VERification-oriented & component-based model Driven Engineering for real-time embedded systems

F2.2.4–**VERDE Methodology V1.1**

••

Document version no.:	1.1
Edited by:	CEA, TCF, TRT, SINTEF, FZI, Bosch, FHG, SCS, TAS, OBEO, ALS, AST

History

Document version #	Author(s)	Date	Remarks
Version 0.1	SCS	15.11.2011	Initial release based on F2.2.3
Version 0.2	SCS	05.12.2011	Update on Modelling Standards based on partner input
Version 0.3	SCS	09.12.2011	Added new patterns provided by CEA and TRT
Version 0.4	FZI	12.12.2011	Update on System C and added a new pattern for Connector Deployment
Version 0.5	TRT, TAS	19.12.2011	Update on Current Practices and Modelling Standards
Version 0.6	SCS	21.12.2011	Update on Modelling Standards
Version 0.7	TRT	26.01.2012	Added new pattern for performance analysis
Version 1.0	SCS	08.02.2012	Included Reviewers feedback
Version 1.1	TRT, OBEO	01.06.2012	Update pattern for performance analysis

Table of Contents

Table of Figures

Executive Summary

The growing complexity of real-time embedded systems is combined with constantly growing quality and time-to-market constraints. This creates new challenges for development projects in this domain. Traditionally most development processes used in this area are based on the V Model and other domain specific standards, where validation and verification activities start when implementation and integration is completed. Problems related to the architecture of a system are often identified late in the project cycle and are therefore more difficult and more expensive to correct.

The goal of this document is to provide information on a more iterative and incremental approach to software development, driven by the early validation and verification activities. Beside modelling of functional components and modelling of the execution platform, activities like capturing of timing constraints and modelling of non-functional properties are addressed as well.

Chapter 2 gives a brief overview of current practices used by the industrial partners in the various domains. This is a good source to find out similarities and differences in the processes and to define the expected improvements.

The VERDE methodology is based on modelling standards, mainly UML2 and its extensions. All standards that are taken into account are described in chapter 3.

An overview of existing work and previous EU projects deliverables can be found in chapter 4. These results can be seen as a good input and a good starting point for the results defined in this task of the project.

The main part of this document is chapter 5 where methodology patterns are described. These patterns provide practical and “easy to put in practice” modelling solutions for concrete modelling issues, that can be selected by end users when needed.

This version of the deliverable in an update of F2.2.3, delivered end of 2011. Beside corrections in chapter one, two, three and four, the main updates are in chapter five, where additional patterns have been added. These patterns are derived from other work packages and are also based on feedbacks from WP1 industrial cases.

1. Introduction

This document describes Methodological Patterns for iterative, incremental design of real-time and embedded systems (RTES). The modelling languages selected for representing components, applications, context, deployment and configuration plans, non-functional properties, tests, variability and in general any of the artefacts envisioned in the VERDE process are UML2 and its extensions for RTES. These languages propose a large number of modelling constructs that may be used in a variety of situations and for different validation techniques. The aim of this task is to select and restrict the modelling practices to those necessary to cope with the iterative, incremental and validation-driven design of component based RTES, and more specifically to their implementation and deployment on the concrete platforms proposed in the project. This is necessary to drive into industrial practice the large amount of research in the field, reducing complexity when possible and posting directions to follow in concrete situations.

1.1 Relation to other Work Packages and Tasks

This deliverable is the result of task 2.2 of work package 2. Figure 1 shows the relationship between WP2 and the other work packages.

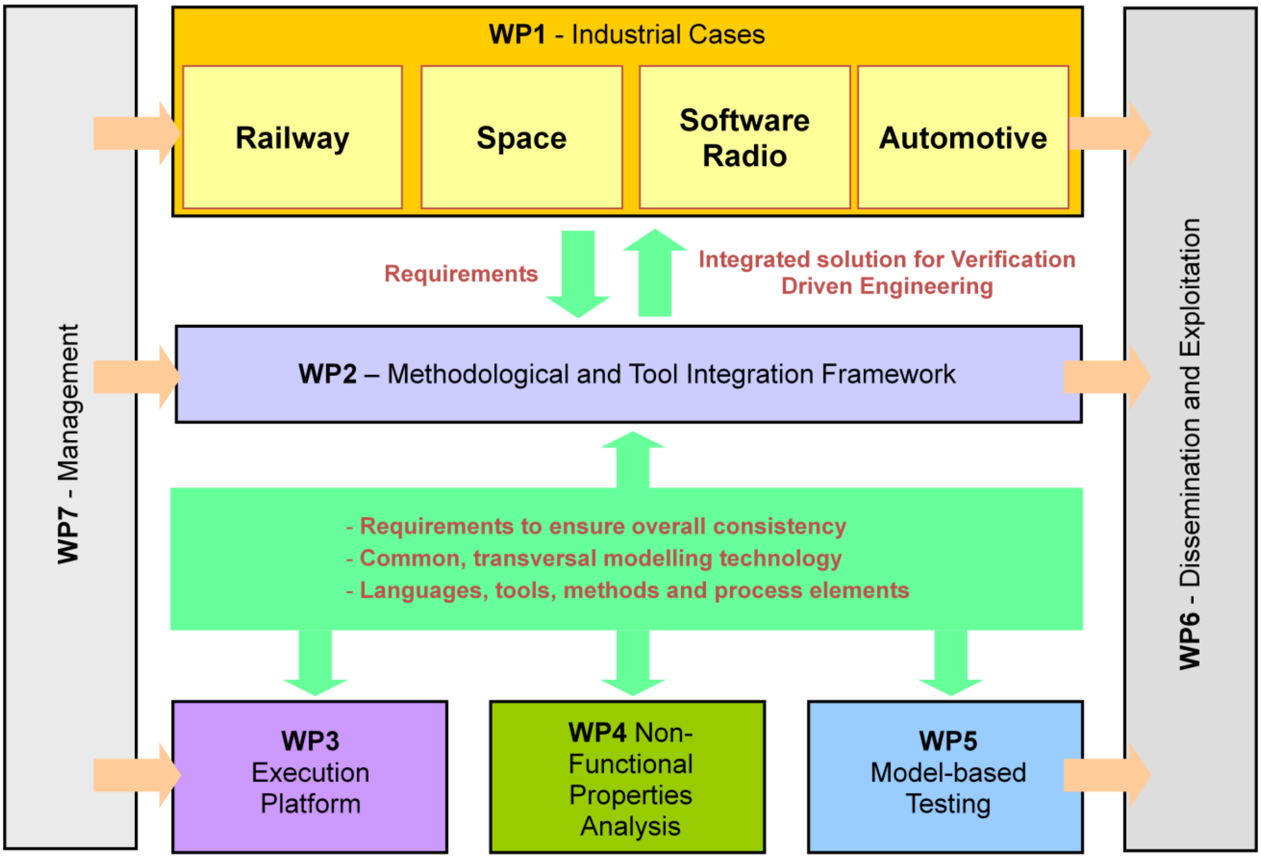

WP1 - Industrial Cases
Railway
Space
Software Radio
Automotive
Requirements
Integrated solution for Verification Driven Engineering
WP2 – Methodological and Tool Integration Framework
- Requirements to ensure overall consistency
- Common, transversal modelling technology
- Languages, tools, methods and process elements
WP3 Execution Platform
WP4 Non-Functional Properties Analysis
WP5 Model-based Testing
WP7 - Management
WP6 - Dissemination and Exploitation

Figure 1: Relation to other Work Packages

The Requirements that drive the outcome of this task are defined in work package 1. This task also provides information on current practices in the various domains. Results of this task will be proved in domain specific use cases, also defined in work package one.

This deliverable is related to other deliverables of work packages 3, 4 and 5. Information from these deliverables is included in this document.

2. Current Practices (Industrial Partners)

The industrial partners in the VERDE project provided information on the current development processes and the used modelling techniques. This chapter provides for each of the industrial domains an overview extracted from the VERDE deliverable F1.6.1: Evaluation Criteria.

2.1 Current Practices in the Radio Communication Domain

2.1.1 Current Development Process at TCF

We focused in the following section on the description of the software process description, as it is currently applied at TCF (mainly GPP based). The enhancement proposals to this process will include System requirements allocated to the hardware in order to validate early in the process non-functional properties.

The current development process is described in the figure below.Each arrow implicitly associates decision points to evaluate the capability to go further in the process.

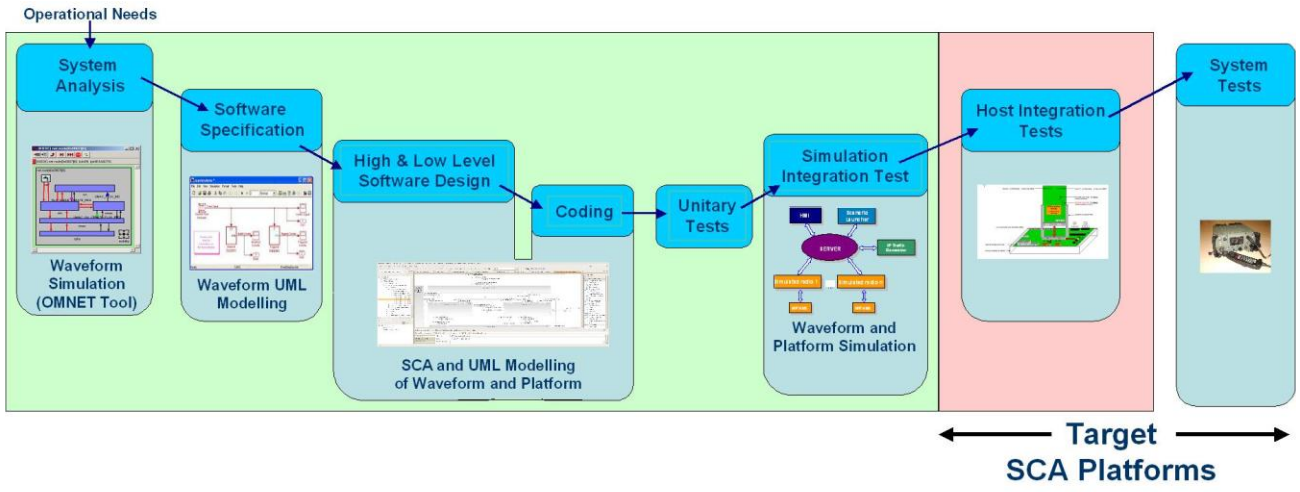

Operational Needs
System Analysis
Waveform Simulation (OMNET Tool)
Software Specification
Waveform UML Modelling
High & Low Level Software Design
SCA and UML Modelling of Waveform and Platform
Coding
Unitary Tests
Simulation Integration Test
Waveform and Platform Simulation
Host Integration Tests
System Tests
Target
SCA Platforms

Figure 2: Current design process for Software Defined Radio (SDR)

The first step is based on System analysis as validation of the hypothesis of the specific protocol layers from the System Requirements Allocated to the Software. These validations are done from specific modelling tools, which could be either Matlab for Signal processing specific algorithms or protocol simulation layers in order to validate parts of protocol layers. This first step aims at the definition and specification of the desired waveform as well as the definition of a preliminary software architecture, taking account the characteristics of the target SDR radio.

The second step is based on a definition or use of already implemented Radio Devices and Radio Services on the platform, which corresponds to the definition of the different SCA based components to be implemented on the platform and used by the resources defined to design the waveform. This software application also implements the interface with the SCA based platform (standardized API). The interface code generation as IDL files is the result of this Software specification phase. This second step, software development, allows for the development of several software components that will be executed on different processing elements and need to be connected together.

The third step reuses the interface definition, and goes further in the SCA component decomposition, in particular in describing the resources used by the waveform, and intra SCA components decomposition. The components deployment on the platform is defined at this stage. XML code generation results from the description of this specification.

The following fourth step corresponds to the encoding of the waveform code, called business code or application code, describing the components behavior. Unit testing of these components follows.

A next verification step (Simulation integration test) validates on host the behaviour of several radio units, with the physical layer simulated. This verification validates only behaviour on complex scenarios involving data transmissions and control information exchange. This test aims at testing the developed software in a “host test environment” prior to integration on the target SDR. The waveform will be validated with communication rules (based on CORBA exchanges) common with the ORB implemented on the based SCA compliant target platform. The compliancy with the adaptation layer (set of un-standardized drivers on legacy radio implementation) is there validated on host by construction.

When the Software is verified on host, it is integrated on target, on which the different components (Physical layers, vocoder, crypto modules, radio protocols are validated as a whole, taking care of non-functional (mainly timing) results. This step finishes the system level testing of the waveform from a functional and performance point of view.

This phase validates if the software and hardware are (at least) qualified on further ground tests.

2.1.2 Expected Improvements

The evolution of this process can be folded on the following major topics:

- Uniform notation based on component based design to specify, design and encode the waveform.
- Modelling of the software and hardware components and their allocation in order to validate non-functional properties on host.
- Automatic test generation in order to complete the manual tests in the software integration phase.

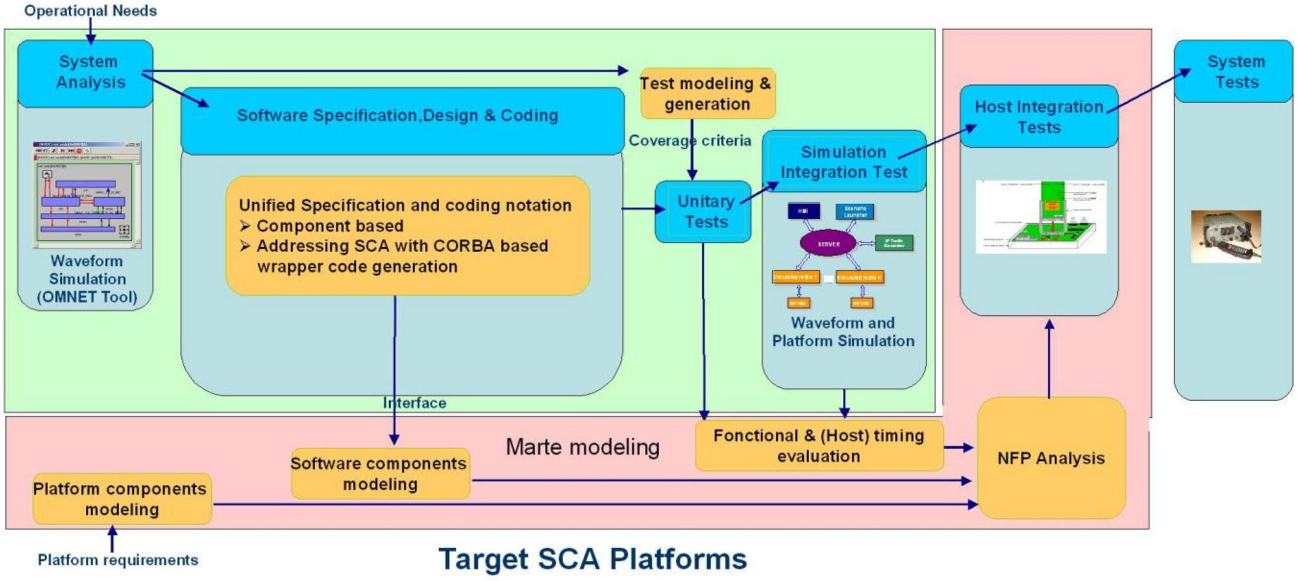

Operational Needs
System Analysis
Waveform Simulation (OMNET Tool)
Software Specification,Design & Coding
Unified Specification and coding notation
➢ Component based
➢ Addressing SCA with CORBA based wrapper code generation
Test modeling & generation
Coverage criteria
Unitary Tests
Simulation Integration Test
Waveform and Platform Simulation
Host Integration Tests
System Tests
Interface
Marte modeling
Software components modeling
Platform components modeling
Fonctional & (Host) timing evaluation
NFP Analysis
Platform requirements
Target SCA Platforms

Figure 3: Improved design process for Software Defined Radio

An improved process is shown in the figure above.

2.2 Current Practices in the Space Domain

2.2.1 Current Development Process at Thales Alenia Space

The Thales Alenia Space On board software development is based on the V Cycle development. In this development process, the left part of the V (or the descending part) represents the development part. And the right part (or the rising part) represents the validation part. In the following a closer look to the main development process steps is presented.

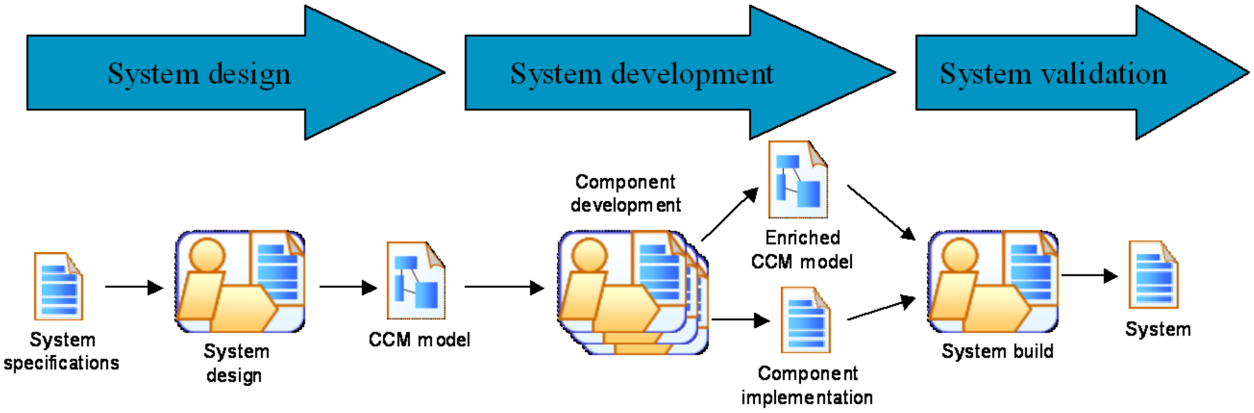


Figure 4: Current design process for Space Software Systems

System specification

The first step of the software development is the software specification. During this phase, the requirements and specifications coming from the system engineers are analyzed and some software level specifications are produced.

High level software design

During this phase, the software architects design the software according to the incoming specification. The software architecture is modeled in the CORBA components model, using the Melody CCMinternal Thales modeler. The system is split in functional components who communicate through defined interfaces to provide required and provided services.

Low level software design

Once each component has been defined, each component is detailed in a stereotyped UML class model.

This model will contain the implementation definition of the component (detailed design). This model will define the different operations provided by the component (or application). The UML model can be generated automatically from the CCM model. The design can be refined by adding packages or internal operations.

This model can also contain some non -functional properties such as real time information.

Coding

Once the modelling steps have been done some code is automatically generated for the component interfaces and internal procedures. The software developers now have to implements the functional code in the generated skeleton.

2.2.2 Expected Improvements

The major expected improvements shall allow the modelisation, the generation, and the execution of tests, based on the system model.

Therefore, the test framework should allow to express test objectives (non-functional properties, or expected system behaviour) related to the system requirements. It should provide the capability to define the interactions of the environment with the System under test.

Based on the above elements, test cases should be generated, and the capability to refine them should be provided. The framework should support the execution of the test set, and provide results and feedback.

The main process evolution topics are:

- Capability to model / generate tests from the system model
- Capability to annotate the system model with non functional properties in order to validate them
- Capability to execute tests and give a feedback

2.2.3 Current Development Process at EADS Astrium

As illustrated in Figure 5, the generic process starts with a concept definition (Phase 0) and a feasibility investigation (Phase A). When having proved feasibility the design will be industrialized, first through a detailed definition of the system design (Phase B activities). This phase is based on the preliminary system design as defined in the Phase A and refines the system design with respect to the various engineering discipline in an iterative manner.

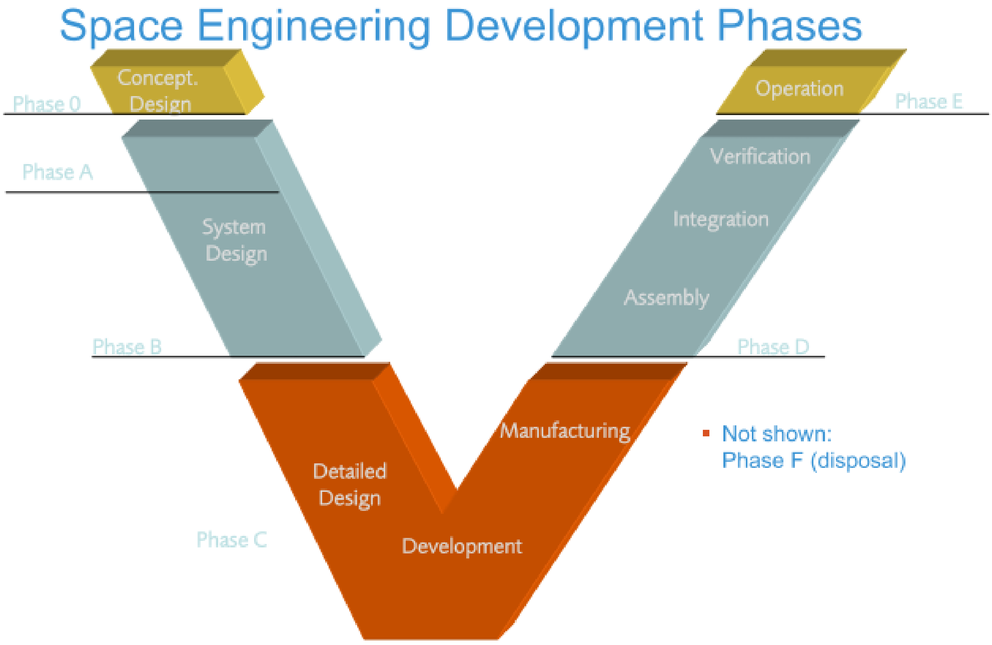


Figure 5: V-Model of ECSS-E-10 Process Phasing

At the end of Phase B and based on the design, the equipment specification is derived and suppliers are selected.

After the supplier selection the S/C design as well as the AIT plan has to be updated accordingly and verified subsequently. This is due to the fact that the design evolves more and more from specification to as-built. During the Phase C and Phase D the spacecraft equipments and related flight software are developed and assemblies are integrated more and more to form the complete spacecraft. These phases are concluded by comprehensive environmental and functional tests to verify and validate the spacecraft system and its components.

As it can be seen from above explanations, the flight software is developed in parallel of the remaining spacecraft equipments. The enabling technology to do so is called Astrium in-house the Model Based Design and verification environment. This environment allows to support the flight software development as well as its verification through extensive positive and negative test on a variety of test benches such as Software Verification Facility (SVF), Electrical Functional Model (EFM), hybrid test benches with H/W and S/W in the loop.

2.2.4 Expected Improvements

The Verde project results will have a direct effect on the ASTRIUM GmbH model based infrastructure. It is conceived that the system simulation development, the flight software development as well as the verification planning will benefit from Verde activities.

For instance the system simulation as well as the flight software development requires a certain system model representation to efficiently develop the SW. A core system model itself describes the overall system architecture including interfaces which can be subdivided into:

- System analysis
- System topological design
- System physical design
- System AIT design
- Operational concepts

This model then shall be automatically transferred to engineering domain models where detailed analysis can be executed. Also the results of the various analysis activities will be provided back to the design activities to contribute to the consolidation of the design.

Through this, one approach is to use the same principle for verification planning to derive automatically verification cases which are established via requirements in the system model. However this requires a strong correlation between the system model the transformation rules either to engineering domain or the verification model in both aspects syntax and semantic.

2.3 Current Practices in the Railway Domain

2.3.1 Current Development Process at ABB

ABB has together with SINTEF, Oslo developed a graphical train station description called Train Control Language (TCL) in the MoSiS project. Interlocking code for simple train stations can automatically be generated based on this language, a corresponding Meta model, and a predefined library for all basic elements.

- The final code is run on ABBs AC800XA system placed on each station.
- The code is run on a PC when it is under development and testing.
- Development testing of the safety code for Interlocking is done in several steps.
- First of all basic, library elements tested by formal methods and manually inspection.
- Then the code for an individual station is tested for correct operation.
- Finally, all safety features are retested.

