eDIANA
Embedded Systems for Energy Efficient Buildings

Grant agreement no.: *100012*

D6.1-A Modelling Guidelines for Building Analyzable/Testable Models

Author(s):

Leire Etxeberría	**MU**
Goiuria Sagardui	**MU**
Adrián Noguero	**ESI**
Huáscar Espinoza	**ESI**

Issue Date	30 November 2009 (m10)
Deliverable Number	D6.1-A
WP Number	WP6
Status	Delivered

Dissemination level	
X	**PU** = Public
	PP = Restricted to other programme participants (including the JU)
	RE = Restricted to a group specified by the consortium (including the JU)
	CO = Confidential, only for members of the consortium (including the JU)

Document history			
V	***Date***	***Author***	***Description***
0.1	*26-10-2009*	*MU*	*ToC*
0.2	*25-11-2009*	*ESI*	*ToC modified and detailed*
0.3	*18-01-2010*	*MU*	*Variability management for V&V section*
0.4	*18-01-2010*	*ESI*	*3.1, 3.2 and 3.3 sections*
0.5	*19-01-2010*	*MU*	*First version of section 2*
0.6	*20-01-2010*	*MU*	*Summary, introduction and conclusions added*
0.8	*21-01-2010*	*ESI*	*Additions in sections 1.1 and 1.2*
0.9	*28-01-2010*	*ATOS*	*Comments and corrections*
1.0	*01-02-2010*	*MU*	*Last version*

Disclaimer

Summary

The D6.1A Modelling Guidelines for Building Analyzable/Testable Models is a public document delivered in the context of WP6, task 6.1 with regard to providing guidelines for the description of the architecture of eDiana systems of systems via analyzable/testable models that can be used for early validation of quality attributes such as performance.

These guidelines for Building Analyzable/Testable Models can be used for performing early analysis of non-functional requirements. An overview of different languages for modelling is presented in section 2: MARTE, SysML, AADL, EAST-ADL2 and AOM. Existing tools for analysis of performance and timing (schedulability) are explored in section 3. And finally a proposal for modelling and managing the variability in analyzable models is specified in section 4.

Contents

Abbreviations

AADL	Architecture Analysis & Design Language
AOM	Aspect-Oriented Modelling
AOSD	Aspect-Oriented Software Development
DRM	Detailed Resource Modelling
DSL	Domain Specific Language
eDIANA	Embedded Systems for Energy Efficient Buildings
FODA	Feature-oriented domain analysis
GCM	Generic Component Model
GQAM	Generic Quantitative Analysis Modelling
GRM	General Resource Modelling
HLAM	High-Level Application Modelling
HRM	Hardware Resource Modelling
LQN	Layered Queuing Network
LQNS	Layered Queuing Network solver
MARTE	Modelling and Analysis of Real-time and Embedded systems
MDD	Model Driven Development
MDE	Model Drivel Engineering
OMG	Object Management Group
PAM	Performance Analysis Modelling
PEPA	Performance Evaluation Process Algebra
RTEA	Real Time & Embedded Analysis

RTES	Real Time and Embedded Systems
SPT	Schedulability, Performance and Time
SAE	Society of Automotive Engineers
SAM	Schedulability Analysis Modelling
SRM	Software Resource Modelling
SUT	System Under Test

Table of Figures

1. Introduction

Embedded systems development is getting more and more complicated. Software shows the highest growth rate within embedded systems. The estimated average annual growth rates between 2004 and 2009 are 16% for embedded software [1].

The eDiana system of systems is an example of a system of embedded systems. Software validation from early development stages is crucial in this kind of systems. Nowadays, there are tools that can help us do software validation from software models, even before writing a single line of code. MDD is a software development paradigm based on models that can facilitate embedded software development. MDD methodology abstracts from system complexity creating easy to understand models. These models can be validated at early stages of the development without having the need to implement the final product. Model analysis can detect problems early in the development life cycle and reduce cost and risk, besides improving quality and shorten time-to-market.

"Validation is the process of evaluating a system or a component during or at the end of the development process to determine whether a system component satisfies specified requirements" [2]. The Model Drivel Engineering paradigm facilitates early validation through Model-Based Analysis and Model-Based Testing.

Model based analysis is based on annotating or adding information specific to the property to be evaluated, and then transforming the annotated model into a formal model which can be analyzed with known analysis techniques and tools. In order to validate quality aspects, systems have to be modelled and annotated in a particular way. Different mechanisms have been defined for this purpose; 1) UML and profiles; and 2) DSL's. UML is a generic modelling language while DSL is a domain specific language. UML profiles are the mechanism provided by UML to extend its syntax and semantics to express specific concepts of particular application domains. The profiles are based on three elements (stereotypes, tagged values and constraints) that UML includes to manage this extension. Different profiles have been defined for different domains and specific problems and standardized by OMG. MARTE (UML Profile for Modelling and Analysis of Real-Time and Embedded systems) [3] or its predecessor profile SPT (UML Performance Profile for Schedulability, Performance and Time) [4], both standardized by OMG define quantitative performance annotations (such as resource demands made by different software execution steps, performance requirements, etc.) to be included in a UML model (architecture, behaviour and deployment views).

Model-based testing is a variant of testing that relies on explicit behaviour models that encode the intended behaviour of a system. The model-based testing allows the automatic generation of test cases from those behavioural models of the SUT (System Under Test).

This document focuses on model-based analysis for providing guidelines for building analyzable models. For this purpose, different languages for modelling have been explored: MARTE, SysML, AADL, EAST-ADL2 and AOM. Domain exploration of performance and timing (schedulability) evaluation, focusing on existing tools, has been made. An approach for modelling and managing the variability in analyzable models is also proposed.

1.1 Requirements to be fulfilled by this methodology

This section will cover the requirements for V&V in eDIANA to be covered by the model-based methodology that will be provided in WP6.

The methodology will particularly focus on four kinds of quality requirements which are key to guarantee the trustworthiness and evolvability of the embedded systems involved in eDIANA:

- Conformance. Conformance refers to the conformity of the functional model with the functional requirements of the system and composition of systems involved in eDIANA, and all operational specifications including aspects such as cost, robustness, maintainability, privacy and security, etc.

- Performance. Performance is related to soft real-time characteristics of embedded systems involved in energy management operations and user related tasks. Evaluating the performance of a system is a difficult task due to the great variety of existing variables used to do so. Therefore, the performance of a system can be measured in terms of the time it takes to perform an action in the worst case, the mean throughput it provides, the quantity of resources it consumes... In the context of eDIANA we will focus on the timing aspects of performance, that is:

    - Throughput. The number of data units processed within a unit of time.

    - Response Time. The average time a system or component requires to provide a response.

  In order to be able to analyze the system in these terms it is necessary to include a set of performance specific annotations into the design models. These annotations will be extracted from the analysis of specific performance analysis methods and tools in section 3.

- Timeliness. Timeliness is related to hard real-time constraints applicable to some parts of the eDIANA architecture, such as controllers or power grid interactions. Hard real-time constraints require the system to respond and

execute its functionalities within a concrete period of time. Within the context of eDIANA we will focus on schedulability analysis. Schedulability analysis methods aim at verifying that the deadlines of all the tasks of the system (i.e. the time within which a task must provide its results) can be met. To perform schedulability analysis on a system, the designers must enrich the models with specific timing annotations regarding timing aspects, such as:

  - Periods, offsets and worst case execution times. These parameters characterize the system tasks, their activation times, etc.
  - Deadline. For each task, periodic or aperiodic, we must provide the maximum time frame within which it should provide its results.
  - End-to-end flows. If a task is composed of several subtasks and global deadlines have to be met, the designer should provide information about the characteristics of these flows.

  To extract the required annotations to perform schedulability analysis we will study a couple of schedulability analysis tools in section 3.

- Variability. Variability is understood as both functional (variation of functionalities) and quality variability (variability on quality requirements: different priority levels of performance and timeliness requirements depending on the product). This is needed from the models to assure that further changes in the requirements, which often occur, can be inserted with reasonable ease, without breaking entirely the original model.

Another requirement is the usage of UML MARTE as a basis for the eDIANA system of systems description. In WP2 MARTE profile was selected as the modelling language. It can be necessary its extension or enhancing with other well-know languages or formalisms to support more flexible modelling mechanisms.

In order to obtain information related to V&V (validation and verification) from eDIANA industrial partners, an elicitation questionnaire have been fulfilled by several partners (Fagor, Ikerlan and ZIV). The conclusions and requirements obtained from those questionnaires are:

- A methodology for variability should be provided. Variability that may be present in eDiana systems of systems has been identified:
  - Variations derived from country specific requirements or communications interfaces.
  - Variability in configurations: number of cells, number of devices, type of devices, topologies...

- Requirements for performance and timing analysis
    - A methodology for performance and timing analysis should be provided.
    - The methodology should be supported by a tool that allows designers to easily test their systems.
    - Tools should provide user friendly interfaces, hiding its complexity when it is not needed, yet providing a flexible and extendible V&V framework.

1.2 Definition of the eDIANA application concepts

In this section we should achieve an abstract application model for eDIANA that contains all the concepts used in eDIANA applications (e.g. component, ports, tasks, scheduler, etc.). These concepts will guide the methodology provided in this WP. These concepts will be broad enough to cope with different scenarios, assuring that the work done specifically by the partners of eDiana is extensible to a major part of embedded systems design.

As stated in D2.1A deliverable the eDIANA platform principles state that any eDIANA deployment must be strictly component oriented. Following this criterion, the eDIANA MDE methodology proposes a component oriented framework for designing eDIANA devices and applications. In order to create the component-based design framework it is a requirement to fully characterize the whole range of eDIANA platform and application components, including all the different scenarios and hierarchical levels (i.e. Cell and MacroCell). The component repositories described in the process model will store the eDIANA components collection. Yet, to use these components in an MDE environment, it is also necessary to create eDIANA compliant models that can be reused during model-driven design stages.

Depending on their nature, application or platform components, the modelling constructs required for them vary. Application components are basically composed of:

Job. The job is the concrete functionality provided by the component. A job may be further split into tasks; however, from the component designer's point of view, the job of a component implies certain behaviour and a set of non-functional properties.

LIF (Linking InterFace). The LIF specifies the messages and signals consumed and provided by the job. The LIF of a job (i.e. a component) is conformed by all the individual interfaces it provides and consumes.

On the other hand, platform components provide the physical entity that hosts/contains the logical application component. It is also possible that some platform components (e.g. software libraries, middlewares...) provide services used by the applications (i.e. APIs).

The eDIANA Platform identifies a number of components that can be extended in the future. The analysis requirements of these components are also different depending on their category. From this point of view we can distinguish:

- At the Cell level:
    - Cell level Monitoring and Metering. In this group any application component intended for measuring physical values is included; namely sensors, smart meters, etc. Components in this group are susceptible to be analyzed as part of end-to-end flows in timing analysis.
    - Cell level Control and Actuation. This group includes application components capable of interacting with concrete physical appliances, such as lamps, blinds, washing machines, air conditioning systems, etc. Similarly to sensors, these components can also be analyzed as part of end-to-end flows. Also, actuators can be analyzed in terms of throughput, regarding the maximum number of commands an actuator can handle.
    - Cell level User Interface Channels. This group of application components includes the application components related to interfacing the users with the Cell Device Concentrator. These components have performance requirements regarding maximum response times, since they interact directly with the final users.
    - Cell level Generation & Storage. This group of application components is focused to the power generation and storage functions of the eDIANA devices. Since these components are likely to interact with other devices and the energy providers, it is possible that these elements have to meet strict hard real-time constraints.
    - Cell level Concentrator (Policy manager). This group includes the application components that provide the CDCs of their functionality; namely control algorithms, sensor data gathering, etc. Being such a complex component, CDC components will require both performance and timing analysis.

- At the MacroCell level:
    - MacroCell level Concentrator. This group of application components provides the MCC of its basic functionality. Similarly to the CDC components, MCC components are likely to be constrained with both

performance and timing constraints; therefore both analysis will be required. Some of the MCC application components have been separated from this group due to their importance. They are integrated in the following two groups.

- Data gathering component. This application component is devoted to data gathering and management at the MCC level.
- Control strategy manager. This application component manages the global control strategy of an eDIANA platform, disregard of the scenario in which it is deployed.

Provided that the eDIANA components will be defined following a model-based approach, as defined in deliverable D2.1-A, it is necessary to develop a methodology for model-based analysis of the eDIANA designs at an early stage. The objective of this document is, therefore, to analyze the modelling guidelines and annotations required to obtain analyzable models from architectural ones. This deliverable is an initial description of the analysis domain of the eDIANA platform. This information will be used to develop the V&V model derivation techniques that will be described in D6.1-B.

2. Languages for analysis oriented modelling

Here we will cover different annotation and modelling languages capable of modelling different non-functional aspects of the systems that can be used afterwards for system analysis. The languages proposed are the following ones:

2.1 UML-MARTE

MARTE (Modelling and Analysis of Real-time and Embedded systems) [1] is a UML profile that adds capabilities to UML for model-driven development of Real Time and Embedded Systems (RTES). This profile provides support for specification, design, and verification/validation stages. This new profile is intended to replace the existing UML Profile for Schedulability, Performance and Time (STP).

The profile is structured around two main concerns, one to model the features of real-time and embedded systems and the other to annotate application models so as to support analysis of system properties. These are shown by the "MARTE design model" and "RTEA (Real Time & Embedded Analysing)" packages in Figure 1. These two packages uses "MARTE foundations" that is about common concerns with describing time and the use of concurrent resources.

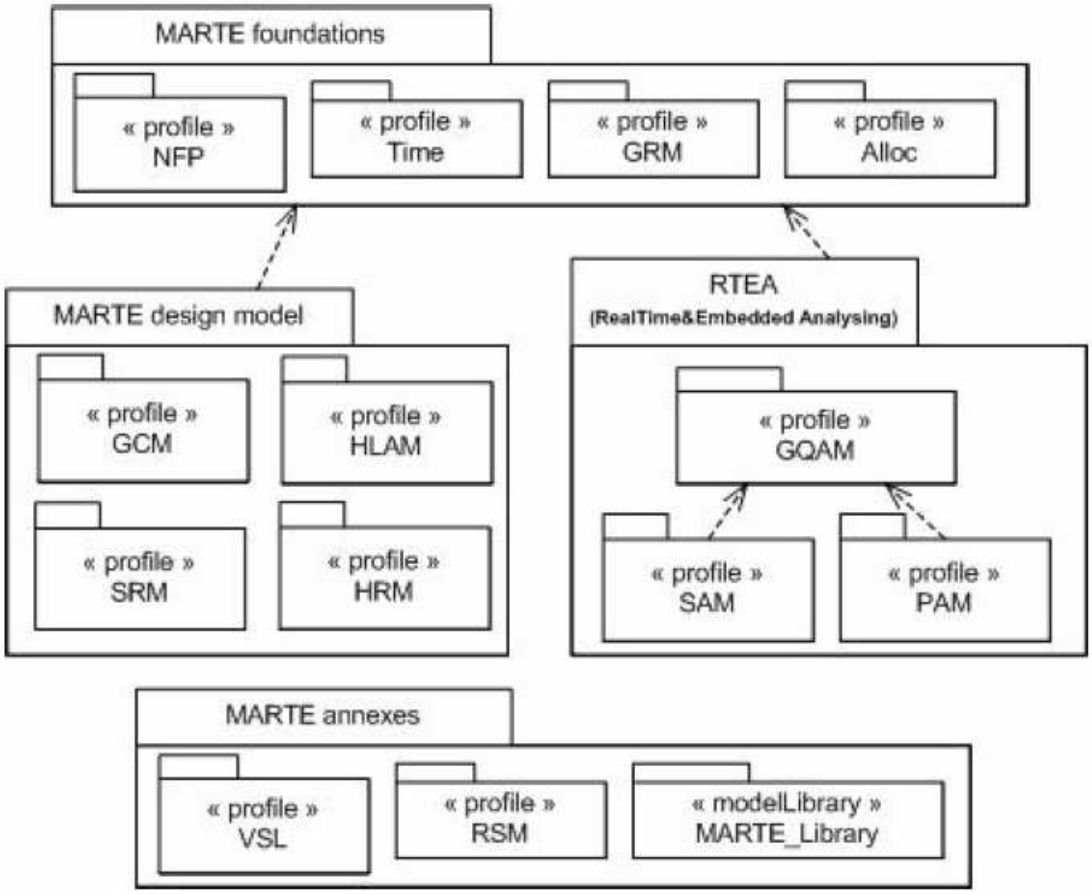

Figure 1: Packages of MARTE profile

MARTE foundations

The four sub-profiles that provide common concerns in MARTE are:

- Non-functional Properties Modelling (NFPs): This sub package of the MARTE specification provides a general framework for annotating models with quantitative and qualitative non-functional information.
- Time Modelling (Time): defines the time as used within MARTE
- Generic Resource Modelling (GRM): The objective of this package is to offer the concepts that are necessary to model a general platform for executing real-time embedded applications.
- Allocation Modelling (Alloc): defines concepts required to describe allocation concerns.

