WP3 - D3.1:
Foundations for
Model-Based Runtime Methods

Milestone: M6  
Last Edited on 29/09/2017 by ABO
With contributions from SOFT, SMA, ARM, PAU, ATOS, UCAN, FTS, UAQ, RO, ABO, SSF, CON, VTT, MDH, BUT, SICS
Reviewed by: BUT, CSY
Global Identifier: http://hdl.handle.net/20.500.12004/1/P/MMART2/D3.1
**Executive summary**

This deliverable provides a succinct overview of the foundations of model-based runtime methods and technologies in order to support the innovation tasks of Work Package 3 of the MegaM@Rt2 project. The main focus of WP3 is on how different artifacts necessary at runtime level can be obtained and used in the MegaM@Rt framework for activities such as model execution, runtime verification, online testing and runtime/real-time analysis.

More specifically, we are discussing on the one hand, how runtime artifacts are obtained from design artefacts and from execution logs. In the first category, we overview approaches for generating runtime code and models from design models via code generation and, respectively, model transformations. In the second category, we discuss approaches for creating or improving the runtime artifacts by analyzing the runtime execution logs of the system via methods like machine learning and data analytics.

On the other hand, we discuss how runtime artifacts are used at runtime either by executing them as part of the system at runtime or by using them to generate tests or monitor the system during its operation.

Throughout this deliverable, we scrutinize the state-of-the art, the state-of-practice, and the baseline technologies which are available for the project participants to innovate on. To this extent, the deliverable will investigate current methods and tools for their benefits and limitations. The results of this work will lay the basis for defining new concepts, methods and tools for coping with these limitations and successfully deploy runtime methods to industrial settings. This deliverable provides input for the specification of the MegaM@Rt runtime tools to support automated code generation and model execution, log analysis, runtime verification and testing activities.

In addition, this document also includes a collection of relevant solutions and tools provided by the MegaM@Rt2 consortium members as baseline technologies in the project.
Table of Contents

Executive summary 2
Table of Contents 4
Acronyms 6
1. Introduction 8
2. Automated code and model generation for runtime 9
   2.1 Aspect-oriented approaches 9
      2.1.1 Aspect-oriented programming 10
   2.2 Methodologies, tools and frameworks for code generation 14
   2.3 Model Execution 16
3. Runtime trace analysis 18
   3.1 Monitoring Solutions 20
      3.1.1 Software monitoring solutions 21
      3.1.2 Hardware monitoring solutions 21
   3.2 Analyzing test execution logs 23
      3.2.1 Predicting test execution times 23
      3.2.2 Visualizing and debugging test traces 23
   3.3 Runtime trace analysis in partitioned systems 24
   3.4 Automatic runtime feedback into the model and learning-based log analysis in real time 25
4. Runtime verification and Online testing 27
   4.1 Runtime verification 27
   4.2 Model-based testing 29
   4.3 Combining automatic test generation and verification 30
   4.4 Static analysis techniques 30
   4.5 Provided technologies by project partners 31
      4.5.1 CompleteTest 31
      4.5.2 Online Model-Based Testing via Model Mutation using UPPAAL-TA 32
      4.5.3 Online Model-Based Performance Evaluation 33
5. Model-based system monitoring through “models@run.time” 34
6. Baseline technologies and tools provided by project partners 36
   6.1 SeaFox 36
   6.2 Conformiq Designer 36
   6.3 Modelio 37
   6.4 CertifIt / MBeetle 38
   6.5 PauWare 40
   6.6 CompleteTest 41
   6.7 Lime 42
7. Conclusions 43
References 44
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alamo</td>
<td><strong>A LIGHTWEIGHT ARCHITECTURE FOR MONITORING</strong></td>
</tr>
<tr>
<td>AOM</td>
<td><strong>ASPECT ORIENTED MODELING</strong></td>
</tr>
<tr>
<td>AODM</td>
<td><strong>ASPECT-ORIENTED DESIGN MODEL</strong></td>
</tr>
<tr>
<td>AOP</td>
<td><strong>ASPECT-ORIENTED PROGRAMMING</strong></td>
</tr>
<tr>
<td>AOSD/UC</td>
<td><strong>ASPECT-ORIENTED SOFTWARE DEVELOPMENT / WITH USE CASES</strong></td>
</tr>
<tr>
<td>API</td>
<td><strong>APPLICATION PROGRAMMING INTERFACE</strong></td>
</tr>
<tr>
<td>ASIC</td>
<td><strong>APPLICATION SPECIFIC INTEGRATED CIRCUIT</strong></td>
</tr>
<tr>
<td>ARINC</td>
<td><strong>AERONAUTICAL RADIO INCORPORATED</strong></td>
</tr>
<tr>
<td>CTF</td>
<td><strong>COMPACT TRACE FORMAT</strong></td>
</tr>
<tr>
<td>DSML</td>
<td><strong>DOMAIN SPECIFIC MODELING LANGUAGE</strong></td>
</tr>
<tr>
<td>DSL</td>
<td><strong>DOMAIN-SPECIFIC MODELING LANGUAGE</strong></td>
</tr>
<tr>
<td>DynaMICs</td>
<td><strong>DYNAMIC MONITORING WITH INTEGRITY CONSTRAINTS</strong></td>
</tr>
<tr>
<td>FDIR</td>
<td><strong>FAULT DETECTION, IDENTIFICATION, AND RECOVERY</strong></td>
</tr>
<tr>
<td>GNU</td>
<td><strong>GNU’S NOT UNIX</strong></td>
</tr>
<tr>
<td>HLS</td>
<td><strong>HIGH LEVEL SYNTHESIS</strong></td>
</tr>
<tr>
<td>HM</td>
<td><strong>HEALTH MONITORING</strong></td>
</tr>
<tr>
<td>IDE</td>
<td><strong>INTEGRATED DEVELOPMENT ENVIRONMENT</strong></td>
</tr>
<tr>
<td>Jass</td>
<td><strong>JAVA WITH ASSERTIONS</strong></td>
</tr>
<tr>
<td>JPaX</td>
<td><strong>JAVA PATHEXPLORER</strong></td>
</tr>
<tr>
<td>JPDD</td>
<td><strong>JOIN POINT DESIGNATION DIAGRAMS</strong></td>
</tr>
<tr>
<td>KPI</td>
<td><strong>KEY PERFORMANCE INDICATORS</strong></td>
</tr>
<tr>
<td>LTL</td>
<td><strong>LINEAR-TIME TEMPORAL LOGIC</strong></td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>MDE</td>
<td>MODEL-DRIVEN ENGINEERING</td>
</tr>
<tr>
<td>MOP</td>
<td>MONITOR-ORIENTED PROGRAMMING</td>
</tr>
<tr>
<td>MTL</td>
<td>METRIC TEMPORAL LOGIC</td>
</tr>
<tr>
<td>MBT</td>
<td>MODEL-BASED TESTING</td>
</tr>
<tr>
<td>MBPeT</td>
<td>MODEL-BASED PERFORMANCE TESTING</td>
</tr>
<tr>
<td>NVM</td>
<td>NON-VOLATILE MEMORY</td>
</tr>
<tr>
<td>OOP</td>
<td>OBJECT-ORIENTED PROGRAMMING</td>
</tr>
<tr>
<td>OS</td>
<td>OPERATING SYSTEM</td>
</tr>
<tr>
<td>PC</td>
<td>PROGRAM COUNTER</td>
</tr>
<tr>
<td>PIM</td>
<td>PLATFORM-INDEPENDENT MODEL</td>
</tr>
<tr>
<td>PSM</td>
<td>PLATFORM-SPECIFIC MODEL</td>
</tr>
<tr>
<td>PTA</td>
<td>PROBABILISTIC TIMED AUTOMATA</td>
</tr>
<tr>
<td>RAM</td>
<td>RANDOM ACCESS MEMORY</td>
</tr>
<tr>
<td>RV</td>
<td>RUNTIME VERIFICATION</td>
</tr>
<tr>
<td>RR</td>
<td>RUNTIME REFLECTION</td>
</tr>
<tr>
<td>REST</td>
<td>REPRESENTATIONAL STATE TRANSFER</td>
</tr>
<tr>
<td>SEAT</td>
<td>SOFTWARE EXPLORATION AND ANALYSIS TOOL</td>
</tr>
<tr>
<td>SM</td>
<td>STATE MACHINE</td>
</tr>
<tr>
<td>SD</td>
<td>SEQUENCE DIAGRAMS</td>
</tr>
<tr>
<td>SUT</td>
<td>SYSTEM UNDER TEST</td>
</tr>
<tr>
<td>UML</td>
<td>UNIFIED MODELING LANGUAGE</td>
</tr>
<tr>
<td>UPPAAL-TA</td>
<td>UPPAAL TIMED AUTOMATA</td>
</tr>
<tr>
<td>VU</td>
<td>VIRTUAL USER</td>
</tr>
<tr>
<td>WCET</td>
<td>WORST CASE EXECUTION TIME</td>
</tr>
<tr>
<td>WSC</td>
<td>WEB SERVICE COMPOSITION</td>
</tr>
</tbody>
</table>
1. Introduction

In the last decades the Model Driven Engineering (MDE) has become increasingly popular in industry due not only to the use of abstract models that focus on different perspectives of the system and the model transformations which are used to refine the specifications, but also due to mature tool support especially in the area around the Unified Modeling Language (UML) and its profiles like MARTE, SysML, etc. This trend has been observed at all levels of the software development life cycle, starting from requirements modeling, analysis, design and ending with validation and verification activities.

MDE aims at promoting models as the main artefacts throughout the software life cycle, including at runtime. Having models present at runtime is made with several techniques and objectives in mind (Szvetits & Zdun, 2016) (Blair, Bencomo, & France, 2009). Among these techniques, we mention model execution, runtime verification, and testing. Model execution consists in defining a model that is interpreted with a dedicated engine implementing an execution semantics (Cariou, Le Goaer, Barbier, & Pierre, 2013). With model execution, the ability to run a model prior to its implementation is a timesaving and henceforth cost-saving approach, making possible detecting and fixing problems at design time by simulating the model and directly reusing the model at runtime through its execution.

Runtime validation consists of validating a design at runtime through additional monitoring mechanisms and some kinds of recovery procedures. In general, this does not require understanding the cause of the bug, but only the symptom and the remedy (Bayazit & Malik, 2005). In turn, runtime verification (K. Havelund & Rosu, 2001), (Leucker & Schallhart, 2009) is the process of verifying that certain properties of the system hold in certain states reached during the execution of the specification. It combines both formal specifications and software testing to validate a system. System properties can be specified by means of a formal language and automatically translated to a monitor, which checks their correctness against the implementation. As the verification result, the monitor assigns a verdict that shows whether the implementation satisfies the properties or not. This type of runtime verification task deals with the observed executions of the implementation, by executing finite test traces within a limited time. There is a large body of work on Model-based runtime verification, which is an extension of the traditional runtime verification in which the runtime behaviour of a system is checked against a model, which satisfies certain properties of the system specification (Zhao & Rammig, 2009). Several researchers suggested as beneficial the combination of mega-modelling and runtime verification (Vogel, Seibel, & Giese, 2011). Recent reports show that mega-modelling enable the industrial adoption of complex MDE solutions (Bastarrica, Simmonds, & Silvestre, 2014), and that it has already been applied to modelling evolvability (Favre & NGuyen, 2005), performance (Fritzsche et al., 2009), product lines (Vierhauser, Grünbacher, Heider, Holl, & Lettner, 2012), as well as for data analysis (Ceri et al., 2013).

Another way to handle models at runtime is to have models reflecting the driving the system behavior and being causally connected with it. Such models (named “models@runtime”) can be used for managing the system adaptation (QoS aspects, environment changes, etc.). As shown in (Filieri, Ghezzi, & Tamburrelli, 2011) the increasing maturity and success of models@runtime techniques tends to blur the traditional distinction between design and runtime, creating a long continuum that includes design, runtime, maintenance and evolution. A recent systematic literature review (SLR) published by Szvetits and Zdun in (Szvetits & Zdun, 2016) provides a comprehensive overview of the state-of-the-art on the use of models at runtime answering the following research questions:

- For what purposes are models used at runtime?
- Which techniques and kinds of models are used when processing models at runtime?
- Which problems are addressed by using models at runtime?
- Which architectures exist for processing models at runtime?
- Which research methods are most frequently used for evaluating runtime model approaches?
- Which empirical evidence has been reported?