The Verde project will help to make the testing more efficient by using the TCL description to be the base for more automatic test generation in the development phase of the project.

In addition we will incorporate the results from a parallel project called Cesar to make the link between the requirement specification and testing.

The Cesar project will include the formal requirements and methods for developing tools to generate safety code according to EN61129.

2.3.2 Expected Improvements

- The test patterns should be generated automatically if possible.
- A link must exist between tests and requirements specifications.

- Manual interaction shall be easy with graphical environments.

2.3.3 Current Development Process at Alstom

Alstom Transport systems are developed from productized platforms complemented with specifically developed products. The product platforms are already the results of the integration of hardware and software platforms. This development strategy is the logical consequence of a strict safety conformance and the high level of reliability expectations. Platforms evolve over time through an incremental approach of new releases.

Developments (of software and hardware) are ruled by CENELEC norm which defines the mandatory activities and the evidences that must be produced and documented. The basics of the associated development process are those from the “V” Model life cycle. Applying this constraining paradigm leads, at the end of the development cycle, to a secure result and a reasonable confidence about the functional match with the needs.

However, the level of safety is also the outcome of the selection of platforms (i.e. software and hardware) and the strong interaction between them. This commonly leads to a late discovery of gaps or errors after several steps of test and integrations have been achieved. As a consequence long-lasting iterations are often required due the numerous developments steps and evidences to produce and document.

Software development process of safety related software:

This process classically defines milestones complying with company’s life cycle. The milestones are the triangles in the following diagram:

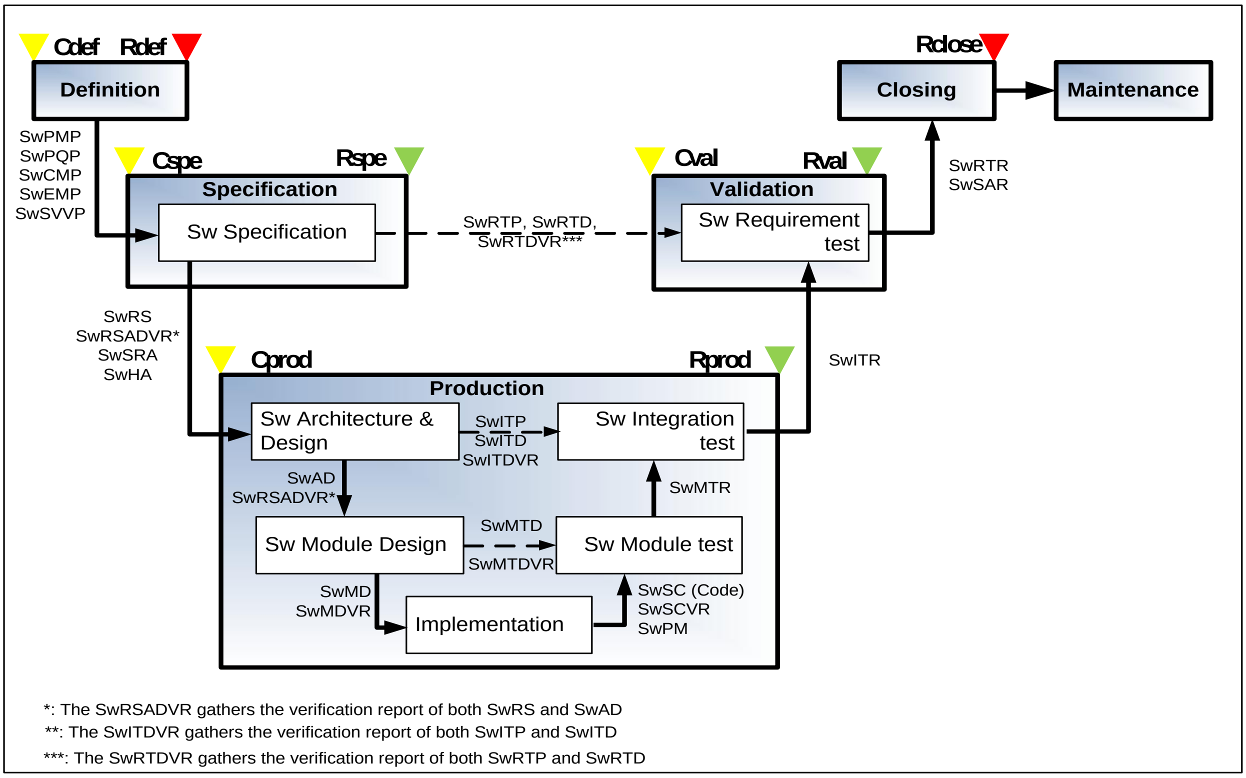
- Cxxx correspond to the start of phase check points
- Rxxx correspond to the end of phase reviews

The development process is phased as follow:

- Software project definition (D)
- Software specification (S)
- Software preliminary design (C)
- Software production (P), being composed of:
- Software Architecture and Design (AD)
- Software Module Design (MD)
- Software Implementation (IM)
- Software Module testing (MT)
- Software Integration testing (IT)
- Software Validation (V)
- Software Project closure (PC)

Each activity results have to be documented and verified: verification activities aim at checking that the results can be traced from entries with a full coverage.

The resulting documents are tagged with “VR” in the following diagram:



*: The SwRSADVR gathers the verification report of both SwRS and SwAD

**: The SwITDVR gathers the verification report of both SwITP and SwITD

***: The SwRTDVR gathers the verification report of both SwRTP and SwRTD

Figure 6: Category B4 life cycle

Even if activities can overlap to some extent, this development cycle is mostly sequential. Major developments are organized into releases implementing then the functional roadmap of the system to cover business needs.

2.3.4 Expected Improvements

The upstream part of the current development process provides an efficient definition and documentation of requirements, top level design and analysis.

The largest improvement expected is the capability to perform meaningful requirement testing during the descending (design) part of the “V” Cycle. A continuous validation along the design steps should allow checking the behaviour of the product regarding the functional requirements but also the non-functional ones. In other words, it would mean specifying timing constraints, time tolerances and scheduling to characterize the expected behaviour. It is also necessary to specify the platforms behaviour (Hardware + low level software) to take into account performance constraints induced by safety execution mode (MPC, redundancy).

The behaviour monitoring at each step of the design would allow for optimizing convergence lead time.

The test framework should allow the definition of test objectives and their allocation to design items from models or from existing software components (source code).

2.4 Current Practices in the Automotive Domain

2.4.1 Current Development Process at Bosch

As depicted in Figure 7, current design processes can be divided in sequential design steps. Starting with a first idea of a new product or even the next generation of an existing one concrete requirements have to be acquired. This can be based on requirements of existing products and available experiences. The abstraction level of the requirements vary from high level demands (e.g. if the brake pedal is pressed, the car has to slow down) to very detailed specifications (e.g. the voltage on pin 23 has to be in the range of 2.7 and 3.6 mV).

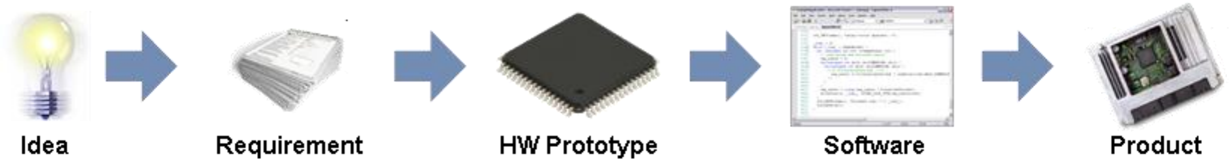


Figure 7: Design flow in the automotive domain

Current design flows are mainly sequential and do not allow parallelization of design steps. After the formulation of the requirements, a first hardware prototype can be implemented to allow further design steps. The step to this prototype includes a very high implementation effort of the digital and analog hardware components. This also includes the simulation and verification of these components or even small parts of the system. The interaction of the components and the correct interaction with its environment can be investigated not until the first prototype. Therefore, the implementation of the necessary software starts with the availability of an adequate prototype. Due to the fact that some design limitations are become visible not until the software implementation, there is usually more than one prototype. However, after the final integration and several extensive test and verification steps the final product can be transferred to the production lines.

2.4.2 Expected Improvements

The largest improvement expected is the parallelization of hardware and software development. The parallel execution of design steps reduces time to market, improves quality, avoids expensive redesigns, and enables global optimizations.

Another expected improvement is the reduced Verification and integration effort at the end of the design process. This should be reached by a continuous refinement enabled through a living exchange between the parallel design steps using (executable) models.

Using executable models also substantiates assertions about functionality and performance of the entire system in early design phases. The application of the rule-of-thumb can be reduced and decisions become more and more comprehensible.

3. Modelling Standards

The VERDE methodology is based on modelling standards, mainly UML2 and its extensions. The semantic alignment of UML extensions will be done in Task 2.4 and the restriction to those modelling practices necessary in the context of Verde will be based on these results. Nevertheless, this chapter gives a brief overview of the standards that are used.

3.1 UML

The UML (Unified Modelling Language) defines a general purpose modelling language standardized by the Object Management Group (OMG). UML provides notations to model static and dynamic aspects of any application field (application structure, behaviour, architecture) and supports behaviour diagram (activity diagram, sequence diagram, state machine diagram, use case diagram) and structure diagram types. It allows also to model business process and data structure the application area is not restricted on Software Development.

Standard UML can be extended by defining stereotypes, tag definitions, and constraints that are applied to specific model elements, such as Components, Operations, Activities and States. Extension mechanisms allow refining standard semantics in an additive manner, so that they do not contradict standard semantics.

A Profile is a collection of such extensions that collectively customize UML for a particular domain. In the following chapters several profiles used in the context of VERDE are described.

Specification: http://www.uml.org/

3.2 SysML

The OMG Systems Modelling Language (OMG SysML™) is a general-purpose graphical modelling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities. In particular, the language provides graphical representations with a semantic foundation for modelling system requirements, behaviour, structure, and parametrics, which is used to integrate with other engineering analysis models. SysML represents a subset of UML 2 with extensions needed to satisfy the requirements of the UML™ for Systems Engineering RFP. SysML leverages the OMG XML Metadata Interchange (XMI®) to exchange modelling data between tools, and is also intended to be compatible with the evolving ISO 10303-233 systems engineering data interchange standard.

Specification: The formal public version of the OMG SysML™ v1.1 was published by the OMG as an "Available Specification" in December 2008. The OMG document numbers are formal/2008-11-01 (with change bars) and formal/2008-11-02 (without change bars). All files for the SysML 1.1 specification are linked from the specification page at http://www.omg.org/spec/SysML/1.1/.

3.3 MARTE

MARTE is a profile for the UML2 language dedicated to the definition of real-time embedded systems. It consists of a set of sub-profiles dedicated for different aspects, globally divided into foundations, design, analysis and annexes. The foundation part defines general concepts such as non-functional properties and time. The modelling part contains general concepts, such as the possible communication mechanisms between components or high level modelling of time properties of service invocations. It also defines sets of stereotypes to model all the entities that are involved in a real-time embedded architecture (execution resources, computation nodes, timers, data types, etc.). These stereotypes can be characterized with non functional properties such as bandwidth/jitter of busses, periods of tasks, size of memories, etc. These properties can be processed to configure applications or perform analysis. The main advantage of MARTE is to provide a standardized way of describing all this information, which can therefore be shared between tools.

Specification: http://www.omgmarte.org/

3.4 UML Testing Profile

The UML Testing Profile is a graphical modelling language for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts of test systems.

This profile is based upon UML 2.0 and is divided in three sub-packages:

- test behaviour, which addresses the observations and activities during a test
- test architecture, containing the elements and their relationships involved in a test
- test data, structures and of values to be processed in a test.

The system under test (SUT) is not specified as part of the test model. In order to run black box tests, the architecture package imports the complete design (UML) model of the SUT to get access to the elements to be tested. The SUT can be exercised via its public interface operations and signals by test components.

Specification: http://www.omg.org/technology/documents/formal/test_profile.htm

3.5 Object Constraint Language

The Object Constraint Language (OCL) is a formal language to describe expressions on UML models. OCL is an extension to UML and allows to be more precise in System or Software Models. While UML focuses on structures and relationships between objects, OCL can be used to define additional conditions on model element.

OCL expressions can be used to specify invariant conditions in class diagrams, conditions in Sequence diagrams or pre- and post conditions for Methods. If these expressions are evaluated, they do not have side effects, i.e., their evaluation do not alter the state of the corresponding executing system.

Specification: http://www.omg.org/spec/OCL/2.2/

3.6 SPEM

SPEM (System and Software Process Engineering Metamodel) provides graphical notations to capture system and software processes. It has been standardized by the Object Management Group as a metamodel as well as a UML profile. The core principle of this language is that any process is ruled by collaboration between abstract entities (roles) performing operations (activities) on concrete entities (work products) (Figure 8).

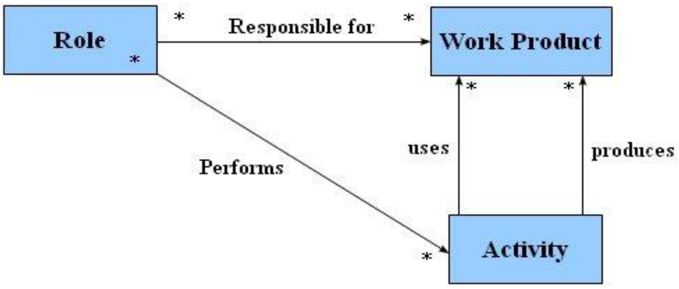


Figure 8: SPEM basics

Like the norm ISO-12207, from which it is inspired, this language can be used to describe a wide range of processes. The main use cases of this language are:

- Support for management of reusable libraries of methodological patterns,
- Support for application of methodological patterns into specific processes (including possible adaptations),

- Support for configuration of methodology and process patterns,
- Support for process enactment.

One major feature of SPEM 2.0 is a clear separation between method and process descriptions.

Specification: http://www.omg.org/spec/SPEM/2.0/

3.7 QVT

The QVT (Query/View/Transformation) standard provides declarative and imperative syntaxes to specify model-to-model transformations. Thus, it is an important component of the Model-Based Engineering approach. It allows the specification of transformation chain that capitalizes part of the know-how needed to transform a model into another one applying transformation rules.

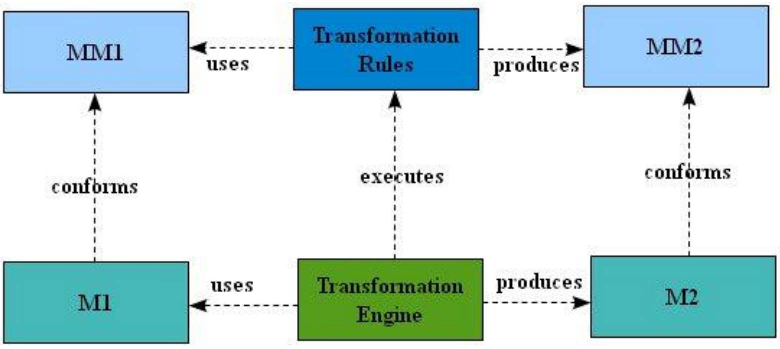


Figure 9: QVT basics

The Figure 9 sketches a simplified QVT model: a transformation is performed by a transformation engine which processes a source model M1, conforming to a meta-model MM1, and produces a model M2 conforming to a meta-model MM2. Transformations are specified by transformation rules that establish relationships between elements of meta-models MM1 and MM2. The transformation is said « endogenous » when MM1 and MM2 are the same and «exogenous» otherwise.

Implementations: Eclipse M2M, SmartQVT, ATL, Borland Together.

Specification: http://www.omg.org/spec/QVT/1.0/

3.8 MOF-M2T

MOF-M2T (Model-to-Text) provides a template language to specify model-to-text transformations. Like QVT, it is thus an important component of the Model-Based Engineering approach as it allows the production of any textual artefact (code, documentation, etc.). The Figure 10 sketches a simplified MOF-M2T model: a generation is performed by a template engine which processes a model M conforming to a meta-model MM to produce a textual artefact. The generation is specified by a template that gives production rules.

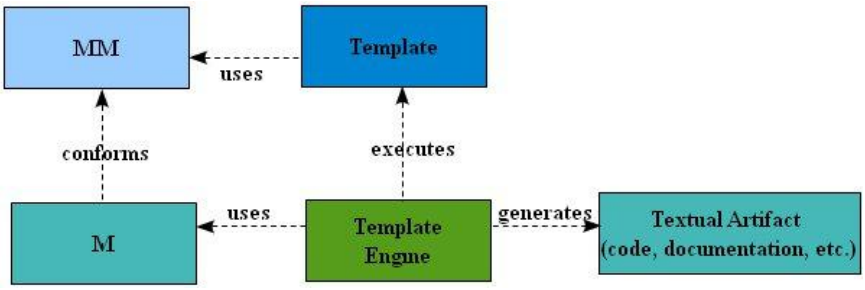


Figure 10: MOF-M2T Basics

Implementations: Acceleo, Eclipse M2T.

Specification: http://www.omg.org/spec/MOFM2T/1.0/

3.9 UML Profile for Corba and Corba Component Model

The UML profile for Corba and Corba Component provides an extension to UML allowing the specification of Corba Components and their required features. This profile represents actually a projection of CCM concepts into UML, allowing the usage of UML syntactic facilities to capture any CCM architecture. Then, from such UML models, all CCM artifacts (idl interfaces, implementations, configuration files, etc.) can be generated thanks to model-to-text transformations.

Implementation: Papyrus

Specification: http://www.omg.org/spec/CCCMP/1.0/PDF/

3.10 AUTOSAR

AUTOSAR is an international development partnership consisting of a multitude of car manufacturers, suppliers and tool vendors, which define concepts and workflows, how electronic automotive software-related systems can be formally specified and processed. AUTOSAR focuses on a software architecture that decouples application software and hardware by offering a runtime environment and a basic software layer. The application software is implemented within software components. These software components communicate via well defined interfaces. The goal is to make the application software completely independent from the underlying hardware architecture to allow an arbitrary distribution onto different ECUs (Electronic Control Units). Configuration and generation processes build the final ECU software.

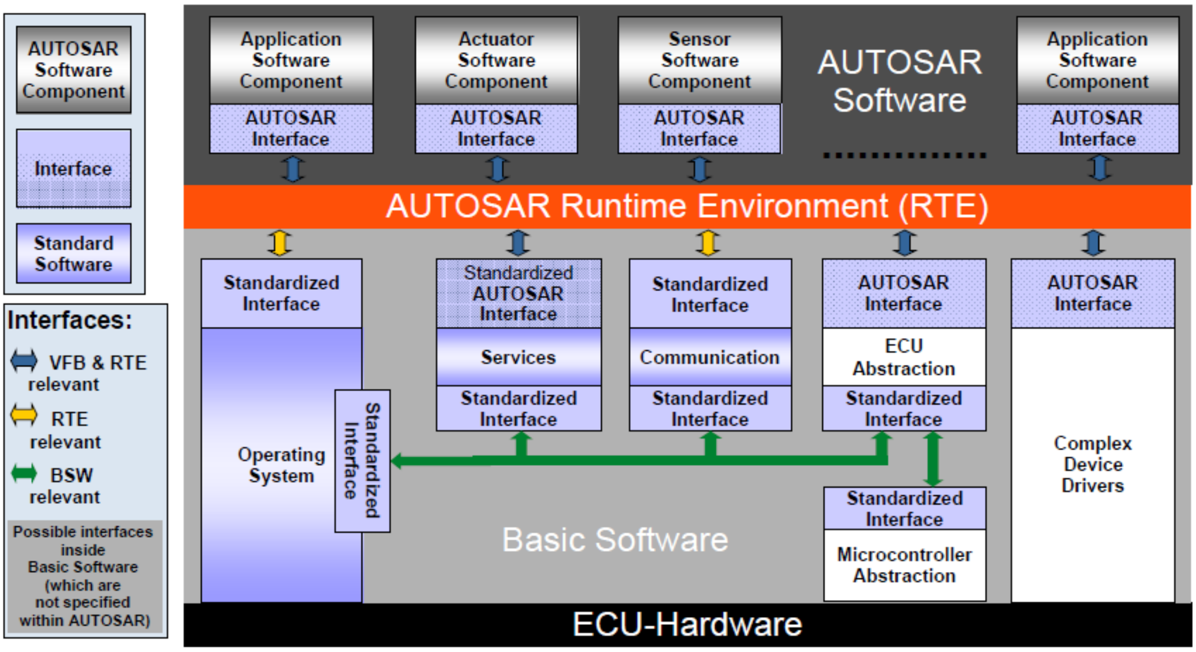


Figure 11: AUTOSAR System Architecture Overview

In contrast to the current state-of-the-art development approach, which is ECU-centric, AUTOSAR focuses on the entire system. As illustrated in Figure 11 one fundamental feature is the separation of application and infrastructure which allows for a model-driven architecture like methodology, i.e. a platform independent software development of functionality. Applications can exist and communicate independently of a particular infrastructure and mapping onto ECUs in an environment called Virtual Functional Bus (VFB). Furthermore, AUTOSAR comprises even more: it specifies methodologies and workflows on how to come from the system living in the VFB to software running on particular ECUs as part of a multilayered ECU architecture.

An AUTOSAR conform architecture consists of an application layer (called AUTOSAR Software), a middleware layer (called Runtime Environment -- RTE), and the infrastructure layer (called Basic Software --

BSW). Assuming that the components of the application layer behave exactly the same way like in the VFB, the RTE and BSW implement the VFB for a particular ECU.

Properties of AUTOSAR applications are described with a specific language, called AUTOSAR Software Component Template (as part of the entire AUTOSAR meta-model). In general, the AUTOSAR Software component template is arranged into three parts regarding the structure, the behavior and the implementation of models.

3.11 **SystemC**Fehler! Hyperlink-Referenz ungültig.

SystemC is a C++library extending the pure software specification capabilities of C++ by hardware and system specification features. The extension consists of a simulation kernel, which enables a pseudo-parallel execution of inherently parallel hardware and system models. The library introduces constructs for parallelism and concurrency, model topology, hardware and system-level communication, and hardware-related data types. The SystemC specification library is standardized in IEEE 1666. Although SystemC covers a full-featured RTL and hardware modelling methodology - including hardware synthesis and analog-mixed-signal modelling - the focus of the language lays on system specification on higher levels of abstraction (transaction-level modelling, TLM). SystemC fits into all modern sophisticated design flows, which handle the increasing system complexity by introducing higher levels of abstraction together with refinement and mapping paths for the system implementation. SystemC allows powerful simulations for algorithm validation, model verification, system-level performance analysis, and software development on virtual hardware and system platforms.

Transaction-level modelling (TLM) is the key technology to raise the abstraction levels for the modelling of complex embedded systems and cyber physical systems. SystemC 2.x incorporates such a TLM mechanism enabling signal-based low-level modelling on RTL as well as high-level modelling on higher levels of abstraction like system level. SystemC 2.0 introduces 5 abstraction levels: Algorithmic level (AL), communicating processes (CP), programmer’s view (PV), programmer’s view timed (PVT), and cycle callable (CC). Within this set of levels, CP, PV, and PVT form the core TLM levels. On one hand side, TLM allows a clear separation of communication and functionality, which enables an easy replacement and reuse of components and modules, and on the other side, TLM allows the abstraction of the communication itself. This leads to a separation of the communication process and its implementation. Models can now express complex communication scenarios without any need to specify the implementation of the communication. This especially avoids the disintegration of complex data types into single bits or bytes, which would be necessary for a hardware implementation. As a consequence a TLM specification has to deal with complex user-defined or application-specific data structures. These data structures are far more complex than single bits, especially in terms of their relation to system or simulation time. In SystemC the communication process itself will be modeled as one single function call or a sequence of function calls, depending on the level of abstraction, e.g. programmer’s view or programmer’s view timed or depending on the TLM (TLM-2.0) modelling style like loosely or accurately timed.

4. Existing Methodologies

This chapter gives an overview of existing works and previous EU projects deliverables. These results are a good input and a good starting point for further work.