MARTE Design Model

To model the features of real-time and embedded systems, four sub-profiles are provided in MARTE profile:

- Generic Component Model (GCM): The MARTE General Component Model presents additional concepts (w.r.t usual component paradigms) that have been identified as necessary to address the modelling of artefacts in the context of real-time and embedded systems component based approaches.
- High-Level Application Modelling (HLAM): The concern of the HLAM package is to provide high-level modeling concepts to deal with real-time and embedded features modeling.
- Detailed Resource Modelling (DRM): The concern of the DRM package is to provide specific modeling artefacts to be able to describe both software and hardware execution supports. It specializes generic concepts offered by General Resource Modelling (GRM).

    - Software Resource Modelling (SRM): which intends to describe application programming interfaces of software multi-tasking execution supports.
    - Hardware Resource Modelling (HRM): which intends to describe hardware execution supports, through different views and detail levels.

MARTE Analysis Model or RTEA (Real Time & Embedded Analysing)

MARTE Analysis Model provides facilities to annotate models with information required to perform specific analysis. Especially, MARTE focuses on performance and schedulability analysis. But, it defines also a general analysis framework which intends to refine/specialize any other kind of analysis.

MARTE Analysis Model contains the following sub-profiles:

- Generic Quantitative Analysis Modelling (GQAM)
- Schedulability Analysis Modelling (SAM)
- Performance Analysis Modelling (PAM)

The generic analysis domain includes specialized domains in which the analysis is based on the software behaviour, such as performance (PAM) and schedulability (SAM), and also power, memory, reliability, availability and security. Although analysis domains have different terminology, concepts, and semantics, they also share some foundation concepts, which are expressed in this sub-profile, in order to simplify the profile and make it easier to add new analyses. Generic modelling defines basic modelling concepts and Non-Functional Properties (NFP), using the NFP annotation framework.

MARTE analysis is intended to support accurate and trustworthy evaluations using formal quantitative analyses based on sound mathematical models, which may supplement designer intuition and "feel". Model analysis can detect problems early in the development life cycle and reduce cost and risk.

GQAM is used for creating sub-profiles for:

- Schedulability analysis, to predict whether a set of software tasks meets its timing constraints and to verify its temporal correctness, e.g. RMA-based techniques (SAM).

- Performance analysis, to determine if a system with non-deterministic behaviour can provide adequate performance, usually defined by some statistical measures (PAM).

Extra annotations needed for analysis are to be attached to an actual design model, rather than requiring a special version of the design model to be created only for the analysis.

2.2 SysML

SysML (Systems Modelling Language) [5][6] is a general-purpose graphical modelling language for systems engineering applications. SysML is designed to provide simple but powerful constructs for modelling a wide range of systems engineering problems. It is particularly effective in specifying requirements, structure, behaviour, and allocations and constraints on system properties to support engineering analysis. The language is intended to support multiple processes and methods such as structured, object-oriented, and others, but each methodology may impose additional constraints on how a construct or diagram kind may be used.

SysML represents a subset of UML2 with extensions needed to satisfy the requirements for modelling Systems (see Figure 2).

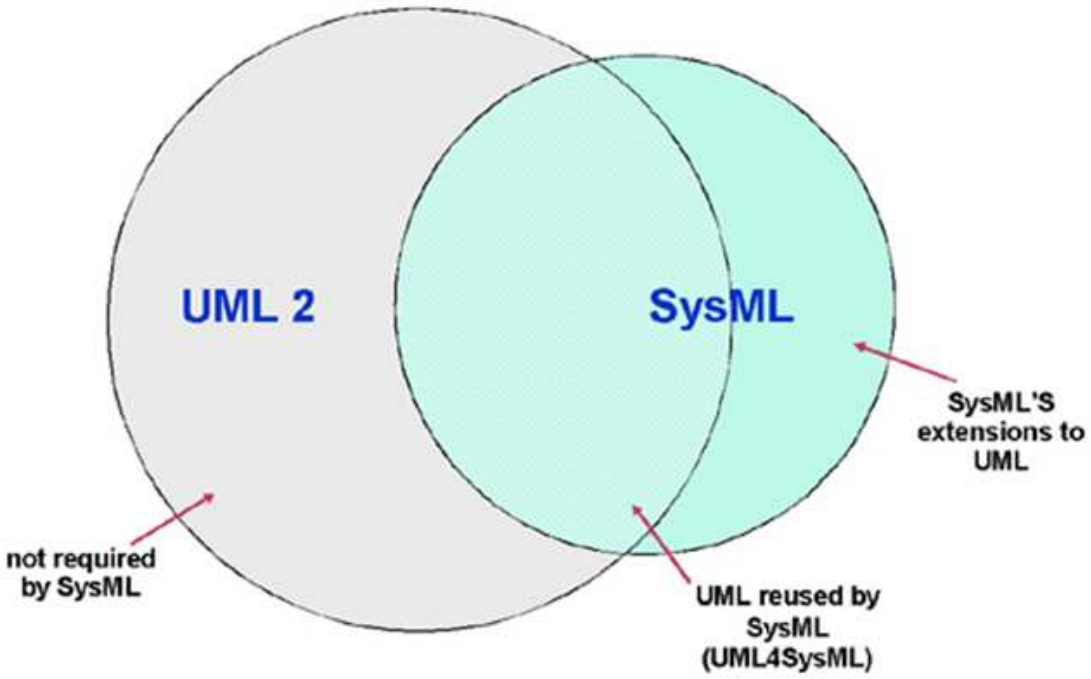

Figure 2: Relationship between SysML and UML

The SysML diagram types are identified in Figure 3.

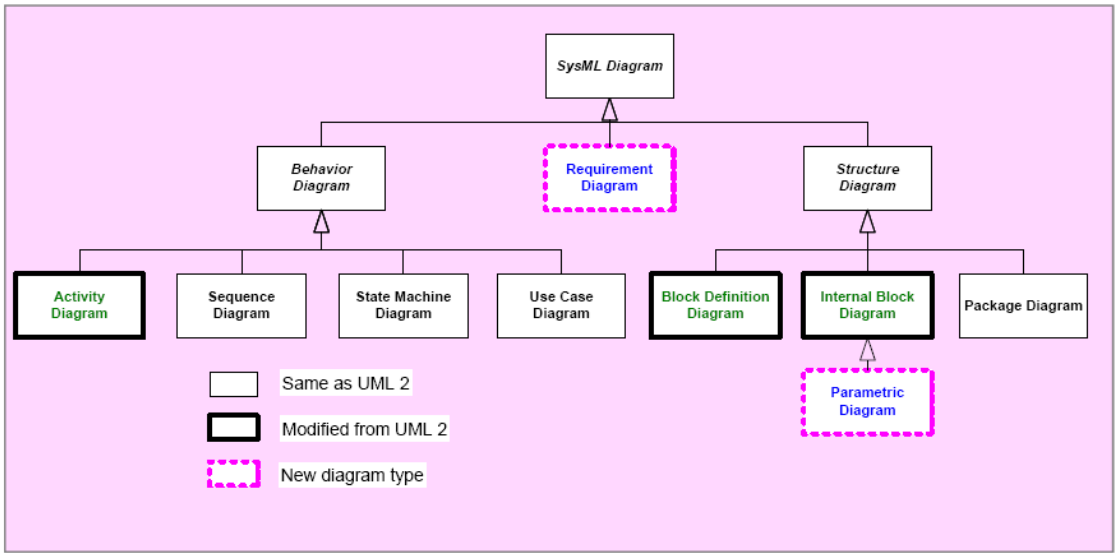

Figure 3: SysML Diagram Types

The «block» is the basic unit of structure in SysML and can be used to represent hardware, software, facilities, personnel, or any other system element. The system

structure is represented by block definition diagrams and internal block diagrams. A block definition diagram describes the system hierarchy and system/component classifications. The internal block diagram describes the internal structure of a system in terms of its parts, ports, and connectors. The package diagram is used to organize the model.

The behaviour diagrams include the use case diagram, activity diagram, sequence diagram, and state machine diagram. A use-case diagram provides a high-level description of functionality that is achieved through interaction among systems or system parts. The activity diagram represents the flow of data and control between activities. A sequence diagram represents the interaction between collaborating parts of a system. The state machine diagram describes the state transitions and actions that a system or its parts perform in response to events.

SysML includes a graphical construct to represent text based requirements and relate them to other model elements. The requirements diagram captures requirements hierarchies and requirements derivation, and the satisfy and verify relationships allow a modeller to relate a requirement to a model element that satisfies or verifies the requirements. The requirement diagram provides a bridge between the typical requirements management tools and the system models.

The parametric diagram represents constraints on system property values such as performance, reliability, and mass properties, and serves as a means to integrate the specification and design models with engineering analysis models.

SysML also includes an allocation relationship to represent various types of allocation, including allocation of functions to components, logical to physical components, and software to hardware.

2.3 AADL

Architecture Analysis & Design Language (AADL) [7] is a language developed by the Society of Automotive Engineers (SAE), which is designed for the specification, analysis, and automated integration of real-time performance-critical (timing, safety, schedulability, fault tolerant, security, etc.) distributed computer systems. It provides a new vehicle to allow analysis of system designs (and system of systems) prior to development and supports a model-based, model-driven development approach throughout the system life cycle.

AADL, like its predecessor MetaH, produces language based modeling artefacts. AADL was developed as a programming language not only to define the textual representation of software architecture but also (and more importantly) to formally

define the syntax and semantics. Moreover, AADL permits textual and graphical system representation.

The key specification elements of AADL are summarized in Figure 4 [8]. In AADL, components are defined through type and implementation declarations. A Component Type declaration defines a component's interface elements and externally observable attributes (i.e., features that are interaction points with other components, flow specifications, and internal property values). A Component Implementation declaration defines a component's internal structure in terms of subcomponents, subcomponent connections, subprogram call sequences, modes, flow implementations, and properties. Components are grouped into application software, execution platform, and composite categories. Packages enable the organization of AADL elements into named groups. Property Sets and Annex Libraries enable a designer to extend the language and customize an AADL specification to meet project or domain specific requirements.

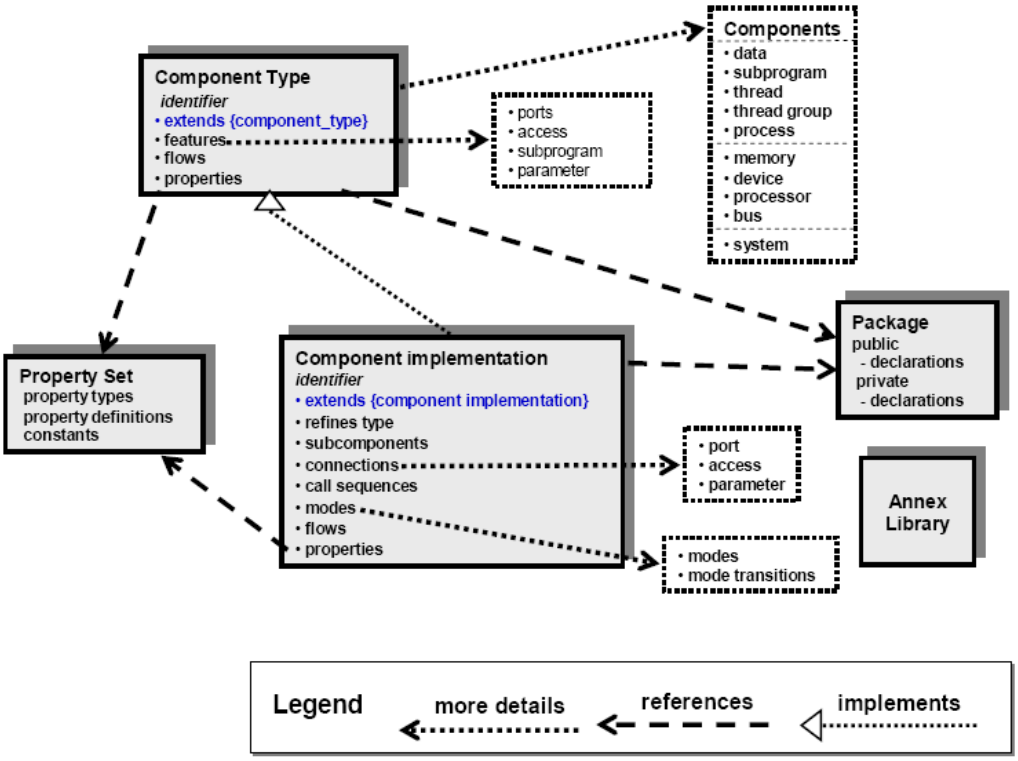

Figure 4: AADL elements [8]

AADL supports the early prediction and analysis of critical system qualities—such as performance, schedulability, and reliability. For example, in specifying and analyzing

schedulability, AADL-supported thread components include the predeclared execution property options of periodic, aperiodic (event-driven), background (dispatched once and executed to completion), and sporadic (paced by an upper rate bound) events. These thread characteristics are defined as part of the thread declaration and can be readily analyzed.

In [9] EAADL (Extended AADL) is presented, extended approach of AADL for embedded system product lines that allow annotating quality requirements.

2.4 EAST-ADL2

EAST-ADL2 [10][11] is Architecture Description Language for Handling all engineering information required to sustain the evolution of vehicle electronics. The language is compliant with the Automotive standard AUTOSAR [12]. The EAST-ADL2.0 is a revision of the initial EAST-ADL system modeling approach that was defined in the EAST-EEA project (http://www.east-eea.net/).

EAST-ADL2 is a System Modelling Approach that is a template for how engineering information is organized and represented, provides separation of concerns and embrace the de-facto representation of automotive software –AUTOSAR.

The purpose of the EAST ADL is to capture the software and electronics architecture with enough detail to allow modeling for documentation, design, analysis and synthesis. These activities require system descriptions on several abstraction levels, from top level user features down to tasks and communication frames in CPUs and communication links. Moreover, the activities also involve the expression of non-structural aspects of the system under development, e.g. requirements, behaviour and validation and verification. The EAST-ADL2 abstraction layers are shown in Figure 5.

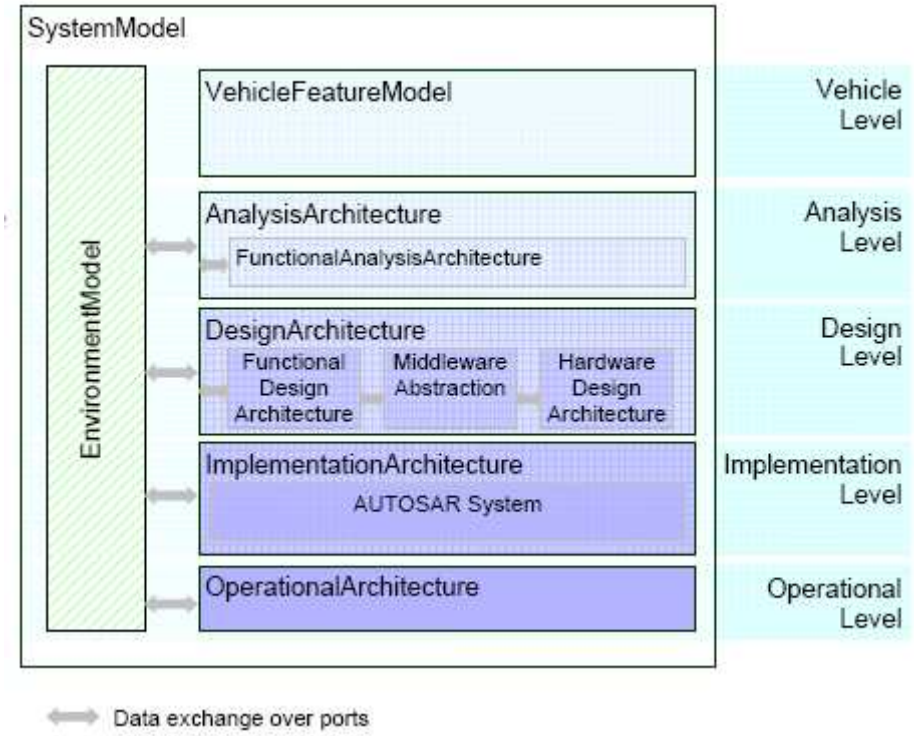

Figure 5: EAST-ADL Structure

The SystemModel is the top level container of an EAST-ADL2 model (see Figure 5). It represents the electronics and software of the vehicle, and its environment, and concepts related to the various abstraction level of models used in EAST-ADL2. It is mainly based on both concepts: Models and architectures.

- VehicleFeatureModel represents the features of the vehicle, i.e. the externally visible properties.
- The AnalysisArchitecture is the abstract functional description of the vehicle electronics.
- The DesignArchitecture contains the functional specification and hardware architecture of the vehicle electronics: Hardware entities/topology, Concrete Functional structure & behaviour and Function-to-ECU allocation
- The Implementation Architecture contains the software architecture and components and the hardware architecture of the vehicle: AUTOSAR constructs.
- The Operational Architecture represents the actual software and electronics in the manufactured vehicle.

Variability modelling is supported in EAST-ADL2, it provides variability mechanism to allowing the inclusion of different vehicles: Product Line Architecture.

2.5 Aspect-Oriented Modelling (AOM)

Aspect-Oriented Software Development is a development paradigm that enables the creation of a modular architecture for a system focusing on crosscutting concerns. Cross-cutting concerns are aspects of a program which affect (crosscut) other concerns. Cross-cutting concerns are software functionalities (e.g., security, distribution, synchronization) that can not easily be implemented with traditional development paradigms because they finish spread over the modules of the application.