We defer the answer to these research questions to the above mentioned study. However we emphasize one of the conclusions of the study applicable to the MegaM@aRt2 project:

"While researchers put a lot of effort into the development of monitoring, adaptation, and synchronization mechanisms, we think that there is a strong need for process-related enhancements when designing and implementing systems which are augmented by runtime models......We believe that a tool-assisted selection of architectures, techniques, kinds of models, and testing mechanisms would boost the development and adoption of applications using models at runtime."

In the following, we discuss different state-of-the-art methods and tools which complement the above study in order to provide the foundations for developing novel methods and tools in the MegaM@Rt2 project for using models at runtime. The document is structured in five parts. In Section 2, we discuss methods for automated code generation and method for building executable models which are used at runtime. In Section 3, we overview a set of methods which allow to obtain runtime models via log and trace analysis techniques, whereas in Section 4 we discuss methods for using these runtime models for runtime verification and testing. Section 5 briefly discusses techniques in which models are used as part of the system implementation and which are used to take decisions on how the system should behave. Last but not least, Section 6 includes a list of tools provided by the MegaM@Rt2 consortium partners as baseline technologies for the MegaM@Rt2 project. We conclude with a list of identified problems faced by the current technology and planned to be addressed in the context of the MegaM@Rt2 project.

2. Automated code and model generation for runtime

This section discusses different techniques for obtaining some parts of the final and running system directly and automatically from design models. In the following we present a set of approaches for automatically deriving the runtime artefacts either using a traditional UML-based modeling approach, an aspect-oriented approach, or via model execution.

2.1 Aspect-oriented approaches

The Aspect Oriented Software Development (AOSD) explores the idea of Separation of Concerns (SoC) to identify concerns in software development and encapsulate them in appropriate modules. Particularly, those concerns that span over different parts of the application, namely cross-cutting concerns, require different ways of modularization, beyond the traditional ones close to programming language structures (i.e. classes). AOSD addresses the effects of crosscutting concerns on software artefacts: scattering -specifications related to one concern are distributed over several units, and tangling - a given unit contains specifications related to several concerns

Different AOSD frameworks have been released in last decade, and some of them have got distinct popularity (e.g., AspectJ). Examples of how to use AOSD to inject support for cross-cutting concerns, including logging are available in the literature (Chiba & Ishikawa, 2005).
2.1.1 Aspect-oriented programming

Aspect-Oriented Programming (AOP) (Kiczales et al., 1997; Kiczales & Hilsdale, 2001) is an extension to Object-Oriented Programming (OOP) that modularized behavioral and structural crosscutting concerns as so-called aspects. AOP complements OOP by providing another way of thinking about program structure to allow for higher levels of separation of concerns. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns such as transaction management that cut across multiple points in your object model. (Such concerns are often termed crosscutting concerns in AOP literature.) As a development methodology, AOP recommends that you abstract and encapsulate crosscutting concerns. Therefore, aspects facilitate the isolation and modularization of the implementation of crosscutting concerns in base modules (i.e., advice classes).

Different frameworks giving support to AOP emerged soon after the idea was first proposed (Brichau et al., 2005; Hanenberg, 2005; Kersten, n.d.). That work have been especially fruitful in the Java domain, as three mature AOP frameworks already exist: AspectJ ((Laddad, 2003)), Spring AOP Framework (Mak, Long, & Rubio, 2010) and JBoss AOP (RedHat, n.d.).

2.1.2 Aspect-Oriented Software Development Using Models at Runtime

With the advance of model-based methods for software engineering, such as UML (Unified Modeling Language), developers were empowered to raise the level of abstraction during the development using high-level models and to automatically generate executable code. This new methodology promised productivity, quality and time-to-market gain in software delivery. Modeling languages were specifically developed for Object-Oriented Analysis and Design (OOA/D) methodologies. The ultimate goal was to automate programming tasks, that is, bringing design models into code. Due to the close relationship between AOP and OOP, aspects (at code level) become also a natural target for abstraction and generation.

The problems of code-level (static) and executable-level (dynamic) weaving in AOM are central in automated aspect-oriented code generation (code synthesis), i.e. code than can be executed by AOP frameworks. Many different modeling approaches supporting AOP have been proposed, mainly for AspectJ, the most popular AOP framework. Those languages capture at a higher level of abstraction the conceptual models of AOP frameworks. They do not envisage weaving of aspects at modeling level but defer it to the implementation phase, delegating the task to the AOP frameworks themselves (Wimmer et al., 2011).

Aspect Oriented Modeling (AOM) combines the ideas behind AOSD with those of model-based software development, where the main focus is placed on how different concerns of the system can be modelled independently and combined later on via composition mechanisms (e.g., model transformations). The benefits of using AOM in industrial context as an enabler for scalable modelling, reducing modelling complexity and facilitating model evolution have been noted in (Shaukat Ali, Briand, Arcuri, & Walawege, 2011). In addition, controlled experiments have shown that using aspect-oriented modelling techniques creates better quality models (S. Ali, Yue, & Briand, 2013) and significantly improves the readability of specifications (Shaukat Ali, Yue, & Briand, 2014).
As a research discipline, AOM is concerned with different weaving levels, that is, the level of abstraction where separation of crosscutting concerns is given up and weaving of concerns (aspects) occurs (Evermann, 2007):

1. in the case of design (model) level weaving, crosscutting concerns are separately modelled, then woven together to yield a non-aspect-oriented model, which is then transformed to non-aspect-oriented code;
2. in the case of code-level weaving, the separation of concerns is maintained on the implementation as aspect-oriented models are transformed to aspect oriented code and then woven by AOP engines;
3. finally, modern AOP frameworks can also maintain the separation of concerns on the executable and carry out executable-level (runtime) weaving by acting on the JVM or CLR bytecode.

With the maturity of the MDE field, approaches that brought good practices in modeling, model transformation and code generation to aspect-oriented code generation appeared. They proposed development methodology-agnostic (Evermann, 2007; Fuentes & Sánchez, 2006) and/or AOP platform-independent (Evermann, Fiech, & Alam, 2011; Mosconi, Charfi, Svacina, & Wloka, 2008) solutions. These approaches propose to use a domain-specific modeling language (DSL) capturing AOP concepts in the form of a metamodel that is then realized as an UML Profile. They also showed a promising perspective for aspect-oriented code generation following:

2. Seamless integration of AOP concepts into existing MDE processes and technology (Evermann et al., 2011; Mosconi et al., 2008).
3. Improved understandability and maintainability of platform independent models leveraging AOP (Mosconi et al., 2008).
4. Productivity gains and portability in AOP as model-driven aspect oriented code generators are written just once for each AOP framework (Evermann et al., 2011).

There is a diversity of mature AOM approaches proposed in the literature, which roughly could be classified into i) asymmetric, which support the distinction between cross-cutting and non-crosscutting concerns, ii) symmetric, which do not. Due to this diversity of approaches, a conceptual reference model (CRM) it is proposed (Wimmer et al., 2011) that provides a common understanding for the basic ingredients of AOM concepts, aiming at supporting the comparison of the different approaches.

The Aspect-Oriented Design Model (AODM) of Stein et al. (Stein, Hanenberg, & Unland, 2002a, 2002b, 2002c) is one of the first examples of incorporating AOP concepts in OOA/D; in particular, AspectJ’s. Its modeling language is an extended version of UML, which is extended with both lightweight and heavyweight extension mechanisms. Structural and behavioral modeling is achieved by employing class diagrams, UML 1.x collaborations and sequence diagrams. In addition, sequence diagrams are used for visualizing join points, for example, messages, while use case diagrams and collaborations demonstrate AspectJ’s composition semantics. The approach claims rapid modeling support by a wide variety of CASE tools (Stein et al., 2002b), which is due to using UML’s lightweight extension. “This is, however, questionable, since the authors also extended UML’s metamodel” (Wimmer et al., 2011).

Crosscutting concerns are introduced in AODM and represented by a stereotype <<aspect>> which is derived from the UML Class. In addition, several meta-attributes capture the peculiarities of AspectJ’s aspects, for example, the instantiation clause. As said before, in Stein’s approach, the composition
actually is deferred until the implementation phase. Nevertheless AspectJ’s composition semantics has been redefined for the modeling level to a limited extent, for example, in terms of UML Use Case and Collaboration diagrams (Stein et al., 2002a). It is possible to manage interactions (and conflict resolution) by manually specifying the order for composing aspects in terms of a stereotyped dependency relationship between aspects (Stein et al., 2002b).

In later works, AODM was extended to incorporate graphical ways to select join points in UML. For this they have introduced join point designation diagrams (JPDD) (Stein et. al. 2006; Stricker et al. 2009), which basically are UML diagrams (i.e., Class and Object diagrams, as well as, State Machines, Sequence, and Activity diagrams) enriched with for example, name and signature patterns, and wildcards. “They represent an independent pointcut mechanism that can be applied to any UML-based AOM language, allows to select all kinds of join points (i.e., structural-static, structural-dynamic, behavioral-static, and behavioral-dynamic) as well as supports composite pointcuts” (Wimmer et al., 2011).

The Aspect-Oriented Software Development with Use Cases (AOSD/UC) of (Jacobson & Ng, 2004) is another example of AOP concepts represented at design level in OOA/D. It extends UML 2.0 metamodel and comes with a systematic process that focuses on the separation of concerns throughout the software development life cycle, that is, from requirements engineering with use cases down to the implementation phase. It this case, the methodology fosters traceability between all phases through explicit trace dependencies between models. During the whole software development life cycle the approach makes use of different UML diagrams including Use Case, Class and Communication diagrams for analyzing the requirements. For the design phase, component diagrams can be refined into class diagrams, while Sequence diagrams are used to model behavioral features. The language extensions reflect the influence by the Hyper/J (Ossher & Tarr, 2000) and AspectJ (Laddad, 2003) languages. Concerns are modeled with the <<use case slice>> stereotype, which is derived from the UML Package. The use case slice—inspired by the Hyper/J language—encapsulates modeling artifacts of one phase in the software development life cycle, that is, concerns are kept separately until the implementation phase. Artifacts include Class and Sequence diagram elements as well as aspect classifiers (i.e. classifiers with stereotype <<=aspect>>).

AOSD/UC supports the pointcut-advice and open class weaving mechanisms from AspectJ (Laddad, 2003). In fact, composition is deferred until implementation, thus, a composite model is not available at modeling level. Its join point model is similar to that of AspectJ. Classifiers are used as structural-static join points, while behavioral join points are identified with the AspectJ pointcut language. Pointcuts are specified in a separate “pointcuts” compartment of an aspect classifier. The pointcut then is specified with AspectJ code allowing for usage of name patterns, type patterns, and wildcards. Advices are represented as “class extensions” in a separate compartment of the aspect classifier. The behavioral advice is further detailed within sequence diagrams. The methodology also incorporates a strategy to resolve conflicting interaction between aspects.

Neither Stein’s AODM nor AOSD/UC from Jacobson and Ng come with tool support. Since composition is deferred until implementation by means of the AspectJ framework, composition support within a tool (at design time) is not the authors’ focus. Moreover, modeling extensions made to the UML metamodel are not currently supported within a tool. Finally, neither AODM nor AOSD/UC do not consider code generation support despite they are thought to be completed with AspectJ code generation (Wimmer et al., 2011).
That is not the case for the JAC Design Notation proposed by Pawlak et al. (Pawlak et al., 2004; Pawlak, Duchien, Florin, & Seinturier, 2001) as it was early proposed as a solution for incorporating AOP concepts at design level in the JAC Framework. JAC is, an open source framework that includes a complete IDE with modeling support and serves as a middleware layer for aspectual components. Similar to the AODM approach of Stein et. al. (Stein et al., 2002a, 2002b, 2002c), the JAC Design Notation supported traceability from design to implementation” (Wimmer et al., 2011). Since it was developed out of a pragmatic need to express crosscutting concerns in the JAC Framework, “the authors do not claim full compliance with UML but aim at keeping it intuitive and simple” (Wimmer et al., 2011). The authors provide no information on the UML Class diagram version used; however, the extended UML metamodel in Pawlak et al. (2005) “indicates the usage of a UML version prior to version 2.0” (Wimmer et al., 2011). The JAC Design Notation realizes the pointcut-advise composition mechanism. The stereotype <<<aspect>>>, which is derived from the UML Class, is used to represent the crosscutting concern, and both behavioral and structural advice are represented as methods of aspect classes. The kind of advice is indicated by the stereotype of the advice’s operation. The stereotypes <<<before>>>, <<<after>>>, <<<around>>>, and <<<replace>>> indicate behavioral advice and the stereotype <<<pointcut>>>, which is derived from UML Association explicitly model the pointcut.