4.1 Modelplex

The European Project Modelplex (Modelling Solutions for Complex Software Systems) funded partially by the European Commission in the 6th Framework (Contract n° 034081) was mostly concerned with the industrialization of the model-driven development. Although the project aimed at complex system engineering, some of the approaches and finding of this project are also reference and input for VERDE.

4.1.1 Development Environment

One of the key aspects of Modelplex was the creation of a development environment which allows the application of various development methodologies for all four industrial case studies in Modelplex in a similar way. This development environment is called Modelplex workbench and it was created by strongly focusing on models as key aspect of a development process. Figure 12 shows the general architecture of the Modelplex Workbench.

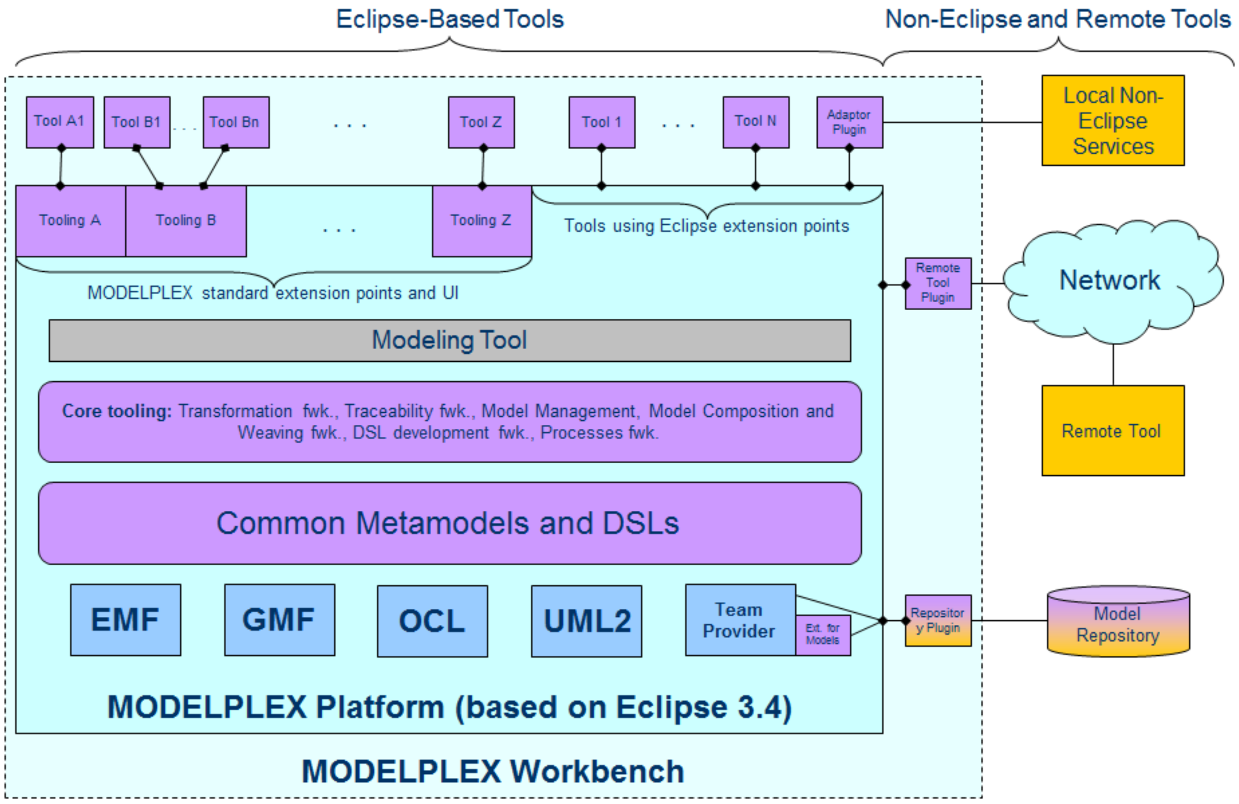

Eclipse-Based Tools
Non-Eclipse and Remote Tools
Tool A1
Tool B1
Tool Bn
. . .
Tool Z
Tool 1
. . .
Tool N
Adaptor Plugin
Local Non-Eclipse Services
Tooling A
Tooling B
. . .
Tooling Z
Tools using Eclipse extension points
MODELPLEX standard extension points and UI
Remote Tool Plugin
Network
Modeling Tool
Core tooling: Transformation fwk., Traceability fwk., Model Management, Model Composition and Weaving fwk., DSL development fwk., Processes fwk.
Remote Tool
Common Metamodels and DSLs
EMF
GMF
OCL
UML2
Team Provider
Ext. for Models
Repository Plugin
Model Repository
MODELPLEX Platform (based on Eclipse 3.4)
MODELPLEX Workbench

Figure 12: General Architecture of Modelplex Workbench

The Workbench in itself is very much Eclipse-centric for two important reasons. Firstly, Modelplex developed a couple of tools as Open Source and within a limited time frame, so the benefit of having Eclipse as a tool framework which allows the rapid development of tools was well appreciated. Secondly, Eclipse with its various model-driven technologies and frameworks allow creating a homogeneous model-driven development environment.

Nevertheless, it has been identified that there is still a strong need for the integration of non-Eclipse based tools in order to realize a real industrial development environment.

The idea of the Modelplex Workbench is to have a number of potentially integrated tools co-located with a number of other technologies and techniques. This set of technical assets can be used for instantiating the

Modelplex Workbench for a specific development process. This instantiation means basically to select which of the potentially available technical assets needs to be part of the workbench instance.

4.1.2 Key aspects of Modelplex Solution

Taking the idea of the Modeplex Workbench, the industrial partners used a customized version of it for conducting their case studies. This means that each industrial partner defined a couple of development process steps and the corresponding methods, tools and technologies. However, it has been turned out that there was a strong need for the automation of development steps and for combined persistency of models and other development artifacts such as source code. Both issues are considered as key elements in order to cope with the complexity of a development process for complex software systems.

4.1.3 Development Methodologies in Modelplex

Industrial partners in Modelplex have used a number of methods and techniques in order to realize the development process of their case studies. The following list highlights most of them:

- Guideline checking and constraint modelling: Using guidelines and constraints (based on constraint languages like EVL or OCL) in order to ensure certain properties of work products before further processing
- Test case generation: Various strategies have been used in order to generate test cases out of systems and test models
- Automatic test execution: Automatic execution of test cases (generated or manually created) and the presentation of test results
- Verification: Verification of certain (system) model properties (static and dynamic ones) in order to ensure software quality
- Model transformation: The transformation of models to other models. The mapping between elements of these models can be described by specialized languages or by using general purpose programming languages
- Code generation: A special kind of model transformation, which targets the creation of source code
- Model traceability: Follow links between work products of development process in order to analyze impact of changes
- Model composition: Compose work products coming from different process steps in order to create a new (composed and integrated) work product
- Model based performance analysis: The analysis of behaviour models in order to get performance indicators prior to the deployment of a system
- Domain specific modelling: Creation of specific languages for certain aspects of development process
- General purpose languages and customizations: The usage of UML and UML Profiles (e.g. MARTE)
- Architectural refinements: Definition of systems at various levels of granularity
- Viewpoints: Definition of system properties at various viewpoints
- Model debugging: Follow the flow of execution on behaviour models
- Orchestration: Definition and automated execution of certain process steps in the development process
- Process enactment: Execution of a formally defined development process
- Knowledge discovery: The presentation of knowledge (information) which is only implicitly present in a system or a development process

4.2 D-Mint

The ITEA project D-Mint (Deployment of Model-Based Technologies to Industrial Testing) targets the development, enhancement, and deployment of high performance methods and tools for quality assurance of large and software-intensive systems. The project was focused on the testing part of a development process.

The D-Mint results are collected, summarized and presented in the D-Mint Asset Box.

4.2.1 General Model-Based Testing methodology

D-Mint has promoted the usage of the model-based testing approach in order to achieve high quality and high performance in the work products of the testing process steps. The Figure 13 outlines the general idea of the model-based testing approach.

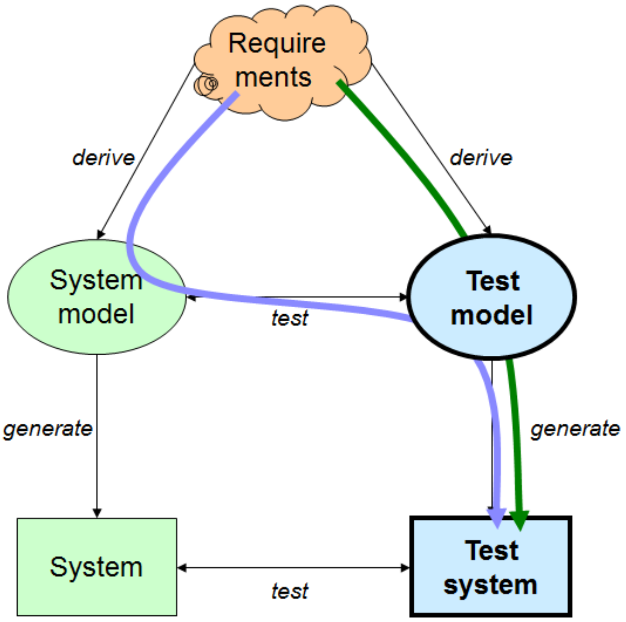


Figure 13: Model-Based Testing

Similar to model-based system development the key concept of model-based testing (MBT) is to generate the test cases out of a test model. Although there are a number of issues to be addressed during the test case generation (e.g. what is the test execution environment) the key question of MBT is: How to create the test model.

The test model may be derived from the requirements (similar to the system model but with a different purpose) or it can be partially derived from system model. Both ways do have pros and cons.

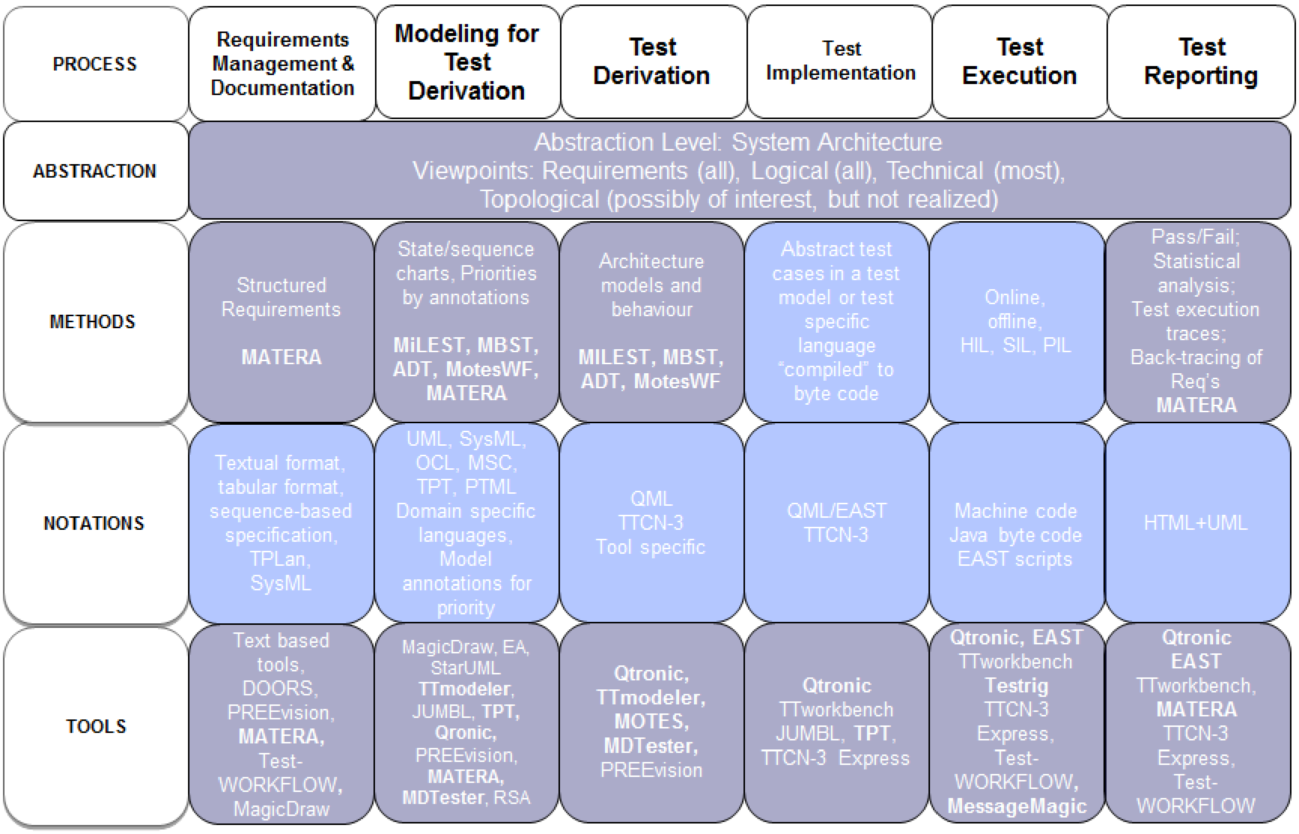
4.2.2 The D-Mint Common Approach (Asset Box)

The results coming from D-Mint as well as results coming from other sources are summarized in the D-Mint Common Approach which constitutes an overall D-Mint Methodology and builds a kind of Asset Box at the same time. This Asset Box can be used in order to approach the domain of model-based testing on a practical level. The Asset Box offers methods and tools which can be chosen in order to realize the different test related process steps. Figure 14 outlines the D-Mint Asset Box.

Further details on the D-Mint Common Approach are described in the D-Mint White Paper (http://www.d-mint.org/public/CommonApproach_WhitePaper_DMINT.pdf). However, there are a couple of innovations which can be listed here explicitly:

- Architecture-Driven Testing: A particular test derivation strategy which takes the different architectural viewpoints into account and which focuses on integration specific faults
- Pattern-oriented model-driven test engineering: The exploitation of a pattern approach in order to facilitate the engineering of test models
- Test Management: Integration of MBT tools and methods in state of the art test of management tools

- Test Quality: The assessment of the test models in the context of specific goals, guidelines and regulations
- Test Process Evaluation: The assessment and metrication of the test process as such



PROCESS	Requirements Management & Documentation	Modeling for Test Derivation	Test Derivation	Test Implementation	Test Execution	Test Reporting
ABSTRACTION	Abstraction Level: System Architecture Viewpoints: Requirements (all), Logical (all), Technical (most), Topological (possibly of interest, but not realized)					
METHODS	Structured Requirements **MATERA**	State/sequence charts, Priorities by annotations **MiLEST, MBST, ADT, MotesWF, MATERA**	Architecture models and behaviour **MILEST, MBST, ADT, MotesWF**	Abstract test cases in a test model or test specific language "compiled" to byte code	Online, offline, HIL, SIL, PIL	Pass/Fail; Statistical analysis; Test execution traces; Back-tracing of Req's **MATERA**
NOTATIONS	Textual format, tabular format, sequence-based specification, TPLan, SysML	UML, SysML, OCL, MSC, TPT, PTML Domain specific languages, Model annotations for priority	QML TTCN-3 Tool specific	QML/EAST TTCN-3	Machine code Java byte code EAST scripts	HTML+UML
TOOLS	Text based tools, DOORS, PREEvision, **MATERA,** Test-WORKFLOW, MagicDraw	MagicDraw, EA, StarUML **TTmodeler,** JUMBL, **TPT, Qronic,** PREEvision, **MATERA, MDTester**, RSA	**Qtronic, TTmodeler, MOTES, MDTester,** PREEvision	**Qtronic** TTworkbench JUMBL, **TPT**, TTCN-3 Express	**Qtronic, EAST** TTworkbench **Testrig** TTCN-3 Express, Test-WORKFLOW, **MessageMagic**	**Qtronic EAST** TTworkbench, **MATERA** TTCN-3 Express, Test-WORKFLOW

Figure 14: D-Mint Common Approach (Asset Box)

4.3 MARTES

MARTES stands for Model-Based Approach for Real-time Embedded Systems. It is an EUREKA-ITEA project aiming to provide « the definition, construction, experimentation, validation and deployment of a new model-based methodology and an interoperable toolset for Real-Time Embedded Systems development, and the application of these concepts to create a development and validation platform for the domain of embedded applications on heterogeneous platforms architectures ».

The Figure 15 gives an overview of the MARTES methodology, which is based on a ah-doc UML profile which extends UML with new capabilities to model Hardware and Software platforms as well as Non-functional properties and allocations.

Links: http://www.martes-itea.org/public/news.php

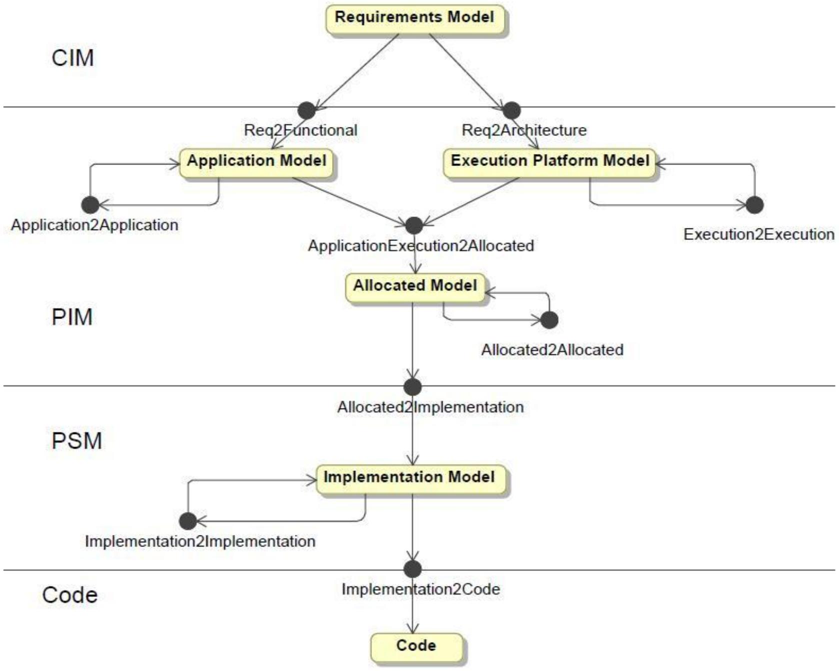


Figure 15: MARTES Methodology Overview

4.4 MoPCoM

MoPCoM stands for Modelling and specialization of platform and components MDA. It is a French ANR project that aims to help designers to deal with high complexity systems by designing their developments using modelling approaches for their applications and platforms. To provide efficient solution, the goal of the MOPCOM project is also to target modelling techniques driven by application domains in order to provide specific design guides and rules. Refinement techniques are essential to promote these approaches as they help designers along the design process.

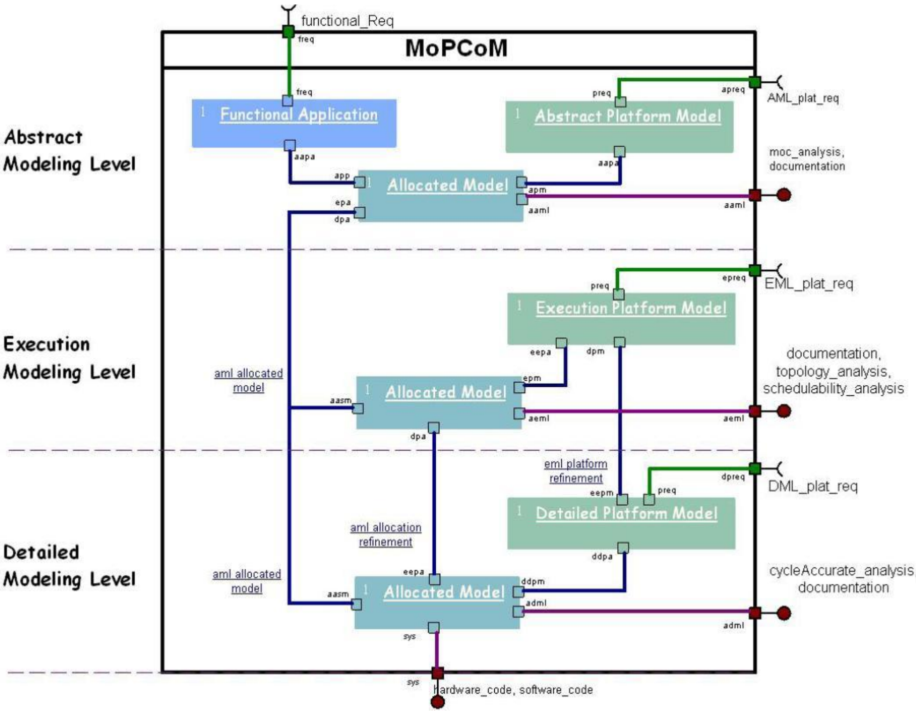


Figure 16: MoPCoM Methodology Overview

The Figure 16 gives an overview of the MoPCoM methodology as a set of interconnected process components. Briefly, the MoPCoM methodology is split into 3 levels of abstraction, each dedicated to a family of analysis:

- The Abstract Modelling Level (AML) is intended to provide the description of the expected level of concurrency and pipeline through the mapping of functional blocks onto a virtual execution platform,
- The Execution Modelling Level (EML) is intended to provide a generic platform defined in term of execution, communication or storage nodes in order to proceed to coarse grain analysis,
- The Detailed Modelling Level (DML) is intended to provide a detailed description of the platform in order to proceed to fine grained analysis. It allows RTL code generation for hardware (VHDL) and software (C) parts including glue logic (drivers).

Regarding the MARTES methodology, the MoPCoM methodology has introduced an essential level of abstraction focusing on Models of Computation and Communication (MoCC) related issues.

Links: http://www.mopcom.fr/doku.php

5. Methodological Patterns

Analyzing the current practices in the various domains addressed in the context of VERDE, it is pointed out that all industrial partners have well established development processes in place, which are mostly variations of the V Model.

On one hand, the used processes provide some similarities for activities like requirements capturing and modelling of a system architecture. On the other hand, they are also specialized for a certain domain, especially for activities like verification and testing with a clear request for improvements here. Therefore, it is not the intention within the VERDE project to define a unified process for all domains. Nevertheless VERDE describes generic spiral process as shown in the following graphic.

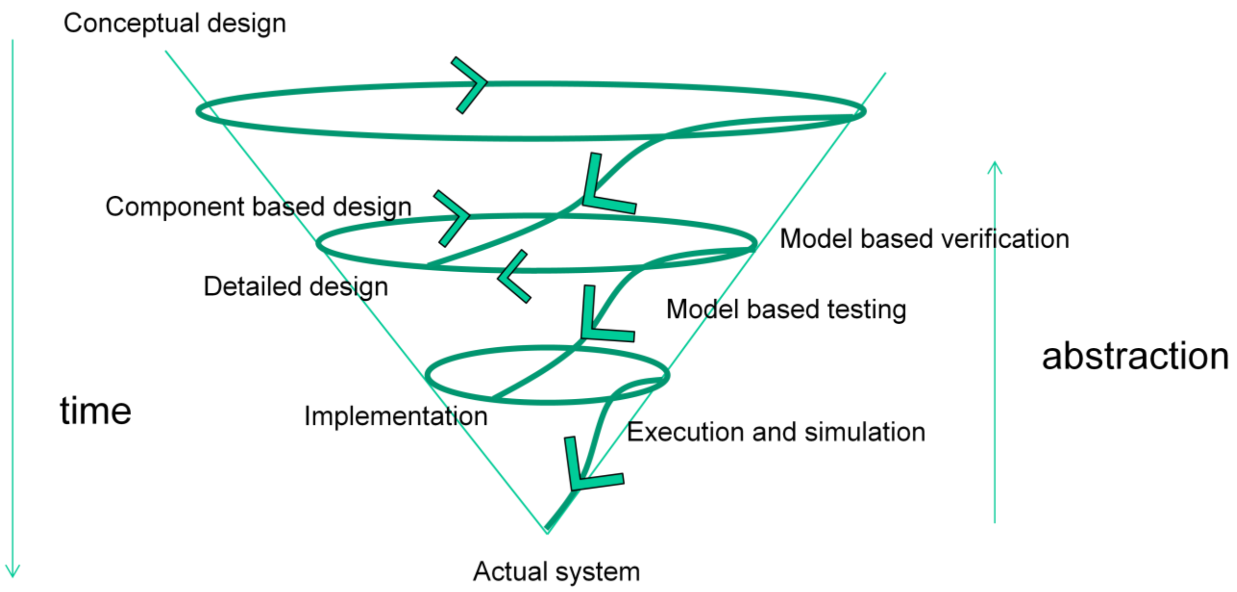


Figure 17: VERDE Spiral Process

This process consists of common activities or phases well known from standard processes and also described by the industrial partners in the current practices chapter of this document. The clear focus in the context of VERDE is Model based testing and verification. Therefore more detailed steps are defined for the activities, to describe what is needed to get there and to address the requirement tickets defined by industrial partners.