Cross-cutting concerns or aspects are much related to quality attributes because the nature of several operational qualities (security, performance...) is cross-cutting.

Model-Driven Engineering (MDE) is a software development methodology which aims to raise the abstraction level of system specifications and increase automation in system development. It uses models at different levels of abstraction for raising the abstraction level. Automation is achieved by using model transformations: higher-level models are transformed into lower level models. One kind of model transformation is code generation.

Aspect-Oriented Modelling (AOM) combines ideas from AOSD and MDD. Modelling aspects (which can be quality attributes) facilitates the validation and verification process.

3. Domain exploration for analyzable models

In this section we will explore different non-functional analysis domains, namely, performance and timing (schedulability). Domains will be explored using existing analysis tools as starting point.

3.1 Overview of the eDIANA V&V context

The early validation phase of real-time embedded systems, such as the ones used in eDIANA, aims at reducing the number of design errors. Correcting these errors in later phases of the development process will typically involve greater effort than the effort required to correct them early. For example, real-time embedded systems typically need to fulfil the temporal constraints specified on requirement phase. When a system is able to execute all its tasks before their deadline expires, then that

system is said to be schedulable. A wrong design may cause some of these tasks to miss their deadlines; eventually leading the system to behaving incorrectly. Correcting these errors may require deep design changes that may cause many parts of the code to become useless. Similar examples can be found regarding performance or variability in product-lines.

The effort spent in the last decades in the development of analysis and validation techniques has resulted in many specific analysis tools that are frequently used by modern real-time application engineers. Despite the fact that many tools share many theories and techniques, it is common to find that different tools are not interoperable with each other.

Modelling languages, as the ones discussed previously in this document, provide eDIANA system designers means to capture implementation aspects of the software and hardware of these systems. Models capture the most significant structural, behavioural and non-functional information of the systems. This information enables not only source code generation through transformations, but also early testing and analysis.

It is common that different analysis tools often employ different input data sets making it difficult to integrate them in an analysis framework. However, the use of a more abstract language along with model transformations provides the framework the required *"glueware"* to achieve this integration.

The following sections will try to go over different timing, performance and variability analysis tools to explore the analysis domain in order to define the set of modelling notations required to integrate the three analysis domains.

3.2 Performance evaluation

In the past few years research in software engineering has witnessed significant interest in the performance evaluation of software models. This has become an even more compelling and exciting issue as development practices shift towards model-driven methodologies. Indeed, the use of model-driven approaches enables engineers to extract conclusions about the performance of the software even at early stages of the development.

Several formalisms and techniques have been developed in order to develop performance tests on software. Many of these approaches rely either in the queue networks theory or in Petri-nets in order to analysis the performance of the software systems. In the following sections we will discuss the concept sets employed by these methodologies and their respective tools to do performance analysis in order to extract a common concept set to be applied in eDIANA models. Namely the methodologies that will be studied are: PEPA and LQN.

3.2.1 Performance Evaluation Process Algebra, PEPA

The Performance Evaluation Process Algebra, PEPA for short, is a performance modelling language for systems developed by Jane Hillston. The theory behind PEPA can be found in her book [24].

In PEPA a system is described as an interaction of components and these components engage, either singly or multiply, in activities. The components will correspond to identifiable parts in the system, or roles in the behaviour of the system. They represent the active units within a system; the activities capture the actions of those units. For example, a queue may be considered to consist of an arrival component and a service component which interact to form the behaviour of the queue.

A component may be atomic or may itself be composed of components. Thus the queue in the above example may be considered to be a component, composed of the atomic arrival and service components. We assume that there is a countable set of possible components, C. Each component has a behaviour which is defined by the activities in which it can engage. Actions of the queue might be *accept*, when a customer enters the queue, *service*, or *loss*, when a customer is turned away from a full buffer.

When talking about PEPA we use the term activity to distinguish it from the usual process algebra notion of an instantaneous action. Every activity in PEPA has an associated duration which is a random variable with an exponential distribution. In this thesis the term action will relate to the behaviour of the system.

Each activity has an action type (or simply type). We assume that each discrete action within a system is uniquely typed and that there is a countable set, A, of all possible such types. Thus the action types of a PEPA term correspond to the actions of the system being modelled. If there are several activities within a PEPA model which have the same action type then they represent different instances of the same action by the system.

There are situations when a system is carrying out some action (or sequence of actions) the identity of which is unknown or unimportant. To capture these situations there is a distinguished action type, τ, which can be regarded as the unknown type. Activities of this type will be private to the component in which they occur. These activities are not instantaneous; each instance of an activity with action type τ will have an associated duration, as with any other type. However, unlike all other types, multiple instances of τ type activities within a PEPA model do not necessarily represent the same action by the system.

Since an exponential distribution is uniquely determined by its parameter, the duration of an activity, an exponentially distributed random variable, may be

represented by a single real number parameter. This parameter is called the activity rate (or simply rate) of the activity; it may be any positive real number, or the distinguished symbol T, which should be read as unspecified.

PEPA models can be transformed into analyzable Markov Chains. From these chains it is possible to extract, using several analysis techniques, such as steady-state analysis and time series analysis, the performance figures of a system.

Jane Hillston's team has implemented the PEPA language and some analysis tools on top of Eclipse [25]. Internally, the PEPA plugin for Eclipse has implemented the PEPA language using a specific metamodel. An excerpt of this metamodel can be seen in Figure 6, and Figure 7 presents the User Interface of the plugin.

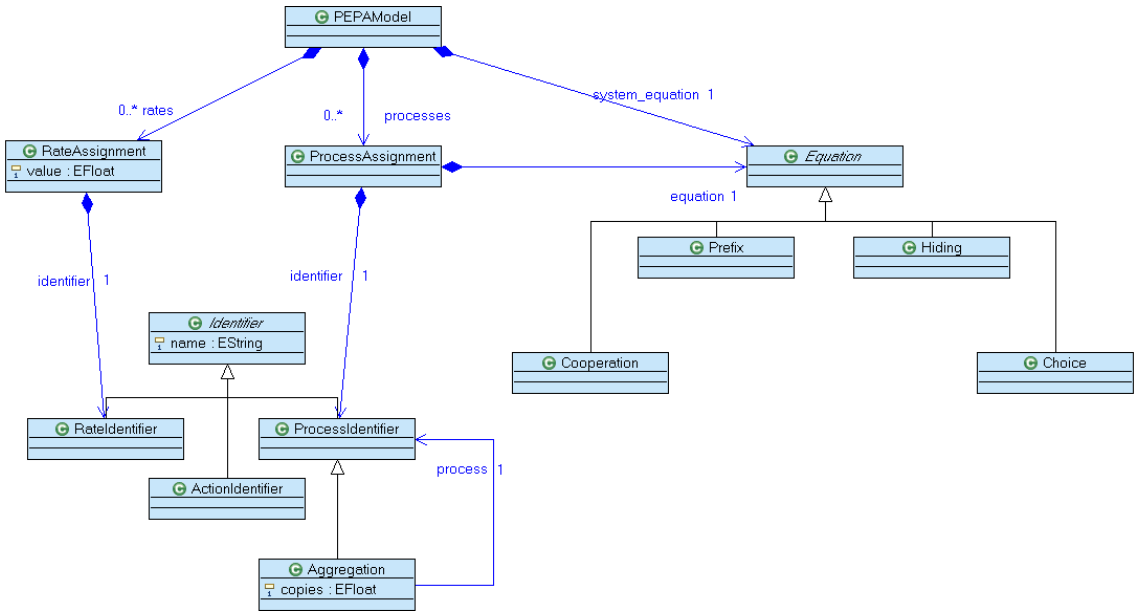

Figure 6. Excerpt of the PEPA metamodel

This tool is freely available and includes the basic toolkit to develop and analyze performance models using PEPA, namely a specific PEPA perspective, an editor, an engine for plotting the results of the analyses, and a set of built-in performance analyses applicable to Markov chains, which are derived from the PEPA models. Additionally, the PEPA plugin includes some extra functionalities around the PEPA models, such an EMF exporter, a UML importer, Matlab files export, etc.

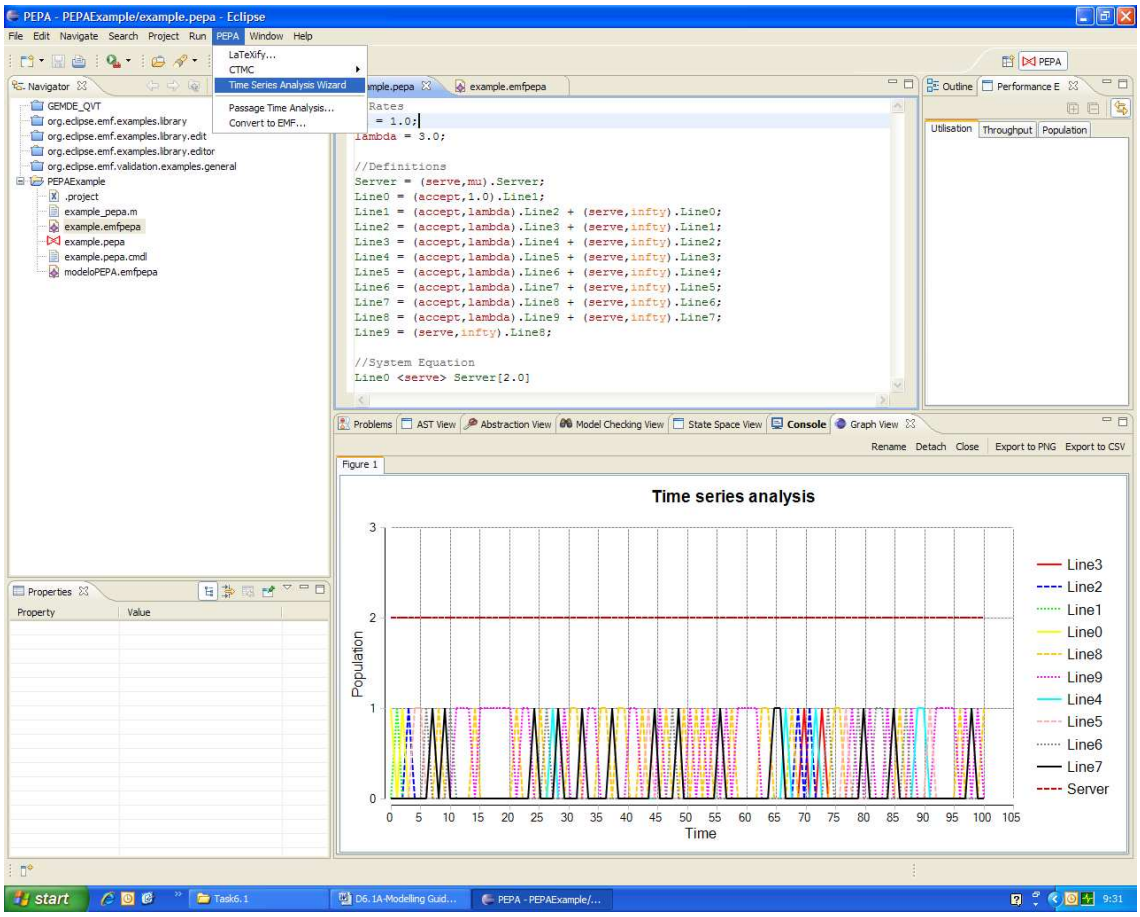

Figure 7. User interface of the Eclipse plugin for PEPA

3.2.2 Layered Queuing Network, LQN

The Layered Queuing Network theory aims at modelling software systems in such a way that they can be analysed in terms of performance. The LQNS (LQN Solver) tool, developed by Carlenton University, implements many analyses that are applicable to systems modelled using the LQN notation. The LNQS is a command-line tool implemented for Linux, HP-UX and Windows (using a GNU toolkit). The input to this tool is provided using plain text files; however, LQNS also accepts XML models using a specific schema (i.e. a LQN metamodel). The LQN toolkit developed by Carlenton University also includes a LQN model simulator caller LQNSIM.

The LQN theory considers layered systems (software systems, and other kinds of systems too) that are made up of servers (and other resources which we will model as servers); the generic term used for these entities is "task". A server can be either a pure server, which executes operations on command (for instance a processor), or a more complex logical bundle of operations, which includes the use of lower layer services. Such a bundle of operations may be implemented as a software server.

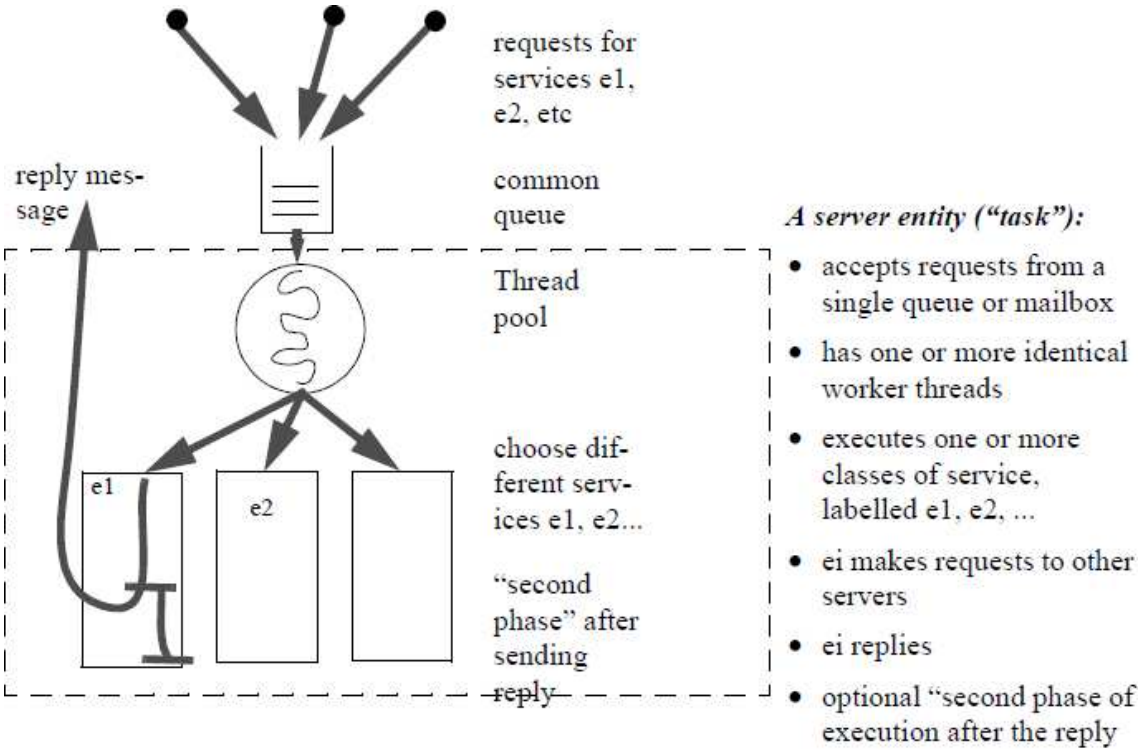

Figure 8. Elements of a software server in a LQN model

Figure 8 illustrates the elements of a software server, as they might be implemented in a software process. Each server has a single queue for all incoming requests as shown in the figure. Threads are servers to the queue, and requests take the form of interprocess messages (remote procedure calls, or the semantic equivalent), and entries describe or define the types of service the server provides. The assumption in this theory is that each thread has the capability of executing any entry. The execution of the entry can follow any sequence of operations, and can include any number of nested requests or calls to other servers. The latter figure includes an optional second execution phase for the entries, the concept behind this second execution phase is that software servers often send the reply as early as possible, and complete some operations afterwards (e.g. database commits).

The execution of the server entity is assumed to be carried out by a single processor or a multiprocessor, called its "host" (not shown in the Figure). Once the request is accepted, the execution of the entry is a sequence of requests for service to the host and to other servers, and the essence of layered modelling is to capture this nesting of requests. Each request requires queuing at the other server, and then execution of a service there.

The following figure shows a small excerpt of the LQN metamodel. This metamodel can be checked in [23], for further details.