Regarding tool support, modeling tools for designing base and aspect classes as well as their relations using the proposed UML notation are provided. Like Stein’s AODM nor AOSD/UC from Jacobson and Ng, the weaving is not available at modeling level (design time) but deferred until implementation by means of a AOP framework, in this case, the JAC Framework. Consequently, the composition semantics are those of the JAC framework. however, unlike them, the IDE also supports code generation (i.e., Java) for the JAC framework. Both modeling of interactions as well as conflict resolution are not addressed at all by the approach (Wimmer et al., 2011).

In general, we can say that a seamless integration between MDE and AOP is far from realized. Some prerequisites for a valuable integration of these approaches are still to be realized (Mehmood & Jawawi, 2013). They have also shown that existing aspect-oriented code generation techniques and tools need to be improved from various perspectives; they are not mature-enough nor integrated-enough with any of the standard AOP frameworks as to be used outside academic environments. In order to be used practically, in industrial scenarios, those tools need to effectively work in conjunction with existing production-ready integrated development environments (IDE) and AOP frameworks (Mehmood & Jawawi, 2013).

AOM at modeling level has been significantly discussed in existing literature while more research is needed in the area of aspect-oriented code generation (AOM at implementation level). This is an underdeveloped area of research comparing to AOM at design time. Furthermore, research on the latter has frequently focused on proposing solutions and thus “there is need for research that validates and evaluates the existing proposals in order to provide firm foundations for aspect-oriented model-driven code generation” (Abid Mehmood, 2013).

MDD as defined by the OMG promotes separation of the abstractions layers for functional specification or Platform-Independent Model (PIM) from execution platform specification and integration in Platform-Specific Model (PSM), from which the code is generated. Both PIM and PSM are models managed by modelling environments such as UML tools. This approach is not sufficiently scalable for large industrial systems (Kolovos, Paige, & Polack, 2009). For example, for a PIM containing 1200 domain classes, this PIM will be transformed in 3600 PSM classes when considering a CORBA framework. Then, the business logic is added to the code generated for CORBA stubs and skeletons. Thus, we obtain a PSM which is 3x or 4x times bigger than PIM, and where the functional
code is buried in CORBA depended classes. Consequently, if requirements change all the chain should be re-synchronized including all manual changes done in the PSM. That operation is mostly manual, which is a nightmare for a large and complex system. Furthermore, over time the business logic tends to closely intertwine with CORBA dependent classes. Thus, it becomes tedious to migrate the system to a new execution framework, such as Enterprise JavaBeans.

Figure 3: MDD + AOP productivity approach

Figure 3 illustrates the code generation approach of Naval group, previously known as the “Direction des Constructions Navales et Systèmes” (DCNS). DCNS This approach has been developed in order to overcome the issues related to the migration from an execution framework to a new one. Naval group is was one of the first companies to apply MDD with AOP in an industrial context (Grivot E., n.d.). Within this approach, the platform classes in aspects are kept separately from functional classes. The integration with the execution frameworks is conducted through aspects weaving. This approach enables the use of MDD for functional specification and generation of fully executable functional code. The link with the platform is ensured by the platform aspects. For the large systems, this means that teams have always an access to an updated functional specifications containing all the business logic. The platform specific issues are managed in the centralized way. The platform aspects can be replaced to migrate the whole system to a new execution framework.

This approach, developed on top of Modelio tool (see Section 6.3) and its Java code generator, is limited to the platforms specific of the Naval group. The challenge, inside MegaMart, is to generalise this approach to address a wide spectrum of industrial systems and to cover more execution frameworks. This challenge implies to develop a library of pluggable aspects for several popular execution frameworks.

2.2 Methodologies, tools and frameworks for code generation

Automatic code synthesis from high-level models has been an active research subject for decades, and a great deal of efforts has been spent on it. In order to provide platform independency and reusability, high-level models, such as UML, SystemC or Matlab models, have been proposed for use as input to this process. There are several works in this area. In (Hartmann, Grüttnner, Ittershagen, & Rettberg, 2011) and (Abdi et al., 2010) it is presented a generic framework for HW/SW communication of functional tasks with shared resources. Additionally, (Herrera, Posadas, Sánchez, & Villar, 2003) and (Herrera & Villar, 2006) introduce methods for systematic embedded software generation from SystemC. All these works take advantage of the executable capabilities of SystemC in order to generate real executable implementations. As a result, the designer’s effort is reduced. However, there are two main disadvantages. First, these approaches use SystemC as system specification language , something which is not always the most suitable approach. Furthermore, they are limited in terms of exploration capabilities, since the evaluation of different design alternatives performed
during the design space exploration, such as concurrency exploration or SW components and cores mapping, usually requires the code to be manually modified.

Different solutions have been proposed for embedded SW generation from Matlab/Simulink, both from the Mathworks company (“Embedded Code Generation - MATLAB & Simulink Solutions - MATLAB & Simulink,” n.d.) and from the research community (Chindris & Muresan, 2006). However, the use of Matlab/Simulink is specially oriented to the development of system components based on intensive data algorithms, but not to the development of the entire system, which is one of the challenges for the MegaM@Rt tools.

As an alternative, the application of UML models to embedded system design has gained increasing interest ((Vanderperren, Mueller, & Dehaene, 2008) and (Luciano, Grant, & Bran, 2003)). UML models have been demonstrated to be an adequate approach for creating high-level models from which implementation code can be generated. Several automated synthesis processes based on UML are characterized by the creation of state machine models or variations of them (Harel, Kugler, & Pnueli, 2005). In (Zhenxin & Weng-Fai, 2009), a method is proposed for synthesizing interfaces for heterogeneous IP (Intellectual Properties) block integration from UML models. Heterogeneous IP blocks are IP blocks provided by different vendors and offering different interfaces. The framework enables the generation of communication links among the system blocks from UML profiles used to model the system-level communication interfaces.

As an alternative, eSSYN (www.essyn.com) try to reduce the developers efforts, eSSYN is a software synthesis tool that automatically generates platform specific executable binaries from a component based model of a software application and an easy-to-produce model of the target hardware platform (supporting complex multi-core heterogeneous platforms). With eSSYN, and starting from a standard based application and platform model, embedded system designers can generate a complete set of target binaries for an embedded application in a matter of minutes. With the use of different system models (such as software component based model or hardware platform model) eSSYN (Figure 1) can generate all required code and system calls implementing communications among software components, and the files necessary to compile automatically the code. The tool will significantly shorten development time and allow for future reuse. However, the eSSYN methodology (Microelectronics Engineering Group & University of Cantabria, 2017) does not enable the automatic generation of source code and specific executable binaries with information that enables the integration of runtime verification. It is not possible to reuse the feedback generated by the execution of the system to enable the introduction of validation aspect in eSSYN that allows the automatic generation of runtime verification code

All the previous techniques are oriented to generating completely fixed models, especially in their concurrent structure. However, exploration of concurrency is not considered, which requires great effort on the part of the designers to find optimal design alternatives.
2.3 Model Execution

The Model-Driven Engineering aims to bring models as productive artifacts for the software development. Having productive models means that they are directly the base for obtaining the final software system, for instance through the ability to generate code from them. **The ultimate challenge within productive models field is to skip the implementation stage so that the models at design-time are fully reifying the whole system at run-time.** This can be partly achieved through model execution. This enables a seamless development between design and runtime levels as design models are directly becoming full parts of the running system without requiring dedicated implementation tasks.

Recent initiatives at the OMG such as fUML (Semantics of a Foundational Subset for Executable UML Models) (Fuml, n.d.) or ALF (Concrete Syntax for a UML Action Language: Action Language for Foundational UML) (Alf, 2017) enable adding executable behavior, for instance, to UML class diagrams, which are basically static structures. In addition, one can define executable DSMLs (Domain Specific Modeling Language). Such DSML are called i-DSML for interpreted DSML (Clarke et al., 2012) or xDSML for executable DSML (Combaremale, Crégut, & Pantel, 2012). Similar to other DSML, the executable DSMLs suffer from limitations like the ability to capture all the concepts defined by domain experts they are difficult to evolve when there are changes in the domain due to paradigm shifts or new regulations (Bryant et al., 2011; Sprinkle, Mernik, Tolvanen, & Spinellis, 2009).
The classical dichotomy between using a general purpose or standardized language (UML, fUML…) and defining our own modeling language fully dedicated to our needs exists also in the context of model execution.

Model interpretation or execution, is then a generalized idea where a model can be actioned or animated, prior to any kind of code generation, for simulation and validation purposes in the very early phases of the software development cycle. Indeed, with model interpretation, a model produced at design-time can be reused as-is at run-time, just as an input of a tailored execution engine (a.k.a model interpreter).

![Diagram](image)

**Figure 2: i-DSML Conceptual characterization of i-DSML**

Model execution has been studied by several works, notably (Breton & Bézivin, 2001; Cariou et al., 2013; Clarke et al., 2012; Combareale et al., 2012). All these works built a consensus on the rationale of model execution and how to design an i-DSML. **Figure 2** is the resulting conceptual characterization of (Cariou et al., 2013). An i-DSML is a specific kind of DSML whose metamodel contains two types of elements: Elements called ‘static’ which describe the steady structure of a model, and elements called ‘dynamic’ which indicate the global model state at a given execution step. In the case of a state machine for example, the static elements are states and transitions while the dynamic elements are the current active state(s). A precise execution semantics is attached to an i-DSML. It specifies how to make evolving the model at runtime or during simulation, acting only on the dynamic elements of the model under execution. Reconsidering the state machine example, it specifies how the transitions have to be fired according to both the current active state(s) and an incoming event, leading to modify the current active state(s). This execution semantics is implemented through an execution engine. The engine takes an executable model (conforming to an i-DSML) in input and is in charge of its interpretation, that is, making evolving it, thereby generating a sequence of execution steps (a.k.a an execution trace). In an EMF environment, the i-DLSC can be defined with an Ecore meta-model and the execution engine can be implemented in Java based on the EMF API or with a model-oriented action language such as Kermeta.

It is worthwhile mentioning that the aforesaid characterization assumes that the current state of a model under execution is stored in the model itself. This is not always the case, since knowledge about the current state can also be internally managed by the engine. As an example, PauWare is an execution engine for UML state machines, written in Java. A detailed description of PauWare can be found in Section 6.5. Nevertheless, the genuine UML specification did not planned to store in a state machine diagram what are the current active state(s). Consequently, the PauWare engine is
responsible for this. Storing the execution state inside the model rather than inside the engine has the advantage that it provides a self-contained execution trace. Thanks to that, it is possible to perform failure recovery or to apply verification techniques onto the trace.

Embedded at runtime in the final system, an executable model defines the behavior of the running system, that is, when its business actions have to be executed and under what conditions (Cariou and al. 2016). For instance, a finite state machine can control the activation of elements of an elevator system (opening and closing its doors, winding/unwinding the cable to reach a given floor...). As another example, a travel booking system inserts customers data into database or call Web services provided by air transport companies. Similarly, the calls to the various Web services may be orchestrated through a BPEL (Web Services Business Process Execution Language (“WS-BPEL,” n.d.) or BPMN (Business Process Model and Notation (“BPMN,” n.d.) model.

The main challenge in the MegaM@Rt project with respect to model execution will be in defining new or improving the current modeling languages and their associated tool-support to focus on different aspects of the system (both functional and non-functional) and to allow working seamlessly with large system specifications.

3. Runtime trace analysis

Preservation of properties is an important aspect in model-driven development. Throughout the chain of transformations from high level models to lower level ones until executable code is generated, as well as during the execution of the code on its platform, various mechanisms need to be in place to ensure that properties of interest are preserved and respected. In other words, it is important to verify that the actual behavior and properties of the system at runtime are in compliance with those expected and defined at the model level. For this purpose, a runtime monitoring and log analysis framework is necessary in building and providing a round-trip support in the development chain, constituting from design models down to code and its execution on the platform and back again to the models. In such a round-trip chain, monitoring information can be processed and propagated back to the design models where the expected behavior is then compared with the actual behavior. If necessary, refinements and modifications may be done to the model(s) in order to improve system with respect to the desired set of properties and behaviors. This may well be used towards optimizing design models for specific properties; e.g, for system performance (Mehrdad Saadatmand, 2015).