For each step a set of methodological patterns is described. Patterns provide practical and “easy to put in practice” modelling solutions for concrete modelling issues, that can be selected by end users when needed and integrated in domain specific processes.

5.1 Conceptual Design

Projects mostly start with a conceptual design phase, where requirements and a first architectural breakdown are defined.

5.1.1 Definition of Requirements

5.1.1.1 Requirements Configuration

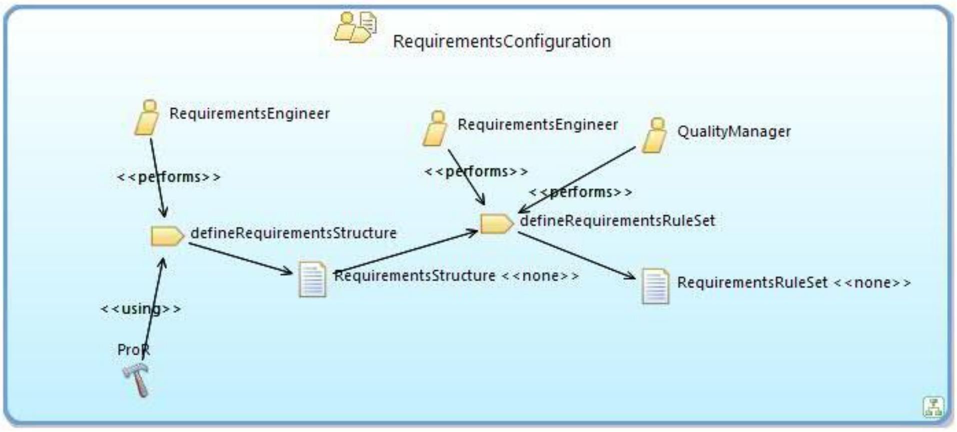
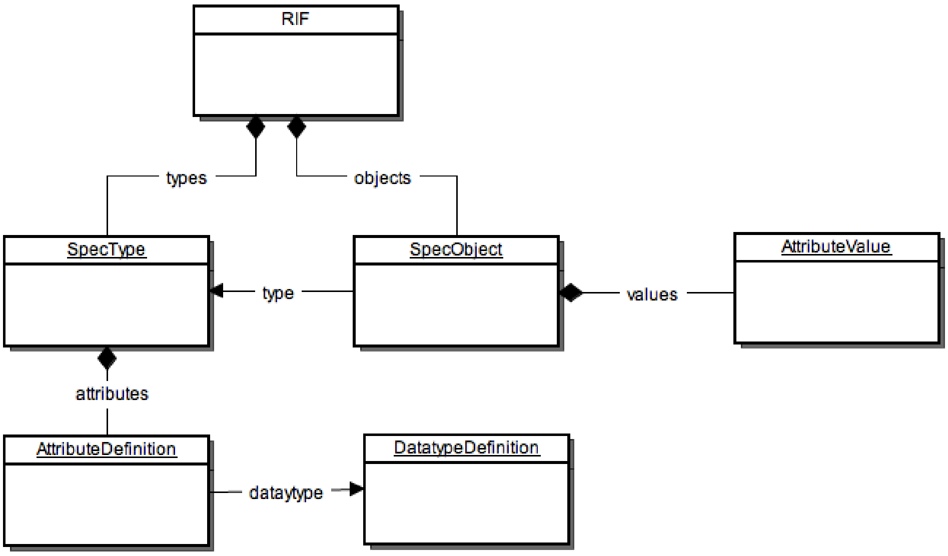
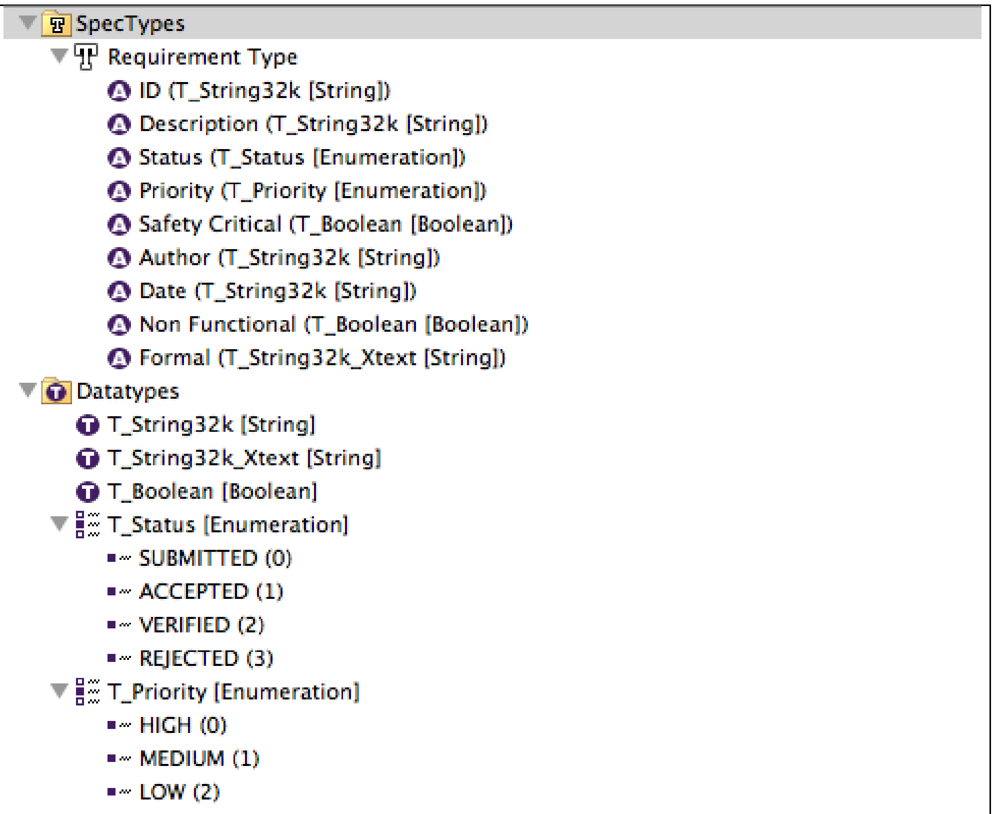

RequirementsConfiguration
RequirementsEngineer
<<performs>>
defineRequirementsStructure
<<using>>
ProR
RequirementsStructure <<none>>
RequirementsEngineer
<<performs>>
QualityManager
<<performs>>
defineRequirementsRuleSet
RequirementsRuleSet <<none>>

Figure 18: Requirements Configuration

Requirements Configuration	
Overview	In order to maintain requirements in a structured way, the structure must be defined initially. In addition, rules based on this structure will be defined to do automated analysis of the requirements.
Assigned Tickets	#33, #76, #93, #117, #242, #237
Language	RIF/ReqIF, SMM, OCL
Tools	ProR,Xtext,Metrino
Parameters	Requirements Structure, out Requirements Rule Set, out
Pre-Conditions	Requirements Engineer Knowledge, Quality Manager Knowledge
Post-Conditions	
Steps	**Define Requirements Structure:** The requirements will be managed using the RIF/ReqIF (Requirements Interchange Format) standard. The RIF/ReqIF standard defines a meta-model including an internal DSL to define the concrete structure of the requirements. The following picture shows the main concepts of the meta-model:

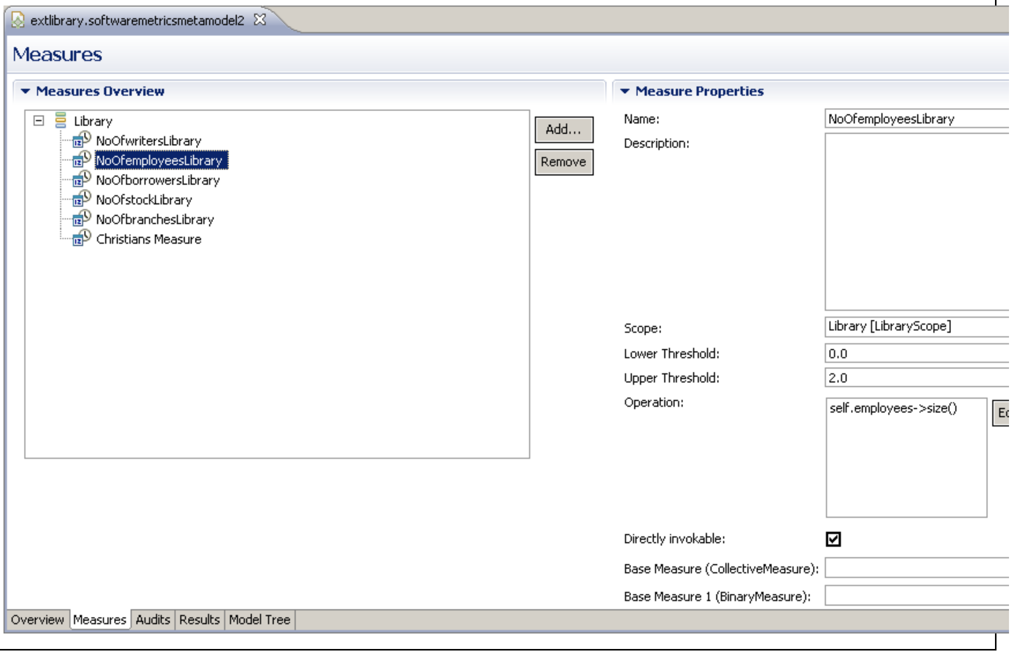
	 RIF types objects SpecType type SpecObject values AttributeValue attributes AttributeDefinition datatype DatatypeDefinition The required spec types, their attribute definitions, and the used datatype definitions must be defined in this step. While defining datatype definitions theXtext framework can be used to provide textual modelling for more formal specifications. Please look into the user guide of ProR to get detailed instructions about the tool usage. **Define Requirements Rule Set:** The rule set that is used for the checking of consistency and well-formedness of requirements models is defined by using the Structured Metrics Metamodel (SMM). The SMM defines the way how to organize such rule sets. Each particular rule is linked to a specific evaluation action which can be expressed by using formal languages like OCL. The set of rules consist of more general ones which target on the general soundness and more specific ones which focuses more on the specific project. (e.g. description field of a requirements shall not be shorter than 20 characters)
Example	**Define Requirements Structure:** Below is a screenshot from the ProR configuration view showing a possible configuration including one spec type with a few attributes and their data types.



- SpecTypes
  - Requirement Type
    - ID (T_String32k [String])
    - Description (T_String32k [String])
    - Status (T_Status [Enumeration])
    - Priority (T_Priority [Enumeration])
    - Safety Critical (T_Boolean [Boolean])
    - Author (T_String32k [String])
    - Date (T_String32k [String])
    - Non Functional (T_Boolean [Boolean])
    - Formal (T_String32k_Xtext [String])
- Datatypes
  - T_String32k [String]
  - T_String32k_Xtext [String]
  - T_Boolean [Boolean]
  - T_Status [Enumeration]
    - SUBMITTED (0)
    - ACCEPTED (1)
    - VERIFIED (2)
    - REJECTED (3)
  - T_Priority [Enumeration]
    - HIGH (0)
    - MEDIUM (1)
    - LOW (2)

Define Requirements Rule Set:

The following screen shot shows the Metrino rule editor, which allows the definition of rules with the help of OCL. Each rule (Measure) consists of a number of attributes, like name, scope, and thresholds.



Name: NoOfemployeesLibrary

Description:

Scope: Library [LibraryScope]

Lower Threshold: 0.0

Upper Threshold: 2.0

Operation: self.employees->size()

Directly invokable: ☑

Base Measure (CollectiveMeasure):

Base Measure 1 (BinaryMeasure):

5.1.1.2 Requirements Engineering

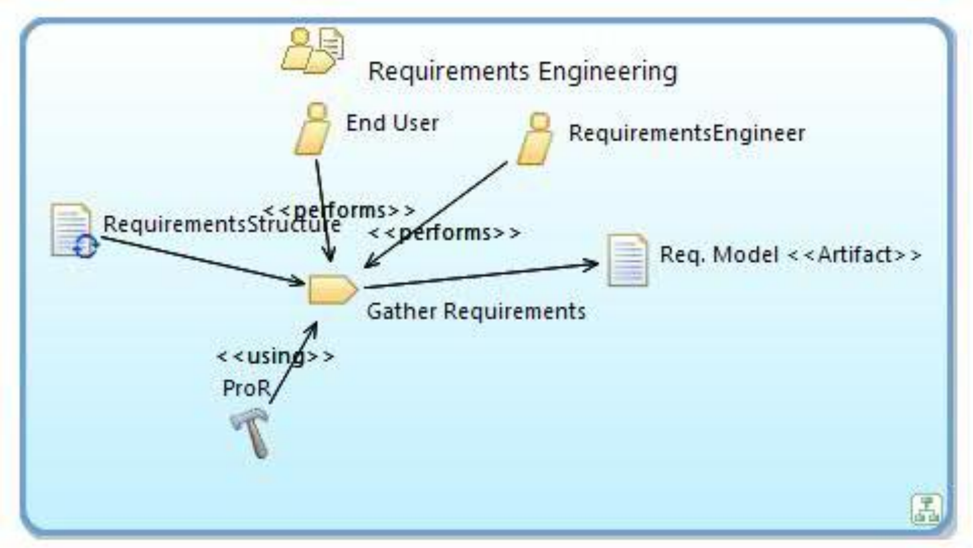
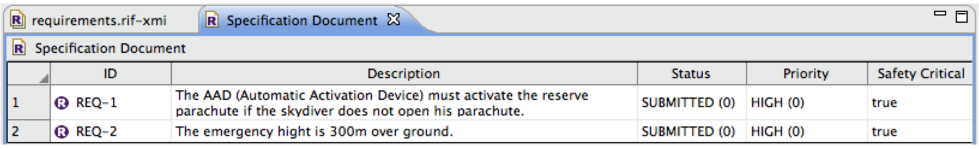


Figure 19: Requirements Engineering

Requirements Engineering	
Overview	Requirements Engineering is typically the initial activity in a development process.
Assigned Tickets	#33, #76, #93, #117, #242, #237
Language	RIF/ReqIF (Requirements Interchange Format)
Tools	ProR
Parameters	Requirements Structure, in Requirements Model, out
Pre-Conditions	End Users Knowledge
Post-Conditions	
Steps	**Gather Requirements:** The requirements for a product, for a process, or for persons involved in a process are gathered here or it could be that a customer can deliver the requirements, which will be imported and maybe extended. Consider that there can be attributes in the requirements structure that can have an impact on the process while processing requirements. For example, an attribute "status" could require a validation of the input by a second person. Please look into the user guide of ProR to get detailed instructions about the tool usage.
Example	Here is a screenshot from the ProR Specification Editor including two requirements. 

	ID	Description	Status	Priority	Safety Critical
1	REQ-1	The AAD (Automatic Activation Device) must activate the reserve parachute if the skydiver does not open his parachute.	SUBMITTED (0)	HIGH (0)	true
2	REQ-2	The emergency hight is 300m over ground.	SUBMITTED (0)	HIGH (0)	true

5.1.1.3 Requirements Consistency Checking

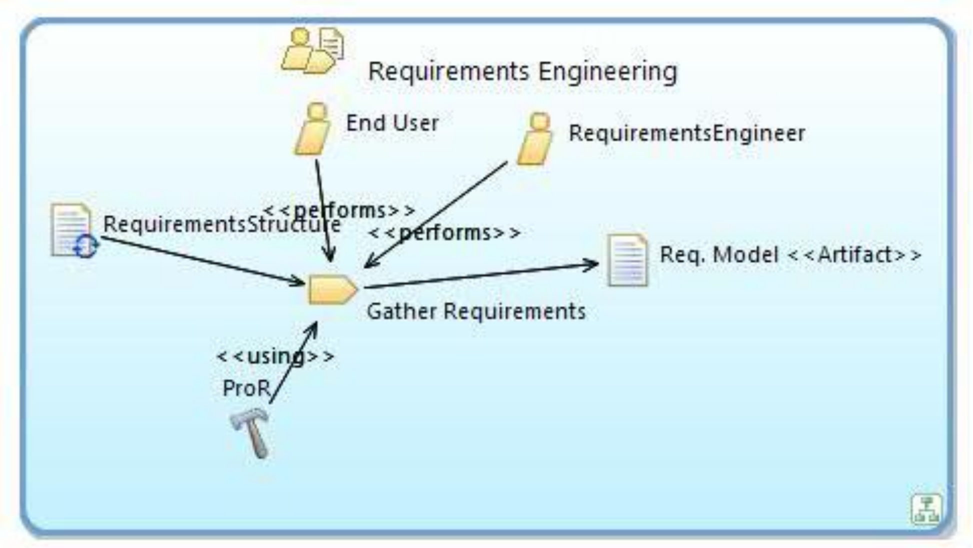
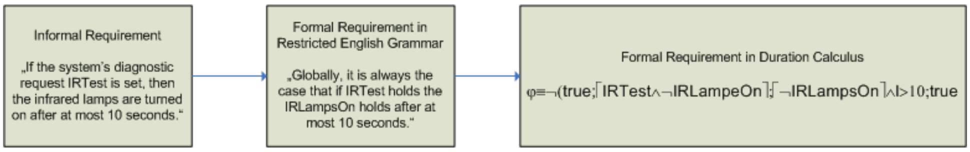


Figure 20: Requirements Consistency Checking

Requirements Consistency Checking	
Overview	Requirements Consistency Checking follows the initial requirements baseline in a development process. It may be repeated after each change, especially after adding new requirements.
Assigned Tickets	#33, #76, #93, #117, #242, #237
Language	RIF/ReqIF (Requirements Interchange Format)
Tools	RAT
Parameters	Requirements Documents, in Former formalizations of requirements documents, in Sets of inconsistent requirements, out
Pre-Conditions	End Users Knowledge
Post-Conditions	Consistent set of requirements, c.f. 5.4
Steps	**Formalize Requirements:** The requirements for a product, for a process, or for persons involved in a process have to be translated into a formal language using an intermediate step. The intermediate step involves restricted natural language that can be automatically be translated into timed automata. The sum of automata can be checked for consistency. **Analyze Requirements Formalizations:** The result of such an automatic analysis reveals that requirements are obsolete are partly redundant or that the requirements are too restrictive such that no system can be build that satisfies all of them **Reflect Analysis Results:** The results of the analysis have to be evaluated in measures have to be taken to improve the consistency. This means, that requirements have to be modified, deleted, or in some instances, that additional requirements have to be added. It is

	best practise to restart the process after any modification until no more analysis problems are indicated.
Example	Here is a screenshot how one requirement is translated into a formal notation.  Informal Requirement „If the system's diagnostic request IRTest is set, then the infrared lamps are turned on after at most 10 seconds." Formal Requirement in Restricted English Grammar „Globally, it is always the case that if IRTest holds the IRLampsOn holds after at most 10 seconds." Formal Requirement in Duration Calculus $\varphi=\neg(true;\lceil IRTest\wedge\neg IRLampeOn\rceil;\lceil\neg IRLampsOn\rceil\wedge l>10;true$

5.1.2 Traceability of Requirements

5.1.2.1 Requirements Tracing High Level Design

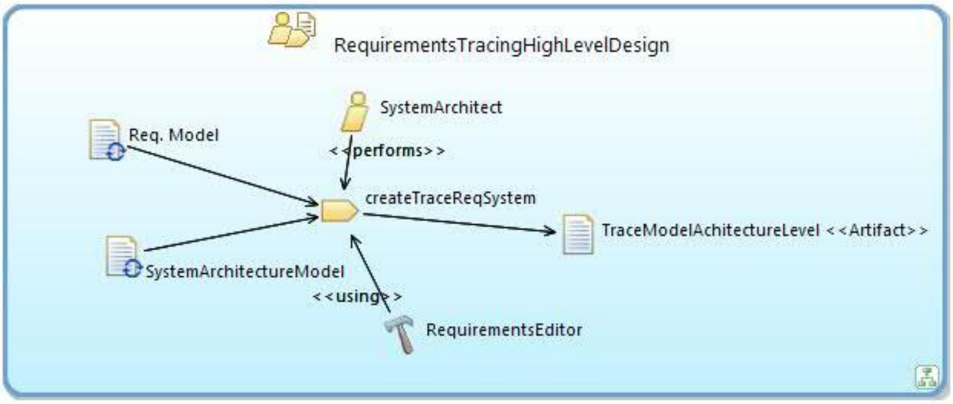
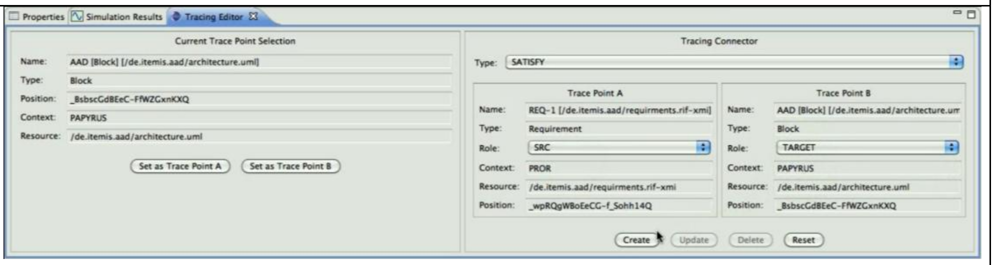


Figure 21: Requirements Tracing High Level Design

Requirements Tracing High Level Design	
Overview	Validation and verification of the development is the basis for quality assurance and requires the tracing from requirements to the implementation including tests.
Assigned Tickets	#33, #76, #113, #93, #114, #116, #237
Language	TracingMM
Tools	YakinduCReMa
Parameters	Requirements Model, in System Architecture Model, in Trace Mode, out
Pre-Conditons	
Post-Conditions	
Steps	**Create Traces:** The traces between the requirements and the system architecture elements should be created as soon as possible. This step requires disciplined work. Whenever the system architect creates a new system architecture element, he should trace it to the corresponding requirement. This ensures that the implemented system does not include unwanted features. The system architect can use customization features like filtering, sorting, and grouping on attributes defined in the tracing meta-model to analyze the traces. Additionally there is a

	synchronization support, which shows all available traces for the current selection in a supported tool. Another analyze feature is the generic reporting. Please look into the user guide of YakinduCReMa to get detailed instructions about the tool usage.
Example	 Properties Simulation Results Tracing Editor Set as Trace Point A Set as Trace Point B Create Update Delete Reset Current Trace Point Selection Name: AAD [Block] [/de.itemis.aad/architecture.uml] Type: Block Position: _BsbscGdBEeC-FfWZGxnKXQ Context: PAPYRUS Resource: /de.itemis.aad/architecture.uml Tracing Connector Type: SATISFY Trace Point A Name: REQ-1 [/de.itemis.aad/requirments.rif-xmi] Type: Requirement Role: SRC Context: PROR Resource: /de.itemis.aad/requirments.rif-xmi Position: _wpRQgWBoEeCG-f_Sohh14Q Trace Point B Name: AAD [Block] [/de.itemis.aad/architecture.um Type: Block Role: TARGET Context: PAPYRUS Resource: /de.itemis.aad/architecture.uml Position: _BsbscGdBEeC-FfWZGxnKXQ

5.1.2.2 Requirements Tracing Detailed Design

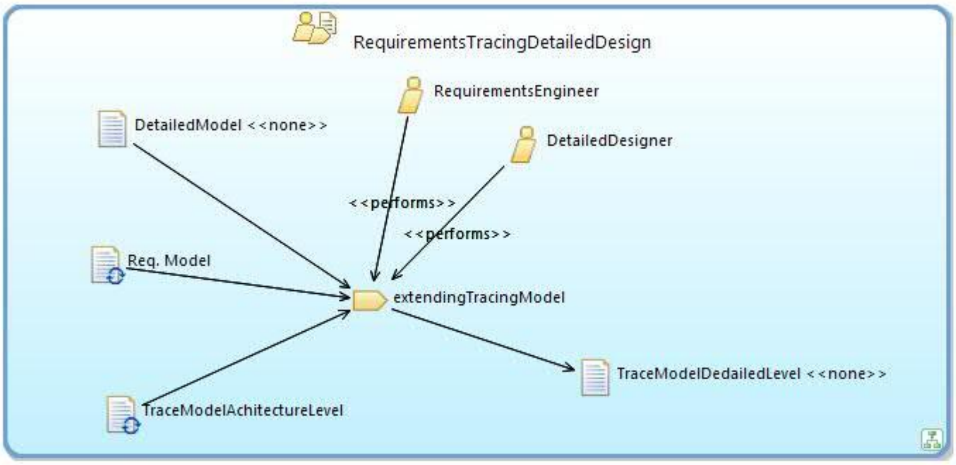


Figure 22: Requirements Tracing Detailed Design

Requirements Tracing Detailed Design	
Overview	Validation and verification of the development is the basis for quality assurance and requires the tracing from requirements to the implementation including tests.
Assigned Tickets	#33, #76, #113, #93, #114, #116, #237
Language	TracingMM
Tools	YakinduCReMa
Input	Requirements Model, in Trace Model, Detailed Model, in Trace Model, out
Pre-Conditions	
Post-Conditions	
Steps	Editing Tracing Model: The existing trace model is extended in this step to include the dependencies to the detailed design.