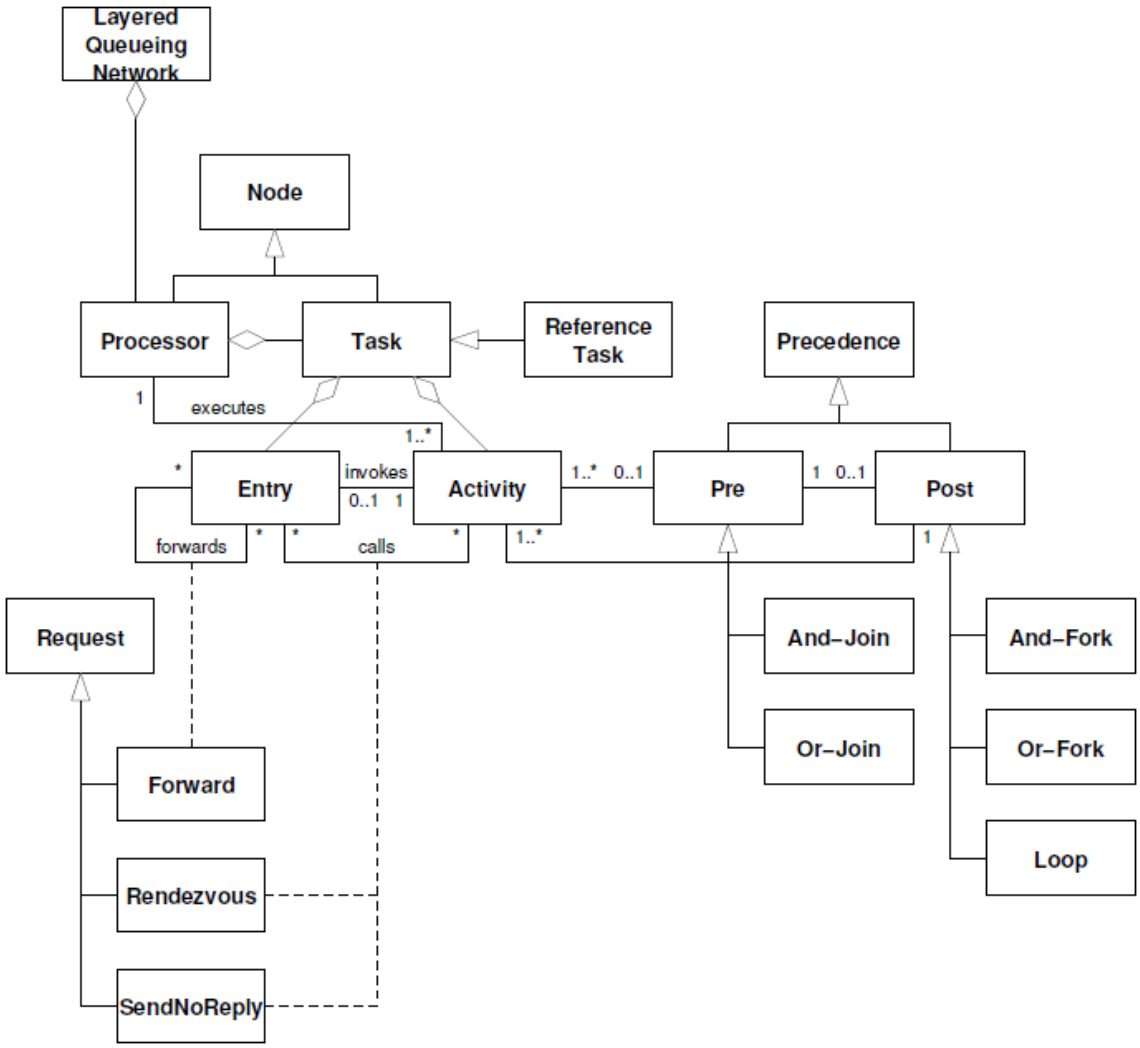

Figure 9. Excerpt of the LQN metamodel (obtained from [23])

The concepts in Figure 9 are the ones used by the LQN toolkit to represent the systems and evaluate their performance figures. Their definitions are as follows:

Processor. Processors are used by the activities within a performance model to consume time. They are pure servers in that they only accept requests from other servers and clients. They can be actual processors in the system, or may simply be place holders for tasks representing customers and other logical resources. Each processor has a single queue for requests. Requests may be scheduled using one the following queuing policies: FIFO, preemptive and non-preemptive priority based scheduling, round-robin processor sharing scheduling and random scheduling.

Task. Tasks are used in layered queuing networks to represent resources. Resources include, but are not limited to: threads (or processes) in a computer system, clients, buffers, and hardware devices. In essence, whenever some entity requires some sort

of service, requests between tasks involved. A task has a queue for requests and runs on a processor. Requests can be served using the following scheduling methods: FIFO and priority based scheduling (preemptive and non-preemptive). Different classes of service are specified using entries. Tasks may also have internal concurrency, specified using activities. These two elements will be further explained later.

One subclass of task exists: ***Reference Task***. Reference tasks are used to represent clients in the layered queuing network. They are like normal tasks in that they have entries and can make requests. However, they can never receive requests and are always found at the top of a call graph.

Entry. Entries service requests and are used to differentiate the services provided by a task. An entry can accept either synchronous or asynchronous requests, but not both. Synchronous requests are part of the closed queuing model whereas asynchronous requests are part of the open model. Entries also generate the replies for synchronous requests. Typically, a reply to a message is returned to the client who originally sent the message. However, entries may also forward a request to another entry making a chain. The next entry which accepts the forwarded request may forward the message again, or may reply back directly to the originating client.

Activity. Activities are the lowest-level of specification in the performance model. They are connected together using *Precedence* elements to form a directed graph to represent more than just sequential execution scenarios.

Activities consume time on processors. The service time is defined by a mean and variance. The service time between requests to lower level servers is assumed to be exponentially distributed, so the total service time is the sum of a random number of exponentially distributed random variables.

Activities also make requests to entries on other tasks. The distribution of requests to lower level servers is set by the call order for the activity which is either stochastic or deterministic. If the call order is deterministic, the activity makes the exact number of requests specified to the lower level servers. The number of requests is integral; the order of requests to different entries is not defined. If the call order is stochastic, the activity makes a random number of requests to the lower level servers. The mean number of requests is specified by the request rate. Requests are assumed to be geometrically distributed.

For entries which accept rendezvous requests, replies must be generated. If the entry is specified using phases, the reply is implicit after phase one. However, if the entry is specified using activities, one or more of the activities must explicitly generate the reply. Exactly one reply must be generated for each request.

Precedence. Precedence elements are used to connect activities within a task to from an activity graph. Referring to Figure 9, precedence is subclassed into ***Pre*** and ***Post*** each of them referring to connections happening before and after an activity. Thus, to connect one activity to another, the source activity connects to a pre-precedence. The pre-precedence then connects to a post-precedence which, in turn, connects to the destination activity.

Request. As we already announced, service requests from one task to another can be one of three types: ***Rendezvous*** (i.e. synchronous), ***Forward***, or ***SendNoReply*** (i.e. asynchronous).

3.3 Timing evaluation

As we already discussed, early detection and correction of system vulnerabilities and errors is becoming dramatically important for software developers; especially for those developers working with safety critical software. An early detection of software defects may reduce the costs derived from the correction of the error. Among the many vulnerabilities that can be detected in an early development stage, schedulability is one of the most recursive topics in the researchers' community.

Several methods and tests have been developed to analyze the schedulability in real-time systems [18] [19] [20] [21]. These tests often require different information of the analyzed system as input. Models provide a good means of capturing this information in a structured way. In this section we will try to extract the main concepts requires to do timing analysis on models. This information will be extracted by analysing the following schedulability analysis tool suites: Cheddar [14], MAST [15], TIMES [17], RT-Druid and RapidRMA. Each of the latter tool suites employs a different set of concepts to create the input models for their simulation and analysis tools. In the following lines we provide a short overview of the tool suites as well as a brief description concepts involved in their metamodels.

3.3.1 Cheddar

Cheddar is an open source schedulability analysis and simulation toolkit. It was first conceived to be an AADL models analyzer. It has been developed on top of OCARINA [1], a tool suite for manipulating AADL models.

Cheddar provides a graphical user interface (see Figure 10) that allows users to model the application they want to analyze and a simulator which computes simulated schedules and feasibility tests. Although Cheddar supports a great number of scheduling policies and schedulability tests, there are cases where existing schedulers do not match the particularities of a given system. For those cases, Cheddar offers the possibility of defining new schedulers and it is able to analyze and simulate the systems according to new scheduling policies.

The schedulers supported by Cheddar for simulation and feasibility analysis are: Rate Monotonic (RM), Deadline Monotonic (DM), Earliest Deadline First (EDF), Least Laxity First (LLF), the POSIX 1003b fixed priority scheduler and Maximum Urgency First (MUF). The tool also supports the inclusion of shared resources into the system models.

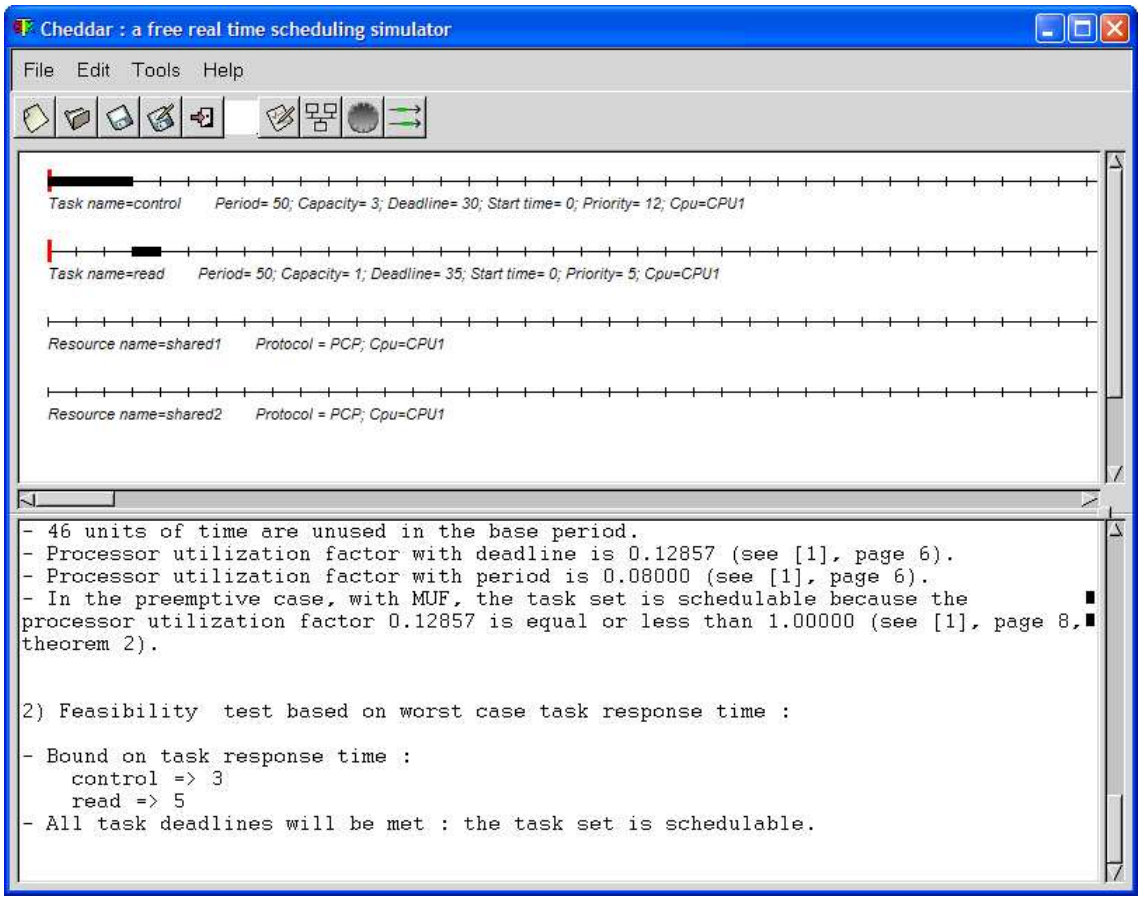

Figure 10. Cheddar user interface

In order to perform schedulability analyses, Cheddar uses system XML models as input. These models are conformed following a precise tool-internal metamodel. We will briefly overview the concepts included in the Cheddar metamodel depicted in the following figure. These concepts are used to create the system models that Cheddar will analyze and simulate.

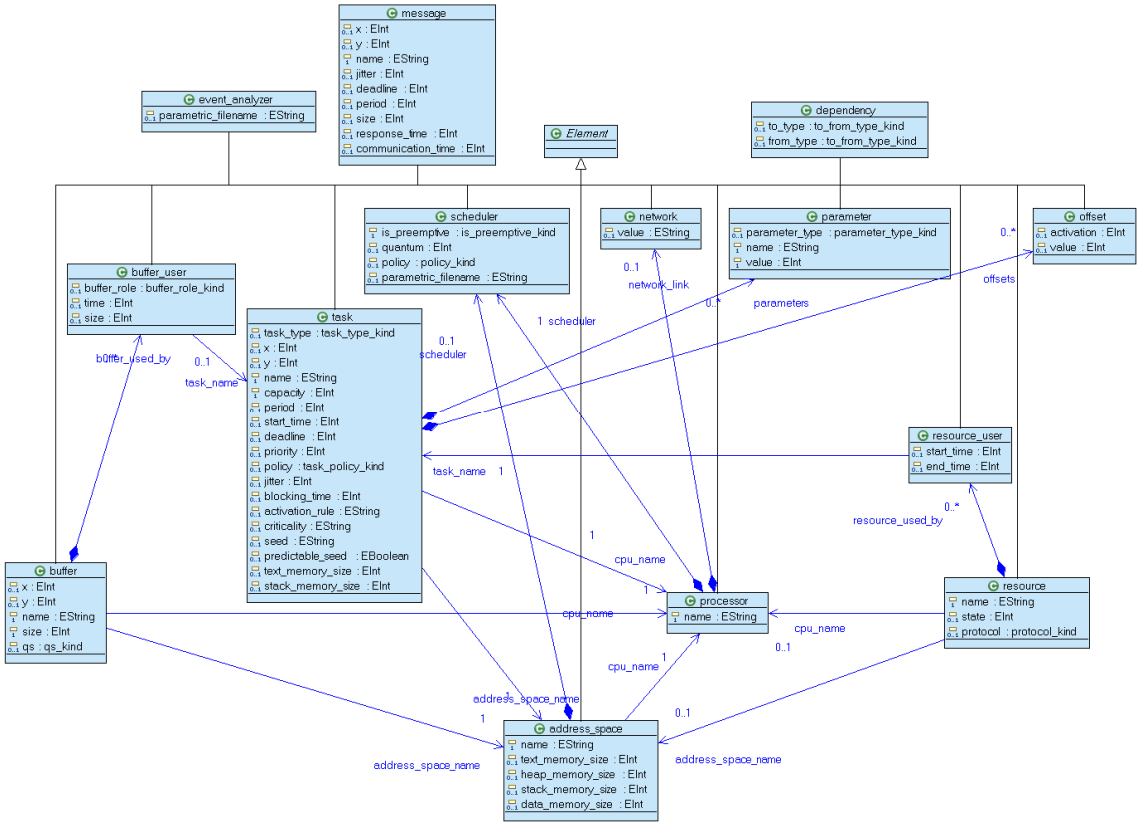

Figure 11. Excerpt of the Cheddar metamodel

Processor. Computing resources (i.e. processing cores or CPUs) are modelled in Cheddar as *Processors*. Each *Processor* has associated a *Scheduler* element and a name. *Schedulers* can be defined as preemptive or non-preemptive, and also a quantum value can be specified to define the maximum time a task may stay active before a rescheduling takes place.

Address Space. *Address Spaces* model memory areas reserved for a certain process in a *Processor*. *Tasks* are allocated inside some *Address Space* associated with a *Processor*. An *Address Space* element has a name and must have a hosting *Processor*. Additionally, an *Address Space* may be given a *Secondary Scheduler* element that will override the primary one for the tasks allocated in it. Lastly, memory properties can be specified in order to perform memory utilization tests.

Task. A *Task* element represents a thread running within a process. It is referred to a hosting *Address Space* and it must be characterized by several parameters (e.g. worst case execution time, period, priority, etc.) that affect them in different scheduling contexts.

Resource. The *Resource* element of the Cheddar metamodel represents resources shared by different tasks in a system (i.e. critical sections, shared variable and the like). A *Resource* is defined by its name, and its hosting *Processor* and *Address Space*. Moreover, it has a number of extra properties used to specify the number of tasks that may access the resource simultaneously, when do tasks require it and which will the access protocol be.

Task Precedence. Cheddar models allow the insertion of some behavioural aspects in the models. A *Task Precedence* element indicates that a task must be completed before another one may start its execution.

Message Dependency. Cheddar models use *Message Dependency* elements to include message based interactions between senders and receivers. A *Message* element must be defined and related to a *Message Dependency*. A *Message* element is defined by its occurrence properties, its size, its communication timing properties and the tasks sending and receiving it.

Buffer Dependency. Cheddar models use *Buffer Dependency* elements to include buffer based interactions between data providers and consumers in streaming interactions. In a similar manner to *Message Dependency* definitions, *Buffer* elements must be defined and related to a *Buffer Dependency*. It is important to note that buffers may only be defined in Cheddar as inter-task communication systems on a local host. A buffer element is defined by the hosting elements (i.e. processor and address space), its size, its queuing policy and the buffer users (i.e. a set of tasks). Buffers can be analyzed in Cheddar using buffer usage simulations and feasibility tests.

3.3.2 MAST

MAST is a tool suite for modelling and analyzing real-time applications. It has its own metamodels to create the models needed by the analysis and simulation tools. MAST tools make use of the concepts introduced in the metamodels to analyze and simulate real-time applications and provide the results.

MAST supports a variety of scheduling analysis methods: RM, EDF and Holistic. The tool suite also includes a scheduling simulator engine.