Runtime monitoring of systems can not only help in evaluating the system behavior at runtime but also can be used to mitigate and prevent undesired behavior through runtime adaptation and reconfiguration.

Trace analysis is an important activity when ensuring the quality of a system by allowing one to reason about or predict (potential) failures of the system. Analyzing execution records can be used for a range of purposes, from acceptance testing, debugging and test generation to performance improvement. However this area is still harmed by the lack of appropriate methods for mining large volume logs. In the following, we present a series of techniques for monitoring and log analysis which allow one to observe and reason about the execution of the system at run time.

Time-series data analysis (Jirkovský, Obitko, Novák, & Kadera, 2014) is another technique used. A time series is a series of data points which is indexed in time order. Typically, a time series is a sequence taken at successive equally spaced points in time. Therefore, it is a sequence of
discrete-time data. Time series analysis entails methods for analyzing time series data to extract meaningful statistics and other characteristics of the data. Jirkovsky et al. (Jirkovský et al., 2014) broadly state that time-series data can be classified into two different categories: 1) offline (or batch) data, which is collected, stored and retrievable at a desirable point in time, and 2) online (or real-time) data, which is collected and analyzed directly upon occurrence.

In a different work (M. Saadatmand, Sjödin, & Mustafa, 2012), the authors introduce a solution and a framework which enables one to collect detailed monitoring information about timing behavior of systems processes and detection of critical events such as deadline misses, and execution time overruns. The suggested solution also helps with predictability of the system behavior at runtime in the sense that it enables to observe how close tasks are to missing their deadlines and whether this gap is decreasing or increasing. Based on such analysis of monitoring information, the system can then adapt itself in order to prevent deadline misses. In (Mehradad Saadatmand, Cicchetti, & Sjödin, 2012) an adaptive approach and algorithm for selection of encryption algorithms is suggested for systems which need to balance their services at runtime in order to achieve their timing constraints. The approach works by logging and analyzing timing behavior of an application at runtime and determining which encryption algorithms to use to mitigate violation of timing constraints. Figure 4 represents the steps of a typical log analysis process (Jagreet, 2017).

![Figure 4](image)

- **Data Collection**: Various sources can generate different formats of log files. As a first step in the process of log analysis, we need to capture the generated log data from all sources.
- **Data Cleaning**: Generally, there may be some corrupted log data among the captured logs. Therefore, performing a data cleaning process is necessary, where, irrelevant entries should be removed from the logs.
- **Structuring of Data**: Prior to analysis, the logs need to be preprocessed because log data can be complex, large and generally unstructured, incomplete, noisy and inconsistent. Since, the presentation of log data directly affects their ability to correlate with the other data, we need to be confident about the usage of collected log data.
- **Data Analysis**: As a last step, the structured log data will be analyzed which can be performed through applying different methods such as machine learning, pattern recognition, normalization, classification, correlation analysis and etc. Here, indexing and crawling can serve as two important aspects of the log analysis. Without the mentioned perspectives, updating of data will not occur properly within time and the chance of duplicates values can be increased.

Considering the ever increasing complexity of software systems today, one of challenges in log analysis is how to manage and analyze the huge amount of unstructured data that can be produced and collected from a system. Some of the data mining techniques which can help in this regard and be applied for analysis of large volume logs are briefly mentioned as follows. Data clustering is one of the proposed techniques for analyzing large volumes of log files through mining line patterns from log (Tahvili et al., 2016). A data clustering algorithm works based on similarity
between objects, where a set of similar data objects will be tagged as a cluster. Moreover, the objects in each cluster are dissimilar as possible to objects from other clusters. The objects which do not fit to any clusters should be detected and marked as outliers. Furthermore, analyzing the cluster of outliers could represent some unexpected behavior of the system such as unknown fault, which deserves more accurate investigation. Over the past decades, several clustering algorithms have been designed for solving high dimensional data problem, such as CLIQUE, MAFIA, CACTUS and PROCLUS (Tahvili et al., 2016). One of the prominent points in the mentioned algorithms is the ability of discovering clusters in subspace, where the algorithms do not attempt to measure distance between individual points. Moreover, as a weak point for the proposed algorithms, we can mention the high runtime overhead. Moreover, several algorithms have been proposed that explore the search space in a depth-first manner (Giri, Bhatt, & Bhatt, 2016), for instance we can mention Eclat and FP-growth as some important depth-first algorithms for mining frequent item set (Giri et al., 2016).

Finally, (Tahvili et al., 2016) proposed an online decision support system for test case selection and prioritization based on the dependency information between test cases. In this work, the test results of the executed test cases in the level of integration testing, have been monitored by us. Additionally, a set of decisions for selecting the next test case for execution has been made based on the test results, dynamically.

### 3.1 Monitoring Solutions

In the context of MegaM@Rt2, different activities involved in Runtime Analysis methods and tools, as defined in the FPP, can be schematized as composed of two main actions: information extraction and information manipulation.

More specifically, such activities are the following:

- **Model-Based Runtime Validation**, considering models@runtime. Indeed, in this context models and system contents are causally connected. If a final target is considered for connection with the model, it is necessary to identify an infrastructure for information extraction and a system for correlation of these traces with the model.
- Models used to help in monitoring and storing the current state of the system by defining an execution trace. This trace can be analysed either at runtime, in parallel with the system execution, or afterwards. The traces from the system execution have to be extracted with a monitoring infrastructure.
- **Model Based Runtime Verification**. Properties of a given system can be identified by a formal language and automatically translated to a monitor, which checks their correctness against the implementation. In the monitor translation, it is necessary a monitoring infrastructure.

This section focuses on the concept of monitoring infrastructure and the related mechanisms for information extraction that exist in literature. A monitoring infrastructure can provide information that, depending on the purpose of their goal, are processed in different ways. For example, they can be used to validate a system, to verify a property of a system, to identify a profile of behaviors of system components (buses, cache memories, etc), an action usually called profiling.

Two types of monitoring systems can be identified: Hardware and Software.
3.1.1 Software monitoring solutions

A software monitoring solution is based on the exploitation of the processors that are executing the application under examination to collect data useful for monitoring. For example, the execution time measurement of a task could be carried out in two possible ways: the first is to activate a timer (if available in the system) when the task starts and stop it when the task ends. In general, being the timer a shared resource among processing elements, another approach is the sampling of the program counter at predefined intervals, by using some interrupts. In both cases, there is an instrumentation of the application: in the first the instrumentation is the call to a routine that activates/deactivate the timer and stores information (i.e. the timestamp), while in the second is used to generate interrupts in order to sample the internal state of the system (i.e. the program counter). There are various examples of Software based monitoring systems, that depend of the application: the memory exploitation analysis is one of these, and in this context there is Gprof (Graham, Kessler, & McKusick, 2004), the GNU statistical profiler, that is able to offer a profile of the behavior of the application on the target by counting the number of times a function is executed, and to estimate the execution time of the routines. Another example of software profiling tool is Raptime (“Rapita Systems | On-target software verification solutions,” n.d.), that starting from instrumentation of the code, takes timestamps of processor during execution of the application. Then, it applies some algorithms on the collected information in order to estimate WCET. Another branch of application of software profiling systems is the Runtime Verification (Nelissen, Pereira, & Pinho, 2015).

Software profiling necessarily introduces some overheads on execution time and, considering the sampling approach, it has some grade of statistical inaccuracy.

3.1.2 Hardware monitoring solutions

Hardware monitoring systems are based on dedicated hardware resources able to carry on the profiling action. This means that no source code instrumentation is needed and the software execution by the central processor unit is not altered, thus no overhead on execution time is introduced. For the same reason, hardware solutions can guarantee the best accuracy in performance analysis. However, these solutions require a larger silicon area occupation for system implementation. Other possible disadvantages are the difficulty to correlate low-level measurements to source code performance metrics and the limited number of allocable hardware resources, that often forces to collect desired performance metrics by means of multiple tests. Various examples of hardware based profiling approaches have been presented in literature. For example, in the past years there were SnoopP (Shannon & Chow, 2004) and Airwolf (Tong & Khalid, 2008), two function-level profilers for software applications running on soft-core processors.

There is interest toward the research of smart monitoring systems. In particular, the following characteristics are defined for next generation monitoring solutions (Gibert, Martínez, Madriles, & Codina, 2015):

- **Speed**: collect profiling information in the shortest time possible, in order to allow which use the monitoring information to react in time;
- **Cost**: small overheads in terms of performance and area occupied;
- **Flexibility**: to allow different types of usages, in order to take both architectural events and micro-architectural ones, adapt to different scenarios, working in heterogeneous multi-core environments;
- **Accuracy**: precise correlation with other information about the system;
• **Fine grain information:** capability to collect information related to instruction granularity;

A monitoring system can be useful in different applications, such as support for software developers for debugging systems obtained through HLS tools (Goeders & Wilton, 2015). Monitoring and debugging extra-functional properties such as performance under workload is required in the context of MegaM@Rt2, and there exist tools that allow it for the performance in reconfigurable logic scenarios (Shannon, Matthews, Doyle, & Fedorova, 2015). In ASIC processors, there are various examples of smart monitoring solutions: AMD Lightweight profiling feature (“AMD, ‘Lightweight profiling specification,’” n.d.) and Intel Processor Tracing (Intel, n.d.) are composed of hardware facilities (such as hardware performance counters) and software able to use information acquired at low-level. They are supported by necessary libraries to interpret collected data. In Leon3 based scenarios, Nam et al. (Ho, Kaufmann, & Platzner, 2014) proposed a performance monitoring unit integrated with perf_event API. Xilinx supports system level profiling using SDSoC environment (Xilinx, n.d.-b), using performance counters in ARM Cortex A9 and performance monitoring units in programmable logic side. Moreover, it offers profiling solutions also in MicroBlaze soft-processor (Xilinx, n.d.-a). In order to define a custom monitoring system for embedded applications, solutions based on specific metric definition and implementation of necessary parts have been considered (Moro et al., 2015). This technique conducted to a definition of a library of elements, to be used to compose a hardware monitoring solution tailored for the application (Valente et al., 2016).

In the case of HLS tools such as LegUp (Toronto, n.d.), there is a hardware mechanism that is customized for debugging this kind of circuits and there is series of API that abstract debug done on the final target up to the layer of C code (Goeders & Wilton, 2015). Considering, instead, the runtime analysis for the non-functional properties of a system, such as WCET, Rapita Systems (“Rapita Systems | On-target software verification solutions,” n.d.) tools represent an example of APIs built on top of an instrumentation mechanism.

The future way to perform monitoring solutions is based on hardware/software collaborative approaches where hardware implements the information extraction (hooks and mechanisms) and software implements the functionalities that use collected data (algorithms, complex heuristics, etc). More precisely, future research will have to focus on the following points:

- The first is the possibility to tailor and customize the monitoring system for the system under exam: it depends on when to use the monitoring action (i.e. during the lifecycle to characterize the system or during development phases to support the designer). It depends on platform selected for the system (ASIC, reconfigurable logic). Other considerations should be done referring to non-functional properties of the system itself (how much overhead can be inserted, if a real-time profiling action is required, etc.).
- The second is the development of a framework able to support the designer on the selection of monitoring solution.
- The third is to integrate this framework with a support to provide the best monitoring infrastructure starting from design requirements.

In this context, MegaM@Rt2 will **generalize the concept among different monitoring goals by defining a general reference architecture that can be adapted to different applications** (e.g. runtime verification, runtime validation, etc.). This is the first step to define a methodology that can support the monitor selection adapting to different requirements, while also considering impact on non-functional properties.
3.2 Analyzing test execution logs

3.2.1 Predicting test execution times

When it comes to test generation, one challenge which arises particularly for systems with strict real-time constraints is predicting execution times that are used in test cases for determining faults in the system. In this context, assigning a large timeout value for a test activity can lead to a long testing process which is not efficient in term of using allocated testing resources. On the other side, without allowing a test case sufficient time for execution, we might face an increased risk of failure. Log monitoring and analysis is one of the approaches that we can utilize for assigning a correct timeout value for a test activity. Today, this is done manually, where based on previous experiences, a developer assigns a timeout value for various step of a test cases. In a recent work in (Tahvili, Saadatmand, Bohlin, Afzal, & Ameerjan, 2017), a method for predicting execution time for manual test cases has been proposed. Through analyzing a wide range of log files, it enables to predict the actual required time for test execution by applying linear regression technique.

