameter in the test is used for constraining the inputs in time)
- the test execution results in a coverage report and the execution traces.
Regarding the use case application, CompleteTest tool has been applied to the Bombardier use case and satisfies the following UC requirements:

- **BT_10** The expectation is that MegaM@RT2 provides methods on: ① how to design requirement models such that test cases can be derived; ② how test cases for module testing can be derived; ③ how test cases for higher level integration tests can be derived.
- **BT_13** The expectation is that MegaM@RT2 provides a method on how the generation of test scripts out of test cases can be automated.

CompleteTest has contributed to the following KPIs in a controlled experiment and case study:

- **KPI 1.2**: Improvement of 10%-30% in identification of design problems.
- **KPI 5.2**: Quality Improvement in the range of 10%-30% by improving predictability and conformance to specifications.

CompleteTest process is composed of the following steps:

1. **Model Transformation.** To test an FBD program we map it to a finite state system suitable for model checking. In order to cope with timing constraints, we have chosen to map FBD programs to timed automata.
2. **Logic Coverage Annotation.** We annotate the transformed model such that a condition describing a single test case can be formulated. This is a property expressible as a reachability property used in most model checkers.
3. **Test Case Generation.** We now use the model-checker to generate test traces. To provide a good level of practicality to our work, we use a specific model-checker called UPPAAL\(^1\) which uses timed automata as the input modeling language.
4. **The verification language supports reachability properties.** In order to generate test cases for logic coverage of FBD programs using UPPAAL, we make use of UPPAAL’s ability to generate test traces witnessing a submitted reachability property. Currently UPPAAL supports three options for diagnostic trace generation: some trace leading to a goal state, the shortest trace with the minimum number of transitions, and fastest trace with the shortest time delay.

### 2.1.2. Guidelines for CompleteTest

The documentation for CompleteTest can be found at [http://www.completetest.org/documentation/](http://www.completetest.org/documentation/)  
This first module of the tool used for validating whether the structure of a provided XML file represents a valid PLCOpenXML\(^2\) file containing an FBD Program (see Figure 3).

In addition, the tool needs the user to provide manually the input FB program to be used and to select the type of coverage needed. In addition, the output produced by the tool is a test report. An example of a scenario is shown in Figure 4.

---

\(^1\) [http://uppaal.org/](http://uppaal.org/)
\(^2\) [https://plcopen.org/iec-61131-10](https://plcopen.org/iec-61131-10)
Figure 3: A PLCOpenXML file containing an FBD Program

Figure 4: An example scenario
2.2. S3D

2.2.1. Methodology for S3D

The S3D Framework has been extended in order to support system analysis by runtime traces. Figure 5 shows the positioning of S3D in the WP3 conceptual framework and the tool set architecture. The detailed steps in the S3D methodology are as follows.

Definition

S3D supports data dependency analysis in order to verify whether timing constraints are fulfilled or not. For this analysis S3D uses traces generated during the execution of the application.

Data dependencies are represented using UML Sequence diagrams (see Figure 6) as part of the verification process of the application. Component instances are represented with lifelines, one per part involved in the data path to be analyzed. An execution of a service or function is represented over the lifeline with an Action Execution Specification. Since a specific part (component) can only have one main function, if the Action Execution is not pointed by a service call, it will unequivocally represent the main function of the component. On the contrary, if the Action Execution is directly pointed by a service call, it represents that specific service in question. For periodic functions, each Action Execution represents one iteration of the total run of the service. Service calls are represented using Message Sync/Async and they are linked to the service’s operation. Once all service calls have
been placed in the diagram, a Time Constraint must be added at the beginning of the last Message of the chain, and maximum and minimum time values are specified.

![Sequence Diagram](image)

**Figure 6**: A sequence diagram representing data dependencies

**Generation**

Traces are generated using the Common Trace Format [22], a standardized binary trace format designed for a fast and efficient writing while using few disk space. For trace generation, S3D makes use of the Linux open source tracing framework LTTng [23]. Traces are created when accessing and returning from a service, and store relevant information about current time and service identification.

When building and running the application, S3D will ask if trace generation is desired or not. If so, by default S3D will generate traces on every service call. In order to reduce tracing overhead, S3D will trace only those functions defined in the data dependencies UML diagram (work in progress).

**Analysis**

For trace analysis S3D uses the CTF trace manipulation library from Babeltrace [24]. Since modeling multiple data path chains in the same model are supported, multiple data path analysis is also possible. Thus, the user can select which declared data dependencies have to be analyzed. After trace analysis, S3D generates a report as shown in Figure 7, reporting requested, maximum, minimum and mean time, received data and valid and lost chains. If the requested deadline is always fulfilled, S3D reports the chain in green color, whereas if not, it will show it in red color. If the chain is invalid, it will be shown in grey color.
2.2.2. Guidelines for S3D

Traces definition

Data dependencies are specified in a dedicated package within the Verification View of the model. To define data dependencies UML Sequence diagrams are used. A lifeline should be created per component instance, and linked to its related component using the “Represented” box. Service calls from one component to another are represented using Message Sync/Async. When placing the messages from the client to the server lifelines, a pop-up will appear asking for the operation requested, and it should be selected between those previously declared. To indicate the execution of a service, an Action Execution Specification should be added over the lifeline, representing the main function of the component if it is not pointed by any message, and if so representing the execution of the service specified in the message. Finally, once all service calls have been placed in the diagram, a Time Constraint must be added at the beginning of the last Message of the chain, and maximum and minimum values are specified using a LiteralString value in the form [time unit].

It is important to notice that the Y coordinate of the diagram (vertical) corresponds with time (or causality). The order of the calls in the diagram must correspond with its requested execution during real operation: if call to A must happen before call to B, A must be higher than B in the diagram.

Traces Generation

For trace generation S3D requires the liblttng-ust-dev library from the Linux open source tracing framework LTTng, which should be installed in the host Linux system.

It creates Common Trace Format (CTF) traces. S3D generates a tracepoint when accessing and returning from a service. When building and running the application, S3D will ask if trace generation is desired or not. If so, by default S3D will generate traces on every service call. In order to reduce tracing overhead, S3D will trace only those functions defined in the data dependencies UML diagram (work in progress).

Trace Analysis

For trace analysis S3D uses the CTF trace manipulation library libbabeltrace-dev from Babeltrace, in a custom command called s3d_trace_analysis. This command takes as inputs the traces path and a path descriptor file generated when executing the analysis from the S3D plugin.

From the plugin, user selects whether it wants to analyze traces generated in native, workload or performance modes. Then, it can be chosen which declared data dependencies to be analyzed. After trace analysis, S3D generates a report as shown in Figure 7, reporting requested, maximum, minimum and mean time, received data and valid and lost chains. If the requested deadline is always fulfilled, S3D reports the chain in green color, whereas if not, it will show it in red color. If the chain is invalid, it will be shown in grey color.
2.3. PADRE

2.3.1. Methodology for PADRE

PADRE is an Eclipse-based framework that enables, in a unique environment, the performance antipatterns detection on UML-MARTE software models [PADRE-010] and the refactoring of the latter based on detection results. The role of PADRE in the WP3 conceptual framework is depicted in Figure 8. Specifically, the PADRE engine is able to detect and remove performance antipattern with respect to metrics gathered at runtime. In particular, a filled UML-MARTE is exploited to detect performance flaws.

![Positioning of PADRE in WP3 conceptual framework and tool set architecture](image)

Figure 8: Positioning of PADRE in WP3 conceptual framework and tool set architecture

**Figure 9** illustrates a typical Software Performance Engineering (SPE) process based on UML, in which we have plugged the *Performance Antipattern Detection and Model Refactoring Framework (PADRE)*.

![A typical Software Performance Engineering (SPE) process based on UML](image)

Figure 9: A typical Software Performance Engineering (SPE) process based on UML
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 Figure 9, namely to get performance indices and set the tagged values of the 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., EVL, EPL, and EWL) [25]. 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**: EPL allows to execute sequentially a set of antipattern detection rules and refactoring actions (see Figure 10). The process can be repeated once (i.e., standard mode) or until no more antipattern occurrences are found (i.e, iterative mode).

![Figure 10: An information dialog in PADRE](image)

**User-driven multiple refactoring sessions**: EVL allows to execute interactive antipattern detection and refactoring sessions (see Figure 11). 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.
**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 (see **Figure 12**). 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 [26].

---

**2.3.2. Guidelines for PADRE**

PADRE’s architecture is illustrated in **Figure 13**, which basically shows the main actors that it involves, together with its workflow.
The framework is centered on three performance antipatterns 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., $M_1, ..., M_{n-1}$, can be created until a software model that satisfies the performance requirements is obtained, i.e. $M_n$.

PADRE deals with UML models made out 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 13 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 13, 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, that are filled in with the outputs of the performance model solution.

A performance expert has to build the basic knowledge $K_{UML+MARTE}^{PERF}$, 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. A fragment of the UML software model is depicted in Figure 14. The UML Component Diagram in Figure 14 shows an excerpt of the static view of the application by means of software components (each providing a microservice) and their
interconnections. The UML Deployment Diagram in Figure 14 depicts the deployment view of the system, i.e., hardware nodes and how the application artifacts are allocated on them. More detailed guidelines can be found in the PADRE github repository³.

³ https://git.io/SeaLabAQ-padre
2.4. CHESS

2.4.1. 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 [26] UML editor and is made available as open source Eclipse-Polarsys [31] project.

For what concern the WP3 area, as stated in CHESS-100 purpose, CHESS tool, since its baseline version, is comprehensive of an ADA code generator to generate only the ADA infrastructural code, under Ravenscar Profile, based on system model elements such as: interfaces, component declarations, containers and connectors. The container code is designed to embed functional code, that is left for separate generation.

The code generator is a model to code transformation that turn the <PIM, PSM> model pair into source code. The Platform Specific Model (PSM) is automatically generated from the Platform Independent Model (PIM) instantiated on a given platform and enriched with a set of MARTE annotations defining the dependability and timing properties of the system.

Refer to D2.6 for an overall view of the CHESS methodological approach and guidelines. The CHESS-100 capability is not exploited in any MegaM@Rt2 UC but in several past projects like Concerto [32] and Contrex [33].