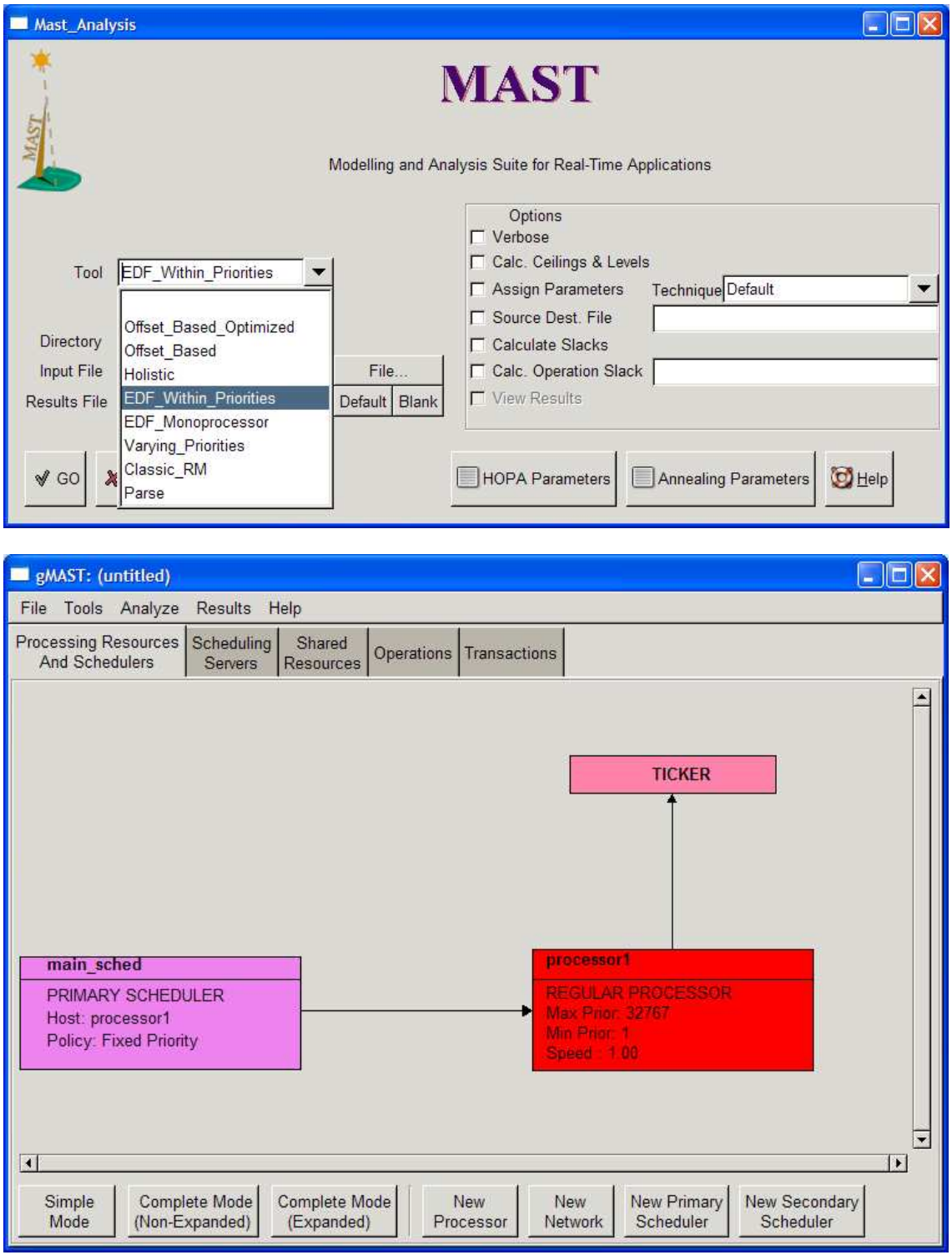

Figure 12. User interface of the MAST tool

As stated before, MAST includes a metamodel for modelling real-time applications and systems and a graphical editor (as depicted in Figure 12). The following paragraphs will briefly cover the concepts included in the MAST metamodel and the properties associated to them (see Figure 13 for further details). Further information about the MAST metamodel can be found in [16] and [22].

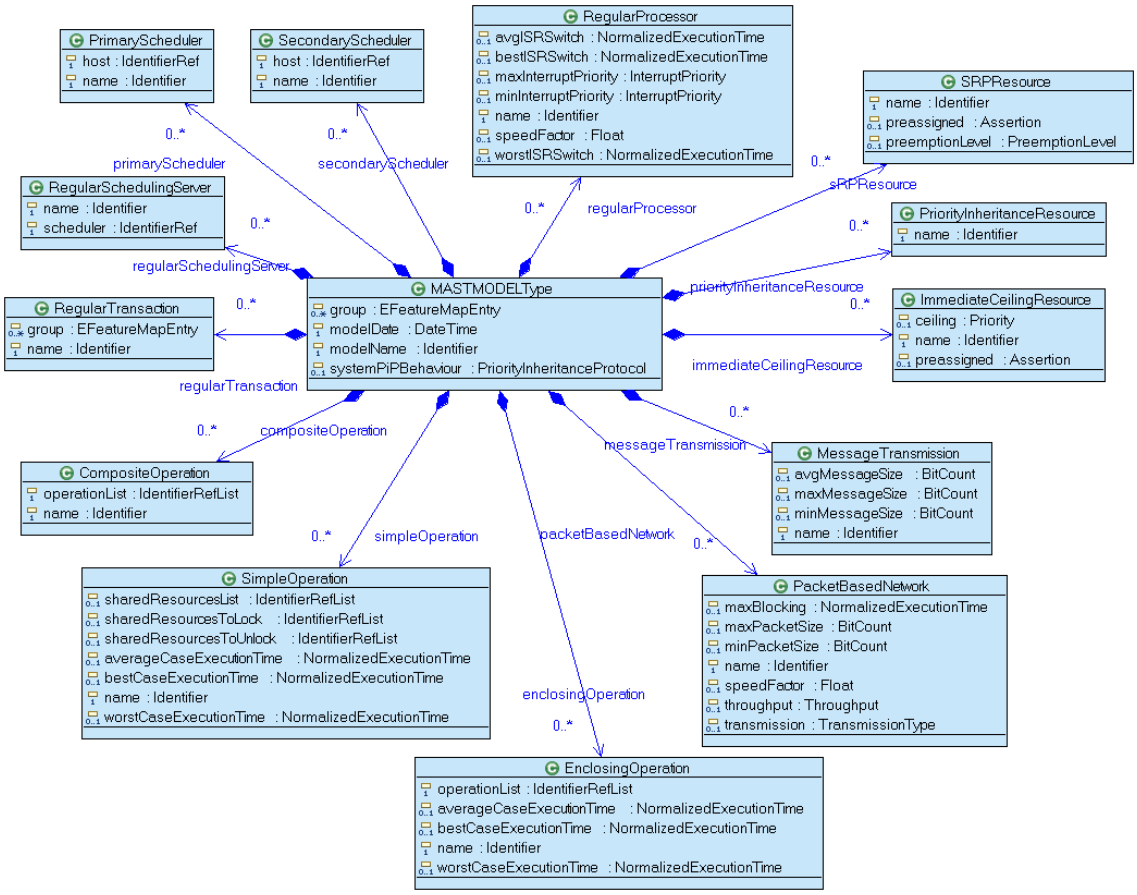

Figure 13. Excerpt of the MAST metamodel

Regular Processor. *Regular Processor* elements represent computing units in real-time application models. A processor in MAST is defined by its name, its timing constraints, its interrupt priority range and its speed factor.

Primary and Secondary Schedulers. A *Primary Scheduler* represents the main scheduling resource in an operating system. It is defined by an identifier, a host processing unit (i.e. a processor or a network) and a scheduler type (i.e. Fixed Priority or EDF). *Secondary Schedulers*, on the other hand, represent schedulers associated threads that have been programmed as virtual processors that execute a list of tasks.

Regular Scheduling Server. A *Regular Scheduling Server* represents the structure of a process in an operating system, that is, the resources that support the creation of threads, and it owns a series of executable code. A scheduling server is defined by its identifier, the scheduler in charge for managing it and the scheduling policy parameters that will be applied to it and to the shared resources accessed by it. Note that these parameters must be compatible with the host scheduler.

Simple Operation. A *Simple Operation* represents a simple amount of executable code which is executed in a regular scheduling server. *Simple Operations* are defined by an identifier and the timing characteristics that affect its execution. A *Simple Operation* may also override the priority defined for the scheduling server and it may also use/lock/unlock a list of shared resources.

Composite and Enclosing Operations. The MAST metamodel enables the definition of small behavioural aspects in the models using similar constructs as Cheddar (refer to the previous section). In order to establish an order of precedence between different *Simple Operations, Composite Operations* are used. This kind of operation is defined by a list of *Simple Operations* that are to be executed consecutively. On the other hand, *Enclosing Operations* represent more complex operations that contain unique code as well as calls to other *Simple Operations*. *Enclosing Operations* must specify their timing parameters independently from the *Simple Operations* enclosed within them.

SRP, Priority Inheritance and Immediate Ceiling Resources. These three elements represent shared resources in MAST. An *SRP* resource represents an unmanaged shared resource or a shared resource managed by a user-defined protocol, while the other two represent shared resources managed by specific priority modification protocols (namely, priority inheritance protocol and priority ceiling protocol).

Packet Based Network. Both the MAST metamodel and its analysis tool support distributed real-time systems modelling and analysis. A *Packet Based Network* represents a packet-based communication media for transmitting messages between tasks located in remote processing resources. A network is defined by a series of parameters: identifier, speed factor, throughput, transmission type, maximum blocking time and maximum/minimum packet sizes. Moreover a network must have a list of network drivers that manage the messages.

Message Transmission. A *Message Transmission* element models a special kind of operation that sends a message through a network. It requires the specification of the message size related to it.

Regular Transaction. A transaction element defines a concrete behaviour in a MAST model. MAST will perform schedulability analyses on each transaction defined. Transactions comprise not only tasks executing on a local computing resource, but

also packet based communications over networks and tasks executing in remote computing resources. Therefore, a transaction defines an end to end workflow performed in a real-time distributed system. Transactions are defined by Activity elements. Each activity has an input event and an output event, which contain the timing constraints related to it (i.e. transaction period, deadline, etc.), an operation element and an execution server. Transactions may also have event servers. Event servers affect the flow of events in different ways (e.g. event multicasting, event barriers, event delays...).

3.3.3 TIMES

The TIMES tool is a software for modelling and analyzing real-time applications. It is not only a schedulability analysis tool but also a systems modeller and a code generator. However, for the analysis presented here only its first two capabilities have been taken into account.

Regarding schedulability, TIMES provides a simulator and a schedulability analyzer. It supports RM, DM, Fixed Priority and EDF policies with shared resources; however, it does not support multiple processors nor distributed systems.

In a similar way as the other tools already revised, TIMES uses its own metamodel for describing real-time systems. The metamodel uses the following concepts (see Figure 14).

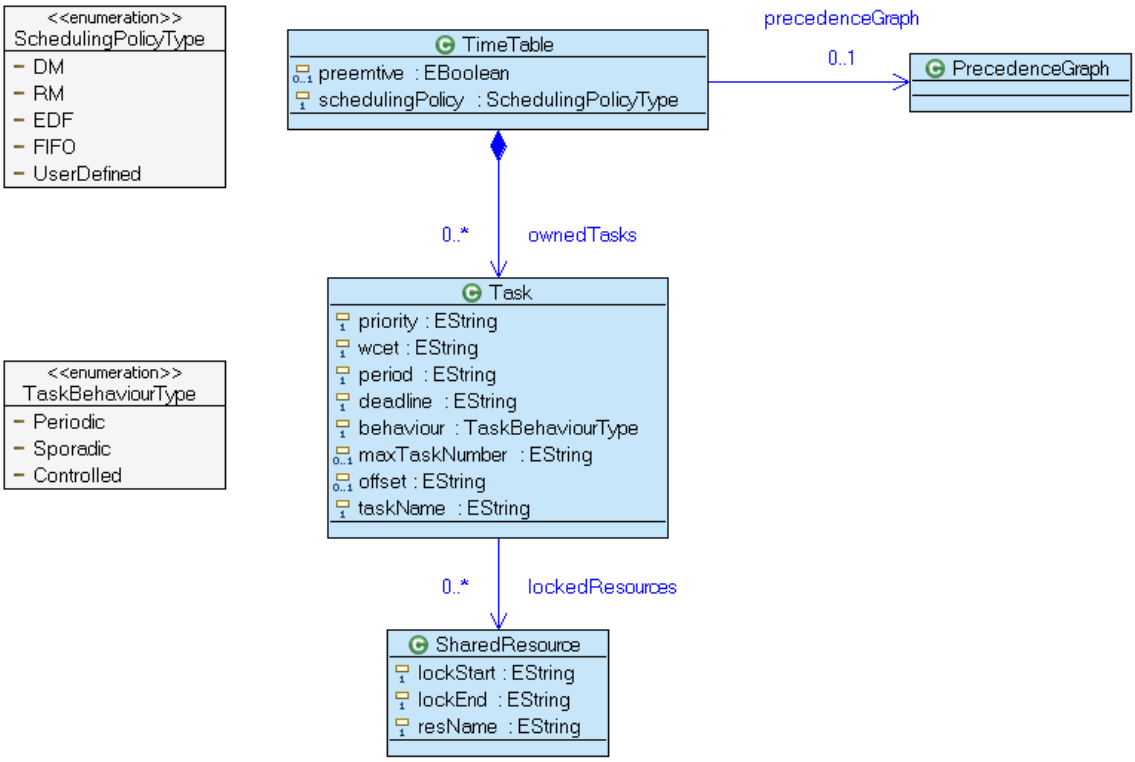

Figure 14. Excerpt of the TIMES metamodel

Task Table. Every TIMES model owns a single *Task Table* element. This element models the resources that will support the execution of the tasks (i.e. the processor, memory, etc.) and it defines the scheduling policy that will be applied to the tasks allocated in it.

Task. A *Task* represents a thread in the system. Since TIMES can only handle single processor systems, all the tasks are allocated in a single task table with a single scheduling policy. A task is defined by its worst case execution time, period, deadline, offset, priority and activation pattern (controlled, periodic or sporadic). A task may also use shared resources. In the latter case, each task must address the instants in which it will be accessing each shared resource.

Semaphores. TIMES allows the definition of resource sharing between tasks using *Semaphore* elements. They are described using only a name, since TIMES does not support any priority modifying access protocols.

Task Precedence. The TIMES metamodel includes the *Task Precedence* element to establish a certain order of precedence between tasks in the system.

3.3.4 RT-Druid

RT-Druid is a schedulability analysis tool implemented as an Eclipse plug-in. It is oriented to automotive applications; however, it is possible to use in any other domains by correctly adapting the metamodel concepts to the targeted domain.

The tool has a very powerful graphical user interface as provided by the Eclipse engine. Moreover, RT-Druid uses advanced modelling plugins in Eclipse, such as EMF, for developing new system models. Internally, RT-Druid manages a set of concepts organized as a metamodel (see Figure 15).

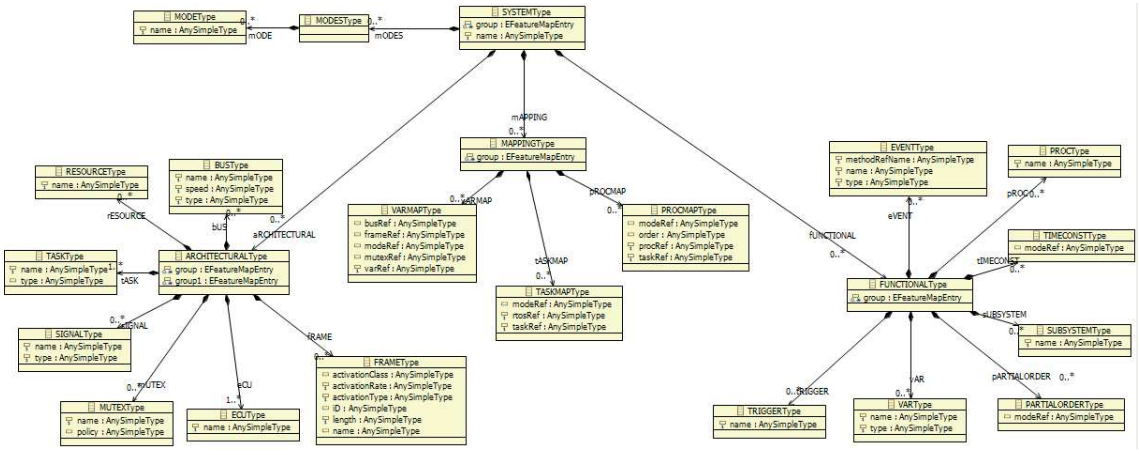

Figure 15. Excerpt of the RT-Druid metamodel

An RT-Druid system is defined by an application part (i.e. the FUNCTIONAL view) which contains the smallest pieces of application code, and a platform part (i.e. the ARCHITECTURAL view) which includes all the platform resources, hardware and software, that are required by the functional elements to build the final system. These two views are then put together by allocation using a special view MAPPING view. The MAPPING view relates each logical element with its architectural counterpart.

Regarding the FUNCTIONAL view, RT-Druid includes the following concept set:

PROC. The PROC element models the smallest piece of executable code in the system. It is possible that a PROC provides some methods to other PROCs in the system. Similarly, a PROC may use a method located in a remote PROC element.

VAR. The VAR element of an RT-Druid model represents an abstract data type. These elements contain internal data structures, as well as a set of methods used to access and manipulate them.

TRIGGER. A TRIGGER element models an external event that can activate one or more methods in a functional element (i.e. a PROC or a VAR).