Example	see Requirements Tracing High Level Design

5.1.3 System/Subsystem Modelling

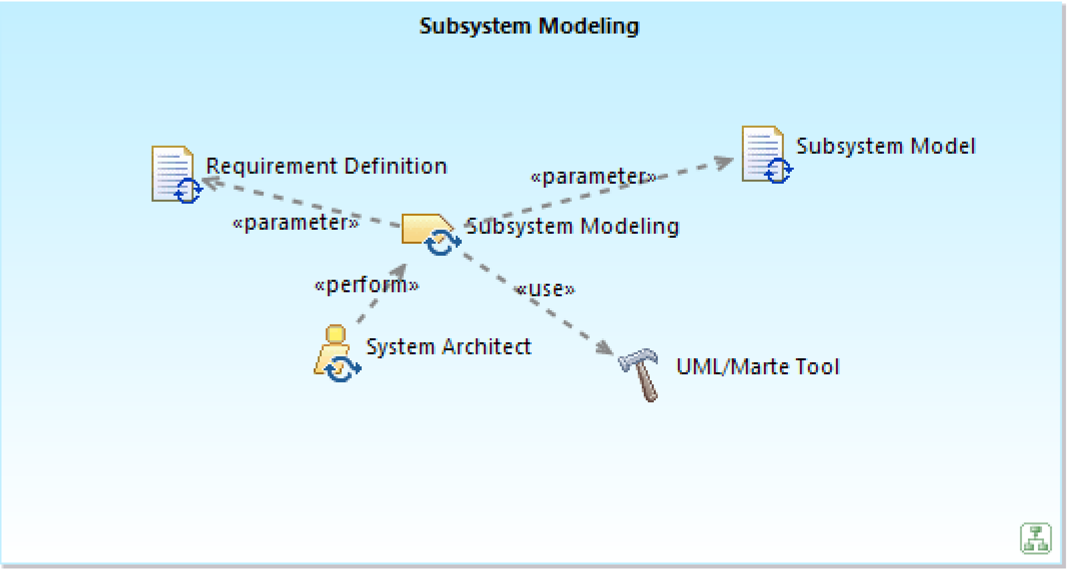
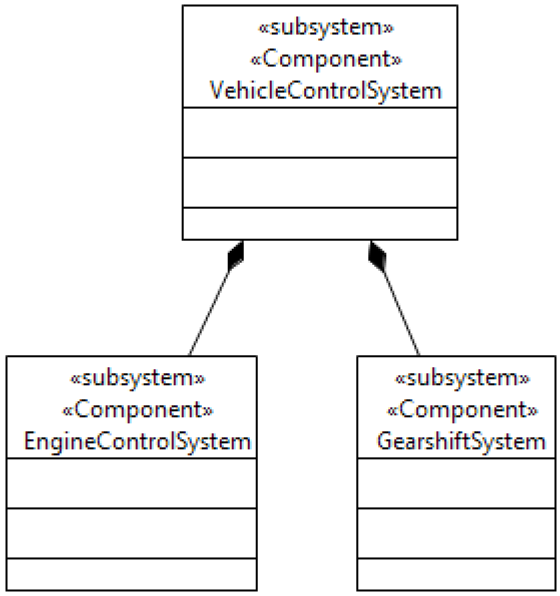


Figure 23: Subsystem Modelling

Subsystem Modelling	
Overview	For large scale system development it might be useful to decompose the system into smaller parts so called subsystems. There should be a loose coupling between subsystems so that a development in parallel can take place.
Assigned Tickets	#43 #49, #76, #218, #264, #145 #34 #77
Language	UML for Marte profile, VERDE profile definition
Tools	Papyrus MDT or any tool supporting the standard (full compliance)
Parameters	Requirement Model, in Subsystem Model, out
Pre-condition	Requirement definitions
Post-condition	
Steps	**Subsystem Definition**: To model subsystems a UML stereotype <<Subsystem>> can be assigned to components in UML class diagrams. **Subsystem Decomposition**: A system or subsystem can consist of none or many subsystems. In order to show that a subsystem is decomposed into other subsystems a composition association is modeled in a UML class diagram.

Example	«subsystem» «Component» VehicleControlSystem «subsystem» «Component» EngineControlSystem «subsystem» «Component» GearshiftSystem

5.2 System Design / Component based Design

5.2.1 Modelling of the Execution Platform

5.2.1.1 Modelling of Hardware Baseline

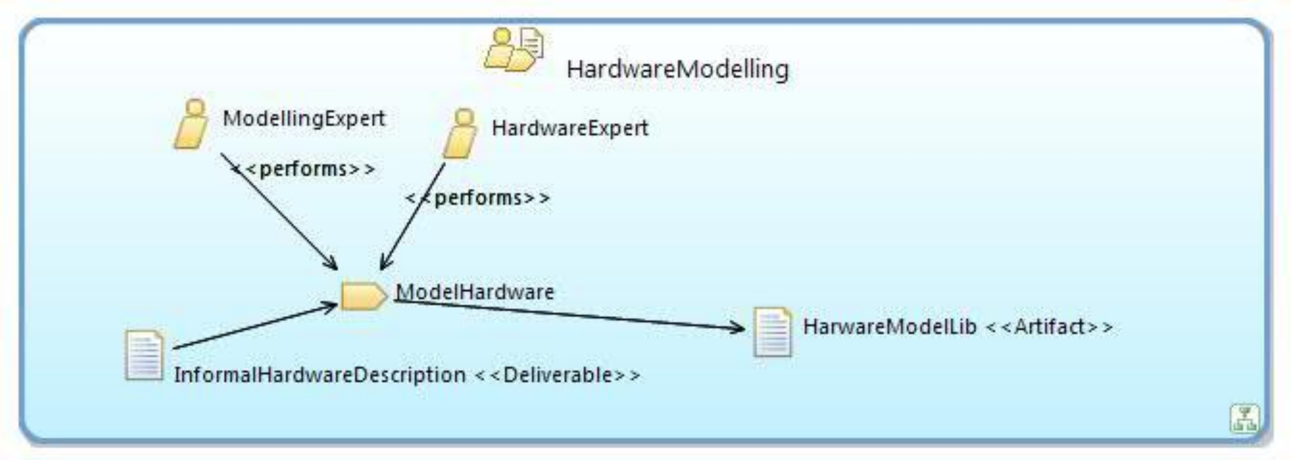
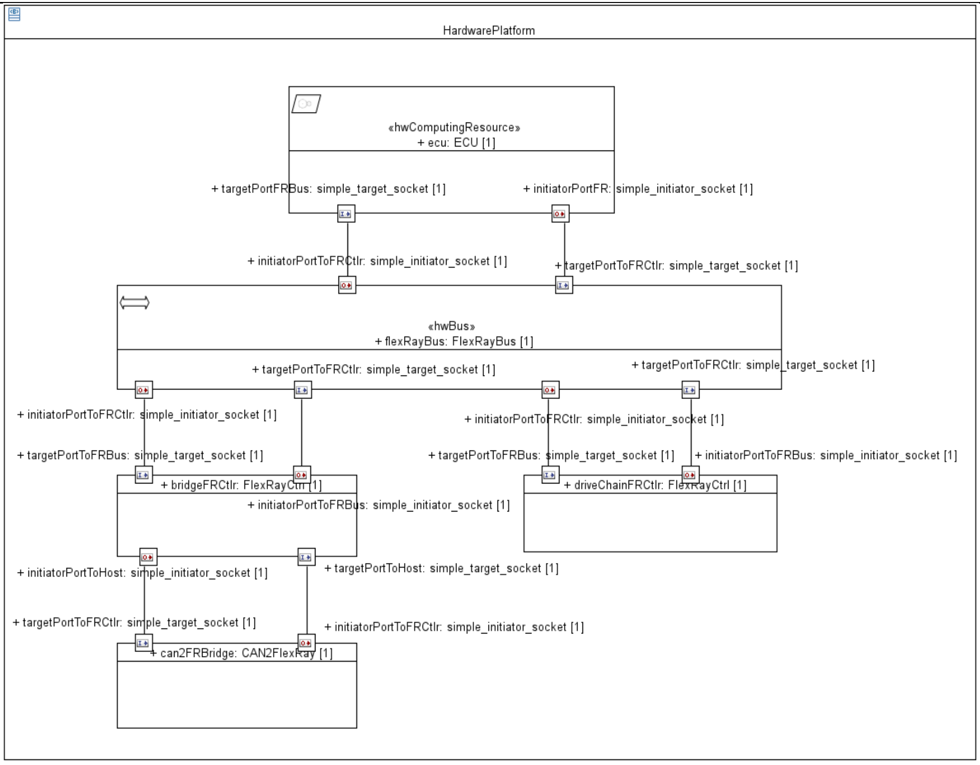
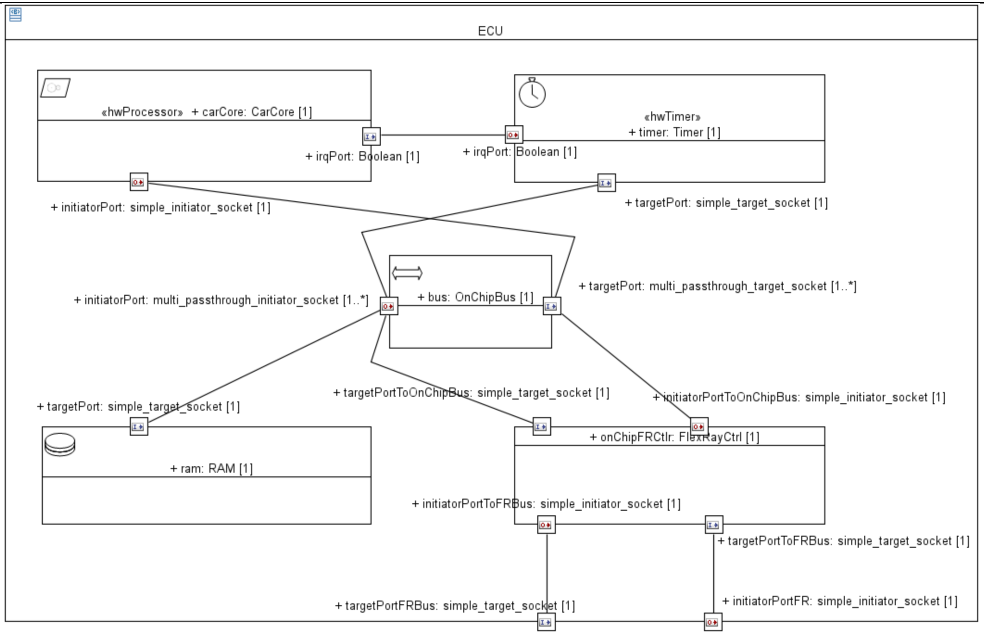


Figure 24: Modelling of Hardware Baseline

Modelling of Hardware Baseline	
Overview	For an accurate estimation of non-functional properties in software-intensive real-time systems the consideration of the underlying hardware platform is an essential part during an iterative development process.
Assigned Tickets	hardware / software integration to one system (Ticket 58) VERDE shall offer different views (Ticket 76) VERDE shall support partitioning (Ticket 56) Different bus architectures, protocols and arbitration schemes (Ticket 67)
Language	UML for Marte profile, SysML
Tools	Any tool supporting the standard (full compliance)
Parameters	Informal hardware description, in hardware module specifications, in SysML hardware model, out
Pre-Conditions	
Post-Conditions	
Steps	**Import UML2SystemCAdapter library** For being able to provide communication ports of the hardware baseline with communication primitives (e.g. blocking or non-blocking) a library based on transaction-level communication patterns has to be imported. Note that in general this library is independent from the language which is used to build the execution platform. **Identify hardware subsystems** First of all, the hardware modeler should be aware of (already existing) hardware subsystems, e.g. electronic control units (ECU), communication buses, and controllers. For each identified hardware subsystem a UML package is created. **Define hardware modules as SysML blocks** Hardware modules are defined as a SysML block in a SysML Block Definition Diagram (BDD). On that level the ports of

	the hardware module are of a special interest such that each of these hardware modules ports is modeled by a UML port which is added to the corresponding SysML block. The UML ports are stereotyped with a SysML flow port indicating the direction of the information flow. To specify the interaction pattern of the port, the port has to be typed with a class defined in the adapter library representing different communication primitives (e.g. synchronous blocking communication at transaction level). **Assign MARTE stereotypes to SysML blocks** To configure the hardware module the hardware-specific MARTE stereotype (e.g. *HwProcessor*, *HwBus*) from the MARTE library *MARTE::MARTE_DesignModel::HRM::HwLogical*is assigned to the SysML block and the tagged values of this stereotype are set according to the hardware module specification. **Instantiate SysML blocks** The SysML blocks representing hardware modules are instantiated in a SysML Internal Block Diagram (IBD). Therefore, an IBD is created for every SysML block on the next higher level block, e.g. the top module level or any subsystem, which is supposed to be modeled in detail. **Connect ports** After instantiation of the SysML blocks, the ports of the instances are connected among each other in the IBD by using standard UML connectors.
Example	

HardwarePlatform
«hwComputingResource»
+ ecu: ECU [1]
+ targetPortFRBus: simple_target_socket [1]
+ initiatorPortFR: simple_initiator_socket [1]
+ initiatorPortToFRCtlr: simple_initiator_socket [1]
+ targetPortToFRCtlr: simple_target_socket [1]
«hwBus»
+ flexRayBus: FlexRayBus [1]
+ targetPortToFRCtlr: simple_target_socket [1]
+ targetPortToFRCtlr: simple_target_socket [1]
+ initiatorPortToFRCtlr: simple_initiator_socket [1]
+ initiatorPortToFRCtlr: simple_initiator_socket [1]
+ targetPortToFRBus: simple_target_socket [1]
+ targetPortToFRBus: simple_target_socket [1]
+ initiatorPortToFRBus: simple_initiator_socket [1]
+ bridgeFRCtlr: FlexRayCtrl [1]
+ driveChainFRCtlr: FlexRayCtrl [1]
+ initiatorPortToFRBus: simple_initiator_socket [1]
+ initiatorPortToHost: simple_initiator_socket [1]
+ targetPortToHost: simple_target_socket [1]
+ targetPortToFRCtlr: simple_target_socket [1]
+ initiatorPortToFRCtlr: simple_initiator_socket [1]
+ can2FRBridge: CAN2FlexRay [1]

ECU
«hwProcessor» + carCore: CarCore [1]
+ irqPort: Boolean [1]
+ irqPort: Boolean [1]
«hwTimer»
+ timer: Timer [1]
+ initiatorPort: simple_initiator_socket [1]
+ targetPort: simple_target_socket [1]
+ initiatorPort: multi_passthrough_initiator_socket [1..*]
+ bus: OnChipBus [1]
+ targetPort: multi_passthrough_target_socket [1..*]
+ targetPort: simple_target_socket [1]
+ targetPortToOnChipBus: simple_target_socket [1]
+ initiatorPortToOnChipBus: simple_initiator_socket [1]
+ ram: RAM [1]
+ onChipFRCtlr: FlexRayCtrl [1]
+ initiatorPortToFRBus: simple_initiator_socket [1]
+ targetPortToFRBus: simple_target_socket [1]
+ targetPortFRBus: simple_target_socket [1]
+ initiatorPortFR: simple_initiator_socket [1]

5.2.2 Modelling of Functional Components

MARTE has the notion of component, which is mapped on the UML component concept: there is no specific MARTE stereotype for components. The stereotype component of UML2 is not sufficient, as it is likely to be used not only for functional components, but also for other kinds of components (processors, buses, etc.). Therefore we need a way to specify the nature of functional components more accurately.

A good choice is to use **stereotype ComponentType from FCM**, which corresponds to the notion of functional component declaration.

Component types do not specifically correspond to software of hardware components: they only represent functional components.

MARTE does not provide any specific stereotype for functional components implementations. For the same reasons as for component types, one needs more specific semantics for modelling.

We can use **stereotype `ComponentImpl` from FCM**, which corresponds to the notion of functional component implementation.

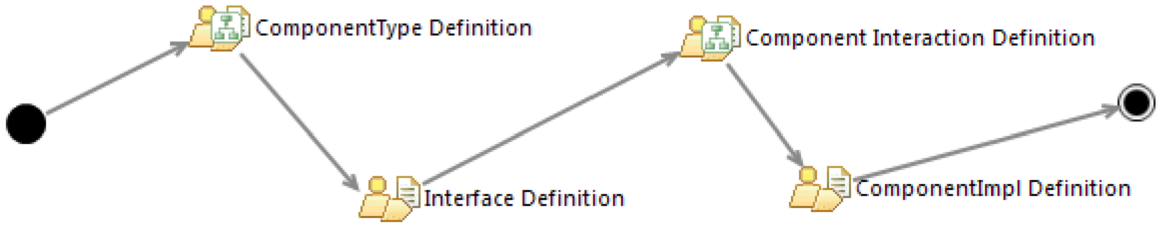


Figure 25: Workflow Modelling of Functional Components

The definition of Components starts with the definition of Components Types and its Interfaces, followed by the definition of Component interactiosn. Since each component type can be implemented in several ways, a component implementation can be modeled as well.

5.2.2.1 Definition of Component Types and Interfaces

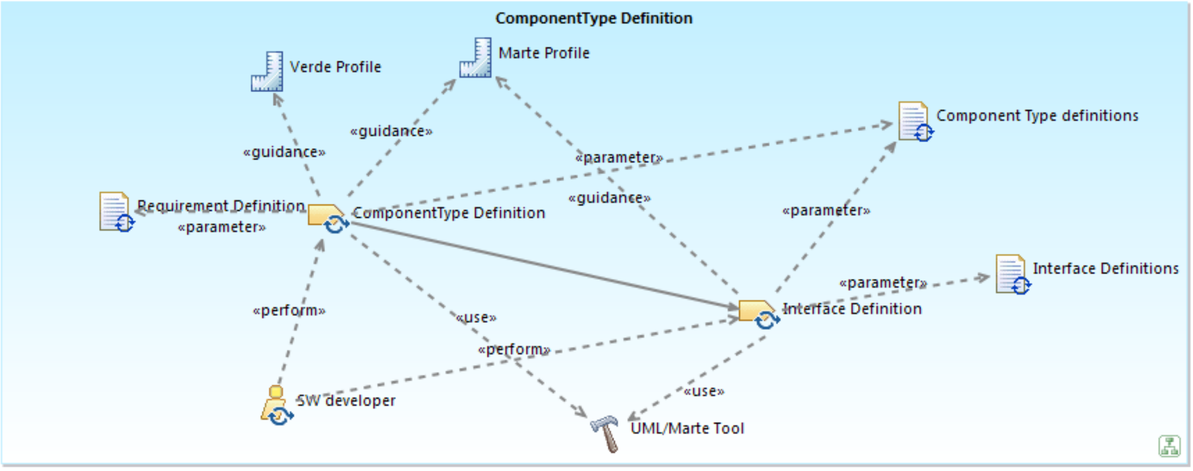
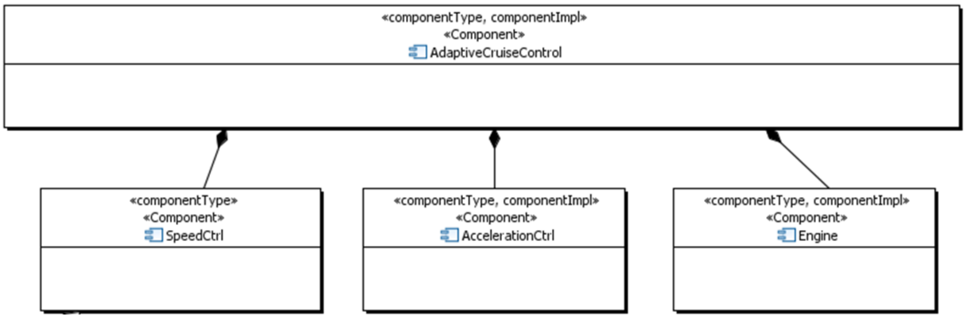
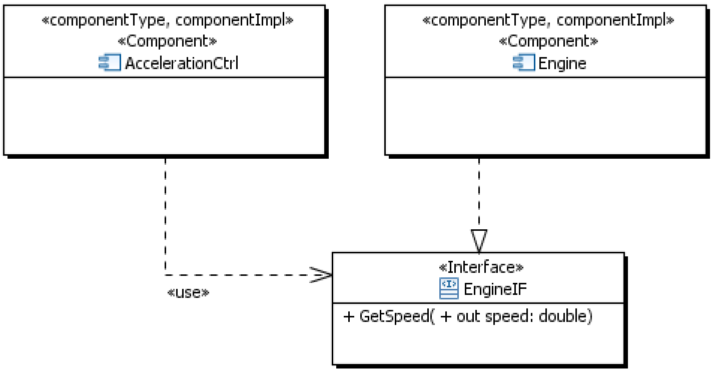


Figure 26: Definition of Component Types and Interfaces

Creation of Component Types	
Overview	Functionality of a system to be developed is encapsulated in functional components. Component types are identified using the FCM stereotype ComponentType applied on UML components. No additional information is required. Only component types should have ports.
Assigned Tickets	#62, support of control or data oriented architectures / systems #146, Definition of generic components #264, VERDE shall be able to refine components during specifications phases #217, support of analog components and simulators
Language	UML for Marte profile, VERDE profile definition
Tools	Papyrus MDT or any tool supporting the standard (full compliance)
Parameters	Requirement definition, in Component Type definition, out
Pre-condition	Requirement definitions
Post-condition	Component Types are defined in a UML Class Diagram
Steps	To define a new component type open an UML Class Diagram, create a new UML Component and add stereotype `ComponentType` from the Verde profile to this component. Optional relationships between components like Dependencies can be defined in class diagrams as well.