2.4.2. Guidelines for CHESS


2.5. XPM

2.5.1. Methodology for XPM

XMP uses the information contained in the XMCF file (configuration file of XtratuM [19]) to extract information about the inter partition communication (see Figure 15). This information is used to generate the code of the creation of the communication mechanisms (e.g. sampling or queueing channels) in the partition source code.
This is an example of XMCF with two partitions that communicate through a sampling channel.

```
<PartitionTable>
  <Partition id="0" name="Partition1" flags="system" console="Uart">
    <PhysicalMemoryAreas>
      <Area start="0x06000000" size="512KB"/>
    </PhysicalMemoryAreas>
    <PortTable>
      <Port name="Port1" type="sampling" direction="source"/>
    </PortTable>
  </Partition>
  <Partition id="1" name="Partition2" flags="system" console="Uart">
    <PhysicalMemoryAreas>
      <Area start="0x16000000" size="512KB"/>
    </PhysicalMemoryAreas>
    <PortTable>
      <Port name="Port2" type="sampling" direction="destination"/>
    </PortTable>
  </Partition>
</PartitionTable>

<Channels>
  <SamplingChannel maxMessageLength="32B" validPeriod="1s">
    <Source partitionId="0" portName="Port1"/>
    <Destination partitionId="1" portName="Port2"/>
  </SamplingChannel>
</Channels>
```

The automatic code generation involves the creation of the channel in the main function of the partitions, and the result would be for Partition1.c:

```c
#include <ARINC_653/apex_services_np.h>
#include <ARINC_653/apex_sampling.h>

int main(void) {
  SAMPLING_PORT_ID_TYPE port1;
  RETURN_CODE_TYPE ret;
  CREATE_SAMPLING_PORT("Port1", 32, SOURCE, SECONDS(1),&port1, &ret);
  if (ret!=NO_ERROR)
    printf("Error creating port: %d\n", ret);
}
```

And for Partition2.c:

```c
#include <ARINC_653/apex_services_np.h>
#include <ARINC_653/apex_sampling.h>

int main(void) {
  SAMPLING_PORT_ID_TYPE port2;
  RETURN_CODE_TYPE ret;
  CREATE_SAMPLING_PORT("Port2", 32, DESTINATION, SECONDS(1),&port2, &ret);
  if (ret!=NO_ERROR)
    printf("Error creating port: %d\n", ret);
}
```
2.5.2. Guidelines for XPM

This guideline can be found in the webpage of Fentiss megamart tools (http://wks.fentiss.com/megamart/).

Inputs: XtratuM configuration file (XMCF) generated by Xamber [21]. 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, the used execution environments and 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)

2.6. Papyrus Extension for AOM

2.6.1. Methodology for Papyrus Extension for AOM

This papyrus extension supports the realization, using automatic code generation, of cross-cutting concerns (such as logging, verification, etc.) as aspects, for a concrete target Aspect Oriented Programming (AOP) technology, such as AspectJ. In this way, these concerns are implemented (and modeled) in isolation from the target system code (and its representing model) so that they are not entangled within it.

This Papyrus extension has been applied to one of the Ikerlan (IKER) use cases, for injecting logging and message verification of the IOT Gateway component of their Skyline system. This tool satisfies the IKER_02 and IKER_11 requirements. This tool contributes to the KPI3.3.

In the context of MegaM@RT2, we have developed a Papyrus extension that supports the automated generation of aspect classes (as instances of AspectJ framework) and manages the injection of advices in order to tackle cross-cutting concerns. In the context of WP2 (see D2.6), we extended Papyrus to model cross-cutting concerns in UML models, using an Aspect Oriented Modeling (AOM) profile for AspectJ. In this context, we solve the problem of instantiating AOM-based Papyrus models into AspectJ, by using model to text (M2T) technologies, such as Acceleo, for code generation. This extension creates an AspectJ aspect class for each one in the target UML model that has been annotated as an aspect (using the AOP profile for UML). This approach largely simplifies the instantiation of AspectJ aspects from user’s UML models representing his/her applications or systems.

---

2.6.2. Guidelines for Papyrus Extensions

This Papyrus extension for AOM/AOP with AspectJ requires a UML model of the target system that includes a representation of AOM aspects, which are annotated with the AOM Profile for UML described in D2.6. See Figure 16 an example of UML aspects for logging and message verification for the IKER IoT Gateway.

![Figure 16: Aspect UML model for IKER IoT Gateway](image)

The way this Papyrus extension for AOM/AOP with AspectJ works is the following (see Figure 17):

1. The user searches for the UML model that describes his/her aspects for his/her target system in the Eclipse Project Explorer view and selects it.
2. The user opens the contextual menu on this model, by right-clicking the mouse and selects the menu entry: Acceleo Model to Text/Generate AspectJ Code
3. After few seconds, the user is notified that the AspectJ classes for his/her model aspects have been generated into the default folder src-gen
Generated AspectJ aspects are stored within the src-gen model (Figure 18). One aspect class is generated for each UML class annotated as an AOM aspect. See in Figure 19 an example of generated AspectJ class: an aspect for injecting logging.

```java
privileged aspect LoggingAspect {
    pointcut logging():
    void around()
    logging():
        System.out.println("logging");
        String name = thisJoinPoint.getSignature().getName();
        Object[] objects = thisJoinPoint.getArgs();
        String info = "Starting method : " + name + "\n\n >> Input :
        for (Object ob : objects) {
            info += ob.getClass().getName() + " = " + ob.toString();
        }
        PrintUtils.log(info);

        try {
            proceed();
            PrintUtils.log(name + " finish without exceptions");
        } catch (Exception e) {
            PrintUtils.error("" + name + " threw an exception");
            e.printStackTrace();
        }
}
```

Figure 19: generated aspect for logging concern for the IKER IoT Gateway
Additional information about how to install and use this tool can be found at: https://github.com/megamart2/tool-papyrus-extensions

2.7. AIPHS

2.7.1. Methodology for AIPHS

AIPHS, acronym of Adaptive Profiling HW Sub-system, is basically conceived to support designers on the development of On-Chip Monitoring Systems (OCMSs) able to satisfy given Monitorability Requirements, namely requirements about the possibility to observe the behaviour of a system with the goal of collecting metrics.

It is a flexible framework that targets SoCs implemented on Field Programmable Gate Arrays (FPGAs), or on Integrated Circuits (ICs) integrating some reconfigurable logics.

OCMSs developed with AIPHS can generate logs for timing performance measurements on targets with multi-core processors, running bare-metal and Linux based applications (e.g., logs for WCET analysis).

The relation of AIPHS with the WP3 conceptual framework and tool set architecture is shown in Figure 20, where red arrows highlight the Megamart2 framework relations, while green arrows show what AIPHS provides:

![Figure 20: Relation of AIPHS with the WP3 conceptual framework and tool set architecture](image)

AIPHS was applied to UC 04_TEK and 01_TRT. It implements AIPHS_10 (TRT_01), AIPHS-020 (TEK_07, TRT_01), AIPHS-030 (TRT_01), AIPHS-040 (TEK_07), AIPHS-050 (TRT_01), and AIPHS_60 (TRT_01). The last tool requirement was planned for the final tool release. AIPHS contributed to KPI 2.1 (REDUCTION OF VALIDATION EFFORT BY AUTOMATED TRACE COLLECTION AND ANALYSIS) and KPI 2.2 (REDUCTION OF TIME FOR MONITORS SETUP BY AUTOMATED PROBES INJECTION). The implementation of AIPHS requirements have been done in collaboration with UCAN. The collaboration between UCAN and UAQ has been described in D3.5, while the interoperability conceptual pattern will be presented in WP5 - D5.3.

AIPHS works internally by exploiting a generalization of the concept of monitoring among different OCMSs, by defining a general reference architecture that can be adapted to different applications. Details are reported in Deliverable D3.2 [27], Section 6.7.
2.7.2. Guidelines for AIPHS

AIPHS tool takes as input a set of monitorability requirements, a description of the platform to be monitored and an application to be monitored, and it provides, as output, a suitable monitoring system to satisfy those requirements.

The requirements are manually expressed, and they can be provided with a reference to a set of metrics supported by AIPHS: metrics for memory exploitation, metrics for HW communications, metrics for overall execution time and metrics for code coverage. More details on metrics are provided in Deliverable D3.2 [27], Section 6.7.

For example, the user can provide the following information as input:

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Platform description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement of overall execution time spent executing specific segments of code on a single core</td>
<td>Type of physical implementation: reconfigurable logic</td>
</tr>
<tr>
<td></td>
<td>Software platform: bare-metal</td>
</tr>
<tr>
<td></td>
<td>Hardware platform: Leon3</td>
</tr>
<tr>
<td></td>
<td>Allowed software overhead: 0 %</td>
</tr>
<tr>
<td></td>
<td>Coverage: architectural</td>
</tr>
<tr>
<td></td>
<td>Allowed hardware overhead: 30 %</td>
</tr>
<tr>
<td></td>
<td>Type of event: ID + timestamp</td>
</tr>
</tbody>
</table>

Supposing that the application to be monitored is executed on Leon3 processor, the output from AIPHS would be a monitoring system described in VHDL able to satisfy the requirements indicated above.

Manual steps are required to provide the requirements, filling opportune tables. In particular, the steps are as follows:

1) To provide the metrics to be monitored and the related platform information (as in the table above);
2) To take the resulting monitoring system provided by AIPHS and to implement it depending on its characteristics. In particular:
   a) If it is a hardware monitoring system described in VHDL, connect it within the rest of the hardware architecture and implement it using a suitable tool.
   b) If it is a software monitoring system, compile the modified source files.

For example, consider the following case, taken from 04_TEK use-case and manually provided:

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Platform description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement of overall execution time spent executing specific segments of code on a single core</td>
<td>Type of physical implementation: fixed</td>
</tr>
<tr>
<td></td>
<td>Software platform: bare-metal</td>
</tr>
<tr>
<td></td>
<td>Hardware platform: ARM Cortex M4</td>
</tr>
</tbody>
</table>
The application to be monitored is written in C code (with source code provided). The output from AIPHS is the list of application source files opportunely instrumented.

2.8. JTL

2.8.1. Methodology for JTL

JTL is an Eclipse EMF\(^7\)-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 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.

The role of JTL in the conceptual framework is depicted in Figure 21. Specifically, the JTL traceability engine is able to relate design models with execution traces by creating a traceability model. Such model contains links to elements in both design models and execution traces.

![Figure 21: Relation of JTL with the WP3 conceptual framework and tool set architecture](image)

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

\(^7\) https://www.eclipse.org/modeling/emf/
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\(^8\) 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 22. 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 that conform 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.

![Diagram](image.png)

**Figure 22: Process underlying the generation of traceability information**

The JTL tool components are:

- **Bidirectional Model Transformation Engine**: that allows to execute 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 execution and maintain them in dedicated models.

\(^8\) [https://www.mat.unical.it/dlv-complex](https://www.mat.unical.it/dlv-complex)

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

2.8.2. Guidelines for JTL

When executing a bidirectional transformation, JTL requires as input:

- **MM<sub>source</sub>**: a source metamodel in EMF format
- **M<sub>source</sub>**: a source model in EMF format
- **MM<sub>target</sub>**: 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<sub>target</sub>**, a target model in EMF format and conforming to **MM<sub>target</sub>**, as well as a traceability model. JTL can also be used to just generate traceability information by providing the **M<sub>target</sub>** model as additional input. In this case, the JTL traceability engine will only generate a traceability model containing the links that relates the elements in **M<sub>source</sub>** with the elements in **M<sub>target</sub>**.

JTL can generate target models as well as traceability models. As an example, Figure 23 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 23: 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](image-url)
Examples, tutorials and further documentation can be found on the JTL website\(^9\).

### 2.9. CertifyIt

#### 2.9.1. Methodology for CertifyIt

CertifyIt is a tool suite that automatically generates test cases based on a model of system requirements. It supports UML/OCL models as the specification modeling language, and generates test cases to cover all the identified requirements that are used as test objectives.