EVENT. EVENT elements are used to represent the order between different executions or timing constraints associated to a method execution. They are linked to methods, and may represent the moment at which a method has activated, the moment at which a method started or the moment at which a method has ended.

PARTIALORDER. This element is a container of ORDER elements; each of which describes a precedence relationship between two events.

TIMECONST. The TIMECONST element represents a timing constraint applied too one or two previously defined events. The following constraints are available: deadline, period, minimum interarrival time, jitter and offset.

SUBSYSTEM. The SUBSYSTEM element allows the designer to introduce modularity in system models. A SUBSYSTEM may include a set of internal PROCs and VARs. Moreover, a subsystem has to define the required and provided interfaces. These interfaces are defined via methods.

On the other hand, the following concepts are included in the ARCHITECTURAL view of the RT-Druid metamodel:

ECU. The ECU element represents a complete computing unit, with a set of CPUs and an operating system. The characteristic of the operating system will define the scheduling behaviour of the system.

TASK. A TASK element models either a thread in an operating system or a thread dedicated to the interruption service routines of the hardware system. A TASK is characterized by a set of timing parameters and a set of scheduling parameters which will define it behaviour during scheduling analysis.

RESOURCE. A RESOURCE element is used to model the methods of the shared resources used by the threads in the system. A RESOURCE always has one or more references to a MUTEX element. This MUTEX is in charge of assuring mutual exclusion in the access to the resource.

BUS. The BUS element models the bus of the ECUs, that is, the communication media between different ECUs. (This element is not currently used).

FRAME. A FRAME element represents a set of messages transmitted through a network. (This element is not currently used).

SIGNAL. A SIGNAL element represents alarms and signals in operating systems (e.g. POSIX signals or OSEK alarms). (This element is not currently used).

MUTEX. A MUTEX element represents a binary semaphore used to establish mutual exclusion between concurrent accesses of different TASKs to shared data structures.

3.3.5 SymTA/S

SymTA/S is a schedulability analysis tool, developed by Symtavision, for systems, communication resources and complex systems. Moreover the tool provides some mechanisms to calculate resource loads, end-to-end latencies, worst-case response times and transmission times for CPUs and buses respectively.

The SymTA/S tool provides the user of a friendly user interface to model and analyse his models using a small set of elements. As Figure 16 depicts, the tool provides some palettes to create the application and architectural model of the system, which will be used afterwards as input for the analysis mechanisms.

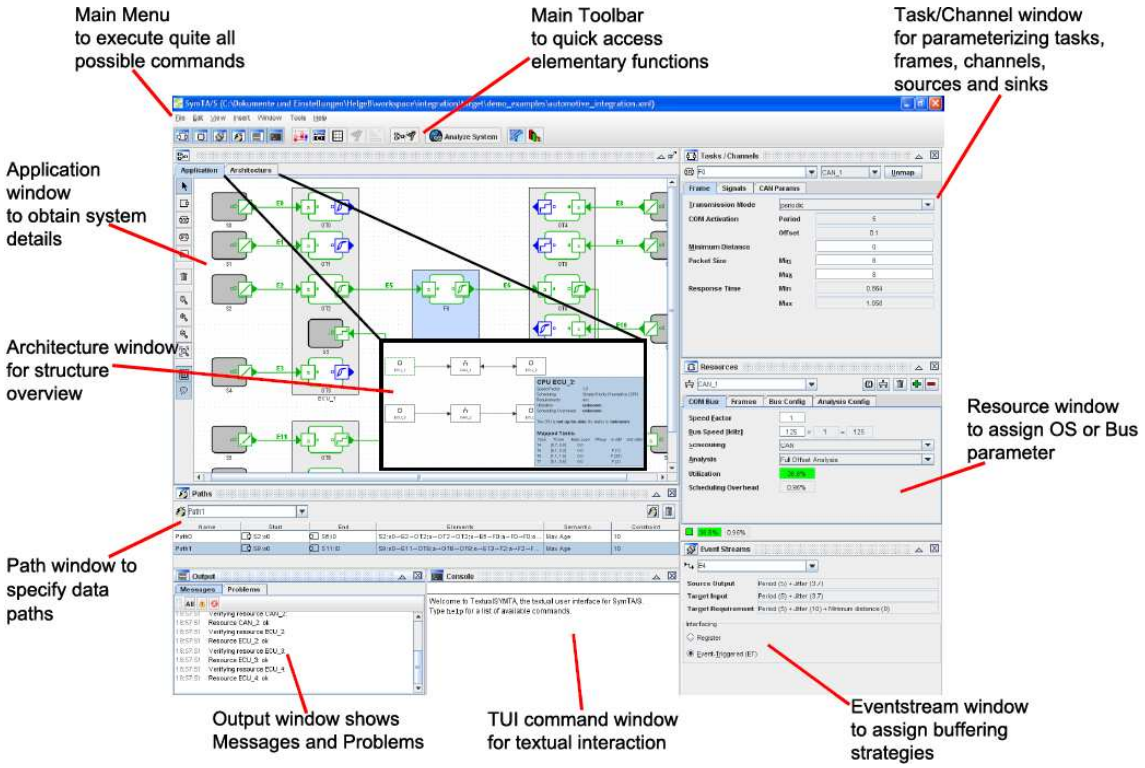

Figure 16. Modelling view in the GUI of the SymTA/S tool

The tool uses an internal metamodel to manage the data for the analyses, similarly to other schedulability analysers already covered in this document. The models are split into two different views: application and architecture. Regarding applications, SymTA/S provides the following components to create applications: Tasks, Channels, Sources, Sinks, Ports and Event Streams. These elements are represented in Figure 17. In the following paragraphs we will go over the elements of the metamodel, explaining their rationale and their use inside SymTA/S analyses.

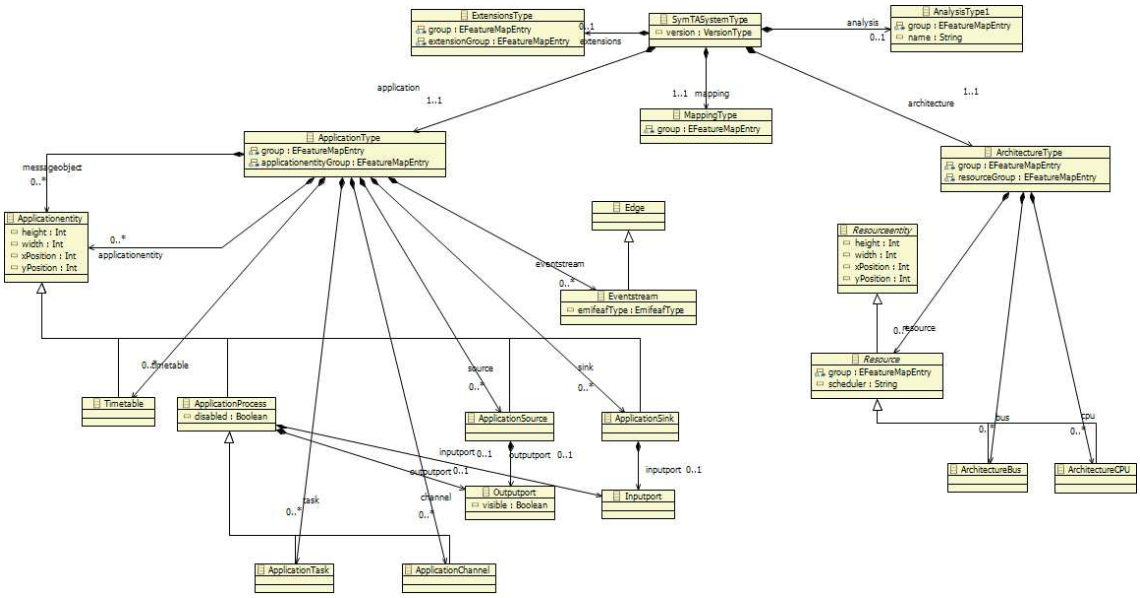

Figure 17. Excerpt of the SymTA/S metamodel

Application Sources and Sinks. Sources and Sinks are summarized as activation elements. On the one hand, Sources are responsible for activating a connected execution element, like a task or a frame. The output assertion of a Source's output port is propagated via an event stream to a connected execution element's input port. On the other hand, Sinks enable the definition of restrictions for output event models. SymTA/S will display an error, if the defined restriction is violated. This can be helpful, if an unknown system should be driven by your system, but you don't know any about system configuration except the essential input parameters. Sources and Sinks have only a single output and input port respectively.

Application Task and Channels. Task and Channels are summarized as execution elements. Tasks represent pieces of executable code that runs on some processing resource (i.e. CPU) in a SymTA/S system. Therefore Tasks are mapped on top of CPUs in the Architecture models. Channels, on the other hand, model communication media used to transfer messages for a task to another. Similarly, Channels are mapped on top of buses/networks in the Architecture models. Both, Tasks and Channels, may have an input port and an output port.

Ports and Event Streams. Execution and activation elements interact with each other using ports. Ports are interconnected using event streams. Ports leave data and activation events in the event streams interconnecting them. These two elements are purely logic, and they are not mapped on any kind of architecture element.

The SymTA/S toolkit also requires the user to provide a brief model of the software/hardware platform supporting the execution of the application elements. To do so the SymTA/S metamodel provides two elements: CPUs and Buses.

CPU. A CPU element represents not only a processing resource, but also the scheduling policy that manages the computing power of that resource. Task elements are mapped (i.e. allocated) on these resources.

Bus. A Bus element models any kind of communication media (i.e. a bus or a network) capable of transmitting messages from a source task to a target task. The Bus element in SymTA/S includes also the scheduling policy for dispatching the messages on the bus.

Lastly, SymTA/S includes an extensive library of preconfigured automotive ECU and bus elements. These elements can be directly used in the diagrams of the tool to obtain immediate timing results regarding our system. Note that, event though SymTA/S is oriented to the automotive domain, it is possible to use it in any real-time domain.

4. Variability management for Analyzable Models

In most cases, software is hardware-dependent and has to run under different configurations (communicating with different number and kind of devices; in the case of eDiana different number of cells and equipped with different devices for example). Thus, embedded software validation in a real environment becomes more difficult. All hardware and configuration aspects have to be configured and as the testing is done in final stages of development, to fix a bug is more expensive. Nowadays, in the development of embedded systems, approximately 50% of total development effort is spent on testing activities.

Software validation from early development stages is crucial in this kind of systems. Nowadays, there are tools that help perform software validations from software models, even before writing a single line of code.

UML profiles such as MARTE (Modelling and Analysis of Real-time and Embedded Systems) or its predecessor SPT (Schedulability, Performance and Time) provides facilities to annotate models with information required to perform specific analysis such as performance or schedulability analysis.

However, to properly validate software under the different configurations for which it may run, it is very important to manage variability of all aspects that affect the validation. As mentioned, embedded software validation is quite complex, one of the factors that influence in this complexity is the context diversity in which the software can be executed. In the eDiana case, software can be executed in multiple contexts

or configurations. For validating, it is required to consider those contexts and it can be considered as a software product line of validation contexts.

In software embedded quality aspects validation of systems that have to run under different configurations, three aspects have to being taken into account:

- Variability in software
- Validation environment variability
- Analysis/Testing scenarios variability

To validate software in a proper way, managing variability on the three aspects mentioned above is needed. Thus software can be validated in different situations i.e., running in different environments, taking into account different software quality aspects, etc. Each of the aforementioned aspects is described in the following paragraphs.

Variability in Software

Embedded software can be developed with different development paradigms:

- **Develop software through software product lines**: Each product configuration will carry specific software that will fit its needs through this approach. The advantages of this approach come from the reuse of core assets, reducing development time, optimized code (limited to the minimal subset needed for each product), reducing memory usage, etc. However, it may require more management regarding product maintenance.
- **Configurable software**: In this case, a unique software is developed, capable of executing in different environments through some configuration settings. The software is configurable to suit the requirements of the environment in which it will run. Maintenance management is not as laborious as in software product lines. There is not different software to manage. But instead, the ability to satisfy all configurations makes it more complex.

From the validation point of view, these alternatives must be considered in different ways. In the first case, it is necessary to instantiate the product from the product line in order to obtain the specific software wanted to validate. There are some proposals to reduce the validation cost by validating some products of the line and extrapolating the results to all of its products [26]. In the second case, the software installed in all products is the same but it usually has to be configured according to the devices connected in the environment. To validate this kind of software, the software must be configured/parameterized so the response of it can be analyzed in

different situations that may face. The choice made in the eDiana case it to produce configurable software.

Validation environment variability

Embedded software usually supports variable environments (which could be seen as varying configurations, which then discards software product lines which would require new deployment for every environment change). It may be connected to a different number of devices, running on different processors and so on. Managing this variability is essential to be able to create the right environment to validate each of the software configurations with its corresponding environment.

In the validation environment, variability may come from:

- Sensor number and kind: We understand sensor as the low-level, probably physical element that captures the data from the source. There are a variety of sensors where some of them correspond to the same functionality. Therefore, a validation environment may contain a variable amount and type of them.
- Actuator number and kind: We understand actuator as the low-level, probably physical element that performs a concrete action or operation at the very end. In the same way as sensors, there will be a variety of actuators that may be at the validation environment.
- Communication mechanism: The use of different communication mechanisms, whether due to a device from the environment may require it or to be suitable for the necessary features.
- Different Processors: The use of different processors with different features to obtain different response times. It has to be noted that distinct processors may produce varying results for identical operations (typical example being the decision to truncate or round results when performing fixed-point division). This requires minute definition of the operation behaviour of the model analysed.

In eDiana, different number and types of sensors and actuators may be present. And communication protocols can also be different.

Analysis/Testing scenarios variability

Not all configurations have the same requirements according to validation. Depending on the configuration, there may be functionalities that are not active and even in some cases, quality attributes may vary. There are certain configurations in which response times can be critical while in other settings may not be important.

With the aim of a proper software validation, it is necessary to take into account these types of variability and the relationship among them, so we are able to obtain all validation environments needed for each software configuration and the necessary tests to validate a particular configuration.

At the validation time, the validation environment will guide the instantiation of variability in other aspects. The validation environment represents a configuration in which the software has to run. Therefore, the first step to perform will be to instantiate the validation environment. This environment will determine:

- The software instantiation or configuration for that validation environment
- The instantiation of testing scenarios for that environment

4.1 Variability

Variability is a key aspect in software product lines but also in systems as eDiana. Variability [27] is understood as modifiability (to allow variation or evolution over time) and configurability (variability in configurations or products). Traditionally the main focus has been on functional variability. On the other hand, quality attribute variability has not received so much attention by researchers. In a product line, there are often products with varying levels of quality attributes. This variability should be modelled and managed from the beginning of the product line development and taken into consideration at different development stages. In a product line different members of the line may require different levels of a quality attribute, for instance they could differ in terms of their availability, security, reliability… One product may require a very high reliability whereas in another reliability is not important. There may also be products that have the same functionality but differ in quality attribute levels. The same happens in eDiana, different configurations (number of cells, number of devices in a cell, type of devices…) can impact on quality attributes.

4.1.1 Modelling quality variability

Quality in software systems has been considered an important issue from the beginning of software engineering. Software quality is the degree to which software possesses a desired combination of attributes [28]: Performance, security, availability, functionality, usability, modifiability, portability, reusability, integrability, testability [29]… And a quality attribute is "a property of a work product or goods by which its quality will be judged by some stakeholder or stakeholders" [30].

Quality in software must be considered in all the phases of development: design, implementation and deployment. But quality attributes must have an important role especially during the design stage, the design and the software architecture has a great influence on the system's final quality as it can inhibit or enable product's

quality attributes. Those quality attributes must be designed and evaluated at architecture level.

If meeting quality attributes is often a challenge in single-systems, in software product lines is much more complicated than in single-systems as products can require different quality levels and the product line can have variability on design that in turn affects quality. "Software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [31].

Quality attributes variability idea is considered in different works [32][33][34]. In addition, there are some approaches to model variability which take into account quality attributes variability [35]. However, there is a lack of quality attributes variability integration at software systematic management as mentioned by [32].

When modelling a system with variability it is essential to represent functional variability, quality attributes variability and impacts that may exist between them and/or with the environment. For modelling variability the following three vertices must be considered:

- Functional Variability: Functional variation can be optional, an instance out of several alternatives or a set of instances out of several alternatives [36].
- Quality Attributes Variability: As functional variability, same happens in quality attributes. Niemelä et al. [34] propose three types of variability for quality attributes:
    - 1) Variability among different quality attributes. For example, for one family member the reliability is important, but for other family members there are no reliability requirements.
    - 2) Different priority levels in quality attributes: For example, for one family member the extensibility requirements are extremely high, whereas for others those requirements are at the lower level.
    - 3) Indirect variation: We consider this type as a point out of quality attributes due to the relationship with functional variability too.
- Impacts: Functional or quality variability can indirectly cause variation in the quality or functional requirements.

Taking into account the three vertices of variability, several requirements, considered important for modelling quality attribute variability, are described below:

- Modelling and automatic reasoning: To provide a way to represent quality attribute variability in order to analyze and reason about the model. Because if so interesting information is captured, it is very reasonable to use it when

deriving or taking other type of decisions. Different reasoning tasks should be interesting: get an approximate value or level for several quality attribute starting from a set of functional requirements, detect impossible configurations starting from a set of functional and quality requirements, detect conflicts among qualities and provide help to performance a trade off analysis... Due to the complexity of this analysis and reasoning, it is very advisable to make it automatic. To achieve automatic reasoning artificial intelligence techniques are need. Three well known problems in the area of automated reasoning are Constraint Satisfaction Problems (CSP), Boolean Satisfiability Problems (SAT) and Binary Decision Diagrams (BDD) [37].