3.2.2 Visualizing and debugging test traces

UPPAAL is an integrated tool environment for modelling, validation and verification (via model-checking) of real-time systems (Behrmann, David, & Larsen, 2004). The environment assumptions and system requirements are modelled as network of timed automata referred as UPPAAL-TA. One of the very useful features of UPPAAL is the simulation engine which allows one to execute the model in abstract time and observe the channel synchronizations between different automata and the values clock and integer variables in the model. The UPPAAL verification engine has been extended by different tools for different purposes, including on-line model-based testing.

For instance, UPPAAL-TRON (Larsen, Mikucionis, Nielsen, & Skou, 2005) is such a tool, which is used for input/output conformance testing of real-time systems. UPPAAL-TRON partitions a given UPPAAL-TA specification, developed in UPPAAL, into a SUT partition and an environment partition. During testing execution, the tool generates symbolic test traces which are projected into test sequences during the interactions with the implementation of the system. In practice, UPPAAL-TRON generates test inputs and checks the test outputs based on to the observable communication between the two partitions. A test adapter is used for converting abstract test inputs to concrete test inputs and, in the opposite direction, for converting concrete test outputs into abstract ones.

While model-based online testing brings benefits in terms of reducing the state space stored in the memory, addressing non-determinism better, and executing long test runs, it suffers from difficulties in analysing and diagnosing the test results in case of long lasting test runs (Hessel et al., 2008). In many situations, especially when a failed or inconclusive test results are encountered, one would like to understand and visualise which traces were explored in the model during the test session, which symbolic states have been visited, and which were the values of both clock and integer variables in each symbolic state.

The test sequences generated by UPPAAL-TRON against the implementation are not reproducible due to the random choices of inputs, time delays, and different optimization techniques used for reducing the symbolic state space used by the test generation algorithm of UPPAAL Tron (Larsen,
Mikucionis, & Nielsen, 2005). However, a generic approach to reconstruct the symbolic trace corresponding to a given conformance testing session of UPPAAL-TRON was proposed (Iqbal, Truscan, Vain, & Porres, 2017). The approach recreates the symbolic trace (sequence of states and state transitions) leading to the verdict state and can be applied to any sequence generated by UPPAAL TRON regardless of its test verdict or number of test events. Symbolic trace reconstruction permits to take advantage of UPPAAL’s capabilities for simulation and visualisation in order to improve the debugging process and to reduce the cognitive effort needed to identify the underlying causes of inconclusiveness or failure.

The SEAT tool (Software Exploration and analysis Tool)(A. Hamou-Lhadj, Lethbridge, & Fu, 2005) implements several operations that can help software engineers understand the content of a large execution trace. It is based on the Compact trace format (CTF), which is a scalable exchange format for representing traces(Abdelwahab Hamou-Lhadj & Lethbridge, 2004).

SEAT user interface is based on the Eclipse platform and consists mainly of a multiple-page editor and a set of auxiliary views. SEAT is developed for exploring large execution traces of routine (or methods) calls. The tool takes traces of routine calls as input and displays them using visualization techniques based on a tree-like control window. To help the user extract useful information, this tool implements several trace compression techniques.

3.3 Runtime trace analysis in partitioned systems

In this section, we discuss a different set of techniques for runtime analysis, this time in the context of partitioned systems. Partitioned systems provide spatial and temporal isolation in systems with different criticality applications. Moreover, in some domains it is needed the integration of a large number of functionalities in the same execution platform. Partitioned systems also eases certification to safety standards.

ARINC 653 (Avionics Application Standard Software Interface) (Engineering, 2007) is a software specification for space and time partitioning in safety-critical avionics real-time operating systems (RTOS). The ARINC 653 standard for partitioned systems provides the LogBook system as part of the Extended services (in Part 2). The abstract notion of a logbook is used to store messages that can be exploited, for example, to generate runtime traces. The logbook retains the stored data after a power failure; the data can be recovered when power is restored to the module. The logbook (Engineering, 2007) content and status is not altered by partition reset. Each logbook is accessible by only one partition. The logbook consists of a buffer in RAM and a Non-volatile Memory (NVM) area.

The kernel of operating system that provides the partitioning to the system can also implement its own trace service. For example, XtratuM provides a mechanism to store and retrieve the traces generated by partitions and XtratuM itself (XtratuM, n.d.). Traces can be used for debugging, during the development phase of the application, but also to log relevant events during the production phase.

In order to enforce resource isolation, each partition (as well as XtratuM) has a dedicated trace log stream to store its own trace messages, which is specified in the @device attribute of the Trace element. Trace is an optional element of XMHypervisor and Partition elements.

Runtime verification in partitioned systems deals with error detection and recovery. The mechanism used in partitioned systems is the Health Monitoring (HM). This facility detects and reacts to anomalous events or states. The purpose of the HM is to discover the errors at an early stage and try
to solve or confine the faulting subsystem in order to avoid or reduce the possible consequences. As defined in ARINC 653 the HM is the function of the OS responsible for monitoring and reporting hardware, application and OS software faults and failures. The HM helps to isolate faults and to prevent failures from propagating.

3.4 Automatic runtime feedback into the model and learning-based log analysis in real time

Although logging system behaviour and analysing logs later offline is a common way to maintain production systems, this method does not usually incorporate high-level considerations that can be mostly done only by system architects. Moreover, architects plan systems before they run in production so the need for redesign cycles forms an integral part of normal development of most systems. A different approach was implemented in the JUNIPER project (M. Rychly, 2015) where an advisor component automatically monitored the production deployment and, based on the model of the system, it gathered performance statistics and generated predictive advices for administrators and architects. Moreover, advices and aggregated information from production usage were imported back into the modeling framework and displayed to the architect within the model of the system.

As a commonly conducted case we consider a distributed platform providing a standard set of tools covering a wide range of tasks, from modeling to monitoring. Using such platform, an architect models the application in a modeling framework and a developer implements the application following the MDE. The application is then deployed on a hardware platform by administrators that have to decide on the right placement of the application’s components. All this requires a deep knowledge on both the software application and the hardware. During the application operation, the administrators have to monitor its performance using various metrics of the infrastructure or metrics built into the application itself, by the developers. This, again, requires highly skilled and experienced architects, developers, and administrators. Finally, the process of application improvement or redesign can only be done through manual analysis of monitoring data, which often means that first changes are made in the design phase based on the pre-production testing of developed application. The Advisor component discussed in (M. Rychly, 2015) in cooperation with heterogeneity-aware scheduler from (M. Rychly, 2014; Rychlý, Škoda, & Smrž, 2015) aims at improving the basic monitoring of the distributed stream processing applications, detecting common problems of distributed applications, and helping to solve these problems.
Figure 4: Tasks of the advisor component—thin arrows show sequence of tasks, thick arrows show dependencies of tasks and platform’s components

The advisor component operates in the following way. Whenever the application is deployed either for pre-production or production use, the monitoring data gathered over time may be exploited for further performance tuning. The advisor component detects through different plugins different suspicious states and behaviours. The gradual process from a modeling to a production use of the advisor component is depicted in Figure 4. For a seamless integration of the advisor into the process of MDE, advisor’s output can be displayed right in the modeling software. The reason for this is mainly an easier process of application’s modification and performance tuning by the architect. To achieve such interoperability, the advisor component uses an XML exchange file that can be loaded by a general MDE environment. The advice’s data are then displayed in the model, tied to corresponding objects in the model.

The architecture of the advisor component is modular with plug-ins. It enables developing specialized analyzing plug-ins for various design errors of distributed systems. A basic set of plug-ins was implemented and successfully tested:

1. Transfer overhead plug-in detects application components that spend the most of their time in data transference instead of computation (i.e., there is a longer communication time than a computation time where the communication time includes a time spend by synchronously receiving and deserializing of the data). This may indicate a very simple program with a high data transference overhead or a program waiting for data most of the time.
2. Out of memory prediction plug-in detects application components where the memory usage is growing over time, and which may eventually run out of memory. The problem is detected by linear regression analysis to identify a linear trend in memory usage.

3. Garbage collection performance plug-in detects application components that spend significant time in garbage collection (a form of automatic memory management in Java). The plugin analyses a total time spent on garbage collection and a number of the garbage collections.

The described approach was successfully employed to detect design flaws in multiple distributed systems which led to significant performance gains. This way, the development process of distributed systems can be simplified and, at the same time, even less experienced architects and developers may design effective distributed systems.

4. Runtime verification and Online testing

Runtime verification (RV) and model-based testing (MBT) are two complementary methods for evaluating the implementation of the implementation under test (IUT). Both approaches use abstract specifications, specified in different formalisms, such as state machines, linear-temporal logic, to describe the expected behavior of the system. However, there are sensible differences between the two methods in the way they interact with the IUT. While in runtime verification the IUT is “observed” and verified against a set of properties, in MBT test inputs are provided to the SUT, while test outputs are observed and checked against the specification. In addition, RV typically looks at a particular execution of the system (a trace) while testing investigates multiple traces. In the following, we overview the state-of the art, current technologies and the tooling available for these two fields.

Different approaches apply runtime verification or online model-based testing independently, where others integrated the two methods. In the following, we discuss those approaches that are more relevant for the MegaM@Rt project.

4.1 Runtime verification

*Runtime verification* is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system (program) under scrutiny satisfies or violates a given correctness property. According to (Grigore, Distefano, Petersen, & Tzevelekos, 2013) runtime verification is the monitoring of program executions to detect specific error traces which correspond to violations of desired properties. Whenever a violation has been observed, it typically does not influence or change the execution of the program, for example, to repair the observed violation.

Three verification techniques have been traditionally used: theorem proving, model-checking and testing with different goals and levels of automation (Leucker & Schallhart, 2009).

In order to understand the wide variety of runtime verification approaches, it is important to notice that checking whether an execution meets a correctness property or not is typically performed by using a monitor. In its simplest form, a monitor decides whether the current execution satisfies a given correctness property by giving a verdict of either yes/true or no/false. An overview of monitoring techniques has been provided in Section 3.1.
One of the distinguishing features of runtime verification compared to other verification techniques is due to the fact that verification, at least in online monitoring, is performed while executing a program. This offers the possibility to react to violations of correctness properties. Furthermore, runtime verification allows to react on faults, before they turn into failures.

Research effort has been spent to deal with the combination of test-case generation and runtime verification, for example, (i) combining the input domain of the program with runtime verification, where execution traces are monitored and verified against properties expressed in temporal logic (Artho et al., 2005), (ii) allowing for high-level specifications of monitors for temporal assertions within the JUnit framework (Decker, Leucker, & Thoma, 2013), or (iii) implementing property and information monitors in AspectJ in order to log several aspects of the system without interfering with its source code (Dimjašević & Giannakopoulou, 2015).

One of the main challenges related to software quality analysis is represented by models to capture the system behaviors. As remarked in (Alalfi, Cordy, & Dean, 2009), for instance, verification and testing of web software require effective modeling techniques that address the specific challenges of web applications. The work in (Alalfi et al., 2009) surveys modelling methods used in web site verification and testing.

Recent challenges and advances in Model-Based Testing (MBT) are discussed in (Alexandre Petrenko, Simao, & Maldonado, 2012; A. Petrenko, Timo, & Ramesh, 2015; Utting et al., 2016). MBT has been applied to test different quality attributes of distributed systems such as security, performance, reliability, and correctness (Saifan & Dingel, 2010). The work in (Häser, Felderer, & Breu, 2014) surveys software paradigms, assessment types and non-functional requirements in model-based integration testing.

Model-based testing is applicable to testing both functional and non-functional properties. An important advantage is that the model itself can provide reference values on expected SUT outputs, which can be used for verdict construction in the oracle step. The disadvantage is that models must be continuously maintained to reflect requirements, design or implementation changes. There are some approaches for generating models based on formal requirements (Maier & Zündorf, 2003; Whittle & Schumann, 2002) but more research is needed on how to accomplish this in a continuous integration environment.