As illustrated in Figure 24, from UML/OCL behavioral models, CertifyIt therefore enables to:

1. Take as input a behavioral UML/OCL test model capturing requirements and functional aspects
2. Generate test cases from several criteria, based on behavioral requirements coverage
3. Manage traceability that gives functional coverage metrics
4. Publish abstract test cases for documentation or manual exploitation.
5. Publish test scripts into a testing environment or an execution language for automated execution.
6. Easily manage the evolution of the test model: update your model and Smartesting CertifyIt will generate the new test cases.

![Figure 24: Positioning of CertifyIt in WP3 conceptual framework and tool set architecture](image)

Figure 25 depicts the global process of the CertifyIt approach that makes it possible to generate test case specifications and derive test scripts from UML/OCL behavioral test models using an offline test strategy.

\(^9\) http://jtl.univaq.it
CertifyIt tool has been applied to UC 03_IKER and UC 06_BT. More details on the CertifyIt usage and the collaboration with IKER and BT could be found in D3.4 and D3.5, respectively. In particular, CertifyIt contributes to KPI_1.3 for IKER case study. The addressed UC requirement is IKER_09. It further contributes to BT use case requirements. More specifically, CertifyIt tool satisfies:

- **BT_10** The expectation is that MegaM@RT2 provides methods on: how to design requirement models such that test cases can be derived; how test cases for module testing can be derived; how test cases for higher level integration tests can be derived.
- **BT_13** The expectation is that MegaM@RT2 provides a method on how the generation of test scripts out of test cases can be automated.

It further contributed to the KPIs:

- **KPI 1.2:** Improvement of 22% in identification of design problems at model level compared to the current way of working.
- **KPI 5.2:** Quality Improvement in the range of 30% by improving conformance to specifications compared to the manual conformance check.

The specific details on the contribution to the KPIs and the case studies requirements will be discussed in the future WP5 deliverable, which will report on the second evaluation phase.

### 2.9.2. Guidelines for CertifyIt

From functional aspects of the system under test, the test model is developed using a UML subset, called UML4MBT [29] (UML for Model-Based Testing), in an eclipse-based modeling environment and is checked for consistency. This test model captures the expected behavior of the system under test, i.e. operations and functions, and specifies the requirements, i.e. constraints on data and function parameters, we want to test. Such a test model is an abstraction of the reality to support the validation engineer in his/her testing effort. Hence, the activity of creating the UML/OCL test models must always be evaluated against a testing goal: “Will this provide the relevant tests required for my system?”.
From this perspective, the validation engineer will make a number of choices while modeling its system, regarding the boundaries of the system, the inputs and outputs relevant for testing the system under test (SUT), what is the amount of information needed to model the behavior of the SUT. In this way, the objective of the validation engineer is not to model all the software system in its most accurate details, but, most of the time, to model just enough and rightly focused system behavior to leverage CertifyIt capabilities to generate relevant test cases.

This is why the use of Smartesting CertifyIt, and the use of models, have been considered as part of a larger perspective on how the target system must be tested, and as a result, the test model should not be considered as design model. It only describes the system as a black box and only formalizes parts of the system relevant for test generation.

On the test model, test selection criteria are chosen to guide the automated test generation so that it produces a test suite aligned with the test strategy. In this context, the system requirements are linked to the elements of the test model, and the coverage of these requirements drives the test generation algorithms. The precise and unambiguous meaning of the UML4MBT test model makes it possible to simulate the execution of the model, to use it as an oracle by predicting the expected output of the system under test, and finally to provide traceability matrix that gives a clear functional coverage metrics. In this way, one or more abstract test cases are produced. Such an abstract test case takes the form of a sequence of steps, where a step corresponds to an operation call representing either an action or an observation of the system under test. The resulting test suite can then be exported for documentation, typically as HTML pages, for specification, typically using XML-based format, and as executable scripts. This last export enables to automate the execution of the generated test cases and to collect the related verdict assignment.

Once the test suite has been generated, the test cases are run. Test execution may be manual, that is by a physical person, or may be automated by a test execution environment that provides facilities to automatically execute the tests and record test verdicts. For example, the tests can be generated into the Java language and automatically executed using the JUnit framework: each abstract test case defines a JUnit test case, while the set of test cases creates a JUnit test suite. To bridge the gap between abstract and executable data, an adapter file, containing the prototype of each operation of the system under test, allow the linking between abstract and concrete structures. The test automation engineer is in charge of implementing each operation and data of this interface. Moreover, the executable test suite embeds traceability information regarding the behaviors as well as the requirements specified in the test model. These traceability links enable engineers to have a precise feedback when a test fails. Further guidelines are provided in the documented embedded into the CertifyIt tool installation. The tutorial specific for MegaM@Rt is available on the Smartesting extranet for MegaM@Rt partners\(^\text{10}\) and the tool access request is given in the MegaM@Rt catalogue\(^\text{11}\).

2.10. MBeeTle

2.10.1. Methodology for MBeeTle

MBeeTle is a Smartesting CertifyIt extension to address online model-based testing. It thus extends the capabilities of the CertifyIt tool, which only supports offline testing approach.

\(^{10}\) [https://extranet.smartesting.com](https://extranet.smartesting.com)

\(^{11}\) [http://toolbox.megamart2-ecsel.eu/content/certifyIt](http://toolbox.megamart2-ecsel.eu/content/certifyIt)
The main difference between offline and online testing, i.e. between CertifyIt and MBeeTle, is defined by the way test cases are generated and executed at runtime. Indeed, using offline approach, the test cases, including operation calls and expected results, are all generated prior to be executed. Figure 26 summarizes these different artefacts.

![Figure 26: Positioning of MBeetle in WP3 conceptual framework and tool set architecture](image)

Therefore, in comparison with the CertifyIt tool, the main issue and added value addressed by MBeeTle to perform online test process consists in combining in a single process test case generation, test case execution and corresponding test results as depicted in Figure 27.

![Figure 27: Online testing approach using MBeeTle](image)

In this way, as the test cases are generated at the same time they are executed, it should be noted that MBeeTle supports innovative algorithms. They allow the test engine to perform on-the-fly dedicated test strategies to produce more relevant and more adapted scenarios regarding the current state or outputs of the system under test.
MBeeTle is compliant with and extends the offline testing capabilities of CertifyIt to:

- Support UML4MBT test models by offering the expressiveness required to model large systems and corresponding requirements.
- Generate test cases from adapted online testing strategies based on requirement coverage, but also regarding current execution results.
- Publish abstract test cases for documentation or manual use.
- Publish executable test scripts into a testing environment that makes it possible to perform automated on-the-fly and step-by-step runtime execution.
- Manage traceability between requirements and generated testing artefacts such as test case specifications, test scripts and related test execution results, including the export of traces into the Common Trace Format (CTF) developed for MegaM@Rt2.
- Provide efficient feedback to end-users to help them validate the system under test. This feedback is thus gathered at runtime and displayed in a dedicated live dashboard, which is now performed with a CTF export that later can be used by analysis tools, such as those provided in WP4.

MBeeTle is experimented with UC 03_IKER and UC 06_BT. More details on the MBeeTle usage and the collaboration with IKER and BT could be found in D3.4 and D3.5 respectively.

In particular, MBeeTle contributes to KPI_2.1 for IKER case study. The addressed UC requirements are IKER_9 & IKER_21. It further contributes to BT to same use case requirements (BT_10 and BT_13) and KPIs (KPIs 1.2 and 5.2), as CertifyIt. The specific contribution of MBeeTle to the KPIs and the case studies requirements will be discussed in the next WP5 deliverable, which will report on the second evaluation phase.

### 2.10.2. Guidelines for MBeeTle

MBeeTle uses the same UML4MBT test models as CertifyIt. As described in the previous methodology section, MBeeTle provides complementary but fundamentally different test generation strategies from CertifyIt. Therefore a user has to follow several preparation activities to run MBeeTle:

- The same model and adaptation layer are used for offline testing with CertifyIt and online testing with MBeeTle. However, depending on the configuration parameters, MBeeTle can run for a long time and generate extremely large set of tests. For this reason, it is recommended to perform first offline testing in order to tackle all the issues about the adaptation layer and the concretizations between the abstract entities of the test model and the corresponding concrete execution elements. In this way, these issues will be systematically addressed on a smaller set of tests but that exercise functionally the tested perimeter of the system.
- MBeeTle stimulates the system randomly and on-the-fly, these operations that may potentially damage the system under test (or put it in a dead state) should be forbidden to MBeeTle. For this reason, a user can configure, in the modelling tool, a dedicated set of operations available for MBeeTle.
- MBeeTle offers a panel of possible configurations of its test strategies. Moreover the user can configure to require from MBeeTle to leave the testing process when one or several stop conditions (requirement coverage, coverage of corner cases, length of test cases etc.) are met (obviously, the user can also decide all the time to stop the testing process by pressing on a dedicated button). It should be noted that these stop conditions should be carefully analyzed and defined to be able to achieve the testing objectives of the project.
Once these preparation activities for online testing are performed, to run MBeeTle for it is necessary to provide the following steps:

1. Select a CertifyIt behavioral UML/OCL test model
2. Select an online test strategy: random, tag coverage, flower etc.
3. Configure the parameters for the test strategy (for instance the test stop conditions: test generation time, length of the test steps, duration of the online test generation etc.)
4. Select an online publisher
5. Start the online generation

To run MBeeTle for online testing, it is required that the existing publisher for CertifyIt implements the CertifyIt online publisher’s API, provided with the tool. The documentation for migrating the existing CertifyIt publisher for online testing with MBeeTle is provided within the tool. This documentation has been updated in respect to the MegaM@Rt2 use case specificities. Furthermore, a MegaM@rt2 tutorial for MBeeTle is available for download from the Smartesting extranet\textsuperscript{12} and access for downloading the tool is available at the MegaM@Rt2 catalogue\textsuperscript{13}.

Finally, when starting a new online testing session, we can import an already existing test configuration that had been exported during a previous session. To support this feature, a test configuration Import/Export manager has been developed.

### 2.11. Convex Hull

#### 2.11.1. Methodology for Convex Hull

Consider a practical example which an engineer may encounter. The engineer has an abundant amount of data, but only a small subset of data points of which the engineer specifically knows that indicates the correct behavior of the information technology system that produces time-series data. The positive samples are read by the Convex Hull tool in addition to the plethora of unknown data. The Convex Hull tool groups all the unknown data points according to the positive samples and indicates the statistically optimal boundary for this group within the provided data. Additionally, the Convex Hull indicates the statistically optimal boundary (or boundaries) in the rest of the data. This is useful in error location within a huge quantity of data. The positive data points points could be automatically ruled out and the median/average/descriptive data points within the other groups could be examined to pinpoint, for instance, an error or error group within the data.