Example	 «componentType, componentImpl» «Component» AdaptiveCruiseControl «componentType» «Component» SpeedCtrl «componentType, componentImpl» «Component» AccelerationCtrl «componentType, componentImpl» «Component» Engine

Creation of Interfaces	
Overview	Verde ML entirely relies on UML for the definition of interfaces. One should use a UML interface with UML operations. In class diagrams a realization and usage relationship denote the dependency to UML components.
Assigned Tickets	#146, Definition of generic components #34, VERDE shall provide a clear definition of software interfaces
Language	UML for Marte profile, VERDE profile definition
Tools	Papyrus MDT or any tool supporting the standard (full compliance)
Parameters	Requirement definition, in Interface definition, out
Pre-condition	Requirement definitions
Post-condition	Interfaces are defined in a UML Class Diagram
Steps	Open a UML class diagram and create a new Interface type. Add operations to the Interface Symbol to describe more details. In class diagrams a realization and usage relationship denote the dependency to UML components.
Example	5.2.3  «componentType, componentImpl» «Component» AccelerationCtrl «componentType, componentImpl» «Component» Engine «use» «Interface» EngineIF + GetSpeed(+ out speed: double)

5.2.3.1 Component Interaction Definition

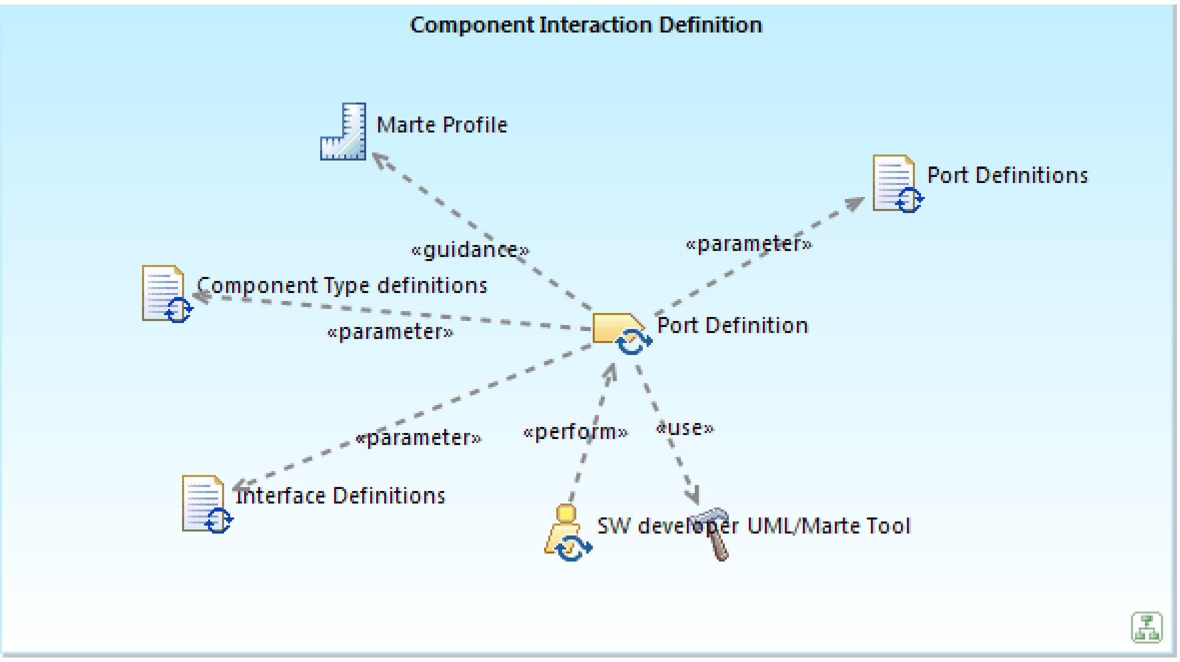
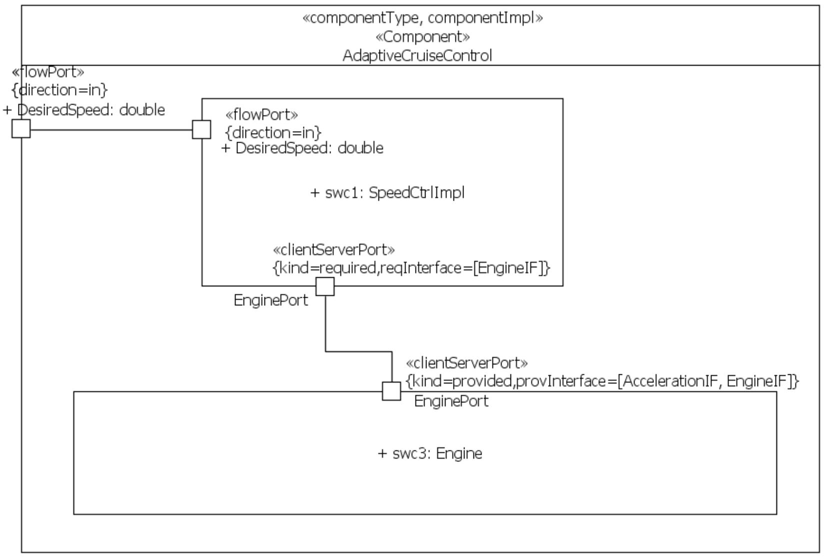


Figure 27: Definition of Ports and Communication Patterns

Ports and Communication Patterns for Components	
Overview	Ports are the interaction points of functional components and are modeled as UML ports with stereotypes from MARTE.
Assigned Tickets	#146, Definition of generic components
Language	UML for Marte profile, VERDE profile definition
Tools	Papyrus MDT or any tool supporting the standard (full compliance)
Parameters	Component Type definition, in Interface definition, in Port definition, out
Pre-Condition	Component Types are defined in a UML Class Diagram
Post-Condition	Ports and Communication Patterns are defined in a UML Composite Structure Diagram
Steps	Ports in Verde ML are UML ports with stereotypes from MARTE. **Operation calls** Communications based on operation calls are modelled by ports with stereotype ClientServerPort from MARTE. A MARTE client server port can provide and/or require several interfaces. In order to specify interfaces that are provided and required, some attributes should be filled in. Attribute kind indicates whether the port provides or requires interfaces. Attributes reqInterface and provInterface are lists of interfaces that are required or provided. Their values should be consistent with the value of field kind.

	As the interfaces are specified using attributes of the stereotype, no type should be associated with the port itself. **Messages** Communications based on message passing (signals or events) are modelled by ports with stereotype FlowPort from MARTE. No specific attribute is required. The type of data transmitted through the port is to be specified by associating a type to the UML port. **Interaction Patterns** Use connectors. A connector is a specific kind of component responsible for interactions. A connector shares all properties of components, i.e. connectors own ports and there is a separation between type and implementation. A difference is that connectors typically need to be adapted to the context in which they are used, e.g. a connector realizing client/server calls needs to be adapted to the interface of the components that should interact. This implies that the port type is generic and needs to be instantiated. One (non-mandatory) solution is to defined connectors in a UML package template and performs the instantiation by means of a model transformation. This approach has been chosen in eC3M.
Example	 «componentType, componentImpl» «Component» AdaptiveCruiseControl «flowPort» {direction=in} + DesiredSpeed: double «flowPort» {direction=in} + DesiredSpeed: double + swc1: SpeedCtrlImpl «clientServerPort» {kind=required,reqInterface=[EngineIF]} EnginePort «clientServerPort» {kind=provided,provInterface=[AccelerationIF, EngineIF]} EnginePort + swc3: Engine

5.2.3.2 Definition of Component Implementations

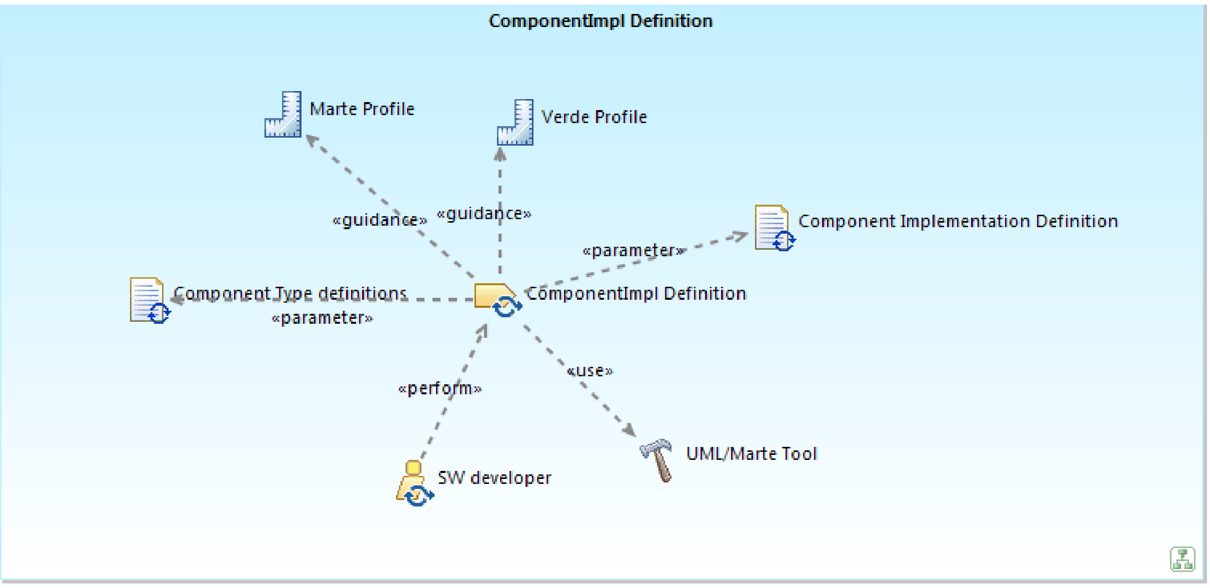
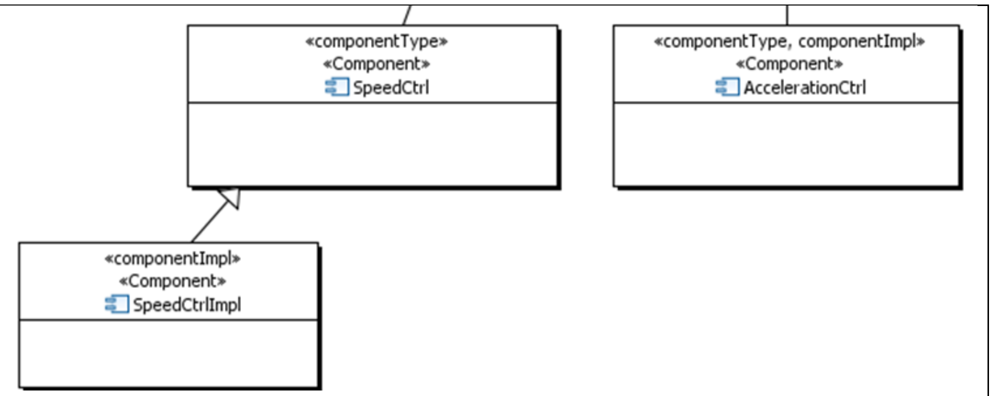


Figure 28: Definition of Component Implementations

Creation of Component Implementations	
Overview	A Component Implementation is supposed to implement a Component Type.
Assigned Tickets	#146, Definition of generic components #145,VERDE shall offer Reuse of precedent design components
Language	UML for Marte profile, VERDE profile definition
Tools	Papyrus MDT or any tool supporting the standard (full compliance)
Parameters	Component Type Definition, in Component Implementation Definition, out
Pre-conditions	Component Types are defined in a UML Class Diagram
Post-conditions	Component Implementation are defined in a UML Class Diagram
Steps	To define a new component type open an UML Class Diagram, create a new UML Component and add stereotype **`ComponentImpl`** from the Verde profile to this component. The relationship between a component implementation and the corresponding component type is inheritance. Using inheritance between component implementation and component type allows the preservation of ports, which are defined at the component type level. No port should be defined in component implementations. Note that both stereotypes `ComponentType` and `ComponentImpl` can be applied to a component declaration. In this case, the component declaration is both a type and an implementation. The attribute `componentType` must then points to the component itself.

Example	«componentType» «Component» SpeedCtrl «componentType, componentImpl» «Component» AccelerationCtrl «componentImpl» «Component» SpeedCtrlImpl

5.3 Detailed Design

5.3.1 Modelling of Internal Behaviours

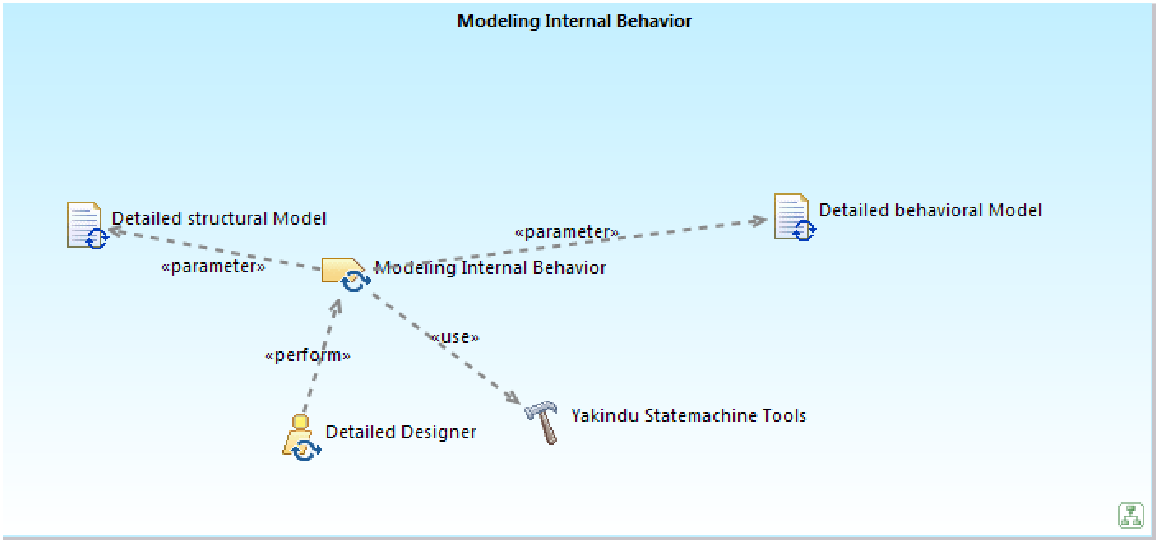
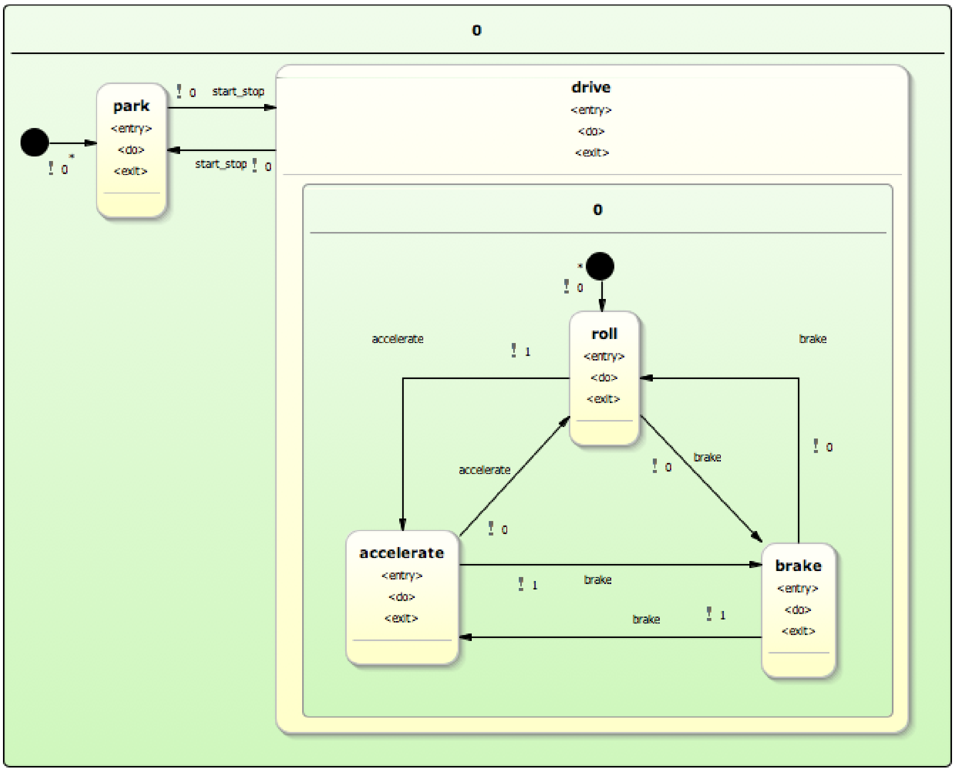

Modeling Internal Behavior
Detailed structural Model
«parameter»
«parameter»
Detailed behavioral Model
Modeling Internal Behavior
«use»
«perform»
Detailed Designer
Yakindu Statemachine Tools

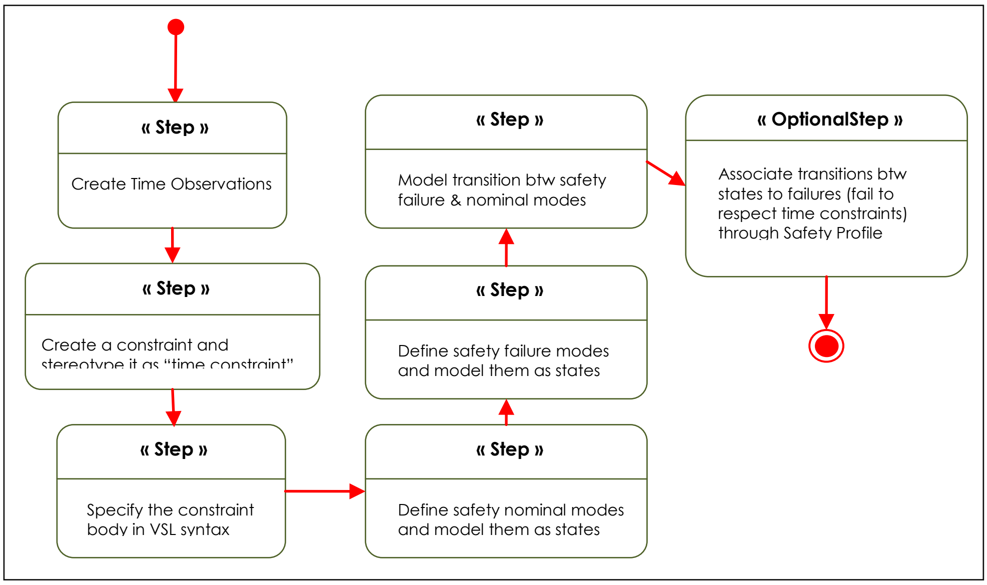
Figure 29: Modelling Internal Behaviour

Modelling Internal Behaviour	
Overview	Behaviour modelling is necessary to reproduce the required behaviour of the original system. It is fundamental to simulate the system and obtain validation and verification results. Later on it is used to generate the behaviour implementation.
Assigned Tickets	#63, #68, #76, #218, #223, #264, #10, #257, #30
Language	Yakindu SM Metamodel, Simulink Metamodel
Tools	YakinduStatemachine Tools, Matlab/Simulink
Parameters	Detailed Model (structure), in Detailed Model (structure and behaviour), out
Pre-Conditions	
Post-Conditions	
Steps	**Model Internal Behaviour:** Behaviour modelling can be done with any type of behaviour modelling techniques. We decided to do behaviour modelling with the Yakindu Statemachine Tools and Matlab/Simulink. Before beginning behaviour modelling the detailed design model should include the system structure. Detailed design modelling is often done iteratively. That is why the detailed design model must include at least those structural elements that will be extended with behaviour.
Example	**Model Internal Behaviour:** The following picture shows a YakinduStatemachine diagram example:

5.3.2 Non-Functional Characterization

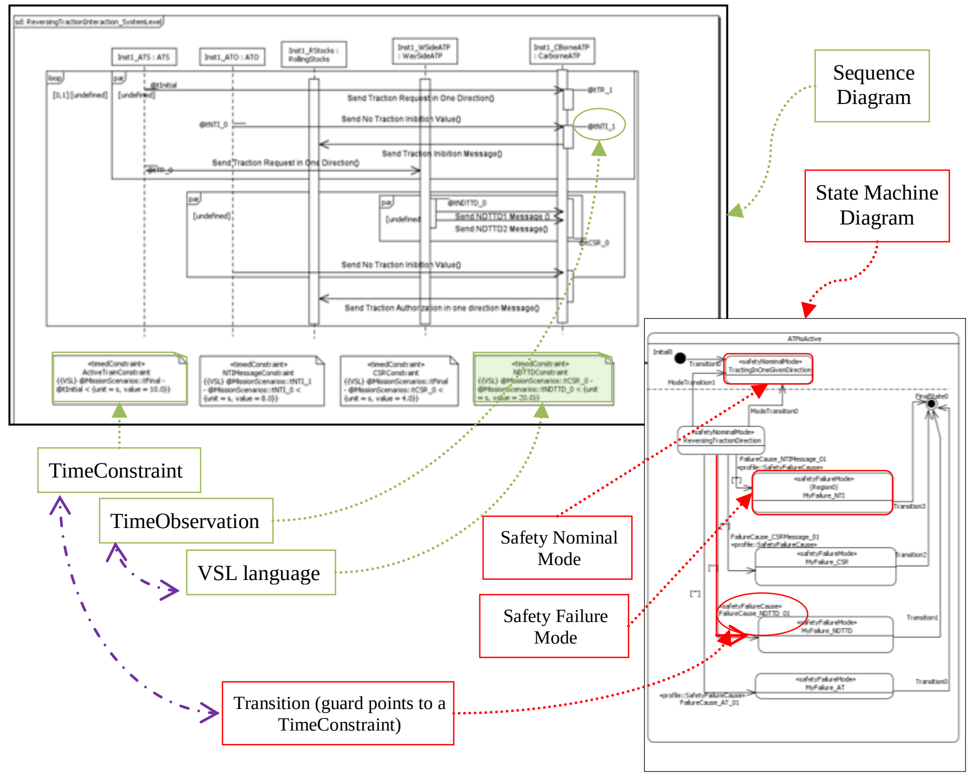
5.3.2.1 Non Functional Characterization for Safety & RAM



« Step »
Create Time Observations
« Step »
Create a constraint and stereotype it as "time constraint"
« Step »
Specify the constraint body in VSL syntax
« Step »
Define safety nominal modes and model them as states
« Step »
Define safety failure modes and model them as states
« Step »
Model transition btw safety failure & nominal modes
« OptionalStep »
Associate transitions btw states to failures (fail to respect time constraints) through Safety Profile

Figure 30: Non Functional Characterization for Safety & RAM

Non Functional Characterization for Safety & RAM	
Overview	Safety & RAM are core issues for train industry. They represent thus an important non-functional aspect of modelling activities. Modelling Safety or RAM characteristics is mainly based on time constraints and (safety) nominal or failure states -and transitions between them- of Systems. Modelling these characteristics is an open door to better understanding and analysis of a System's Safety and RAM levels
Assigned Tickets	Support functional and non functional aspects (Ticket 60)
Language	SysML, Marte profile, Safety profile (in-house)
Tools	Any tool supporting these standards (full compliance)
Parameters	System Design Model, in Time constraints, Failure states, stereotypes refereeing to System Design Model, out
Pre-Conditions	
Post-Conditions	
Steps	**Create time observations**: This step consists in creating time observations (in the context of Sequence Diagrams), so that they can be referenced in VSL expressions, formally capturing time constraints. Each time observation will typically refer to a communication event associated with a message from the interaction, or to an execution occurrence. **Create a constraint**: In order to encapsulate an expression that will actually describe the timing constraint, an UML Constraint must be created. This constraint is typically owned by aSysML Sequence diagram, and can additionally refer constrained elements (e.g., the time observations that will be manipulated in the VSL expression). Marte Profile is used to stereotype each constraint as a "timeconstraint". **Specify the constraint body with VSL**: Once the constraint has been created, a VSL expression can be encapsulated in it. The VSL expression is a Boolean expression which will make reference to time observations. For example, if the context interaction defines time observations @t1 and @t2, the following VSL expressions could be specified: @t2 - @t1 < {value = 15.0, unit = ms}. **Define& Model safety nominal modes**: Safety Nominal Modes correspond to operational or functional modes and are modeled with states. They are linked to components (System, sub-system, software…) **Define safety failure modes**: Safety Failure Modes are modeled with states. They correspond to possible types of failure. And a reach from Safety Nominal modes through transitions. **Model Transitions**: Btw Safety Nominal and Failure Modes. They correspond to specific events that trigger the transition and guards. **Associate transitions btw states to failures**: Failures can be due to non respect of time constraints. In such cases, transitions' guards can be associated to time constraints.