Runtime software monitoring is related to a particular runtime verification technique. Unlike Section 3.3 that refers to traces extraction mechanisms, this section talks about runtime verification starting from the extracted traces. In the following, we provide an overview of software monitoring tools for runtime verification that have been described in the literature (Delgado, Gates, & Roach, 2004):

- **Alamo** stands for A Lightweight Architecture for Monitoring. It is developed for C and Icon programs, uses the Icon programming language to specify assertions. The Alamo monitor architecture reduces the difficulty of developing dynamic analysis tools, such as special-purpose profilers, bug-detectors, and program visualizers. (Jeffery, Zhou, Templer, & Brazell, 1998);
- **DynaMICs** stands for Dynamic Monitoring with Integrity Constraints. This is a software tool that facilitates the collection and use of constraints for software systems. In addition, it supports traceability by mapping constraints to system artifacts. Constraint specifications are stored separately from code; constraint-monitoring code is automatically generated from the...
specifications and inserted into the program at appropriate places; and constraints are verified at execution time. These constraint checks are triggered by changes made to variable values. DynaMICs is being designed for C, C++, and Java programs. (Gates, Roach, Mondragon, & Delgado, 2001);

- **Jass** stands for Java with Assertions is a general-purpose monitoring approach that is implemented for sequential, concurrent, and reactive systems written in Java. The Jass tool is a pre-compiler that translates annotated into pure Java programs in which compliance with the specification is dynamically tested. Besides the standard Design by Contract features known from classical program verification (e.g. pre- and postconditions, invariants), Jass additionally supports refinement, i.e. subtyping, checks and the novel concept of trace assertions. Trace assertions are used to monitor the dynamic behaviour of objects in time. (Bartetzk, Fischer, Möller, & Wehrheim, 2001);

- **Java PathExplorer (JPaX)** is a system for monitoring the execution of Java programs. The system extracts an execution trace from a running program and verifies that the trace satisfies certain properties. An execution trace is a sequence of events, of which there are several kinds as we shall discuss below. Two forms of monitoring are supported: temporal verification and concurrency analysis. The concurrency analysis requires no user provided specification. (Klaus Havelund & Roșu, 2004);

- **Temporal Rover** is a specification-based verification tool that uses Linear-Time Temporal Logic (LTL) and Metric Temporal Logic (MTL). In Temporal Rover, the user determines the point at which a property should be checked and inserts an annotation of the property. The Temporal Rover parser converts an annotated program into an identical program with the properties implemented in source code. During application execution, the generated code validates the executing program against the formal specifications. Temporal Rover takes a Java, C, C++, Verilog, or VHDL source code program as input. It enables customizable actions for the program domain. (Delgado et al., 2004).

### 4.2 Model-based testing

In model-based testing, an abstract model is built which reflects the behavior of the system. Later, the abstract model is used to generate test cases (Utting & Legeard, 2010). Depending upon how tests are executed there are two versions of MBT: offline and online. In offline testing, the generated test cases are executed on the SUT after complete creation. The benefit of using the offline testing is that the entire test suite can be optimized before execution and stored for later use. However, in online testing, the test cases are generated and directly executed on the SUT one after the other. As a result, the testers cannot apply post-optimization to the test suite. On the other hand, online testing is adaptive and more suitable than offline testing when the model or the SUT is non-deterministic, and it can execute tests over longer periods (Utting, Pretschner, & Legeard, 2012).

In their work on building a taxonomy of MBT, (Utting et al., 2012) describe several technologies used for test generation among which model-checking, symbolic execution and search based algorithms. In addition, many researchers performed extensive studies on existing MBT methodologies and tools. For instance, (Dias Neto, Subramanyan, Vieira, & Travassos, 2007) provided an overview of MBT approaches, (Aggarwal & Sabharwal, 2012; Broy, Jonsson, Kateon, Leucker, & Pretschner, 2005; Shafique & Labiche, 2013) synthesized literatures on test generation from state-based models, (Fraser, Wotawa, & Ammann, 2009) surveyed different approaches for testing with model-checking tools, while others focused on test generation approaches for test case generation (Dave & Agrawal, 2015).
Most work on MBT has focused on off-line testing. However, Examples of MBT tools capable of online testing include TorX (Tretmans & Brinksma, 2002), TEMA (Jaaskelainen et al., 2009), and UPPAAL TRON (Larsen, Mikucionis, Nielsen, et al., 2005). In the following, we present couple of complementary techniques which complement the previously discussed studies, and which we plan to develop further and expand to suit industrial requirements in the context of MegaM@Rt2.

4.3 Combining automatic test generation and verification

There have been a number of methods and techniques for test generation and verification developed during the past few years (Fraser & Arcuri, 2011; Pacheco, Lahiri, Ernst, & Ball, 2007; Tillmann & de Halleux, 2008; Visser, Pâsăreanu, & Khurshid, 2004). For example Randoop (Pacheco et al., 2007) creates random tests by using feedback information as search guidance. EvoSuite (Fraser & Arcuri, 2011) is a tool based on genetic algorithm for Java programs. Some researchers have proposed the combination of test generation and runtime verification (Arcaini, Gargantini, & Riccobene, 2013; Artho et al., 2005; Dimjašević & Giannakopoulou, 2015; Falzon & Others, 2011; Falzon & Pace, 2013) such that the properties to be verified are checked automatically using the inputs to the software to be tested. In this section, we focus on techniques and tools that combine automatic test generation and verification using model checkers. Model checking task is intended to cover all execution paths through an application. This is also valid for concurrent and multithreaded applications. This technique is very useful in software testing. It allows detecting defects, collecting runtime information and creating interesting test inputs and corresponding test drivers.

When combined with verification, the test generation is based on a model gathered during the actual software execution instead of a model collected during static analysis. The software is executed with some input parameters and during its execution, different runtime information are observed: executed paths, branches, and operators. Based on these observations, new additional input parameter values are generated in order to drive the execution by selecting certain control flow paths (Ferguson & Korel, 1996). Test generation can be used to improve code coverage by using a chaining approach (Ferguson & Korel, 1996). The main drawback of this approaches is that the execution of software has to be performed, which requires the preparation of the whole infrastructure or environment and potentially could not be performed automatically.

A model checker has been used to find test cases to various criteria and from software in a variety of languages (Black, 2000; Hong, Lee, Sokolsky, & Ural, 2002). Black et al. (Black, 2000) discuss the problems of using a model-checker for test generation using full-decision coverage. Rayadurgam and Heimdahl (Rayadurgam & Heimdahl, 2001) defined a method that can be used for coverage-based test generation using a model checker. Rayadurgam et al. (Rayadurgam & Heimdahl, 2003) described a method for obtaining MC/DC adequate test cases using a model-checking approach.

4.4 Static analysis techniques

The use of static analysis (in particular, of program slicing techniques) has been exploited in different context (see survey (Yoo & Harman, 2012)), for example, in order to: (i) identify definition-use pairs that are affected by a code modification (Gupta, Harrold, & Soffa, 1992); (ii) select test cases (Agrawal, Horgan, Krauser, & London, 1993); (iii) prioritize test cases (Jeffrey & Gupta, 2008), or (iv) reduce test suites (Arlt, Podelski, & Wehrle, 2014; Leitner, Oriol, Zeller, Ciupa, & Meyer, 2007;
Enoiu, A. (2016). Program slicing techniques have also been combined with search based techniques for test data generation (McMinn, Harman, Lakhota, Hassoun, & Wegener, 2011). Research efforts have also been spent in order to estimate energy consumption of embedded programs by using static analysis (see, for example, (Grech et al., 2015; Jayaseelan, Mitra, & Li, 2006; Wägemann et al., 2015), detailed here below).

The work in (Grech et al., 2015) uses static analysis to estimate the energy consumed when running a function under different platforms, using different compilers. In (Jayaseelan et al., 2006), it is presented a static analysis technique to estimate the worst-case energy consumption of a task on complex micro-architecture. Finally, in (Wägemann et al., 2015), for the worst-case energy consumption analysis of embedded systems, it is exploited the combination of different techniques, such as static analysis, energy model, and genetic algorithms. Another class of related papers use static properties of tasks (in particular, the Worst Case Execution Time (WCET)) in order, for example, to support stress testing of task deadlines by identifying worst case scenarios where tasks are more likely to miss such deadlines (Alesio, Briand, Nejati, & Gotlieb, 2015).

4.5 Provided technologies by project partners

In this section, we briefly overview different technologies provided by MegaM@Rt project partners in addition to the state-of-the-art discussed above. Additional tools provided in the project are summarized. The challenge in MegaM@Rt2 will be on how to adapt existing technology to serve the scalability and traceability requirements of industrial applications.

4.5.1 CompleteTest

CompleteTest (Eduard P. Enoiu, Čaušević, et al., 2016; E. P. Enoiu, Sundmark, & Pettersson, 2013) is a method in which the model is annotated and the properties to be checked are expressible as a single sequence. In contrast to other approaches, CompleteTest provides an approach to generate test cases for different code coverage criteria that are directly applicable to industrial control IEC 61131-3 software. In CompleteTest, the UPPAAL model-checker is used for automatic test generation based on code (Eduard P. Enoiu, Čaušević, et al., 2016; Eduard Paul Enoiu, Sundmark, & Pettersson, 2013) and mutation coverage criteria (Eduard P. Enoiu, Sundmark, Čaušević, Feldt, & Pettersson, 2016). For a detailed overview of testing with model checkers we refer the reader to Fraser et al. (Fraser et al., 2009).

CompleteTest (Yilmaz et al., 2014) is a tool for automatic test generation based on the concepts of model checking and timed automata. Code coverage criteria can be used by CompleteTest to generate test cases, provide 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 (as provided manually by the tool user). CompleteTest connects as a client to the model checker and verify properties against the model and a trace parser collects a diagnostic trace from the model checker and outputs an executable test suite containing inputs, actual outputs and timing information (i.e., the time parameter in the test is used for constraining the inputs in time). CompleteTest was extended to support mutation testing, a technique for automatically generating faulty implementations of a program for the purpose of improving the fault detection ability of a test suite (Eduard P. Enoiu, Sundmark, et al., 2016). More details about this tool can be found in Section 6.6.
A series of studies (Eduard P. Enoiu, Čaušević, et al., 2016; Eduard P. Enoiu, Sundmark, et al., 2016; E. Enoiu, Sundmark, Čaušević, & Pettersson, 2017) based on industrial use-case scenarios from Bombardier Transportation show the applicability of using automatic test generation in practice. These results indicate that automatic test generation is efficient in terms of time required to generate tests and scales well for industrial IEC 61131-3 software. Automatic test generation can achieve similar code coverage as manual testing performed by human subjects but in a fraction of the time. The results of these studies (Eduard P. Enoiu, Sundmark, et al., 2016; E. Enoiu et al., 2017) support the conclusion that automatically generated tests are slightly worse at finding faults in terms of mutation score than manually created test suites.

### 4.5.2 Online Model-Based Testing via Model Mutation using UPPAAL-TA

In complex systems (i.e., systems with many elements very interrelated), the challenge of the testing does not only rely on the correctness of the implementation, but also on being robust against invalid interactions. One way to assess the robustness of the system is to use mutation analysis technique. Extensive studies on model-based mutation technologies have been provided by several authors (Forostyanova & Kushik, 2013; Fraser & Wotawa, 2007; Lorber, 2015).

Conventionally, mutation analysis is used to inject artificial faults (known as mutants) in the source code of a system to evaluate a test suite. By combining mutation testing technique with model-based testing (MBT), the model of the SUT and thus, the test cases are mutated. In model-based mutation, mutation is done on the behavioral model of the system under test (SUT), instead of the source code. In such case, the mutated tests that can reveal implementation faults that were not found by normal test generations. In addition, model mutations identify defects related to missing functionality, unexpected functionality, or misunderstanding of specifications.

In (Siavashi, Iqbal, Truscan, & Vain, 2017), a model-based mutation testing approach based on UPPAAL-TA is presented for testing web services. In this approach, the UPPAAL-TA model of a web service is verified via model checking and its conformance against the SUT is checked via online testing. A mutation generator tool is developed to apply a set of mutation operators to systematically generate mutants from the UPPAAL-TA model. In order to reduce the number of unsuitable and invalid models and also increase the efficiency of testing, verification rules are applied (deadlock-freeness and reachability) to select a set of mutants (known as valid mutants), which were executed against the original SUT generating mutated test inputs. Some hidden faults are detected during model-based mutation testing.