In detail, Convex Hull attempts to optimize the class boundary in positive-unlabelled learning through the utilization of self-learning with Perceptually Important Points. Given data of which only a few positive samples are available, Convex Hull exploits the positive samples and analyzes a plethora of unlabelled data. The analysis results in a statistically optimal division between the positive and negative class. The Convex Hull tool is a stand-alone tool which can exploit data which is accessible in, for instance, .csv, .xls, etc. format. In comparison to other state-of-the-art solutions, the Convex Hull tool does not require setting any hyperparameters.

In machine learning, Positive-unlabelled learning is a special case within semi-supervised learning. In positive-unlabelled learning, the training set contains some positive examples and a set of unlabelled

12 https://extranet.smartesting.com
13 http://toolbox.megamart2-ecsel.eu/content/mbeetle
examples from both the positive and negative classes. Positive-unlabelled learning has gained attention in many domains, especially in time-series data, in which the obtainment of labelled data is challenging. Examples which originate from the negative class are especially difficult to acquire. Self-learning is a semi-supervised method capable of PU learning in time-series data. In the self-learning approach, observations are individually added from the unlabelled data into the positive class until a stopping criterion is reached. The model is retrained after each addition with the existent labels. The main problem in self-learning is to know when to stop the learning. There are multiple, different stopping criteria in the literature, but they tend to be inaccurate or challenging to apply. We propose a novel stopping criterion, which is called Peak evaluation using perceptually important points, to address this problem for time-series data. Peak evaluation using perceptually important points is exceptional, as it does not have tunable hyperparameters, which makes it easily applicable to an unsupervised setting. Simultaneously, it is flexible as it does not make any assumptions on the balance of the dataset between the positive and the negative class (see Figure 28 and Figure 29).

Figure 28: The original time-series data (X) and the identified Perceptually Important Points (PIPs). The dotted line (Z) discriminates the natural cubic spline between the PIPs. The red circle indicates the optimal PIP.

Figure 29: An illustration of the scaled error series (e) of the difference of the original data and the natural cubic splines. The elbow (the optimal value) indicates the location at which the difference between the scaled error series (e) and the straight line is the greatest.
2.11.2. **Guidelines for Convex Hull**

The Convex Hull tool [9] is self-contained. It only requires machine-readable data (e.g., .csv, .xls, or another similar format) which contains a subset of positive examples. The final result is the distinction of the optimal point (or optimal points.) Additionally, graphs, statistics, diagrams, etc. can be created according to the requirements of the user.

In detail, the Convex Hull tool reads the input files (e.g., .csv, .xls, or another similar format) which contains the data (of which there is a specific set of data which contains only positive sample (example of a behavior which is desirable to identify)) and completely unknown data which is evaluated by the Convex Hull tool to indicate which data points statistically optimally belong to the positive samples. Additionally, the Convex Hull tool also identifies the statistically optimal boundaries (which could be interpreted as group boundaries) of the unknown data. The final outcome is the categorization of positive samples in relation to the unknown data points and the boundaries of the rest of data points in comparison to each other.

2.12. **Clusterability**

2.12.1. **Methodology for Clusterability**

A definite problem in the utilization of state-of-the-art clustering tools is that the tool assumes that the data is clusterable, i.e., clusters can be formed from the data. This is not always the case, for instance, in a situation in which the data contains merely noise. Additionally, many state-of-the-art clustering tools require the user to specify the amount of clusters which should exist in the data. This is yet another huge predicament. The Clusterability tool automatically analyzes the given data (for instance, in .csv, .xls., or similar, machine-readable format,) then outputs an indication of whether the data is statistically clusterable or not, and if the data is clusterable, then the Clusterability tool indicates the amount of statistically discovered clusters (see [Figure 30](#) and [Figure 31](#)).

Clustering is used to gain an intuition of the structures in the data. Most of the current clustering algorithms produce a clustering structure even on data that do not possess such structure. In these cases, the algorithms force a structure in the data instead of discovering one. To avoid false structures in the relations of data, a novel clusterability assessment method called density-based clusterability measure is created. It measures the prominence of clustering structure in the data to evaluate whether a cluster analysis could produce a meaningful insight to the relationships in the data. This is especially useful in time-series data since visualizing the structure in time-series data is hard. The performance of the clusterability measure is evaluated against several synthetic data sets and time-series data sets, which illustrate that the density-based clusterability measure can successfully indicate clustering structure of time-series data (see [Figure 32](#)).
Figure 30: Unclusterable data set (the letter 'O'.)

Figure 31: Clusterable data set (called 'Shapes'.)

Figure 32: the output of the Clusterability tool: a) the unclusterable data set (the main is dominant) and b) the clusterable data set (the three subbranches are dominant.)
2.12.2. Guidelines for Clusterability

The Clusterability tool [10] is self-contained. It only requires machine-readable data (e.g., .csv, .xls, or another similar format) which contains data. The final result is the distinction on whether the data is clusterable or not. If the data is clusterable, then the Clusterability tool also provides the optimal amount of clusters numerically. Additionally, graphs, statistics, diagrams, etc. can be created according to the requirements of the user.

In detail, the Clusterability tool reads the input files (e.g., .csv, .xls, or another similar format) which contains the data which is evaluated by the Clusterability tool to statistically evaluate whether the data is clusterable or not. If the data is clusterable, then the Clusterability tool indicates the statistically optimal value of located clusters within the data. The final outcome indicates whether the data is clusterable or not. If the data is clusterable, then the Clusterability tool also provide a numerical value on the amount of clusters.

2.13. Moka Extension for Logging

2.13.1. Methodology for Moka Extension for Logging

The MegaM@Rt2 Moka14 extension for logging complement the visualization of a fUML15 model simulation with logging and their serialization into the file system for post-mortem analysis. Current Moka version only enables a visual (i.e. using a UML graphical representation) simulation, without showing detailed execution traces, for instance, including the values and types of the inputs and outputs of the actions under execution (e.g. activity diagrams with behaviors, actions, etc). In previous MOKA version, this inspection of the execution details was only possible by debugging the simulation, using breakpoints to stop the execution, and only the objects within the execution context were visible. Moreover, the behaviors associated with states in a state machine were not debuggable. Optionally, the user could visualize specific information of the simulated system model by injecting ALF16 code for logging or by using behavior primitives to print concrete model values into the fUML view. ALF approach is not stable with latest Moka version and using printing primitives would make models much more complex for execution visualization (apart from being a very time consuming and prone to error work).

This Papyrus extension has been applied to one of the IKER use cases, for simulating their model of the IOT Gateway component of their Skyline system. The tool satisfies IKER_02 and IKER_12 requirements and contributes to KPI2.1.

Moka original plugin has been extended to receive notifications of the execution of the fUML models. Information about fUML model execution is collected from notifications to create a model execution pattern, which is presented to the user in a specific Moka Logging view. Moreover, these fUML execution logs are serialized into the filesystem, for each execution, to enable post-mortem analysis.

14 https://marketplace.eclipse.org/content/papyrus-moka
15 https://www.omg.org/spec/FUML/About-FUML/
16 https://www.omg.org/spec/ALF/About-ALF/
2.13.2. Guidelines for Moka Extension for Logging

Moka extension for logging requires an executable fUML model, which should have been modeled using Papyrus (see D2.6 for Papyrus and Moka support for fUML modeling). For example, Figure 33 shows the state machine diagram for the IKER IoT Gateway of the Skyline system, and Figure 34 shows an activity diagram (within the same model) that designs a scenario for testing the Gateway. This fUML model contains a state machine diagram that describes the main Gateway states and their transitions, a number of (more than 50) complex activity diagrams that describe the state behaviors (e.g. entry, do, exit) and behaviors of other Gateway subcomponents. This model must be a valid (i.e. verified, free of errors) fUML compliant model.

Figure 33: State Machine model for IKER IoT Gateway

Figure 34: IKER Skyline behavior for test scenario

To simulate/execute this fUML model and obtain execution traces, the user has to proceed with the following procedure:

1. The user opens the fUML model into the Eclipse IDE, in the Papyrus perspective.
2. The user configures a Moka launch configuration to run the model, specifying:
   a. The fUML model location (within the Eclipse workspace)
   b. The activity to execute
c. The Moka engine to use for simulation. In case the fUML requires to execute a model containing a state machine, the \texttt{org.eclipse.papyrus.moka.fuml.statemachines} engines should be used. See Figure 35 for an example of MOKA launch configuration for the IKER IoT Gateway.

![Figure 35: Moka launch configuration wizard for IKER Gateway](image)

3. The user pushes the run button to start the model execution with Moka. Alternatively, the user can launch a pre-existing Moka launch configuration by selecting a concrete configuration from the list of available ones.

4. Moka execution can be further configured in the \texttt{Moka Animation Configuration} view (see Figure 36), where users can specify:
   a. The simulation animation speed (e.g. animation delay)
   b. Whether or not to open automatically the diagrams currently under execution

![Figure 36: Moka Animation configuration view](image)

5. Soon after the fUML model simulation starts, a new Moka Execution view is opened. This view shows fUML model execution logs (Figure 37). By double-clicking in a log entry, log details are popped up (Figure 38).

![Figure 37: Moka Execution view showing logs](image)
6. Additionally the user can inspect similar execution logs in the fUML console (Figure 39).

7. Upon the finalization of the fUML model simulation, a logging file is saved in the simulations folder (Figure 40).

This Moka extension generates detailed fUML execution/simulation traces (logs) in different views (Moka Execution view, fUML Console) and serialized into disk (in simulation folder) for post-mortem analysis. Figure 41 shows an excerpt of traces stored in the filesystem for IKER Gateway simulation.

Figure 38: Log detail in Moka Execution view

Figure 39: Moka execution logs shown in fUML console

Figure 40: Moka simulation trace files

Figure 41: Serialized Moka execution traces
2.14. MATERA2-MBMÅA

2.14.1. Methodology for MATERA2-MBMÅA

In many situations, real-time systems are deployed in unpredictable environments, where the behavior of the systems is nondeterministic and proving its correctness depends on certain assumptions of the environment which are only available at runtime [17]. In addition, some faults only manifest when the system is deployed in production environments or when the system runs for longer periods of time [18]. We present a runtime verification approach which uses UPPAAL timed automata (UTA) [13] to specify the behavior of the IUT and of its environment. The properties of the model are checked via a set of verification rules in UPPAAL. The Distributed TRON (DTRON) [20] online testing tool is used as a monitor to compare the observed behavior of the IUT with the expected input/output traces created from the model. MATERA2-MBMÅA is capable to detect faults that are not detected by traditional testing techniques.

MATERA2-MBMÅA was applied to UC 06_BT. It implements MATERA2-070 in the baseline (M0) release and MATERA2-030, MATERA2-040, and MATERA2-090 in the final (M32) release. MATERA2-070 satisfies BT_08, BT_13, and BT_14. Moreover, MATERA2-030 and MATERA2-040 also satisfy BT_08. MATERA2-090 does not map to UC 06_BT in WP3, but in WP2 it satisfies BT_06. MATERA2-MBMÅA was also planned to implement MATERA2-050 in the final (M32) release, however MATERA2-050 does not map to UC 06_BT. Moreover, MATERA2-050 was cancelled due to lack of data and test infrastructure in the case studies and the effort was allocated to other tool purposes. Therefore, in UC 06_BT, MATERA2-MBMÅA satisfies BT_06, BT_08, BT_13, and BT_14.