- Quality attribute characterization: Quality attributes have vague definitions. In different domains, one quality attribute may not mean exactly the same or different names are used for the same concept. So it is necessary to concretize and make quality attributes more specific. A mechanism for describing and explaining a quality attribute adequately must be provided: A structure where a quality attribute may be explained through refinement among different levels.
- Optionality: In one product one attribute may be important and in another this attribute not be required. So this attribute is optional in the product line. This may happen at quality attribute level but also at lower level, in the refinements of this quality attribute. For instance, in a quality attribute (performance) that is decomposed into two concerns ("Data latency" and "Transaction throughput"). Those concerns can also be optional or variant. This variability must be represented and not only at product level. It is not enough with specifying this optionality when deriving products.
- Levels: Different priority levels in quality attributes are need. For example, for one family member the extensibility requirements are extremely high, whereas for others those requirements are at the lowest level. However, quality attributes due to their nature are not easy to quantify, only more concrete concerns (refinement results) may be quantified. It is necessary to provide a way to define different levels (high, medium, low) at quality attribute high level and map those levels to more concrete concerns' values.
- Quantitative and qualitative: Indirect variation must be represented with qualitative and quantitative impacts and means must be provided to quantify qualitative influences to be able to do an automatic analysis. Some examples of impacts:
    - To have different languages impacts positively on usability (qualitative).
    - To be local application impacts very positively on availability (qualitative).
    - All features impact on Application Price (quantitative). The price of each feature is known.
- Group impacts: There are some types of influential relationships that are not addressed in all approaches, for instance, the influence of a group of variants. The impact of two variants together is not always the sum of the individual

impacts of those two variants alone. For instance, in some applications the price of some packages that have several features or options together may be cheaper that buying all the features separately.

4.2 Modelling variability with MARTE

MARTE provides facilities to annotate models with information required to perform specific analysis. Especially, MARTE focuses on performance and schedulability analysis. However, MARTE does not provide explicit means of modelling variability in performance and schedulability quality attributes.

In a software product line or system with variability, different levels of performance can be required and functional and design variability can produce products or configurations with different performance and schedulability. In this section, an approach is presented to integrate variability in quality attributes using MARTE.

Below, the required variability types for modelling quality attributes are described.

4.2.1 Types of Variability

Different types of quality variability can be identified. *Niemelä* [34] defines three types of quality attribute variability:

- *Variability among different quality attributes (optionality)*: For example, for one family member the reliability is important, but for other family members there are no reliability requirements.
- *Different priority levels in quality attributes*: For example, for one family member the extensibility requirements are extremely high, whereas for others those requirements are at the lowest level.
- *Indirect variation*: Functional variability can indirectly cause variation in the quality requirements

Starting from this classification, two broad categories has been distinguished and new types of variability for embedded software product lines has been also detected. Below, each variability category is detailed.

Variability in quality attributes

In this category, the variability that is required by quality attributes and their specification is included. This variability must be modelled in order to facilitate the evaluation of the quality of a system or set of systems with variability in quality.

The notation and types of variability proposed by the feature model [38] can be used. The feature model is one of the most used notation for domain analysis and variability modeling in software product lines. Following the variability types

proposed by the feature model, the quality attributes and their elements must allow expressing variability in the following way [38]:

- *Mandatory*: The element is compulsory for all the members of the family or line.
- *Optional*: The element can be present in some products and not in others.
- *Alternative*: The element will be present but can select from a set of variants.

An approach that allows detailing more the variability of elements is to annotate the elements with cardinalities [40]. Cardinality expresses how many copies of an element can exist. Mandatory and optional features are special features with [1..1] and [0..1] cardinality.

Cardinality should be modelled at individual level or at group level, that is to say, grouped elements with cardinalities (as proposed by [40]).

Relationships and variability

This category includes the indirect variation in quality caused by functional variability but also other relations among variability and quality (variability and the impact on quality).

The variability of other system elements can impact on quality attributes and this impact must be analyzed. The final goal is to be able to analyze or simulate a quality attribute. For doing that, it is necessary not only to consider the variability in quality attributes, but also know the aspects of the system that can affect indirectly a quality attribute (and determine its value).

In FODA, the relationships among features are described using the next constraints:

- Mutually exclusive with: the selection of an alternative excludes another.
- Requires: the selection of an alternative forces the selection of another.

Within [41] another new relationship is proposed that allows representing the impact of one or several elements in others.

- Impacts: An impact can be understood as a feature interaction (a feature or a set of features that modify a quality feature) following the definition of [42]: "A feature interaction occurs when one or more features modify or influence other features". The concept of *impact* has been introduced in the variability model to explicitly define impacts among functional, architectural and implementation variants and quality aspects. Those impacts can be qualitative or quantitative:
  - *Qualitative impacts*: Those impacts are defined in a first step and they are designer's and expert's dependant.

- *Quantitative impacts*: They are the result of product evaluations. Some products are evaluated and the achieved data helps to quantify impacts and detect interactions.

Using impacts the relationship among variability and quality attributes can be modelled in order to take them into account during validation. In an embedded software product line the following relationship among variability and quality attributes can appear:

- Variability among quality attributes:
  One of the aspects to consider when addressing quality attributes is the trade-off points that must be considered. It is impossible to optimize all the quality attributes because changing one quality attribute often forces a change in another quality attribute: positively or negatively. Achieving a quality attribute is often based on a cost-benefit relation or the goal is to achieve the optimal point among different quality attributes. For this reason, it is necessary to model the impact that a quality attribute has on another.

- Functional variability and its impact on quality:
  One of the variabilities defined by Niemela is the indirect variation; the functional variability can cause variation on quality requirements indirectly. Therefore, to be able to manage variability in quality attributes, it is important to model the functional variability and its relation with quality attributes.

- Variability in platform or hardware resources and its impact on quality:
  In an embedded system, the platform where the software will be deployed is a key aspect and has a great influence on quality attributes such as performance or schedulability. For this reason, it is necessary to model the relation among platform variability and quality attributes.

4.2.2 Variability in the MARTE Design Model

As mentioned previously the MARTE profile is structured around two concerns, one to model the features of real-time and embedded systems (MARTE design model package) and the other to annotate application models so as to support analysis of system properties (MARTE analysis model package).

In this section, guidelines for variability modelling using the MARTE Design Model enhanced with other approaches are proposed. MARTE will be enhanced with an extended feature model to represent the variability of the product line or configurations including the variability on quality attributes and variability will be introduced in MARTE model using an UML profile for variability.

4.2.2.1 Extended Feature Modelling

In [41] an extended feature modeling approach is proposed to model variability in quality attributes. In this approach, an extension of the quality attribute utility tree has been integrated into the feature model that models the variability of a system. This tree is used in ATAM evaluations [31] to describe and characterize the quality attributes in order to evaluate them. The proposed model for modelling quality variability is an extension of the feature model, a much known technique for modelling variability [38][43][44][45], etc. The proposed extension is based on the FeatuRSEB [46] approach which uses the feature model as a central element to capture the variability during the entire life cycle including design or implementation. This approach distinguishes among three types of features: functional, architectural and implementation features.

Quality attributes can fit in the definition of feature: "A feature is a prominent or distinctive and user-visible aspect, quality, or characteristic of a software system or systems" [38]. However, quality attributes have different meanings depending on the domain and sometimes they have imprecise meaning [31]. For this reason, it is necessary a mean for eliciting and refining quality attribute requirements (quality attribute characterization). Therefore, specifying quality attributes just as features is not advisable and feature models does not support the necessary characterization of attributes. For this reason, the feature model has been extended with special features to address quality aspects in a more adequate way.

A quality attribute characterization mean has been integrated in the feature model. The selected mechanism for describing and explaining quality attributes adequately is an extension of the quality attribute utility tree. Quality attribute utility tree is a model that is used in ATAM (Architecture Trade-off Analysis Method) evaluations [31] and that is oriented to characterize quality attributes to perform software architecture evaluations. A utility tree is a data structure that has a root called *utility*. Nodes below the root are names of *quality attributes* such as performance or security. Nodes below that level are *elaborations*, *refinements* or *concerns* – for example, performance may be elaborated as "high throughput" and "short end-to-end transaction latency". The leaves of the tree are *scenarios* that elaborate still further. Note that a utility tree is not an attempt at defining a rigorous taxonomy of quality attributes. Its purpose is to elicit a definition of system quality requirements in a practical way [31].

The utility tree has a similar structure to the feature model so it can be integrated in the feature model, allowing representing quality aspects as special features.

The utility tree does not provide a way to represent variability and other concepts so it has been extended. The resultant tree is called **quality feature tree** and allows characterizing quality attributes, concerns and scenarios and also more types of nodes that can facilitate characterization and also to express which quality nodes are

optional, alternative or mandatory. This approach is similar to *Olumofin's* [47] proposal of introducing variability in the utility tree.

One of the new types of nodes is the *level.* This new concept is used be able to define alternative groups of quality levels (high, medium, low) which are very useful during derivation.

The quality feature tree is represented as a branch in the feature model, the quality branch of the product line. An example of a feature model extended with quality aspects is shown in the Figure 18.

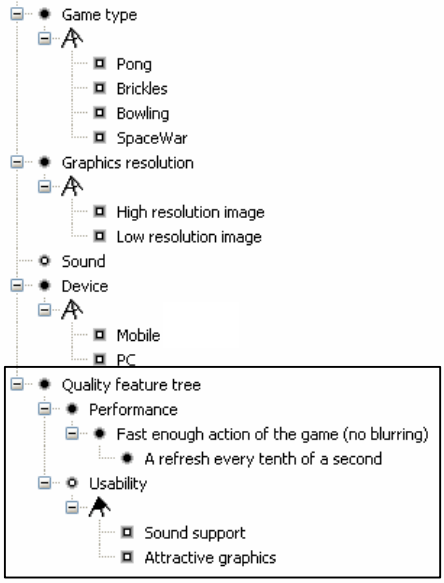

Figure 18: Extract of the extended feature model

This way, both functional and quality aspects can be represented in the extended feature model but a way to represent the *indirect variation* (functional variability that causes variation in the quality requirements) is still required. The concept of *impact* has been introduced in the variability model to explicitly define impacts among functional variants and quality aspects.

[48] also propose to add to feature models "quality features" characterizing design decisions that have impact on the non-functional requirements or concerns. And represent relationships with functional features in the feature dependency diagram. Requires, mutually includes… dependencies are modelled in this diagram. However, indirect variation is not represented.

4.2.2.2 UML notation for variability

As MARTE profile is defined for single systems, there is not any mechanism defined for modelling variability in this profile. Although other UML profiles specified for that purpose can be combined with MARTE profile besides using other UML mechanisms to handle all the requirements needed.

There are several UML profiles for specifying variability and product lines. Some of them focus on functional aspects and extend use cases to specify variability, others extend static models to specify variability and few works model variability in behavioural models.

Gomaa's product line profile called PLUS approach [49] is one of the most complete profile: feature modelling, use cases, static and dynamic modelling. [48] approach uses this profile to model variability and it is defined as "a well developed method applied to real-time systems, and concerned with the behaviour view needed for performance analysis". Ziadi's UML profile [50] for Product Lines is also a representative profile: it extends class and sequence diagrams to include variability and provides support for product derivation via Product Line constraints that guide the derivation process. It is the only one that concerns UML2.0 models and not UML1.x models. UML-F [51]: UML profile for frameworks can be also useful when product-lines have been developed following a framework based approach.

The variability profiles applies stereotypes to include variability in UML models. As stated in [49], it is important to realize that the role a class plays in the application (or in the analysis) and the reuse characteristic are orthogonal that is, independent of each other. Thus a class stereotyped as «SchedulableResource» (a MARTE stereotype for analysis purposes) could also be specified as «optional», «variant» ... of the variability profile.

Moreover, the Tawhid and Petriu's approach [48] combines or applies both profiles (MARTE profile and variability profile) together. They use the PLUS profile with small modifications: they introduce the concept of "quality feature", they use sequence diagrams for behaviour representation instead of collaboration (communication) diagrams, they use deployment diagrams and they modified slightly the Product Line stereotypes and tags in order to represent quality features.

Our proposal also applies PLUS profile together with MARTE profile but combined with the extended feature model explained in the previous section and the impacts for specifying the relationship among variability and quality attributes.

Starting from the diagrams and models proposed by the GENESYS project [52] for design modeling (see Table 1), variability is introduced using the PLUS profile and the extended feature model is added as the general view that will capture the variability of the line.

Table 1: Diagrams proposed by Genesys methodology for design modeling

Genesys methodology	**Views**	**Modelling language**	**Diagrams**
Platform Architecture model	Structural view	MARTE Generic Resource Modelling (GRM) sub-profile	Class diagram of processing units and tasks
			Class diagram of shared resources
			Class diagram of variables and shared memory
			Class diagram of communication resources
			Class diagram of platform black-boxes
			Class diagram of timing resources
		MARTE Software Resources Modelling (SRM) sub-profile	
		MARTE Hardware Resources Modelling (HRM) sub-profile	
	Behaviour view	UML	Behaviour diagrams (i.e. activity, sequence and state machine diagrams)
Application Architecture model	Structural view	MARTE High-Level Application Modelling (HLAM) sub-profile	Class diagram with HLAM and GCM stereotypes and composite diagrams with GCM stereotypes and UML2 ports.
		MARTE Generic Component Model (GCM) sub-profile	
		UML2 constructs	
	Syntactical view	MARTE High-Level Application Modelling	UML2 signal elements and stereotypes of

		(HLAM) sub-profile	HLAM
		UML2 constructs	
	Behaviour view	MARTE High-Level Application Modelling (HLAM) sub-profile	Behaviour diagrams (i.e. activity, sequence and state machine diagrams) with HLAM stereotypes
		UML	
	Semantic view	MARTE High-Level Application Modelling (HLAM) sub-profile	Extended UtilityType of HLAM sub-profile with ontology
Allocated model	Allocation view	MARTE Allocation Modelling (Alloc) sub-profile	Class diagram with allocate stereotype and using the structural views of both application and platform models

4.2.2.3 Traceability among models

To maintain the traceability among features in the extended feature model and classes in the design models will be essential. The PLUS approach takes into account the relations among classes and features with Feature/Class dependencies that are modelled in tables for maintaining the traceability among models.

4.2.3 Variability in the MARTE Analysis Model

As mentioned before MARTE Analysis Model provides facilities to annotate models with information required to perform specific analysis. Especially, MARTE focuses on performance and schedulability analysis.

The core purpose of real-time analysis is to estimate the capability of a system to provide timely responses to requests for (or initiations of) specified system-level operations, which we will call services, and to handle an adequate frequency of requests, under specified conditions. To enable this analysis, a UML model must specify the system-level operations, the frequency of requests, and the conditions of execution (which we may term its environment). Depending on the analysis aim, models are annotated in a different way, that is, different stereotypes are used for different purposes. In case we want to analyze the performance of the SPL modelled,

PaRunTInstance stereotype will be annotated for example. PaRunTInstance stereotype provides an explicit connection between a locality or role in a behaviour definition (a lifeline or swimlane) and a run time instantiation of a process, and optionally defines properties of the process. SaSchedObs stereotype will be used in schedulability analysis. SaSchedObs provides prediction about scheduling metrics such as overlaps, the maximum number of suspensions caused by shared resources or the blocking time caused by the used shared resources. All these metrics are relative to the interval defined by the reference and observed events.

Once the SPL is modelled taking into account the analysis we want to do, critical scenarios must be identified. These critical scenarios are not modelled from scratch, but models before annotated can be used and/or modified if necessary. Once we have got critical scenarios modelled, they can be transformed to the appropriate analysis models. The diagrams more used for analysis are sequence, activity, deployment and class diagrams. Although those are not the only ones that can be used. Different diagrams are used as input for different analysis models.

In an Analysis Model, the following variability types must be considered and addressed:

- Variable value of attributes: Quality attributes with value variation can be modelled by Value Specification Language (VSL). VSL (MARTE expression language) is used to specify the values of constraints, properties and stereotype attributes particularly related to non-functional aspects. In fact, this expression language can be used by profile users in tagged values, body of constraints, and in any UML element associated with value specifications. It deals with:
    - How to specify parameters/variables, constants, and expressions in textual form.
    - How relationships between different parameters/variables, or constant values are to be defined with support on arithmetic, logical, relational, and conditional expressions.
    - How different time values and assertions are to be defined in UML.
    - How to specify composite values such as collection, interval, and tuple values.

  In this way, attributes can be modelled by possible different values once data type has been specified. Data types that can be used for that aim are the following once:

    - IntervalType: IntervalType defines a collection of values, having the same type, contained between two given values. It is possible to use for process priorities in fixed priority processors for example, where priorities have a value from a range.