In (Siavashi et al., 2017), the model-based mutation testing is applied to different types of web services in which the interactions among the environment's entities are important (such as social networks). The testing approach is evaluated by experimenting a Blog web service as a case study. The web service is implemented in REST architectural style and with timing constraints. Combined with the previous results on a different case study in which several faults were uncovered in the implementation (Siavashi, Truscan, & Vain, 2016), it shows this approach has the potential to detect faults that are not found via functional testing. One improvement compared to the previous work would be the automatic detection of equivalent mutants, which was previously done manually and which eliminates unsuitable mutants and reduces the testing time.
4.5.3 Online Model-Based Performance Evaluation

Performance testing is the process which is used to measure the responsiveness and scalability of the system when it is under a certain synthetic workload (Subraya & Subrahmanya, 2000). The synthetic workload is generated by simulating the workload in a real environment. Under synthetic workload, different key performance indicators (KPIs) are monitored to evaluate the performance of the system under test (SUT). For example (Chen, Moreland, Nepal, & Zic, 2008; Vokolos & Weyuker, 1998; Weyuker & Vokolos, 2000):

- Response time: elapsed time between the sending a request and the arrival of a response.
- Error Rate: percentage of total requests that resulted in an incorrect response due to high load.
- Throughput: number of requests processed within a unit of time by the SUT.
- Bandwidth: amount of data sent back and forth to the system during a specified interval.
- Number of concurrent users: number of parallel users that send requests to the system.
- Resource utilization: CPU, disk space, network bandwidth and memory usage.

It is important to generate a workload close to the real workload, otherwise the results would be inconclusive (Menasce, 2002). The challenge lies in simulating a correct user behavior. Every user has a unique and dynamic behavioral pattern, meaning the most users behave differently from each other. Traditional methods have shown difficulties in dealing with these issues (Draheim, Grundy, Hosking, Lutteroth, & Weber, 2006).

Traditionally, performance tests usually last for hours, or even days, and only test a predefined number of prerecorded scenarios that are executed in parallel against the SUT. The major drawback with this approach is that certain inputs that the system can face might be left untested.

In recent years, model-based testing principles have also been applied to the field of performance testing, by using models to specify the expected runtime behavior of the user of the system. The synthetic workload is then generated from these models by letting virtual users execute these models.

For instance, (Abbors, Ahmad, Truscan, & Porres, 2012) propose an approach and a tool for load generation and system monitoring from probabilistic models. The MBPeT tool generates the load by simulating the probabilistic timed automata (PTA) models describing the behavior of groups of virtual users. The load is applied on the system while being generated and different KPIs such as response times, throughput, memory, CPU, disk, etc. are monitored. MBPeT has a distributed architecture where one master node controls several slave nodes which are the actual load generators. This means that the MBPeT is very suitable for a cloud environment where computational nodes can be rented in an on-the-fly manner. Besides monitoring, the tool also produces an performance test report at the end of the test. The report contains information about important KPIs, such as response times, throughput etc., and graphs showing how CPU, memory, disk, network utilization varied during the test session.

The approach has been suggested for and evaluated on web-based systems. In the context of MegaM@Rt2 we will evaluate its applicability to other application domains and in an industrial context.
5. Model-based system monitoring through “models@run.time”

Models embedded during the execution of a software system are referred in a general way as “models at runtime”. They can be used for different purposes but when presented under the acronym “models@run.time”, they are dedicated to a particular use: managing the dynamic adaptation of the running system (Blair et al., 2009). Usually, such models endeavor to be the strict reflects of the running software, each one addressing a specific viewpoint on the latter. Abstracting away from details of the running system and focusing on specific concerns, they allow to perform a large range of analysis. Indeed, decision making is always easier at the model-level than on at the system-level. Decisions can then be eventually enacted onto the system that requires, in turn, the models to be updated accordingly and so on. For instance, (Morin, Barais, Jezequel, Fleurey, & Solberg, 2009) defines metamodels for expressing the features, context, reasoning and architecture of a system as support for adapting component-based applications.

Adaptation is usually carried out through a loop (See Figure 5) and can be for instance inspired from the MAPE-K loop defined by IBM for self-adapting systems (Computing & Others, 2006). The autonomic manager is first monitoring the system. For a MDE-based adaptation, it will lead to produce and update models reflecting the current state of the system. Then, the models are analyzed and if required, an adaptation action is planned and executed on the system.

![Figure 5 MAPE-K loop defined by IBM for self-adapting systems (Computing & Others, 2006)](image_url)

System adaptation is out of the scope of the MegaM@RT2 project; however, models@run.time approaches propose model-based monitoring techniques that can be used as a base for analysing the performance of running systems.

Now consider the specific case where the final running system is an executable model: the loop becomes a model to model loop. In that case, it may be preferable to simplify that loop into a single enriched model representing both the running system and all properties useful for its analysis (Cariou et al., 2013).

Even though the research in the field of models@runtime has made substantial progress in recent years, there are still many challenges to be solved (Bencomo, N., France, R.B., Cheng, B.H.C., Aßmann, U., 2014) in the context of the MegaM@Rt2 project:

- the need for a reference architecture,
- uncertainty tackled by runtime models,
Deliverable D3.1 Foundations for Model-based Runtime Analysis Methods

- mechanisms for leveraging runtime models for self-adaptive software,
- and the use of models at runtime to address assurance for self-adaptive systems.

6. Baseline technologies and tools provided by project partners

Below we provide a list of tools provided by project partners to support the activity of WP3 and on which the MegaM@Rt framework will be developed. Additional tools provided by MegaM@Rt partners have been discussed in Deliverables D2.1 and D4.1.

6.1 SeaFox

<table>
<thead>
<tr>
<th>Provided by</th>
<th>MDH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>SEAFOX is the only available combinatorial test suite generation tool for industrial IEC 61131-3 control software.</td>
</tr>
<tr>
<td>License</td>
<td>open source software</td>
</tr>
<tr>
<td>Documentation Resources</td>
<td><a href="https://github.com/CharByte/SEAFOX">https://github.com/CharByte/SEAFOX</a></td>
</tr>
<tr>
<td>Source Code</td>
<td><a href="https://github.com/CharByte/SEAFOX">https://github.com/CharByte/SEAFOX</a></td>
</tr>
<tr>
<td>Maturity Level</td>
<td>Beta</td>
</tr>
<tr>
<td>Contact</td>
<td><a href="mailto:eduard.paul.enoiu@mdh.se">eduard.paul.enoiu@mdh.se</a></td>
</tr>
</tbody>
</table>

SEAFOX is open source software and is available at https://github.com/CharByte/SEAFOX. SEAFOX supports the generation of test suites using pairwise, base choice and random strategies. For pairwise generation, SEAFOX uses the IPOG algorithm as well as a first pick tie-breaker (Lei, 2008). SEAFOX was used in several studies (Bergström & Enoiu, 2017; Charbachi, Eklund, & Enoiu, 2017) in order to support testing of industrial programs and fault detection. A tester using SEAFOX can automatically generate test suites needed for a given industrial IEC program after manually providing the input parameter range information based on the defined behaviour written in the specification.

6.2 Conformiq Designer

<table>
<thead>
<tr>
<th>Provided by</th>
<th>CON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Conformiq Designer enables automatic generation of functional box tests from system models. Combining best-of-breed mathematical algorithms with an Eclipse-based IDE for Automated Test Design, Conformiq Designer reduces the risk of missed tests by enabling companies to test for difficult and complex system scenarios.</td>
</tr>
<tr>
<td>License</td>
<td>Commercial</td>
</tr>
</tbody>
</table>
Conformiq offers a next generation solution for automatic software testing. Our products fit the needs of Agile software development by adapting quickly to new product requirements and eliminating the time required for laborious test execution script maintenance during short sprints. The low implementation and maintenance requirements of Conformiq products tackle the challenges of traditional test automation and enable your organization to test better, faster, sooner, and cheaper.

Instead of using test cases, Conformiq users have a model, which describes the System Under Test, or the product you want to test. From the model, Conformiq products use highly intelligent algorithms to automatically determine the necessary tests and test data, and automatically generate scripts for automated execution. Conformiq products also automatically create test case documentation in any language, and upload it to your Application Lifecycle Management or Test Management system. On design changes, our products automatically update the scripts test cases identifying which are new, which are the same, and which are no longer valid. The tests generated are optimized for fast execution and create known coverage, improving the quality of your product.

Key Capabilities of Conformiq Products

- Create and update automatically scripts for automated test execution systems.
- Create and update automatically test documentation for Application Lifecycle Management or Test Management systems.
- Automatically optimize tests for faster test execution and improved coverage.
- Flag changes in product requirements to make maintenance faster.
- Automatically create and maintains test data.
- Automate the execution of new and existing tests, and transform manual tests into automatically executable tests
- Extend industry-standard test automation tools, including existing and open source test execution tools

6.3 Modelio

<table>
<thead>
<tr>
<th>Provided by</th>
<th>SOFT</th>
</tr>
</thead>
</table>

| Short Description | Modelio is an open source modeling environment (Eclipse RCP) supporting various standard as UML2, BPMN2, SysML and MARTE (among other). Modelio also provides a broad-focused range of standard based functionalities as document generation, code generation, traceability, etc. |
| License | Commercial and also GPL and APL Licenses for the open source version. |
Modelio is an open source modeling environment (UML2, BPMN2, MARTE and SysML among others). Modelio delivers a broad-focused range of standards-based functionalities for software developers, analysts, designers, business architects and system architects. Modelio is built around a central repository, around which a set of modules are defined. Each module provides some specific facilities which can be classified in the following categories:

- **Core**: Modelio Modeler module is the only module belongs to this category. All other modules depend on this central element.
- **Scoping**: this category is composed of Goals, Dictionary & Business Rules and Requirement Analyst which allow specifying high level business models for any IT system.
- **Modeling**: for example, SysML, MARTE and BPMN are included in this category. The modules belonging to this category are used to model different specific aspects of a system such as Business Process, component architectures, SOA or embedded systems.
- **Code generators**: such as C++ or JAVA. These modules allow users to generate and to reverse the code to/from different programming languages.
- **Utilities**: modules allowing transversal utility facilities like teamwork module or XMI.

This architecture allows Modelio to be flexible and to be configurable simply by adding the Modelio modules. Thus users can dynamically change their configuration at any time simply by changing their choice of modules in the same repository. The Modelio functionalities depend on the user's modules choice. In this section, we highlight three functional sets which seem to be the most relevant in our context. These functional sets are the following:

- **XMI export/import**: The XMI module is the provider of this XMI import/export functionality. The module allows Modelio to exchange models, in XMI format, with external modellers such as Enterprise Architect, Artisan Studio, Topcased, Papyrus etc. Thanks to that, Modelio can exchange XMI models with a wide range of modellers including Papyrus.
- **MARTE models design**: MARTE is an OMG standard for modelling embedded and real time system. The Modelio MARTE Designer project provides dedicated MARTE editors to assist users in the modelling of embedded systems.
- **Generation**: Modelio has powerful code generation and reverse engineering modules for Java, C# and C++ language. Moreover, it is able to generate documentation in several formats (e.g., HTML or OpenXML) which can be stored in the Component repository.

### 6.4 Certifityt / MBeetle

| Provided by | SMA |
### Short Description
CertifyIt is a commercial model-based testing solution allowing to derive executable test cases from models edited by an Eclipse-based UML modeller, and is especially well-integrated into the RSA platform. Recently, Smartesting has developed a plugin prototype, called MBeetle, that combines both test generation and corresponding execution, and dynamically enabling the derivation of the test cases at the same time they are executing. Within the MegaM@Rt project, Smartesting aims at extending and improving this early prototype to provide an efficient and easy-to-use solution dedicated to online testing.

### License
Commercial

### Documentation Resources
www.smartesting.com/fr/certifyit/

### Source Code
N/A

### Maturity Level
CertifyIt: commercial - MBeetle: early beta version

### Contact
bruno.legeard@smartesting.com

Manual test design is labour intensive and error prone; this manual work can be avoided for complex applications by modelling the key concepts (abstraction) and allowing CertifyIt to automate your test design work. Indeed, CertifyIt is a tool suite that automatically generates test cases from a functional model of the system requirements and by applying testing directives. Since the model is more expressive and simpler than the system-under-test, it can more readily be reviewed for correctness and coherence, as well as be updated more easily. CertifyIt supports UML/OCL models as the specification modelling language that allows the models to have a precise and unambiguous meaning, so that the behaviour of those models can be understood and automatically manipulated. OCL expressions indeed provide the expected level of formalization necessary for model-based testing modelling. This precise meaning makes it possible to simulate the execution of the models and to automatically generate test cases from. The test generation process consists to derive test cases by applying well-known structural coverage criteria (all behaviours, all requirements, etc.) and also to cover dedicated test purposes that can be used as test objectives. CertifyIt thus enables to:

1. Import the test model described by a subset of UML/OCL notations.
2. Implement your testing strategy by importing test pattern directives.
3. Generate your tests from several criteria, and manage traceability between the model and the corresponding generated test cases.
4. Publish abstract test cases for documentation or manual exploitation.
5. Publish test scripts into a testing environment for automated execution.
6. Easily manage the evolution of the specification: when the model is updated, CertifyIt will generate the new test cases.