In our approach, different artifacts are derived from the given requirement specifications of the IUT and are used to monitor its behavior as illustrated in Figure 42. The approach is divided into two main stages: Modeling and Monitoring. In the former stage, the system is specified as an UTA model from the given requirements and specification documents. The model is partitioned into system and environment. Each partition can be composed of several processes. In the later stage, we use the DTRON tool to monitor the functionality of the IUT against the UTA model at runtime. The tool connects to the IUT via the Spread network [14] and the adapter, and observes the output messages sent by the adapter. We use a set of mappings to associate model fragments stemming from requirements with entries of the log file. The information is extracted using regular expressions. The result of mapping will be read by the adapter and converted into output messages sent to the UTA model. Whenever a message is received by DTRON a synchronization of the prefixed channels occurs.
Our approach gives one the possibility to track the coverage level of the model with respect to acceptance criteria and model structure. However, tracking the coverage level does not come for free, as introducing tracking variables and additional processes (and clocks) in the UTA model has an impact on the state space. Further details can be found in [6].

2.14.2. Guidelines for MATERA2-MBMÅA

Following guidelines describe the installation steps as well as the tool chain dependencies.

**Tool chain dependencies**

- Download and install Jython 2.5.3 (http://www.jython.org/downloads.html).
- Java SE Runtime Environment 6 or later

**SPREAD**

- Download the binaries of the Spread from http://www.spread.org/download.html
- Unpack the spread-bin-4.x.tar.gz file into the target directory and goto bin directory to run the Spread

**UPPAAL TRON**

- Download TRON 1.5 from http://people.cs.aau.dk/~marius/tron/download.html#download
Dtron


- Set an environment variable TRON_HOME to the directory of UPPAAL TRON e.g. export TRON_HOME=/home/test/uppaal-tron-1.5-linux

- Usage
  - `$ java -jar dtron.jar -f <FILE> -o <UNITS> [-s <HOST>] -u <MSEC> -p <PORT>`
  - `-f,--file <FILE>` UPPAAL model file to be used.
  - `-o,--timeout <UNITS>` TRON timeout in time-units.
  - `-s,--spread <HOST>` Spread host address, with port (default: localhost)
  - `-u,--timeunit <MSEC>` TRON timeunit in msec.
  - `-p,--port <PORT>` UPPAAL Reporter port (default: 666)

Test Tracker

- Usage
  - `$ java -jar TestTracker-0.2.jar -m <file> -coverageTypes <types> -t <templates> -outputFile <file> -details`
  - `-channel <channel>` Channel to listen for messages on
  - `-coverageTypes <coverageTypes>` Requirement types, separated by ",."
    - Available types are: edge, edge-pair, , user-story
  - `-details` Set this flag to print a more detailed coverage report.
  - `-fileOutput` Enable reporting to file
  - `-help` Print a help message and exit.
  - `-m <modelFile>` UTA model file with counters on edges.
  - `-outputFile <outputFile>` File name for coverage output, use auto for automatic file name generation
  - `-silent` No output to console
  - `-spread <spread>` Host of the spread network. (default: localhost)
  - `-t <templates>` Provides templates (UTA) names for test coverage. Separated by ","
  - `-time <time>` Time interval of coverage output in ms. (default: 5000)

Adapter

- Dependencies
  - Dtron jar file should be set in the python path.

Steps to run the tool chain

1. Run Spread with following command:
   a. `$ spread -n localhost`
2. Run dtron to simulate the model:
   a. `$ java -jar dtron-4.9.1.jar -f CLIC.xml -o 10000 -u 1000000 -p 6666`
      where:
      - `jar dtron.4.12.jar` dtron java executable file
-f CLIC.xml UPTA model file
-o 10000 simulation timeout in timeunit
-u 1000000 defined here that 1 time unit should be equal to 1 second
-p 6666 UPPAAL Reporter port

PS: There is no need to specify the Spread host address if the spread and dtro are running on the same machine.

3. Run Test Tracker with desired coverage criteria:
$ java -jar TestTracker-0.2.jar -m CLIC.xml -coverageTypes user-story -t IOAC -outputFile report.txt -details
where:
-m CLIC.xml UPTA model file with counters on edges.
-coverageTypes user-story Coverage criteria is user-story
-t IOAC IOAC template for test coverage.
-outputFile <outputFile> File name for coverage output, use auto for automatic file name generation
-details Set this flag to print a more detailed coverage report.

4. Set following parameters in config.py file to configure the adapter:
   a. speed_factor adjust the time scaling (default : 1 )
   b. log_path path of BTS log file
   c. counter_timeout counter timeout

5. Run adapter with the following command:
$ jython -Dpython.path=dtron-4.9.1.jar -Dpython.cachedir=/tmp/jython Main.py
where:
-Dpython.path set the path for dtron library
-Dpython.cachedir the directory to use for caches
Main.py start the adapter

Example

The following figures show an example of a model of a sensor group (S1), fan speed controller (FSC), and environment model from Temperature Control System presented in [6].

The sensor group 1 is shown in Figure 43. Initially, the control system sends the request to initiate the registration (o_S1_req!) of sensor group 1 and waits for a response message (o_S1_res?) from the implementation. The response message will always be accompanied by a preliminary temperature indication (o_S1_ind) for each sensor in the group. Following a successful registration (S1_reg = true), the sensor will enter the polling mode (S1_poll = true) where the temperature of the sensor group will be indicated every polling interval (S1_poll_Int).

![Figure 43: model for sensor group S1.](attachment:image.png)
The behavior of the fans (Figure 44) is similar to the one of the sensors: on startup, the maximum fan speed is requested from HAL (o_fan1_max_speed_req!) followed by a confirmation response (o_fan1_max_speed_res?). Subsequently, FCS checks if it is able to control the fans by setting their speed to a specific value (o_fan1_speed_req!) and waits for a confirmation that the fan speed request has been received (o_fan1_speed_res!). Immediately after, implementation reports the current fan speed for the group (o_fan1_speed_ind?). When the procedure is completed, the fan registration is considered complete (FANreg = true) and the fan enters the polling mode (FANPoll=true).

![Figure 44: model for Fan Control system.](image)

The environment model (Figure 45) is a canonical model containing all the counterparts of the channel synchronizations in the temperate control system processes. Any synchronization can occur at any moment, if the state of the UPTA model allows it.

![Figure 45: environment](image)

Additionally, we need execution trace of IUT referred as a log file which contains timed stamped entries. An expert from the execution trace of IUT is shown in Figure 46.

```
005469  01:01  01:41:10, SensorGroup1: temperature of sensor 1 is 45
005470  01:01  01:41:10, SensorGroup1: temperature of sensor 2 is 47
005471  01:01  01:41:10, SensorGroup1: temperature of sensor 3 is 47
005472  01:01  01:41:10, SensorGroup1: temperature of sensor 4 is 44
005473  01:01  01:41:10, SensorGroup1: temperature of sensor 5 is 47
005474  01:01  01:41:10, SensorGroup2: temperature of sensor 4 is 44
005475  01:01  01:41:10, SensorGroup2: temperature of sensor 5 is 47
005476  01:01  00:41:11, System1: Sending FAN_SPEED_REQ
005554  01:01  01:41:20, SensorGroup1: temperature of sensor 1 is 50
005555  01:01  01:41:20, SensorGroup1: temperature of sensor 2 is 52
005556  01:01  01:41:20, SensorGroup1: temperature of sensor 3 is 49
```

![Figure 46: Execution trace of IUT](image)
Our tool produces three reports: (1) Abstract report, (2) Concrete report, and (3) Coverage report as shown in Figure 47, Figure 48, and Figure 49, respectively.

![Figure 47: Abstract report](image)

![Figure 48: Concrete report](image)

![Figure 49: Coverage report](image)
2.15. MATERA2-MBPeT

2.15.1. Methodology for MATERA2-MBPeT

Performance characteristics such as throughput, response times, and resource utilization are crucial quality attributes of such applications and systems. The goal of performance testing is to determine how well the system performs in terms of responsiveness, stability, and resource utilization under a particular synthetic workload in a controlled environment. The synthetic workload should mimic the expected workload as closely as possible, once the system is in operational use, otherwise it is not possible to draw any reliable conclusions from the test results.

MATERA2-MBPeT [1] generates synthetic workload using probabilistic models for the user profiles which simplify the creation and updating of test specifications. Owing to a certain level of randomness introduced by the probabilistic models, synthetic workload is closer to the real workload as compared to the workload generated from static scripts, as shown in Figure 50.

Figure 50: MATERA2-MBPeT approach

MATERA2-MBPeT has a distributed architecture. It consists of two types of nodes: a master node and slave nodes. A single master node is responsible for initiating and controlling multiple remote slave nodes. Slave nodes are designed to be identical and generic, in a sense that they do not have prior knowledge of the SUT, its interfaces, or the workload models. That is why for each test session, the master gathers and parses all the required information regarding the SUT and the configuration for each test session and sends that information to all the slave nodes. Once all slaves have been initialized, the master begins the load generation process by starting a single slave while rest of the slaves are idling.

MATERA2-MBPeT was applied to UC 05_NOK. It implements MATERA2-070 and MATERA2-080 in the baseline (M0) release and MATERA2-030 and MATERA2-060 in the final (M32) release. MATERA2-070 satisfies NOK_25 and NOK_26. MATERA2-030 and MATERA2-080 satisfy NOK_25. Moreover, MATERA2-060 satisfies NOK_22, NOK_23, NOK_24, and NOK_25. MATERA2-MBPeT was also planned to implement MATERA2-050 in the final (M32) release, which maps to NOK_25. However, MATERA2-050 was cancelled due to lack of data and test infrastructure in the case studies and the effort was allocated to other tool purposes. As a result, MATERA2-MBPeT does not provide a monitoring tool to predict future problems. In UC 05_NOK, MATERA2-MBPeT satisfies NOK_22, NOK_23, NOK_24, NOK_25, and NOK_26.
2.15.2. Guidelines for MATERA2-MBPeT

The tool takes as input a set of Probabilistic Timed Automata (PTA) \cite{12} user models, adapter, and test configurations.

User Models

The user models (as shown in Figure 51) can be built either by inspecting the requirements of the SUT or Service Level Agreements (SLAs) \cite{7}, or by analyzing the historical usage of the system \cite{2,8}.