  - ChoiceType: In those attributes where value can be chosen from several alternatives, choice type can be useful. ChoiceType generates a data type each of whose values is a single value from any of a set of alternative data types, for example for message size in a communication between two processors.
  - …
- Impacts: The impacts that can be derived from functional variation can be represented by constraints with Object Constraint Language (OCL) [53]. OCL is a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modelled or queries over objects described in a model. Note that when the OCL expressions are evaluated, they do not have side effects (i.e., their evaluation cannot alter the state of the corresponding executing system). OCL expressions can be used to specify operations / actions that, when executed, do alter the state of the system. UML modellers can use OCL to specify application-specific constraints in their models. Thus, OCL can be used to specify impacts in a quality attribute caused by a functional variation in the system. In the same way, the effects derived from one or more impacts in the same quality attribute can be specified with the same mechanism.
- Hw variability and its impact on quality attributes: Hardware resources are very closed to software in embedded systems. Thus, any change in hardware resources impacts on quality attributes. Although it is a kind of impact, we decided to treat it as a different one due to its importance. In the same way as impacts, this relationship can be modelled by OCL.
- Variable Scenarios for analysis: Some scenarios must be defined and modelled with the aim of validating quality attributes. These scenarios must represent critical situations of the systems. As we are modelling a system with variability, variability must be included in these scenarios.
  Behavioural models are suitable to model critical scenarios, within sequence and activity diagrams. Variability mechanisms for functional variation as quality attributes variation are needed.

4.3 Variability in eDiana

In this section, the variability in eDiana platform is identified and the way of modelling this variability is selected.

A feature model will be used for modelling the overall variability (Figure 19) of all possible configurations.

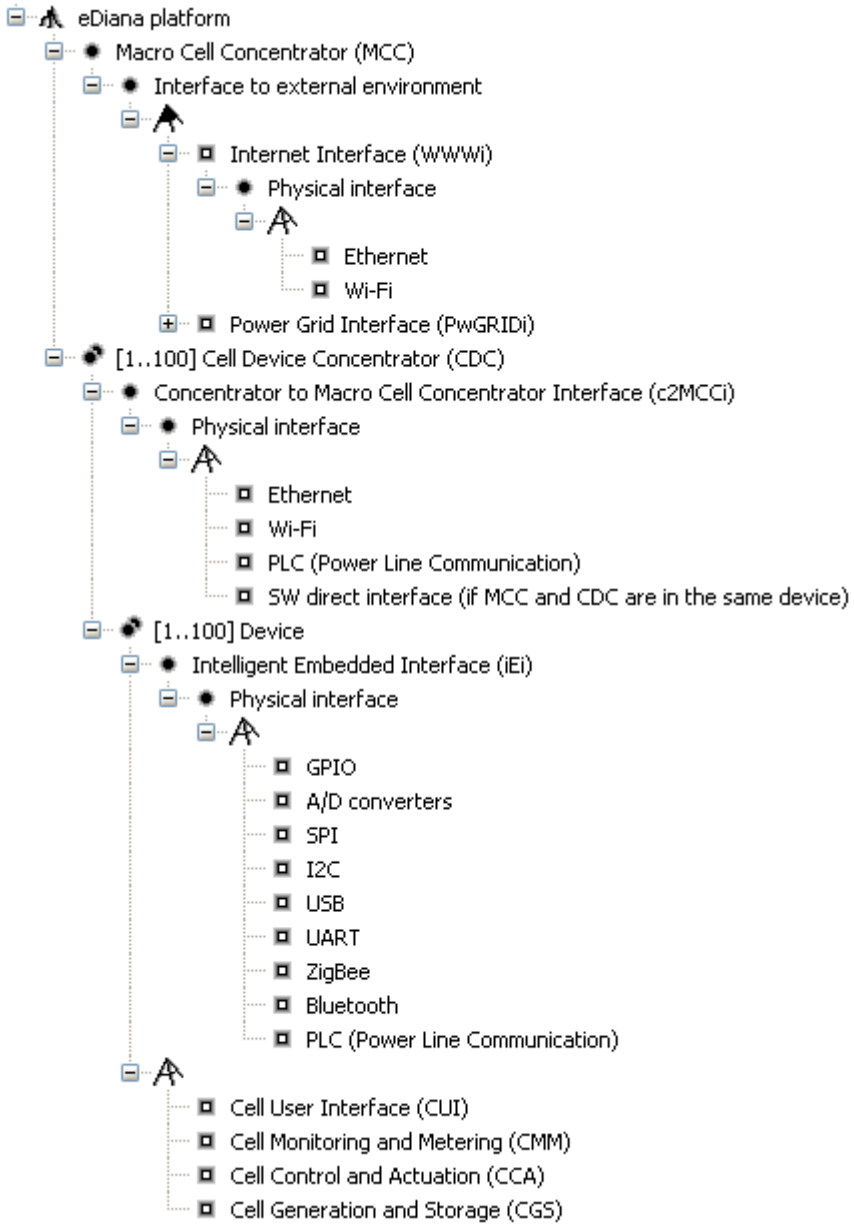

Figure 19: Feature model of eDiana validation software product line

The variability described in the feature model will impact on quality attributes. For instance, the selection of a physical interface or another will impact on performance. Ethernet will be faster than wifi and shared memory communication will be faster than Ethernet. The number of devices and cells of the eDiana configuration will also impact on performance aspects.

As the configurable software is deployed in a configurable environment, the *validation environment variability* must be specified. This variability will be explicit in the Platform Architecture Model where platform resources annotated for analysis are

represented. And the *analysis/testing scenarios variability* will be represented using behaviour models of the Application Architecture Model annotated for analysis.

Validation environment variability: Variability of eDiana platform to be considered during validation will include:

- Number of cells
- Number of device in each cell
- Type of devices
- Type of interface to external environment

Analysis/testing scenarios variability: The variability in environment will cause variability in analysis scenarios. For instance, for validating the scenario: "response time of checking energy consumption <1milisecond in a cell", in this scenario the Cell Device Concentrator has to ask to each device the energy consumption and calculate the aggregated value. The scenario must be variable to support different number and types of devices connected.

In eDiana platform, new devices can be added to the cell in a dynamic way (a new device is connected), so the system must be capable of modifying its own behaviour to take into account this new device. So in some scenarios dynamic variability must also be considered; variability that is bound at runtime.

4.4 Related Work

Some advances and work have been presented related to this topic. Tawhid and Petriu propose [48] a software product line modelling with functional variability and annotated with MARTE profile for performance in a general way (using variables). In order to validate quality aspects, concrete values are assigned to general annotations through ATL transformations that also are used to obtain a concrete product model. This approach focuses only on functional variability, without taking into account quality attributes variability.

MeMVaTEx methodology presented in [54], proposes the decomposition of design process in different abstract levels of EAST-ADL2 framework. For each level, requirements and solution models are created in a separate way. The interrelations between the elements of these models are specified through traceability mechanism of SysML profile while real-time issues and non-functional constraints are specified by MARTE profile. The methodology proposed focuses on requirements traceability from

analysis to implementation phase, taking into account temporal issues and regardless the variability of them.

The work presented in [55] and [56] focuses on the modelling and the performance analysis of hierarchical schedulers with AADL and takes into account MARTE notations. Hierarchical scheduler timing and synchronization relationships are expressed with a domain specific language based on timed automata: the Cheddar language.

Table 2 Different approaches comparative table

	Tawhid and Petriu's approach [48]	MeMVaTEx methodology [54]	AADL + Cheddar [57]
Functional variability	Yes, with SPL profile based on PLUS method	No, although EAST-ADL2 allows variability	No, although in [57] an extension for functional variability of AADL is presented
QoS Modelling	Yes, with MARTE	Yes, with MARTE	Yes, with Cheddar language or MARTE
Quality attributes variability	No	No	No

A summary of the three approaches is done in the table above, trying to highlight requirements we find necessary for modelling and validating quality attributes variability. Tawhid and Petriu's approach is the only one which takes into account functional variability and derives different products with different quality by the use of variables. They explain the followed process in detail. The three approaches compared use MARTE profile. Although these approaches don't show all the potential of MARTE, we can say that VSL (Value Specification Language) is a language that may facilitate modelling variability.

Regarding variability management in validation, [58] presents an approach using the tool Pure:variants for Simulink tool for managing variability and with a connection to simulink where validation is performed.

5. Conclusion

This document presented an analysis of several distinct modelling languages. Details have been provided of the possibilities of UML-MARTE, SysML, AADL, EAST-ADL2, AOM. Guidelines for making performance and timing analysis have been provided, by exploring the existing tools: PEPA and LQN for the former, and Cheddar, MAST, TIMES, RT-Druid, SymTA/S for the latter. UML variability profiles and their combination with MARTE have also been addressed, in order to be able to model systems with variability, which is a crucial requirement of eDiana.

UML MARTE and the rest of modelling languages that allow early analysis for checking non-functional requirements are complex languages and there is a lack of guidelines for applying them. This document answers this need providing guidelines for building the analyzable models (annotate quality aspects), for performing the analysis (existing tools...), etc.

This deliverable and the guidelines described in it are the basis for the next one: D6.1-B Derivation of V&V models from architecture models where a modelling methodology to guarantee the construction of consistent and unambiguous models for embedded systems' architecture or design will be provided.

Acknowledgements

The eDIANA Consortium would like to acknowledge the financial support of the European Commission and National Public Authorities from Spain, Netherlands, Germany, Finland and Italy under the ARTEMIS Joint Technology Initiative.

References

[1] Eurostat: "GDP and main components - Current Prices". Retrieved August, 19th, 2005
[2] IEEE. Ieee standard glossary of software engineering terminology. Technical report, Inst. Electr. & Electron. Eng., New York, NY, USA, 1990.
[3] A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2, OMG Adopted Specification, ptc/2008-06-09, June 2008
[4] OMG, UML Profile for Schedulability, Performance, and Time Specification, January 2005, Version 1.1, formal/05-01-02
[5] SysML, "OMG system modeling language (OMG SysML) V1.0," Tech. Rep. formal/2007-09-01, 2007.
[6] The Official OMG SysML site, http://www.omgsysml.org/
[7] W. SAE AADL, "The SAE AADL Standard," vol. 2008, 2008.
[8] Peter H. Feiler, David P. Gluch, John J. Hudak, The Architecture Analysis & Design Language (AADL): An Introduction, Technical report, CMU/SEI-2006-TN-011, 2006
[9] O. Youngseok, H. L. Dan, K. Sungwon and H. L. Ji, "Extended architecture analysis description language for software product line approach in embedded systems (extended abstract)," in 5th ACM and IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE'07, 2007, pp. 87-88.
[10] V. Debruyne, F. Simonot-Lion and Y. Trinquet, "EAST-ADL – an architecture description language," in Workshop on Architecture Description Languages, WADL'04; IFIP World Computer Congress, 2004,
[11] The ATESST Consortium, EAST ADL 2.0 Specification, 2008-02-29
[12] AUTOSAR web page, http://www.autosar.org/
[13] Ocarina: An AADL model processing suite, http://ocarina.enst.fr/
[14] F. Singhoff, J. Legrand, L. Nana, L. Marcé: Cheddar: A Flexible Real-Time Scheduling Framework. ACM SIGAda Ada Letters, volume 24, number 4, pages 1-8 (2004)
[15] MAST: Modeling and Analysis Suite for Real-Time Applications, http://mast.unican.es/

[16] Drake, J.M., Harbour, M.G., Gutiérrez, J.J., López, P., Medina, J.L., Palencia, J.C.: Description of the MAST model (2008)
[17] Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems, FORMATS 2003, Marseille, France, September 6-7 (2003)
[18] Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. Journal of the ACM, 20, 46—61 (1973)
[19] Tindell, K., Clark, J.: Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. Microprocessing & Microprogramming, Vol. 50, N. 2-3, 117—134 (1994)
[20] Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Trans. on Computers (1990)
[21] Tindell, K.: Adding Time-Offsets to Schedulability Analysis. Technical Report YCS 221, Dept. of Computer Science, University of York, England (1994)
[22] Medina, J.: Metodología y Herramientas UML para el Modelado y Análisis de Sistemas de Tiempo Real Orientados a Objetos. Ph. D. Thesis (2005)
[23] Greg Franks, Peter Maly, Murray Woodside, Dorina C. Petriu and Alex Hubbard. "LQNS User Manual" (2005). Available at: ftp://ftp.sce.carleton.ca/pub/cmw/userman-dec15-05.pdf
[24] Jane Hillston. "A Compositional Approach to Performance Modelling". Cambridge University Press. 1996.
[25] The PEPA Eclipse plugin. Available at: http://www.dcs.ed.ac.uk/pepa/tools/plugin
[26] Leire Etxeberria and Goiuria Sagardui. Variability driven quality evaluation in software product lines. In *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*, pages 243–252, Washington, DC, USA, 2008. IEEE Computer Society.
[27] Steffen Thiel and Andreas Hein. Systematic integration of variability into product line architecture design. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 130–153, London, UK, 2002. Springer-Verlag.
[28] IEEE. Ieee standard 1061-1992. ieee standard for a software quality metrics methodology, 1993.
[29] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
[30] SEI. Software architecture glossary (software engineering institute, carnegie mellon university). Web page: http://www.sei.cmu.edu/architecture/glossary.html, 2007.
[31] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, January 2002.

[32] Varvana Myllärniemi, Tomi Männistö, and Mikko Raatikainen. Quality attribute variability within a software product family architecture. In *2nd International Conference on the Quality of Software Architectures (QoSA)*, 2006.

[33] Günter Halmans and Klaus Pohl. Communicating the variability of a software-product family to customers. *Software and System Modeling*, 2(1):15–36, 2003.

[34] Eila Niemelä. Architecture centric software family engineering. Product Family Engineering seminars, Helsinki, Finland, October 2005.

[35] Leire Etxeberria, Goiuria Sagardui, and Lorea Belategi. Modelling variation in quality attributes. In Klauss Pohl, Pratrick Heymans, Kyo-Chul Kang, and Andreas Metzger, editors, *First International Workshop on Variability of Software-Intensive Systems (VaMos 2007)*, volume Lero Technical report 2007-1. Lero, 2007.

[36] Felix Bachmann and Len Bass. Managing variability in software architectures. In *SSR '01: Proceedings of the 2001 symposium on Software reusability*, pages 126–132, New York, NY, USA, 2001. ACM.

[37] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *10th International Software Product Line Conference (SPLC)*. IEEE Computer Society, 2006.

[38] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, November 1990.

[39] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration using feature models. In Robert L. Nord, editor, *SPLC*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2004.

[40] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities. *Integrated Design and Process Technology, IDPT*, June 2002.

[41] Leire Etxeberria and Goiuria Sagardui. Variability driven quality evaluation in software product lines. In l, editor, *International Conference on Software Product Lines, SPLC*, 2008.

[42] Jia Liu, Don S. Batory, and Srinivas Nedunuri. Modeling interactions in feature oriented software designs. In *Feature Interactions in Telecommunications and Software Systems VIII, ICFI'05, 28-30 June 2005, Leicester, UK*, pages 178–197, 2005.

[43] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[44] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[45] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[46] M. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the rseb. In *5th International Conference on Software Reuse, ICSR 1998*, pages 76–85, Vancouver, BC, Canada, jun 1998.

[47] Femi G. Olumofin and Vojislav B. Misic. Extending the atam architecture evaluation to product line architectures. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 45–56, Washington, DC, USA, 2005. IEEE Computer Society.

[48] Rasha Tawhid and Dorina Petriu. Integrating performance analysis in the model driven development of software product lines. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 490–504, Berlin, Heidelberg, 2008. Springer-Verlag.

[49] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley, 2004.

[50] Tewfik Ziadi, Loc Hélouët, and Jean-Marc Jézéquel. Towards a uml profile for software product lines. In *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, Revised Papers*, pages 129–139, 2003.

[51] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product line annotations with uml-f. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 188–197, London, UK, 2002. Springer-Verlag.

[52] Eila Ovaska, Kari Tiensyrjä, Sergio Campos, Adrian Noguero, Josetxo Vicedo, András Balogh, and András Pataricza. Model and quality driven embedded systems engineering. Technical report, VTT, 2009.

[53] OMG. Uml 2.0 ocl specification. Technical Report ptc/03-10-14, 2003.

[54] Roberto Passerone, Imene Ben Hafaiedh, Susanne Graf, Albert Benveniste, Daniela Cancila, Arnaud Cuccuru, Sébastien Gérard, Francois Terrier, Werner Damm, Alberto Ferrari, Leonardo Mangeruca, Bernhard Josko, Thomas Peikenkamp, and Alberto Sangiovanni-Vincentelli. Metamodels in europe: Languages, tools, and applications. *IEEE Design and Test of Computers*, 26(3):38–53, 2009.

[55] Frank Singhoff and Alain Plantec. Aadl modeling and analysis of hierarchical schedulers. *Ada Lett.*, XXVII(3):41–50, 2007.

[56] Su-Young Lee, Frédéric Mallet, and Robert de Simone. Dealing with aadl end-to-end flow latency with uml marte. In *ICECCS '08: Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems*, pages 228–233, Washington, DC, USA, 2008. IEEE Computer Society.

[57] Youngseok Oh, Dan Hyung Lee, Sungwon Kang, and Ji Hyun Lee. Extended architecture analysis description language for software product line approach in embedded systems. In *MEMOCODE '07: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 87–88, Washington, DC, USA, 2007. IEEE Computer Society.

[58] C. Dziobek, J. Loew, W. Przystas, and J. Weiland. Model diversity and variability - handling of functional variants in simulink-models. *Elektronik automotive*, February 2008.