Based on this mature model-based testing solution, a plugin prototype, called MBeetle, has recently been developed to address online testing. This prototype allows to combine, in a single process, both test generation and corresponding execution. This extension dynamically enables the derivation of the test cases at the same time they are executing. Within MegaM@Rt project, Smartesting targets to extend and improve the MBeetle early prototype to provide a reliable and integrated demonstrator for online testing. The goal is therefore to achieve the required reliability and scalability level for its industrial use and adoption.
6.5 PauWare

<table>
<thead>
<tr>
<th>Provided by</th>
<th>UPAU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>PauWare is a Java API for specifying statecharts and an engine for executing them. PauWare implements the complete UML 2 state machines semantics. It can be embedded into Java SE, Java EE and Android applications.</td>
</tr>
<tr>
<td>License</td>
<td>LGPL license version 3 (<a href="https://www.gnu.org/licenses/lgpl-3.0.en.html">https://www.gnu.org/licenses/lgpl-3.0.en.html</a>)</td>
</tr>
<tr>
<td>Documentation Resources</td>
<td><a href="http://www.pauware.com">http://www.pauware.com</a></td>
</tr>
<tr>
<td>Source Code</td>
<td><a href="http://www.pauware.com">http://www.pauware.com</a></td>
</tr>
<tr>
<td>Maturity Level</td>
<td>Good, can be used in any development project</td>
</tr>
<tr>
<td>Contact</td>
<td>Eric Cariou: <a href="mailto:Eric.Cariou@univ-pau.fr">Eric.Cariou@univ-pau.fr</a></td>
</tr>
</tbody>
</table>

PauWare is a model execution tool based on the UML state machines semantics and implemented in Java. The main idea beyond model execution is that a model is specifying the behavior of a system. By behavior, we mean when and why calling business actions. For instance, a state machine can control the activation of elements of an elevator system, that is, when opening and closing its doors, winding or unwinding the cable to reach a given floor, depending of the interaction of the user with the interface of the elevator system. PauWare enables to program in full plain Java an UML state machine and to associate business methods with states and transitions of this state machine.

Specifying or implementing a UML state machine is made through the PauWare API. The API is defining Java classes and methods for creating states and transitions of a state machine. The complete UML state machines semantics is implemented within the API. One can build state machines with composite states, parallel regions, shallow and deep history states, etc. Transitions between two states are associated with an event (a String value) and optionally a guard, a business operation or a triggered event. States can be associated with business operations (entry, do and exit ones). Guards and business operations are implemented under the form of plain Java methods in a standard Java class.

The other main part of PauWare is the execution engine. Once the state machine built and associated with business methods in a Java program, the engine is able at runtime to execute this state machine. Concretely, one main method “run_to_completion” takes as parameter an event name. When called, the engine is searching the transitions corresponding to this event starting from the current active states, validating the guards of the transitions if existing, and then triggers these transitions to lead to new active states and executes associated business methods.

PauWare is a very light tool as the JAR file integrating the Java API and the execution engine is less than 90 KB. A Java SE and a Java EE versions are available as well as an Android adaptation.

PauWare is distributed with a prototype of a viewer. When used, a graphical view of the running state machine of a PauWare program is automatically displayed in a Web browser. This view is showing the
structural contents of the running state machine and the current active states with the last triggered transitions. However, it is not a real time display as the refresh is made with an interval between 1 and 3 seconds.

Currently, the developer has to implement the state machine in the Java program, by concretely writing the code lines building the states and transitions of the state machine. However, to avoid this task, code generation from UML diagrams or SC-XML files could be carried out. We are currently working on the development of these code generation tools. SC-XML stands for State Chart XML. It is a W3C standard for specifying statecharts in a XML format (https://www.w3.org/TR/scxml/).

6.6 CompleteTest

<table>
<thead>
<tr>
<th>Provided by</th>
<th>MDH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>CompleteTest is a tool for automatic test generation based on model checking. Code coverage criteria and mutation testing can be used by CompleteTest to generate test cases. The main goal of the design of the user interface was to meet the needs of an industrial end user. The function of the user interface is to provide 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 (as provided manually by the tool user).</td>
</tr>
</tbody>
</table>

| License | Free for non-commercial applications in academia only. |
| Documentation Resources | Documentation and resources are available at [http://www.completetest.org/](http://www.completetest.org/) |
| Source Code | Available on request. |
| Maturity Level | Early beta version. |
| Contact | [http://www.completetest.org/](http://www.completetest.org/) |

The tool is built from the following modules: an import editor used for validating the structure of a provided input file, a translation plugin that creates an XML format accepted by the UPPAAL model checker, an server plugin allowing CompleteTest to connect as a client to the model checker and verify properties against the model, and a trace parser that collects a diagnostic trace from the model checker and outputs an executable test suite containing inputs, actual outputs and timing information (i.e., the time parameter in the test is used for constraining the inputs in time).
1 – Step & Time columns are used to present to the user the number of test vectors needed for achieving the maximum coverage. Steps represent a cycle scan in FBDs while Time represent an external clock.

2 – Input values columns in this area are showing the values of input variables to a given FBD. These values are automatically generated by CompleteTest.

3 – Output values. Columns in this area by default showing false for Boolean variables and 0 for Numeric values. It is expected that a user will manually change those values to fit the expected behaviour of the system, based on the input values provided in the previous area.

4 – Achieved coverage. In this field CompleteTest is displaying the percentage of coverage items that are covered by the generated test inputs. It is important to note that the tool always provide the maximum achievable coverage value. This means that if we have a value of 80% this is the maximum that could be covered using the selected logic coverage. Most likely this is a sign of the dead code.

5 – Diagnostic information. In this field, a user is presented with diagnostic information regarding the state space that was explored and memory consumption that was consumed during the model-checking execution.

6 – Validate Test Outputs. Once a user has provided expected values, it is possible to click on Validate Test Outputs in order to compare expected and the actual output vector for each test input vector.

6.7 Lime

<table>
<thead>
<tr>
<th>Provided by</th>
<th>Space Systems Finland Ltd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>The LIME toolset provides novel tools for distributed component-based embedded systems. The challenge is to enhance the design of such systems so that system validation can be started in an earlier stage of the design cycle when the system has been implemented only partially. The toolset provides a lightweight interface specification method and some related monitoring and testing tools. Advanced features are extended interface specification methods and testing techniques, which enable incremental testing, through automatic test generations, for distributed components already in early stages where the systems has been implemented only partially.</td>
</tr>
<tr>
<td>License</td>
<td>open source software</td>
</tr>
<tr>
<td>Maturity Level</td>
<td>Beta.</td>
</tr>
<tr>
<td>Contact</td>
<td>Timo Latvala</td>
</tr>
</tbody>
</table>
The Lime testbench is open source software and is available at [http://www.tcs.hut.fi/Software/lime/](http://www.tcs.hut.fi/Software/lime/). Lime testbench software is only known to work on 64-bit Ubuntu Linux. Lime testbench has been used in several of studies\(^1\) in order to enable incremental testing for distributed components already in early stages where the system has been implemented only partially.

### 7. Conclusions

This deliverable presented the state-of-the-art, state of practice and the baseline technologies available in MegaM@Rt project with respect to model-based runtime methods. Despite of the multiple proposals in the literature, and the different tools available, there are still a number of open issues and challenges that need to be addressed in the context of models used at runtime.

For instance, with respect to model and code generation, we have shown that different methods for code generation or model transformation have been proposed in the context aspect-orientation. Although work has been done in this direction, the integration between MDE and AOP/AOM is not realized completely and current methods and tools are not mature enough to be used outside academic environments.

Regarding model execution, the main challenges stand in the limited support the current modeling languages and their associated tool support, and to allow one to focus on different aspects of the system. Another limitation is that currently such models can only be used to implement at runtime a part of the complete system. The challenge, inside MegaMart, is to generalise this approach to address a wide spectrum of industrial systems and to cover more execution frameworks.

As industrial systems become more and more complex, increasingly large data volumes need to be analyzed at runtime. Current methods and tools lack capabilities to mine large volume logs and analyze (unstructured) data collected from systems. In addition, new methods for visualising these data need to be devised in order to reason about the properties and behavior of the system at runtime. The challenge in MagaM@Rt\(^2\) will be on how to adapt existing technology to serve the scalability and traceability requirements of industrial applications.

Last but not least, in the topic of models@runtime the main identified challenges have been the need for a reference architecture, the need to tackle uncertainty by runtime models, mechanisms for leveraging runtime models for self-adaptive software, and the use of models at runtime to address assurance for self-adaptive systems.

All these challenges will be used as innovation targets in the MegaM@Rt\(^2\) project and addressed the tasks of Work Package 2, as follows:

- **Automated code generation**: The SW synthesis process will include the required Run-Time verification artifacts. Code synthesis for AOM will be developed as well, enabling the synthesis of aspects for reference AOP weaving frameworks, such as AspectJ or Spring AOP.
- **Model execution**: We will develop code generators and model interpreters which will be based on state machines, activity or sequence charts. The code generation and model execution tools will be further extended for monitoring and tracing for analysis, online verification and testing.

\(^1\) [http://research.ics.aalto.fi/publications/lime2.shtml](http://research.ics.aalto.fi/publications/lime2.shtml)
• **Aspect-Oriented Modelling for Models at Runtime**: In MegaM@Rt, by taking advantage of the benefits of aspect-orientation, we will develop methods that will facilitate developing, validating, and maintaining large-scale models as independent components stemming from concerns. Verification of models will also be pursued aspect-wise, and then at integrated (woven) level. Test generation criteria and monitoring techniques will be developed to exploit the expected nature of the models, which will allow not only focusing on the relevant features during testing/monitoring, but also will provide means for improved analysis of logs and localization of faults. In addition, we propose to develop models at design time support, using AOM techniques, to inject or interweave runtime white-box monitoring capabilities, as a cross-cutting concern. Interwoven monitoring code will be generated using the models@design-time code synthesis support. Design model patterns for system monitoring will be implemented as AOM basic modelling building elements.

• **Runtime verification and validation**: In MegaM@Rt, models will be used and exploited during runtime, and combined with advanced tracing and monitoring techniques that will allow analysing information from different sources (e.g., network, applications, system logs) and using different techniques (e.g., statistics, machine learning), for obtaining a more holistic understanding of the system under observation, for verifying the models and assessing the safety, security, performance and quality of the final system.

• **Model-based testing**: The main progress that MegaM@Rt will provide are the following. First, a standard environment for model-based testing will be developed with a specific focus on using the test models for online testing at runtime. Second, by linking test generation with assertion verification, the test generation process will be accelerated while its quality improved. Model-based test generation will cover not only functional properties but extra-functional constraints in such a way that it will be possible to detect if a particular constraint such a delay, a throughput or a power constraint is not met. Some of these properties will be integrated automatically in the synthesized code and verified at runtime.

**References**


and Its Applications, 7(2), 207–218.
https://doi.org/10.1007/s10270-012-0293-5


Deliverable D3.1 Foundations for Model-based Runtime Analysis Methods

(UMR’06) (p. 11 pp.–70). https://doi.org/10.1109/CSMR.2006.43


Automation Conference (pp. 911–914). New York, NY, USA: ACM. https://doi.org/10.1145/1146909.1147140


https://www.xenonstack.com/blog/understanding-log-analytics-log-mining-anomaly-detection


Deliverable 3.8, Juniper - Project Number 318763). Brno University of Technology.


Some challenges and solutions. In 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC) (pp. 1–6). https://doi.org/10.1145/2744769.2747935


http://www.es.mdh.se/publications/4758-


workshop at the 24th Int. Conf. on Software Engineering (ICSE 2002).