![Figure 51: Example of a user model in PTA](image)

Each user model defined in DOT format. For example, the user model in Figure 51 can be represented in DOT language in the following way:

```dot
digraph user_model {
    0 -> 1 [label = "1 / 0 / homepage()" ];
    1 -> 2 [label = "0.6 / 3 / browse()" ];
    1 -> 3 [label = "0.4 / 3 / search()" ];
    2 -> 2 [label = "0.65 / 3 / browse()" ];
    2 -> 4 [label = "0.35 / 3 / details()" ];
    4 -> 5 [label = "0.8 / 4 / comment()" ];
    4 -> 6 [label = "0.2 / 3 / exit()" ];
    3 -> 4 [label = "0.7 / 3 / details()" ];
    5 -> 6 [label = "1.0 / 3 / exit()" ];
    3 -> 6 [label = "0.3 / 3 / exit()" ];
}
```

A user model must have one initial state, which has no incoming edges. The initial state will be used as a starting state during simulation. Each transition has three attributes: probability, think time and action. For example, the following transition in the above model:
2 -> 4 [label = "0.35 / 3 / details()"];

has the following components:

- Probability: 0.35
- Think time: 3 seconds
- Action: details()

Adapter

This is a case-study specific module which is written in Python language. It is used to communicate with SUT. It translates each action interpreted from the PTA model into a form that is understandable by the SUT, for instance a HTTP request. It also parses the response from the SUT and measures the response time. For example, the following snippet of the adapter implements the *homepage* action in Figure 51:

```python
from petadapter import *

class Generic_Adapter(AbstractAdapter):
    def __init__(self, *arg, **kwarg):
        AbstractAdapter.__init__(self, *arg, **kwarg)

@action("homepage")
def open_homepage(self, username, user_id, parameters):
    url = "https://megamart2-ecsel.eu/"
    res = self.session.get(url)
    repeat = False  # Repeat this action
    return res, repeat
```

Test configurations

It is a collection of different parameters that are defined in a *settings.py* file, which is a case-study specific. The file specifies the necessary information about the case-study and this information is later used by the tool to run the experiment. These are some of the parameters:

- Test duration: It defines the duration of a test session in seconds.
- Ramp function: This function specifies the desired number of concurrent virtual users at any given moment during the test session.

Running MATERA2-MBPeT

1) Download the binaries of MATERA2-MBPeT
2) Execute the master node of MATERA2-MBPeT by running *mbpet_gui* or *mbpet_cli* for GUI or command-line mode, respectively. For example:
   
   $ ./mbpet_cli <Project folder Path>
   
   e.g., $ ./mbpet_cli /home/projects/test

   The *project folder* contains the user models, the *adapter.py* file, and the *settings.py* file.
3) Execute the master node of MATERA2-MBPeT by running *mbpet_slave* in command-line like:
   
   $ ./mbpet_slave <MATERA2-MBPeT Server IP address:Port>
   
   e.g., admin@mbpet:$ ./mbpet_slave localhost
Test Report

The tool generates the load against the system under test and produces a test report in HTML format at the end of a test session, which contains response times of the actions (see Figure 52), resource utilization (see Figure 53), throughput (see Figure 54), error rates (see Figure 55), etc.

### AVERAGE/MAX RESPONSE TIME per METHOD CALL ###

<table>
<thead>
<tr>
<th>Method Call</th>
<th>NON-BIDDER_USER (%)</th>
<th>PASSIVE_USER (44.0 %)</th>
<th>AGGRESSIVE_USER (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average (sec)</td>
<td>Max (sec)</td>
<td>Executions #</td>
</tr>
<tr>
<td>SEARCH(STRING)</td>
<td>1.03</td>
<td>16.96</td>
<td>907</td>
</tr>
<tr>
<td>BROWSE()</td>
<td>1.39</td>
<td>19.37</td>
<td>3687</td>
</tr>
<tr>
<td>GET_BIDS(ID)</td>
<td>0.97</td>
<td>18.61</td>
<td>3004</td>
</tr>
<tr>
<td>BID(ID,PRICE,USERNAME, PASSWORD)</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>GET_AUCTION(ID)</td>
<td>0.83</td>
<td>19.36</td>
<td>4828</td>
</tr>
</tbody>
</table>

Figure 52: Average and Maximum response time of SUT per action

Figure 53: SUT CPU and memory utilization w.r.t number of concurrent users
2.16. MATERA2-PerfXGA

2.16.1. Methodology for MATERA2-PerfXGA

The poor performance of web-based systems can negatively impact the profitability and reputation of the companies that rely on them. Finding those user scenarios, which can significantly degrade the performance of a web application, is very important in order to take necessary countermeasures, for instance, by allocating additional resources. Furthermore, one would like to understand how the system under test performs under increased workload triggered by the worst-case user scenarios.
In the article [3], we have suggested a performance space exploration approach for inferring the worst-case user scenario in a given workload/user model which has the potential to create the highest resource utilization on the system under test with respect to a given resource, as shown in Figure 56. We propose two alternative methods: one which identifies the exact worst-case user scenario of the given workload model, but it does not scale up for models with a large number of loops, and another one which provides an approximate solution which, in turn, is more suitable for models with a large number of loops.

MATERA2-PerfXGA was applied to UC 05_NOK. It implements MATERA2-030 and MATERA2-060 in the final (M32) release. MATERA2-030 satisfies NOK_25, while MATERA2-060 satisfies NOK_22, NOK_23, NOK_24, and NOK_25. Therefore, in UC 05_NOK, MATERA2-PerfXGA satisfies NOK_22, NOK_23, NOK_24, and NOK_25.

Figure 56: PerfXGA approach

2.16.2. Guidelines for MATERA2-PerfXGA

The tool requires two inputs: workload/user model as a Markov Chain (MC) Model [28] and the average resource utilization of every action in the given model. Informally, we have extended MC with two additional labels on the edges: probability value and think time. The probability value specifies the chances of that particular edge being chosen according to a probability mass function, whereas the think time represents the amount of time that a user waits between two consecutive interactions. In addition, each state in the MC model is tagged with an action specifying the interaction between the user and the SUT. An action specifies either an HTTP request or a set of HTTP requests that the user sends to the SUT whenever the corresponding state is visited. The MC model in Figure 57 shows a workload model of an auctioning web application, which allows registered users to search, browse, and bid on auctions that other users have created. For instance, after performing a browse(), the user can execute either get auctions() action with a probability of 0.87 (after waiting for 3 seconds) or exit() with a probability of 0.03 after waiting for 2 seconds. In the model, start() and exit() are pseudo-states which are only used to indicate the initial and the optional final state of the model, respectively, and they cause no interaction with the SUT. There are different ways to obtain such workload models either by inspecting the requirements of the SUT or Service Level Agreements (SLAs) [7], or by analyzing the historical usage of the system [2, 8].
In order to compute which path of the model will create the highest resource utilization on the server, we need to know what is the average resource utilization on the server corresponding to each state. For this purpose, we benchmark each action defined in a given workload model with respect to different resource types (e.g., CPU, memory), by executing the model against the SUT for a given period of time. In this work, we record the resource utilization of each action via code instrumentation; however, other means of collecting the resource utilization can be used in case one does not have access to the source code of the SUT. During the test session, each action in the model is executed several times. For each action, we calculate the average utilization of the selected resource type for all its executions. For instance, we have executed the workload model in Figure 57 for 60 seconds against the implementation discussed in the previous sections. Table 2 summarizes the resulting average CPU for each action in the model.

<table>
<thead>
<tr>
<th>Action</th>
<th>CPU (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_bids(id)</td>
<td>0.083</td>
</tr>
<tr>
<td>search(string)</td>
<td>0.088</td>
</tr>
<tr>
<td>browse()</td>
<td>0.178</td>
</tr>
<tr>
<td>get_auction(id)</td>
<td>0.072</td>
</tr>
<tr>
<td>bid(id,price,username,password)</td>
<td>0.578</td>
</tr>
</tbody>
</table>

MATERA2-PerfXGA produces the worst-case user scenario in a given workload/user model. For example, we provide the workload model in Figure 57 and the average CPU utilization of each action listed in Table 2 to the tool, and get the most expensive path in the model (shown in Figure 58) with respect to the CPU utilization.
2.17. MATERA2-iPerfXRL

2.17.1. Methodology for MATERA2-iPerfXRL

Performance bottlenecks resulting in high response times and low throughput of software systems can ruin the reputation of the companies that rely on them. Almost two-thirds of performance bottlenecks are triggered on specific input values. However, finding the input values for performance test cases that can identify performance bottlenecks in a large-scale complex system within a reasonable amount of time is a cumbersome, cost-intensive, and time-consuming task. The reason is that there can be numerous combinations of test input values to explore in a limited amount of time.

We have presented MATERA2-iPerfXRL[4], a novel approach for finding those combinations of input values that can reveal performance bottlenecks in the system under test. Our approach uses reinforcement learning to explore a large input space comprising combinations of input values and to learn to focus on those areas of the input space, which have a high chance of triggering performance bottlenecks, as shown in Figure 59.

Figure 59: MATERA2-iPerfXRL
MATERA2-iPerfXRL was applied to UC 05_NOK. It implements MATERA2-030 and MATERA2-060 in the final (M32) release. MATERA2-030 satisfies NOK_25, while MATERA2-060 satisfies NOK_22, NOK_23, NOK_24, and NOK_25. Therefore, in UC 05_NOK, MATERA2-iPerfXRL satisfies NOK_22, NOK_23, NOK_24, and NOK_25.

2.17.2. Guidelines for MATERA2-iPerfXRL

Tester needs to define four artifacts: (1) the input space of the SUT that specifies the number of numerical input parameters and their ranges; (2) a fitness function which defines how different combinations of the input parameters affects the performance of the SUT; (3) the acceptable performance threshold by analyzing the requirement specifications of SUT; (4) allowed number of input combinations determines the maximum number of input combinations that the MATERA2-iPerfXRL can execute. The performance impact for a certain input combination is calculated using the fitness function; and if the performance impact is higher than the acceptable performance threshold then that input combination is most likely to cause performance bottleneck on the SUT.

For example, let us consider a conceptual example of a web application, which accepts three input parameters. For convenience we will refer to it as Conceptual Black-box SUT (CBS). The parameters of CBS can take integer values between 1 to 50 for the first two parameters, and between 1 and 400 for the third one. Table 3 defines the input space of CBS.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>p2</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>p3</td>
<td>1</td>
<td>400</td>
</tr>
</tbody>
</table>

The fitness function returns the performance impact of a given input combination. For example, the following function in Python language returns the performance impact in terms of elapsed execution time of an HTTP request in seconds:

```python
import requests

def fitness(p1, p2, p3):
    url = "https://example.com/?p1={}&p2={}&p3={}".format(p1, p2, p3)
    res = requests.get(url)
    return res.elapsed.total_seconds()
```

We can set the acceptable performance threshold to 0.05. This means that if the CBS takes more than 0.05 seconds to process a request corresponding to a certain input combination, then the input combination has triggered a performance bottleneck on the CBS.
After executing the allowed number of input combinations, the tool produces a list of input combinations that can trigger the performance bottlenecks in the SUT without any prior domain knowledge of the SUT.

2.18. MATERA2-ADCT

2.18.1. Methodology for MATERA2-ADCT

Conformance testing for reactive systems is mostly related with the formal models that are developed separately from the specifications models. Despite their verification capability, the development of the formal models require special skills, learning new formal languages and consuming time. We present an approach that utilizes the specification models for conformance testing via the introduction of time properties in executable models and by enabling the execution engine to generate partial observable traces. The time properties enable the modeller to use clock expressions in executable models to model the time requirements. Additionally, the generation of partially observable traces is used to conform the IUT with specifications. Furthermore, reusing the specification models for conformance testing helps model developers to focus on the specifications details and reduces the need to learn new formal languages.