Example	 Sequence Diagram State Machine Diagram TimeConstraint TimeObservation VSL language Safety Nominal Mode Safety Failure Mode Transition (guard points to a TimeConstraint)

5.3.2.2 Resource Usage Characterization

Resource Usage characterization (of elements from a system model) is an important aspect of non-functional characterization. It consists in specifying the amount of resource usage (e.g. memory, CPU time, network bandwidth, etc.) required by the run-time manifestation of a given model element (e.g. a model element representing a component instance) in order to enable early analysis and validation of design choices. As depicted in figure 29, this methodological pattern is composed of a single task “Express Resource Usage”. Details about this task are provided in the table below.

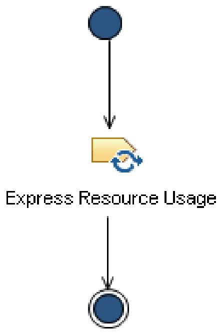
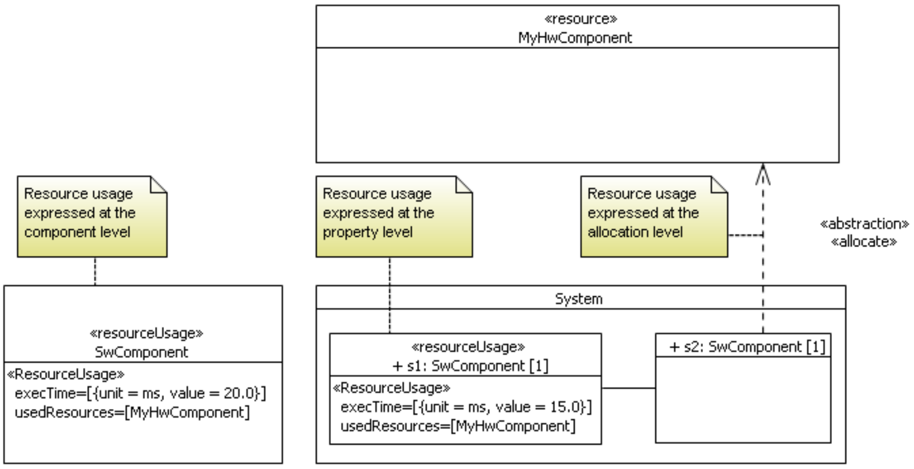


Figure 31: Resource Usage Characterization pattern

Express Resource Usage	
Overview	Associate resource usage information with a UML model element.
Assigned Tickets	#60, #95, #307, #308, #309, #451, #452, #453, #454, #455
Language	UML, MARTE
Tools	MDT Papyrus
Parameters	Inputs: - inoutelementToBeCharacterized : NamedElement [1] - in resources : Resource [0..*]
Pre-condition	None
Post-condition	elementToBeCharacterized has stereotype « ResourceUsage » applied.
Steps	**Apply ResourceUsage stereotype:** This step consists in applying the MARTE stereotype <<ResourceUsage>> to model elements of interest in the system model. This stereotype can be applied to any UML NamedElement. For examples, this includes structural elements (e.g. component implementation, instances or properties) as well as behavioural elements (e.g., activities, actions or messages), or even relationships such as Allocations. The third step of this task provides details on the various kinds of non-functional properties which can be specified via this stereotype. **(Optional) Identify used resources:** This step enables to specify the resources for which the resource usage values (see step 3 for details) are specified. This is captured via property usedResources : Resource [*]. Since the multiplicity of this property is * (i.e., it is potentially empty), the overall step can be considered optional. In this case, the ResourceUsage can be considered underspecified, and could be refined further in the design process. Note that an Allocation (e.g. of a software component onto an execution resource) can also be stereotyped with ResourceUsage. In this case, the used resource is simply the target of the allocation relationship. **Specify the resource consumption with VSL:** This step consists in specifying the various usage values for this particular resource usage. Stereotype ResourceUsage carries the following non-functional properties: • execTime : NFP_Duration [*], the time that the resource is in use due to the usage • msgSize : NFP_DataSize [*], the amount of data transmitted by the resource • allocatedMemory : NFP_DataSize [*], the amount of memory that is demanded from or returned to the resource. It maybe a positive or negative value • usedMemory : NFP_DataSize [*], the amount of memory that will be used from a resource but that will be immediately returned, and hence should be available while the usage is in course. This may be used to specify the required free space in the stack for example. • powerPeak : NFP_Power [*], the power that should be available from the resource for its usage

	• energy : NFP_Energy [*], the amount of energy that will be permanently consumed from a resource due to the usage. The type of each of these properties is an NFP type. The VSL syntax can therefore be conveniently used to specify associated values. Note that the multiplicity of each property is *. Firstly, it means that it is not mandatory to specify a value for all of these properties. Secondly, it means that multiple values can be specified for each property, which enables to have different values characterized with different statistical or source qualifiers.
Example	

«resource»
MyHwComponent
Resource usage expressed at the component level
Resource usage expressed at the property level
Resource usage expressed at the allocation level
«abstraction»
«allocate»
«resourceUsage»
SwComponent
«ResourceUsage»
execTime=[{unit = ms, value = 20.0}]
usedResources=[MyHwComponent]
System
«resourceUsage»
+ s1: SwComponent [1]
«ResourceUsage»
execTime=[{unit = ms, value = 15.0}]
usedResources=[MyHwComponent]
+ s2: SwComponent [1]

5.3.3 Legacy Code Abstraction/ Integration

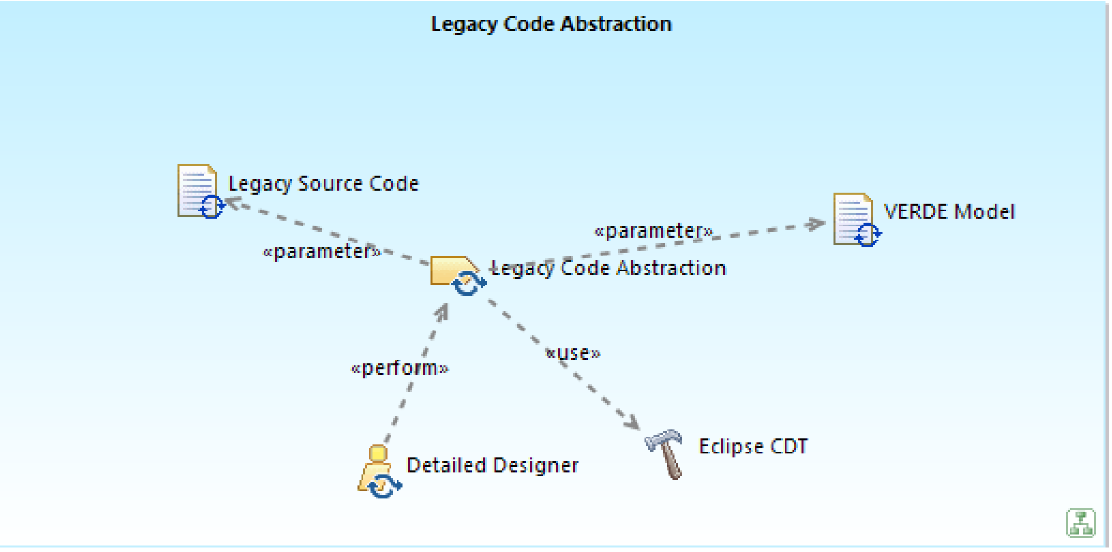
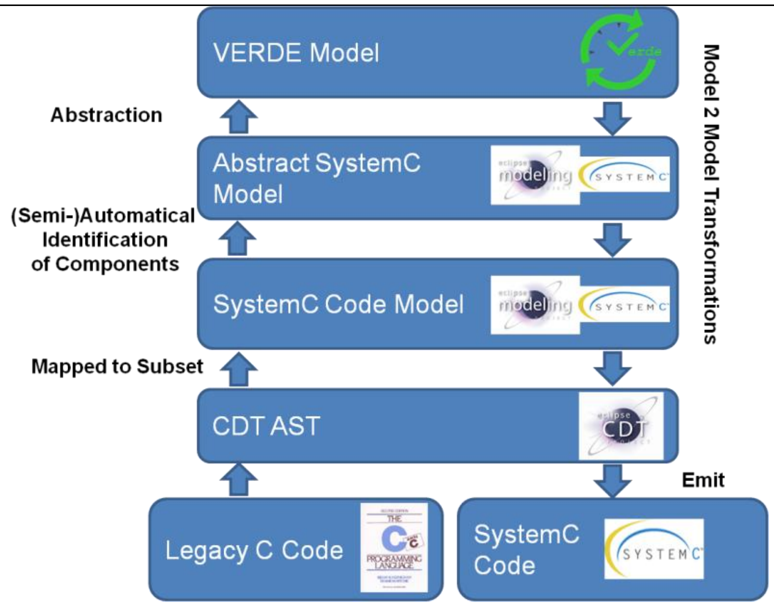


Figure 32: Legacy Code Abstraction

Legacy Code Abstraction	
Overview	For analyzing and testing non-functional properties in software-intensive real-time systems the modelling methodology must be able to capture already existing (source) code and generate an execution platform including this source code. Legacy code abstraction is especially needed for round-trip engineering in a model-based development process.
Assigned Tickets	Functionality described using C/C++ (Ticket 68) Timing and power abstraction of legacy software (Ticket 403) Abstraction of legacy code to components (Ticket 404) NFP Analysis of functionality implemented in C/C++ (Ticket 440)
Language	C/C++/SystemC, UML
Tools	Eclipse CDT
Parameters	C/C++/SystemC Source Code, in (UML/VERDE) Functional Model, out
Pre-Conditions	
Post-Conditions	
Steps	 VERDE Model Abstraction Abstract SystemC Model (Semi-)Automatical Identification of Components SystemC Code Model Mapped to Subset CDT AST Emit Legacy C Code SystemC Code Model 2 Model Transformations **Building AST in Eclipse CDT** By using the Eclipse CDT framework existing source code is parsed and an AST representation is automatically generated on condition that the source code is contained in a valid CDT project. **Mapping to SystemC Code Metamodel** The CDT AST model is elaborated and all relevant nodes in the CDT AST are mapped onto corresponding elements in a SystemC Code Model. Due to the bidirectional M2M transformation with QVT this model can be used to generate code. This feature is mainly applied in a top-down development process starting at UML/VERDE modelling. **Component Abstraction by SystemC Metamodel** The source-code-oriented SystemC code metamodel is mapped onto an abstract SystemC metamodel using the same technology for M2M transformation as in the transformation step before. This abstracted model is bridging the gap between a representation of existing source code artifacts and the component-based modelling approach addressed in VERDE.

5.4 Model based Verification

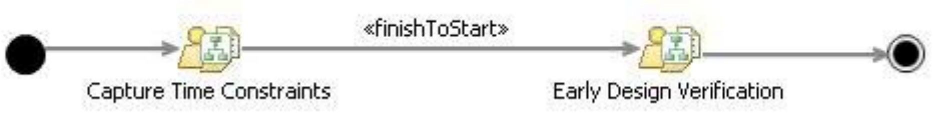


Figure 33: Model Based Verification Pattern

The Model Based Verification starts with the capture on non-functional properties related to real-time constraints. Annotating the models with real-time constraints is a pre-requisite to their early design verification. In the following sections, we describe each of those activities.

5.4.1 Capture time constraints

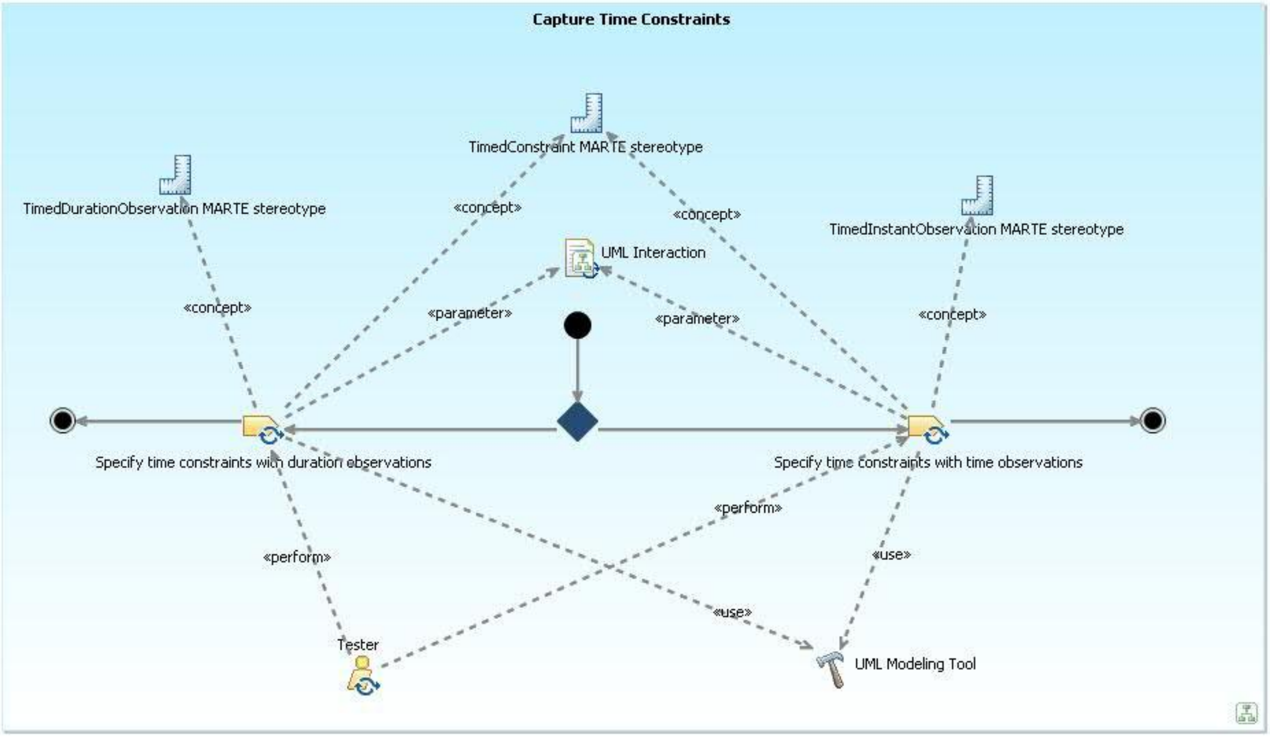

Capture Time Constraints
TimedConstraint MARTE stereotype
TimedDurationObservation MARTE stereotype
TimedInstantObservation MARTE stereotype
«concept»
«concept»
UML Interaction
«concept»
«concept»
«parameter»
«parameter»
Specify time constraints with duration observations
Specify time constraints with time observations
«perform»
«perform»
«use»
«use»
Tester
UML Modeling Tool

Figure 34: Capture Time Constraints

Expression of timing constraints is an important non-functional aspect of system design and validation, e.g. for specifying the maximal duration of a method call or the minimum duration between two occurrences of an event.

5.4.1.1 Products lifecycle

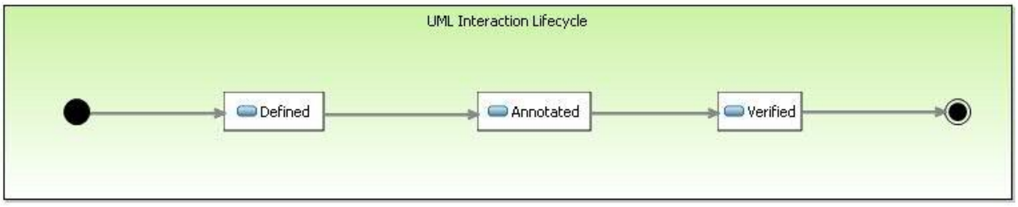
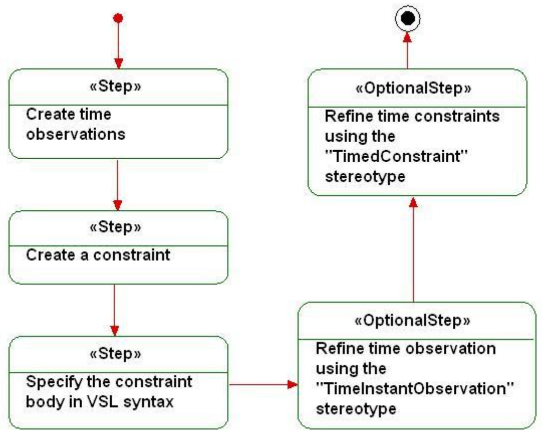


Figure 35: UML Interaction Lifecycle in Capture Time Constraints Activity

In the Capture Time Constraints activity, only one product is transformed and verified. Indeed this activity consists mainly in transforming a well defined UML Interaction product into an annotated UML Interaction (with time constraints). Those annotations are then used to verify the real-time properties of the model.

5.4.1.2 Capture Time Constraints Tasks

Specify time constraints with time observations	
Overview	One way to capture time constraints is to use time observations into UML Interactions. Then, from those observations, we can establish time constraints.
Assigned Tickets	Support functional and non functional aspects (Ticket 60)
Language	UML for Marte profile
Tools	Any tool supporting the standard (full compliance)
Parameters	Interaction:UML Interaction (inout)
Pre-condition	Interaction in state “Defined”
Post-Condition	Interaction in state “Annotated”
Steps	 «Step» Create time observations «Step» Create a constraint «Step» Specify the constraint body in VSL syntax «OptionalStep» Refine time observation using the "TimeInstantObservation" stereotype «OptionalStep» Refine time constraints using the "TimedConstraint" stereotype
Description	**Create time observations**: This step consists in creating time observations (in the context Interaction), so that they can further be referenced in VSL expressions, formally capturing timing constraints. Each time observation will typically refer to a communication event associated with a message from the interaction, or to an execution occurrence. Graphically, a time observation is represented by the symbol "@" followed by the name of the time observation. The fact that a time observation is actually bound to a communication event of a message will be graphically captured by having the time observation located at the corresponding end of the message. **Create a constraint**: In order to encapsulate an expression that will actually describe the timing constraint, a UML Constraint must be created. This constraint is typically owned by the context UML Interaction, and can additionally refer constrained elements (e.g., the time observations that will be manipulated in the VSL expression). Note that these additional references have no semantic impact on the VSL specification of the constraint, in the sense that it does not restrict the set of time observations that can be manipulated in the expression. Therefore, it can be seen as additional information making the model potentially easier to read or exploit. **Specify the constraint body with VSL**: Once the UML constraint has been created, a VSL expression can be encapsulated in it. Encapsulating the VSL expression involves the usage of an OpaqueExpression. The property language of the opaque expression must contain the string "VSL", and the property body must

contain the VSL expression. Since properties language and body are ordered collections, the indexes of "VSL" and of the VSL expression (in their respective collection) must be the same (see Section 7.3.35 OpaqueExpression from the UML superstructure).

The VSL expression must be a Boolean expression (i.e., an expression whose evaluation will produce a result of type Boolean), which will typically make reference to time observations. For example, if the context interaction defines time observations @t1 and @t2, the following VSL expressions could be specified (this list is of course not exhaustive):

- @t1 < @t2, which specifies that the event associated with the time observation t2 must occur before the event associated with the time observation t1 (shortly, @t1 must occur before @t2)
- @t2 > 11:43:45 2010/09/21, which specifies that @t2 must occur after a date literally specified
- @t2 - @t1 < {value = 15.0, unit = ms}, which specifies that the duration between the occurrence of @t1 and occurrence of @t2 must be lower than 15.0 milliseconds
- @t2 - @t1 < 15.0, which roughly specifies the same thing than the previous constraints, without specifying the time unit (i.e., it can be implicit from the context, or can be indirectly obtained from another model element, as illustrated in the last step of this task).

(Optional) Refine time observations with the MARTE stereotype <<TimedInstantObservation>>: As explained in the previous steps, in the context of an Interaction, a TimeObservation is bound to a communication event (i.e., the emission or reception of a message), and can therefore be literally interpreted as a specification of the instant where a message is emitted or received. However, this explanation only remains an interpretation.

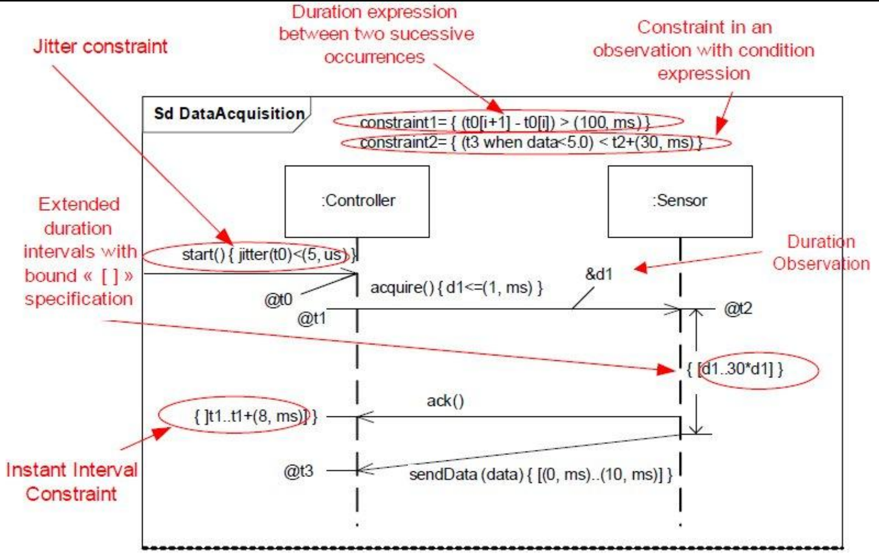
In order to avoid any ambiguity on the interpretation of the event observed via the time observation, MARTE provides a stereotype : <<TimedInstantObservation>>. With the property obsKind:

EventKind of this stereotype (possible values are start, finish, send, receive, consume), it is possible to indirectly characterize the event associated with the time observation. For example, if we have:

- @t1 with <<TimedInstantObservation>> {obsKind = send}, a VSL expression such as @t1 > 17:25 that the emission of the event underlying t1 must be done after a literally specified date
- @t1 with <<TimedInstantObservation>> {obsKind = consume}, a VSL expression such as @t1 < 17:25 that the event underlying t1 must be consumed before a literally specified date. This can be used to specify the validity date of a message.

(**Optional) Refine constraints with the MARTE stereotype <<TimedConstraint>>**: From a given expression context, it is possible to determine if a constraint actually concerns a particular instant (e.g., @t2 > 11:43:45 2010/09/21) or a duration (e.g., @t2 - @t1 < {value = 15.0, unit = ms}). Determining if the constraint refers to an instant or a duration typically requires an interpretation phase (which can be automated since the VSL syntax is formally defined), with an inference mechanism exploiting the content of the expression (e.g., the time events it refers to and the operator which are manipulated) as well as the context in which it is specified.

MARTE provides a stereotype which enables to explicitly tag a constraint as an "instant" and/or a "duration" constraint: <<TimedConstraint>>, which extends the UML metaclass Constraint. By applying the stereotype on a Constraint, it is possible to specify how the constraint must be interpreted, using the property

	interpretation: TimeInterpretationKind (possible values are instant and duration). If interpretation is set to the enumeration literal instant, then the constraint is interpreted as a constraint on instant value. If interpretation is set to the enumeration literal duration, then the constraint is interpreted as a constraint on duration value. Note that the stereotype <<TimedConstraint>> also inherits from stereotypes <<TimedElement>> (see Section 9.3.2.7 of the MARTE specification) and <<NfpConstraint>> (see Section 8.3.2.5 of the MARTE specification). With the property on: Clock (inherited from TimedElement), it is possible to reference a clock, which can itself be associated with a time unit (e.g., seconds, milliseconds, ticks, etc.). Considering a VSL expression such as @t2 - @t1 < 15.0, this can be used to indirectly specify the time unit behind the real literal "15.0". With the property kind: ConstraintKind (inherited from NfpConstraint), it is possible to further characterize the timed constraint (typical values are required or offered). Required indicates that the constraint represents a minimum quantitative or qualitative level. Offered establishes that the constraint represents the space of values that the constrained elements can afford.
Example	 Duration expression between two sucessive occurrences Jitter constraint Constraint in an observation with condition expression Sd DataAcquisition constraint1= { (t0[i+1] - t0[i]) > (100, ms) } constraint2= { (t3 when data<5.0) < t2+(30, ms) } :Controller :Sensor Extended duration intervals with bound « [ ] » specification start() { jitter(t0)<(5, us) } Duration Observation acquire() { d1<=(1, ms) } &d1 @t0 @t1 @t2 { [d1..30*d1] } ack() {]t1..t1+(8, ms)] } Instant Interval Constraint @t3 sendData (data) { [(0, ms)..(10, ms)] } (Taken from the official MARTE tutorial)

5.4.2 Non-Functional Analysis

5.4.2.1 Schedulability Analysis

The purpose of this process pattern is to enable schedulability analysis at early stages of the development process, by identifying the various steps required to refine an input functional model towards an output model carrying schedulability results. The proposed pattern is part of the Optimum methodology, detailed in VERDE deliverable F4.4.1.