Figure 60 present our approach in general. The online test cases are generated via introducing time properties in fUML activity diagrams executed by a customized fUML execution engine. The customize engine enables observation of partial execution traces for conforming SUT to IUT.

MATERA2-ADCT was applied to UC 03_IKER and UC 05_NOK. It implements MATERA2-010, MATERA2-020, MATERA2-040, MATERA2-060, and MATERA2-090 in the final (M32) release. MATERA2-010 does not map to UC 03_IKER and UC 05_NOK in WP3, but in WP2 it satisfies IKER_01 and NOK_01. Similarly, MATERA2-020 does not map to UC 03_IKER and UC 05_NOK in WP3, but in WP2 it satisfies NOK_07 and NOK_08. MATERA2-040 satisfies NOK_07 and NOK_25.
MATERA2-060 satisfies IKER_09, NOK_22, NOK_23, NOK_24, and NOK_25. MATERA2-090 satisfies NOK_25 in WP3, while in WP2 it satisfies IKER_18, NOK_12, NOK_17, NOK_18, NOK_19, and NOK_20. Therefore, MATERA2-ADCT satisfies IKER_01, IKER_09, and IKER_18 in UC 03_IKER and NOK_01, NOK_07, NOK_08, NOK_12, NOK_17, NOK_18, NOK_19, NOK_20, NOK_22, NOK_23, NOK_24, and NOK_25 in UC 05_NOK.

The specification is modeled via fUML AD and elements are annotated via stereotypes. We introduced two stereotypes via an observable profile, as shown in Figure 61.

The <<clock>> stereotype is used to annotate a class property used as a clock in the model. The clock runs independent of the model execution and can be reset during the execution. Additionally, the clock can be used in a relation expression in the activity, modeling a decision node to check a timing requirement. Based on the result of the expression, the model can execute a different branch in the model, representing the corresponding behavior wrt. to some timing requirements.

The <<ObservableIO>> stereotype is used to annotate the activity nodes to be externally observed, since the original fUML specification does not support the connectivity with external tools and the execution engine was originally implemented in Java and run in a JVM container. The <<ObservableIO>> mainly annotates the activity parameter node that is later externally observed. The partially observable trace is generated via the execution of different activities, by providing input parameters and observing the output parameter. Both input and output parameters are externally visible using <<ObservableIO>> annotation.

2.18.2. Guidelines for MATERA2-ADCT

The fUML AD model, annotated with stereotypes from observable profile, is required by MATERA2. Additionally, a Spread [14] demon is required and the configuration e.g., host address, port, user name and group should be done accordingly. Spread is an open source toolkit that can be used in many distributed applications that require high reliability, high performance, and robust communication among various subsets of members. The spread tool kit can be obtained from http://spread.org/.

Inorder to install MATERA2-ADCT plugin, the user is required to install Eclipse Modeling Tools 2018-09 (4.9) and Eclipse Papyrus. MATERA2-ADCT is Eclipse plugin and can be installed from the MATERA2-ADCT update site listed at http://toolbox.megamart2-ecsel.eu/content/matera2 via selecting Install New Software from Help menu in Eclipse.
After successful installation, MATERA2-ADCT can be configured by providing some basic setup information as shown in Figure 62.

![Figure 62: MATERA2-ADCT preference page](image)

Additionally, the user needs to select MATERA2-fUML execution engine in Moka preferences as shown in Figure 63, in order to execute fUML AD with clocks and observable parameters.

![Figure 63: Moka preferences page showing MATERA2-CT as an executable engine](image)

After configuring the MATERA2-ADCT, the user needs to open the fUML models in Papyrus. The nodes of the activity diagrams need to annotated via stereotypes i.e., `<<ObservableIO>>` stereotype is used to annotate the activity parameter node and `<<clock>>` stereotype is used to annotate the class property used as clock in the model as shown in Figure 64.

![Figure 64: Application of observable stereotypes in fUML models](image)
After annotating the model, the user is required to start the spread demon in order to connect the MATERA2-ADCT execution engine to the broadcast network. Afterwards, the user can start the execution of the model via the activity designed to start the simulation. The partial execution traces is observable in fUML console window along with the message broadcast on the spread network. The partial observable traces of the SUT and the interface of IUT must be mapped to provide inputs with the values provided by SUT and to observe the output of the IUT to compare with SUT. This mapping is done via an artifact called test adapter. The adapter receives the inputs and converts them to a format that is compatible with the IUT. It also transforms the outputs of the IUT to model-level output actions. Thus, the I/O conformance of the behavior of the IUT is observed by MATERA2-CT.

2.19. MATERA2-AlfTester

2.19.1. Methodology for MATERA2-AlfTester

The execution and simulation features of Moka allow one to provide input data required to reach and execute all important elements in the graphical fUML and textual Alf [16] models, and to measure model coverage and produce a model coverage report at the end of the simulation process. The work presented in [5] addresses two research questions: (1) How to automatically generate input data needed to simulate all execution paths in fUML Activity diagrams (ADs) containing Alf code? (2) How to generate test cases and test oracle from fUML ADs containing Alf code?

MATERA2-AlfTester was applied to UC 03_IKER and UC 05_NOK. It implements MATERA2-010, MATERA2-020, MATERA2-030, and MATERA2-060 in the final (M32) release. MATERA2-010 does not map to UC 03_IKER and UC 05_NOK in WP3, but in WP2 it satisfies IKER_01 and NOK_01. Similarly, MATERA2-020 does not map to UC 03_IKER and UC 05_NOK in WP3, but in WP2 it satisfies NOK_07 and NOK_08. MATERA2-030 satisfies NOK_25 and MATERA2-060 satisfies IKER_09, NOK_22, NOK_23, NOK_24, and NOK_25. Therefore, MATERA2-AlfTester satisfies IKER_01 and IKER_09 in UC 03_IKER and NOK_01, NOK_07, NOK_08, NOK_22, NOK_23, NOK_24, and NOK_25 in UC 05_NOK.

We present a novel approach for the exhaustive simulation and test generation from fUML ADs containing Alf code. The proposed approach, called MATERA2-Alf Tester (M2-AT) [5], 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. The generated test cases in M2-AT satisfy 100% code coverage of the Java code. The generated input data is used for executing the original fUML and Alf models in the Moka simulation engine. The interactive execution in Moka allows to determine the model coverage of the executable models. In addition, the generated Java code can be reused later on as a starting point for the actual implementation of the system.

The approach, shown in Figure 65, is composed of several steps. First, the fUML ADs and their associated Alf code are converted into Java code. Then, we obtain all the inputs of the Java program to achieve 100% coverage of the code via using an external tool (EVOSUITE [19]). These inputs are used to simulate the ADs. Since the Java code and the ADs are behaviorally equivalent, the input will also satisfy 100% coverage of the AD. During the simulation, one can detect and fix problems in the
specifications. In the next step, the Java code is used to generate input data and a test suite. The proposed approach allows to left-shift testing activities in the software development process. In M2-AT, exhaustive simulation of fUML models helps in validating software specifications and improving their quality at an early stage. Moreover, test cases and test oracles are generated beforehand with 100% of the model prior to the actual implementation of the system.

![Diagram](image)

Figure 65: Approach for MATERA2-AlfTester

2.19.2. Guidelines for MATERA2-AlfTester

The input MATERA2-AlfTester are the design models developed in fUML. The tool dependencies are Eclipse Papyrus, Eclipse Moka, and Papyrus ALF properties view integration plugins. The required dependencies can be downloaded from Eclipse Marketplace. However, it is mandatory to install Eclipse Moka for Co-Simulation plugin since the Moka Core plugin is unable to show fUML ADs in Papyrus. The MATERA2-AlfTester update site can be reached via Eclipse Help menu by selecting the Install New Software option. The update site url is list at MATERA homepage\(^\text{17}\) as shown in Figure 66.

\(^{17}\) [http://toolbox.megamart2-ecsel.eu/content/matera2](http://toolbox.megamart2-ecsel.eu/content/matera2)
Figure 66: Installing MATERA2-AlfTester from update site

The Figure 67 shows an example of fUML activity diagram. It invokes the withdraw method, which creates a new transaction and sets it as the current transaction. Next, it validates the entered pin. If validatePin returns true, the withdrawal transaction is successfully performed and the account balance is updated. Finally, the completed withdrawal transaction is recorded in the system.

Figure 67: fUML AD example

After installing the MATERA2-AlfTester plug, one needs to load the fUML model in Papyrus editor. The initial step is to generate code for the set of fUML activities grouped in a Package since one activity may use other activity to complete its execution. The code generation option can be selected from the context menu in the Project Explorer window as shown in Figure 68.

Figure 68: Context menu option for the code generation

The user need to provide a path to a folder to store the generated code as shown in Figure 69.
An external tool EVOSUITE is required to generate a test input for simulation. The path can be configured by the user prior to the test generation phase as shown in Figure 70. The tool can be downloaded separately from EVOSUITE web site\textsuperscript{18} and executable jar should be placed in an accessible folder.

During the second step, MATERA2-AlfTester will execute EVOSUITE in order to generate test suite by using the Java code generated in the previous step. Figure 71 shows EVOSUITE during the test generation process.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{select_folder.png}
\caption{Selecting folder for generated code}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{select_jar.png}
\caption{Selecting EVOSUITE jar file before test generation phase}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{evosuite.png}
\caption{EVOSUITE during the test generation process}
\end{figure}

\textsuperscript{18} \url{http://www.evosuite.org/downloads/}
Thirdly, user needs to generate Alf test script to exhaustively simulate the fUML activity in Moka. This can be done via accessing the Activity Alf test script context menu by selecting the activity diagrams in project explorer as shown in Figure 72.

The Alf test script to simulate the activity diagram is embedded in the model that can be executed in Moka. Figure 73 shows Moka simulating the model with generated test script and the coverage of the model can be observed visually in the editor.
Additionally, the MATERA2-AlfTester produces the following outputs:

- Test inputs for simulation encoded in Alf Tester script
- Java skeleton for the fUML AD
- jUNIT test cases
- Test Report (XML)

The details for these outputs can be found in [5].

2.20. Modelio

2.20.1. Methodology for Modelio

In the context of the WP3, Modelio covers requirements for the specific code generation (see Figure 74). To address the requirements from IKERLAN case study on the need of support of the design of IoT devices development on the design level, Modelio team developed design and generator solutions for the OMG Data Distribution Service (DDS) Interface Definition Language (IDL) including specific Java code skeletons generation.

As an input, the Modelio DDS designer obtain a System Model. Using Modelio UML DDS Profile users may annotate elements of the System Model. This resulting model can be used to generate DDS interface definitions in IDL as well as the Java code skeletons that are ready for using the DDS framework. Eventually the Java code can be back-reversed to UML to reflect the changes done in the Java code. The obtained System Model" can be fed to the model-based testing tools for example those of SmartTesting.
2.20.2. **Guidelines for Modelio**

The following subsections provide brief guidelines for deploying DDS Designer module in Modelio and generation of Java skeletons.

**Installation**

This module requires an installation of the 3.7 or 3.8 version of Modelio.

**Add the module to Modelio**

Download the .jmdac file joined to this project.

Open the Configuration > Modules catalog… menu, then click Add the module to the catalog… and select the .jmdac file.

The module is now available for your Modelio installation, you can deploy it in any project.

**Deploy the module in the project**

Once the module is added to Modelio (See previous section).

Open the Configuration > Modules… menu, click on the Add button and select the DDS Designer module in the catalog.
**Usage**

**Create System Model with DDS profile**

The DDS profile is applicable on a UML Class diagram with the (DDS) Topic stereotype. Apply the Topic stereotype to the required classes. Generate IDL files describing the topic and used structures, then use an embedded IDL compiler to generate all the DDS Java classes in the JavaDesigner generation folder. Then the generator will add DDS related wrapper around the topic to the model.

**Command: Include in IDL files**

This command is invoked by right-clicking on the following UML elements: Class, Enumeration and Datatypes. Add the corresponding IDL stereotypes and apply the naming convention for such elements.

**Command: Create Topic / Add topic stereotype**

Create or transform a Class into a Class with the Topic stereotype and apply the naming convention.

**IDL modelling details**

The table below UML to IDL mapping table describes the details of the transformation of UML elements in DDS IDL.

<table>
<thead>
<tr>
<th>UML elements with DDS annotation</th>
<th>Mapping with elements generated in DDS IDL</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>Class &lt;&lt; IDLClass &gt;&gt;</td>
<td>Struct</td>
<td></td>
</tr>
<tr>
<td>Class &lt;&lt; Topic &gt;&gt;</td>
<td>Struct</td>
<td>Generated struct has pragma instruction for the generator, so it is compiled has a topic.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Struct member</td>
<td></td>
</tr>
<tr>
<td>Attribute &lt;&lt; TopicID &gt;&gt;</td>
<td>Struct member</td>
<td>Said member will be used has an ID for the topic. It does nothing if the containing class has no Topic Stereotype</td>
</tr>
<tr>
<td>Enumeration &lt;&lt; IDLEnumeration &gt;&gt;</td>
<td>Set of constant String with a typedef</td>
<td>It is generated this way to be used with the embedded IDL compiler. A classic enumeration generation can be achieved by changing the code generation templates</td>
</tr>
</tbody>
</table>

Page 71 of 80
IDL naming conventions

Some of the above-mentioned UML elements (e.g. Class, Enumeration, Datatype) need to be prefixed by the string “IDL”, so that the generated Java class does not override the IDL element when reversed. This naming convention is added automatically if needed upon generation and when adding stereotypes through dedicated commands.

Code generation customization

For the DDS module, the way the IDL code is generated can be customized. Note that this does not apply to the Java code used by DDS. The code is generated using a StringTemplate\(^\text{19}\). The directory containing used templates can be specified in the module parameters.

2.21. VeriATL

2.21.1. Methodology for VeriATL

The methodology for VeriATL is already described in detail within the sibling deliverable D2.6 (cf. the section dedicated to VeriATL in this document). Please refer to this deliverable D2.6 for more insights on the methodological aspects of VeriATL.

2.21.2. Guidelines for VeriATL

The guidelines for VeriATL are already described in detail within the sibling deliverable D2.6 (cf. the section dedicated to VeriATL in this document). Please refer to this deliverable D2.6 for more insights on the guidelines for using VeriATL.

2.22. LIME TestBench

2.22.1. Methodology for LIME TestBench

The LIME Testbench is a testing and runtime monitoring tool originally developed in the project Lightweight formal Methods for distributed component-based Embedded systems (LIME), and it's continuation project LIME2. LIME was developed further in MegaM@art2 project, to make it compatible with the newest development tools, and to extend all its components for the C/C++ programming language. The LIME Testbench implements two separate functions: runtime monitoring of model-based properties, and test case generation. Both functions rely on the instrumentation and monitoring of the execution of the program under test. Instrumenting the program under test allows to monitor and record all function calls, memory access, and register operations during execution without additional effort from the programmer. This in turn allows to add assertions to the program to monitor adherence to properties that can be specified with regular expressions, temporal logic, or finite state machines. It is also possible to automatically generate high coverage test suites by using a constraint

\(^{19}\) https://www.stringtemplate.org/
solver to produce inputs that force the program to take certain branches during execution. The above features work for Java and C/C++ languages.

2.22.2. Guidelines for LIME TestBench

The monitoring component of LIME TestBench requires as input a source program in Java or C/C++ and a specification of a property that the program should satisfy. The property can be specified using regular expressions, linear temporal logic or finite state machines. The tool provides a mechanism to specify the property in the language of the program to be tested (Java or C/C++). From these components the tool compiles an instrumented executable, which when executed it tests if the property is preserved. For example assume that we have a program that uses two functions lock() and unlock() to lock and respectively unlock access to a shared resource. We want to ensure that the program calls lock and unlock in this order, and it never calls for example lock twice before calling unlock. In this case the property can be specified using the regular expression (lock unlock)*, i.e. the program could call lock and unlock successively an arbitrary number of times. Next we introduce the example using a pseudocode to simplify its presentation. Concrete examples as well as instructions for installation are available from the webpage of the tool: https://megamart.ssf.fi/lime/.

```java
void lock() { print("lock"); }
void unlock() { print("unlock"); }
void work() { print("work"); }

regexp monitor() { return (lock ^ unlock)*; }

int main()
{
    lock();
    work();
    lock();
    unlock();
}
```

If we compile this example using LIME, and run it, we will discover that the property is not satisfied, because we call lock twice before calling unlock. This is a simple example with simple control structure, but the tool can monitor the property in any program with complex control structure.

The test generation of LIME TestBench takes as input a program where some variables are assigned a special value to indicate to the test generation to search for suitable values for these variables that ensure covering all executions paths. Next program illustrates this mechanism.

```java
int main()
{
    int a = lct_get_int();
    int b = lct_get_int();

    if (a * b == 781) // 11 * 71
        printf(a, b);
    else
        printf(a, b);
}
```
When this program is compiled with LIME testing tool, an instrumented executable is generated, which when run consecutively will find and report values for variables a and b that enforce the executions via both branches of the if statement. A first execution of the program could find $a = b = 1$ and this covers the else branch. A second execution could find $a = 11$ and $b = 71$, and this covers the first branch of the if statement.

2.23. RCRS

2.23.1. Methodology for RCRS

The methodology for RCRS is described in detail within the sibling deliverable D2.6.

2.23.2. Guidelines for RCRS

The guidelines for RCRS are described in detail within the deliverable D2.6.

2.24. PauWare

2.24.1. Methodology for PauWare

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 and in [11].

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.

Figure 75 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.

![Figure 75: Development of a Java-based software using PauWare and its associated tools](image)

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 the 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
class 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 onTransitionOperation(Transition transition, AbstractAction action);  
    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.

### 2.24.2. Guidelines for PauWare

At the runtime level, the complete running application is available: the code of the state machine has been generated, the business and others Java operations have been implemented and weaved with the states and transitions of the state machine. This weaving requires a human intervention with a particular attention of the management of the parameters of the operations (see this documentation\(^\text{20}\) for instance).

Then, one can execute the build system associated with a monitor than can generate an execution trace of take some decision depending on how the execution is evolving. Invariants can be associated with states of the state machine and evaluated at runtime during a run to completion cycle (the processing of an event). If the monitor is informed that an invariant is violated, it can take a decision on

\(^{20}\) [http://ecariou.perso.univ-pau.fr/MegaM@RT2/Pauware%20presentation.pdf](http://ecariou.perso.univ-pau.fr/MegaM@RT2/Pauware%20presentation.pdf)
the execution of the state machine: going back to the last active states, stopping the execution of the state machine, etc. This enables to verify and handle at runtime that the execution of the state machine is correctly performed.

2.25. CMA

2.25.1. Methodology for CMA

CMA, acronym of Completeness Metric Analyzer, is a command-line tool for runtime monitoring of timing. This tracing mechanisms uses probes in order to providing instruction tracing of a processor. Outputs operation are the information of the processor such as the function timing execution. The trace protocol provides a real-time trace capability for the functionality.

The command line tool compresses the trace information, when the timing data has been captured the command line tool extracts the information from the Trace Port Analyzer or from Embedded Trace Buffer and decompresses it to provide a full disassembly, with the information about the code that was executed. After that a log file was generated with all tracing information listed.

The log file has the following structure:

- `<system>` may be `<state>`
  - attributes: parameters
  - function: may be
  - timing: unit
- at least `<quantity>` times per `<unit>`
  - attributes: quantity, unit
  - function: name

2.25.2. Guidelines for CMA

When command line tool tracing processor execution, the core generate trace for every instruction that the processor commits for execution. This generates trace that is easy to interpret, but it normally requires a high trace bandwidth. The Program Flow Trace internal functionality assumes that any trace decompressor has a copy of the program being traced, and generally outputs only enough trace for the decompressor to reconstruct the program flow. However, its trace output also enables a decompressor to reconstruct the program flow when it does not have a copy of parts of the program, for example because the program uses self-modifying code.

The Program Flow Trace internal functionality also provides full information about exceptions, and the instruction set state, security state, and current Context ID of the processor. It can also provide cycle count information, and timestamping.

Program Flow Trace identifies certain instructions in the program, and certain events, as waypoints. A waypoint is a point where instruction execution by the processor might involve a change in the program flow.

The Program Flow Trace waypoints include:

- all indirect branches
- conditional and unconditional direct branches
• all exception
• any instruction that changes the instruction set state.
• when Halting debug-mode is enabled, entering or leaving Debug state
• synchronization primitives

When a waypoint occurs, the command line tool generates trace data that describes the waypoint. From this data, a trace decompressor can determine how many instructions have been executed since the previous waypoint, and therefore can reconstruct the execution stream. In effect, the tool outputs an indicator at each waypoint, and the decompressor matches these indicators with the waypoints in the program code, to reconstruct the program flow.

3. Conclusions

In this deliverable, we presented methodologies and guidelines for the MegaM@Rt2 WP3 runtime analysis tool set presented in D3.5. We also presented the overall WP3 conceptual framework and tool set architecture and positioned each WP3 tool into the WP3 conceptual framework and tool set architecture.

Each of the presented tool methodology includes (1) a description of the problem that the tool addresses and (2) a brief description of the proposed solution outlining the theoretical basis of the tool along with some important details on the method or process of the tool. In addition, for each tool that has been applied to a use case (UC), we have also included a statement on how the tool was applied and which UC requirements were satisfied. As part of the tool guidelines, we described the tool inputs, outputs, and any manual steps needed to use the tool.
References


[22] https://diamon.org/ctf/.


[27] D3.2: Runtime Analysis Tool Set http://hdl.handle.net/20.500.12004/1/P/MMART2/D3.2


[30] D5.5 Case study evaluation report phase 1 http://hdl.handle.net/20.500.12004/1/P/MMART2/D5.5

[31] https://projects.eclipse.org/projects/polarsys


[33] Contrex project, https://contrex.offis.de/home/