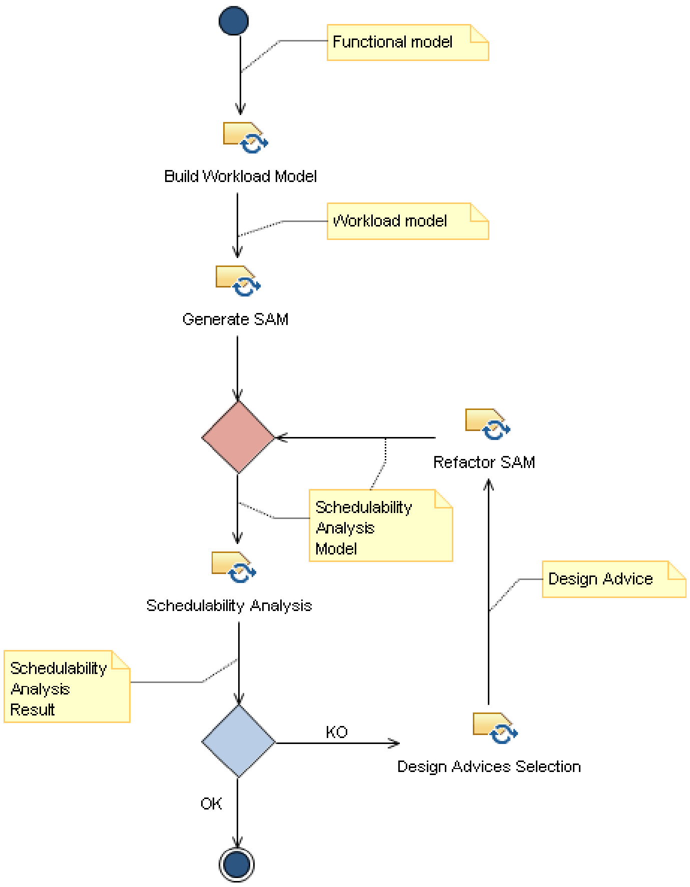


Figure 36: Schedulability Analysis Pattern

Schedulability Analysis Pattern	
Overview	Build a UML/MARTE model analyzable by schedulability analysis tools.
Assigned Tickets	#60, #218
Language	UML, MARTE
Tools	MDT Papyrus, Optimum
Parameters	- in functionalModel : FunctionalModel [1] - out analysisModel : SchedulabilityAnalysisModel [1]
Pre-condition	[TODO: specify rules for the input functional model.]
Post-condition	None
Steps	**Build Workload Model:** The workload behaviour model is build from the input functional model specifying the controlled sequence of actions triggered by external stimuli. The construction of a workload behaviour proceeds by the generation of an UML Activity diagram containing a canonical form of the controlled sequence of actions contained in the function model. Details about this canonical form are given in section 4.2.1.1.1 of VERDE deliverable F4.4.1. In addition to the workload behaviour, a very abstract view of the execution

	platform resources is also needed to have an estimation of computation time budgets for steps. This estimation is used to perform feasibility tests with respect to expressed end-to-end deadlines and external events activation rates. The platform execution resources are modeled with a UML property stereotyped <<SaExecHost>>. The resources types are modeled in a model library that can be reused. The classifier containing platform resources is stereotyped <<GaResourcesPlatform>>. This platform model is refined in the following steps. **Generate SAM:** Schedulability tests in the literature apply on specific characterization of recurrences of each external event (e.g. arrival pattern of the event, deadline for the system response, etc.). In this step, these characterizations are added to the input workload model in order to produce a schedulability analysis model suited to exploitation by analysis tools. It mainly consists in refining the platform resources identified in the previous step and specifying allocation of workload behaviours to platform resources. Details about modelling rules are given in section 4.2.1.2 of VERDE deliverable F4.4.1. **Schedulability Analysis:** In this step, the input SAM is enriched with information needed by schedulability analysis tools such as input numerical parameters, threading strategy and allocation information on potential execution and communication hosts. After analysis by tools, the model is enriched with computed schedulability analysis output parameters such as worst-case response time and hosts' utilization alons with a verdict on the schedulability of this context. Details about this step are given in section 4.2.2 of VERDE deliverable F4.4.1. **Design Advices Selection:** In the case where the schedulability analysis performed by an analysis tool reveals that scheduling requirements of the input SAM cannot be satisfied, the input SAM needs to be refactored. The purpose of this step is to identify and select an analytical design advice that could help the designer in the refactoring of the input SAM. Examples of analytical design advices are given in section 4.2.3.1 of VERDE deliverable F4.4.1. **Refactor SAM:** This step consists in refactoring a non-schedulable SAM by following analytical design advices selected in the previous step. This may for example imply reducing event arrival frequency, bounding execution times or increasing logical concurrency. The way to specify the information in the refactored model follows rules indentified in the other steps of this pattern.
Example	See section 5 of VERDE deliverable F4.4.1

5.4.2.2 Non functional analysis for Safety & RAM

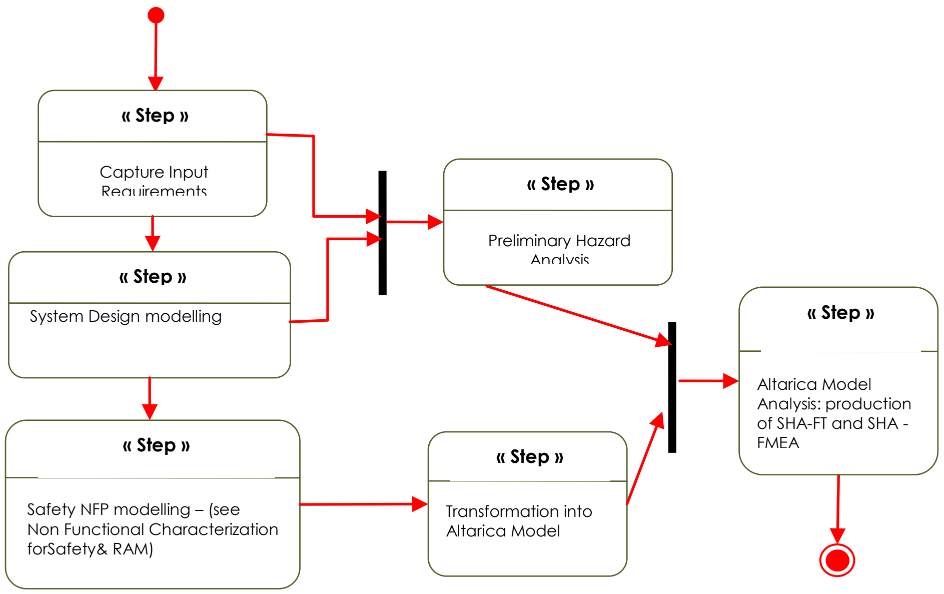
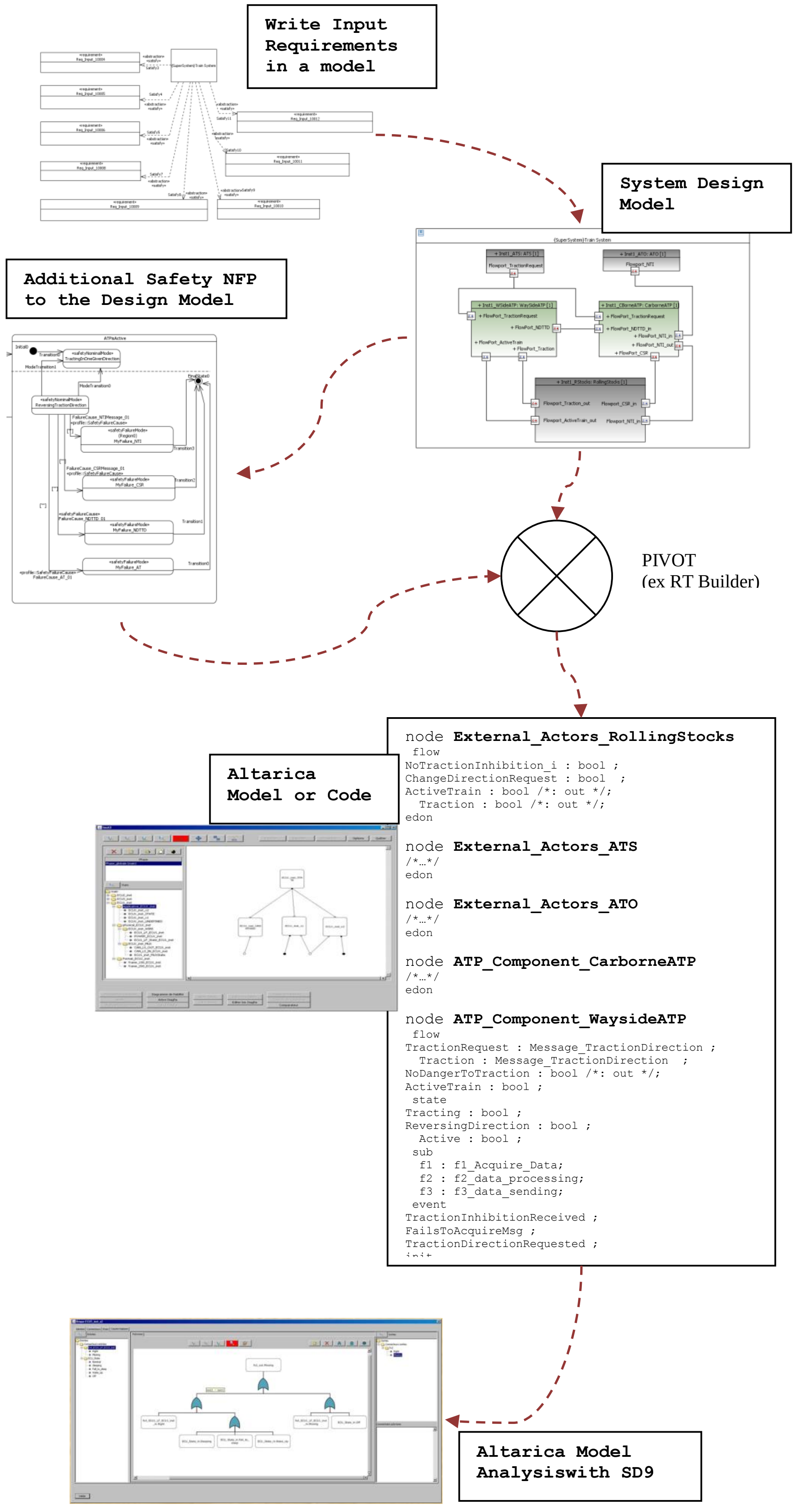

« Step »
Capture Input Requirements
« Step »
System Design modelling
« Step »
Safety NFP modelling – (see Non Functional Characterization forSafety& RAM)
« Step »
Preliminary Hazard Analysis
« Step »
Transformation into Altarica Model
« Step »
Altarica Model Analysis: production of SHA-FT and SHA - FMEA

Figure 37: Non functional analysis for Safety & RAM

Non functional analysis for Safety & RAM	
Overview	Safety is a core issue for train industry. It is thus an important non-functional aspect of modelling activities. Modelling Safety characteristics is mainly based on time constraints and safety nominal and failure states -and transitions between them- of Systems. Modelling safety characteristics is an open door to better understanding and analysis. Safety Analysis exploits Models to build FMEA. RAM analysis manipulates the same kind of NFPs and reveals availability and reliability levels of a System.
Methodological Pattern	[TODO: reference methodological pattern]
Assigned Tickets	Support functional and non functional aspects (Ticket 60)
Language	SysML, Marte profile, Safety profile (in-house), Altarica (Geensoft)
Tools	Any tool supporting UML standards (full compliance) – SD9 (Dassault – Geensoft)
Parameters	System Scpecification, in Altarica Model + Safety Hazard analysis *(Falt tree+ Failure Modes and Effect Analysis)*, out
Pre-Conditions	
Post-Conditions	
Steps	**Capture Input Requirements**: Input Requirements are the initial input of the whole process. They define the end-user specifications of a System (or components). **System Design Modelling**: Refers to a full modelling process used in Alstom which address operational, functional and constructional view modelling.

	Safety NFP modelling: See note *Non-functional characterization for Safety & RAM* **Preliminary Hazard Analysis**: Identification of the main hazards and there possible reasons. **Transformation into Altarica Model**:PIVOT transformer (Geensoft) generates an Altarica model out of design model (SysML). It includes at least nodes, hierarchy, interfaces and states. Ways of modelling Failure Modes effects on the design model are being evaluated and would therefore lead to generate through the PIVOT a full Altarica model. **Altarica Model Analysis**: Using Safety Designer to analyse the Altarica Model allows for generating the Safety Hazard Analysis (Fault Tree) and Safety Hazard Analysis (Failure Mode and Effects Analysis).

Example


Write Input Requirements in a model
System Design Model
Additional Safety NFP to the Design Model
PIVOT (ex RT Builder)
Altarica Model or Code
Altarica Model Analysiswith SD9

```
node External_Actors_RollingStocks
 flow
NoTractionInhibition_i : bool ;
ChangeDirectionRequest : bool  ;
ActiveTrain : bool /*: out */;
  Traction : bool /*: out */;
edon

node External_Actors_ATS
/*…*/
edon

node External_Actors_ATO
/*…*/
edon

node ATP_Component_CarborneATP
/*…*/
edon

node ATP_Component_WaysideATP
 flow
TractionRequest : Message_TractionDirection ;
  Traction : Message_TractionDirection  ;
NoDangerToTraction : bool /*: out */;
ActiveTrain : bool ;
 state
Tracting : bool ;
ReversingDirection : bool ;
  Active : bool ;
 sub
  f1 : f1_Acquire_Data;
  f2 : f2_data_processing;
  f3 : f3_data_sending;
 event
TractionInhibitionReceived ;
FailsToAcquireMsg ;
TractionDirectionRequested ;
init
```

5.4.3 Performance Analysis

In order to be able to perform scheduling and performance analysis for component-based models, the later have to be annotated with information describing the timing characteristics (e.g. core execution times and activation frequencies for threads and methods) and the behaviour (e.g. data dependencies and communication protocols between threads). The set of annotations extending a component-based model is called ***Domain Specific Language*** (DSL). In the following, we describe the DSL we have defined extending the component-based models used by Thales Communication. Note that the DSL we developed is based on the MARTE standard.

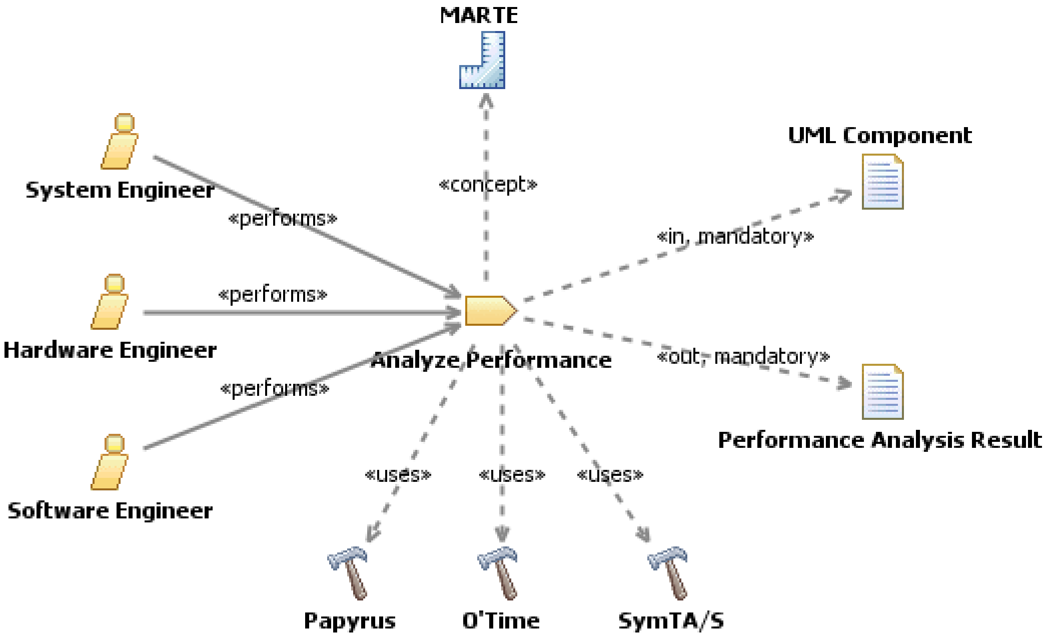


Figure 38: Performance Analysis

Performance Analysis	
Overview	The purpose of this process pattern is to be get performance estimations at early stages of the component-based development process.
Assigned Tickets	#448, #450, #451, #456 and #458
Language	UML, MARTE profile
Tools	Papyrus, O'time, SymTA/S
Parameters	IN: UML Model OUT: Performance analysis results
Pre-condition	None
Post-Condition	None
Steps	Before applying performance analysis to his component-based model, the designer has first to annotate his model with information describing the mapping, the scheduling and the timing characteristics. **Step 1: Threads Scheduling** The user has to select a scheduling policy for each processor. Then, he has to set the scheduling parameters for each thread mapped to a processor. E.g., in case of

static priority preemptive scheduling, the user has to set priorities for the threads.

Step 2: Methods Mapping:

The user has to map the methods to the threads using the methods mapping table. All methods are namely automatically displayed in the left column of the mapping table. Each cell in the right column contains a drop-down list with all available threads. In order to map a method to a thread, the user has simply to select the thread in the method corresponding drop-down list.

Step 3: Timing Characteristics

Timing characteristics needed by the performance analysis have to be set in the thread configuration window and the sequence diagram illustrating the scenario to be analyzed.

Thread internal activation

The internal activation of a given thread is driven by an internal clock which triggers the thread periodically. The internal clock timing characteristics have to be described in the thread configuration window, where the user has to set the a period and eventually a jitter value for the clock.

Thread external activation

The external activation occurs, when an external event (call, signal, data) triggers the root method of the thread. External activation is modeled using input stimuli. Events sent by an input stimulus are either strictly periodic, periodic with jitter or sporadic. A sand timer is used for the graphical representation of an input stimulus. The timing characteristics of the external activating events are set under the sand timer. First, the user has to specify if the external activating event stream is periodic or sporadic. Then, it has to specify the period and jitter value in case of periodic event stream or the minimum inter-arrival distance between events in case of sporadic event stream. In case of sporadic event stream, if no jitter value is set, the jitter is automatically set to 0.

Communication behaviour between methods:

The communication between methods is either synchronous or asynchronous. The communication type is illustrated by the user in the sequence diagram by connecting the caller method and the called method using an arrow: a full arrow for in case of a synchronous communication and a half arrow in case of a asynchronous communication.

Execution times for methods:

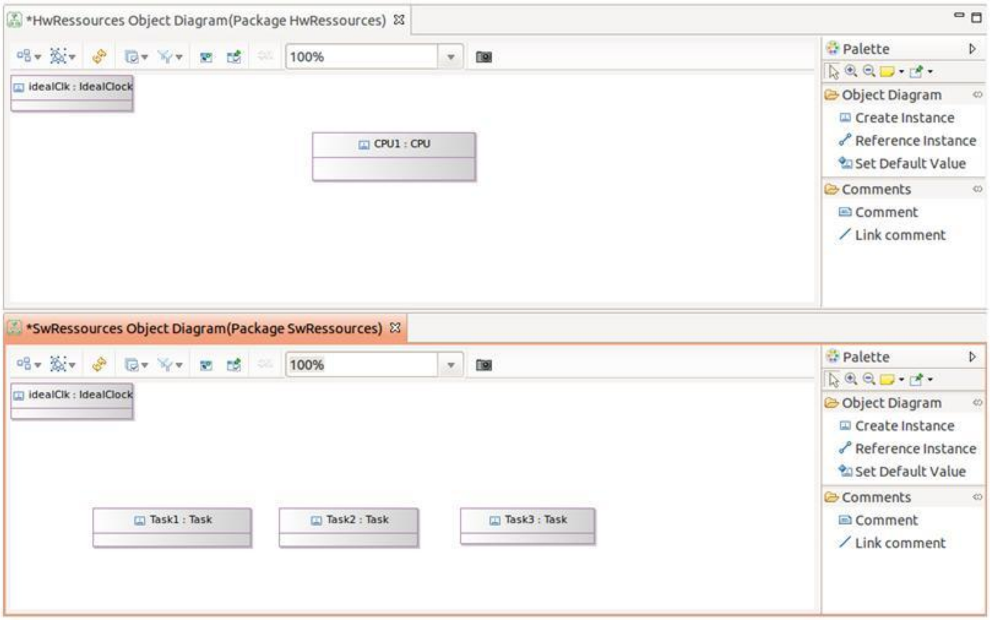
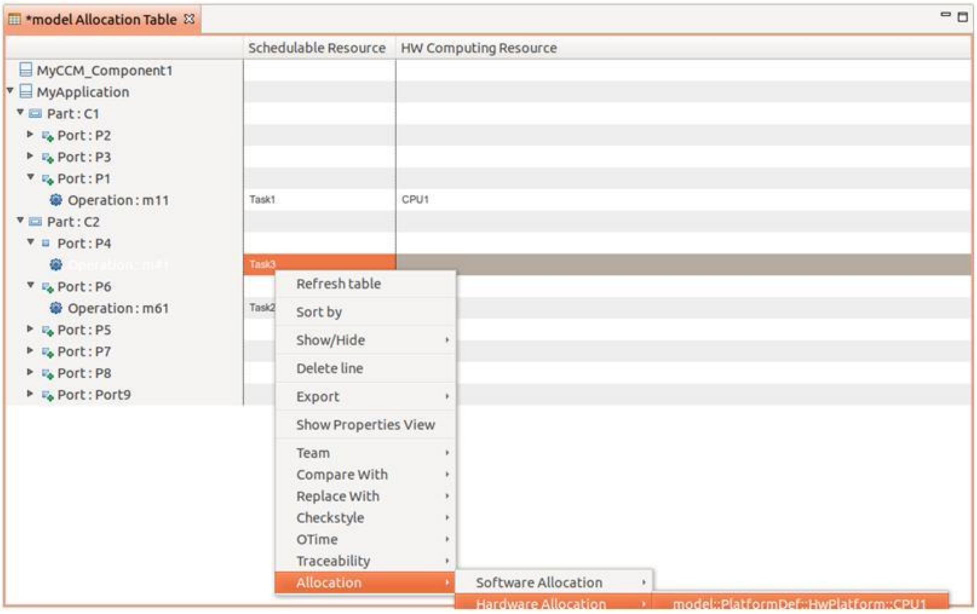
Each method and sub-method is characterized by a core execution time interval which limits its minimum and maximum execution duration. The user has to set the core execution time interval of each method and sub-method at its right in the sequence diagram illustrating the scenario to be analyzed.

Timing requirements:

The user may require to set constraints regarding latencies or jitters at the paths outputs. These timing constraints can be modeled using output stimuli. A sand timer is used for the graphical representation of an output stimulus. The timing constraints are set under the sand timer. In case of a latency constraint, the user has to set a couple (input stimulus, latency interval) under the output stimulus. In case of a jitter constraints, the user has simply to set the jitter value under the output stimulus.

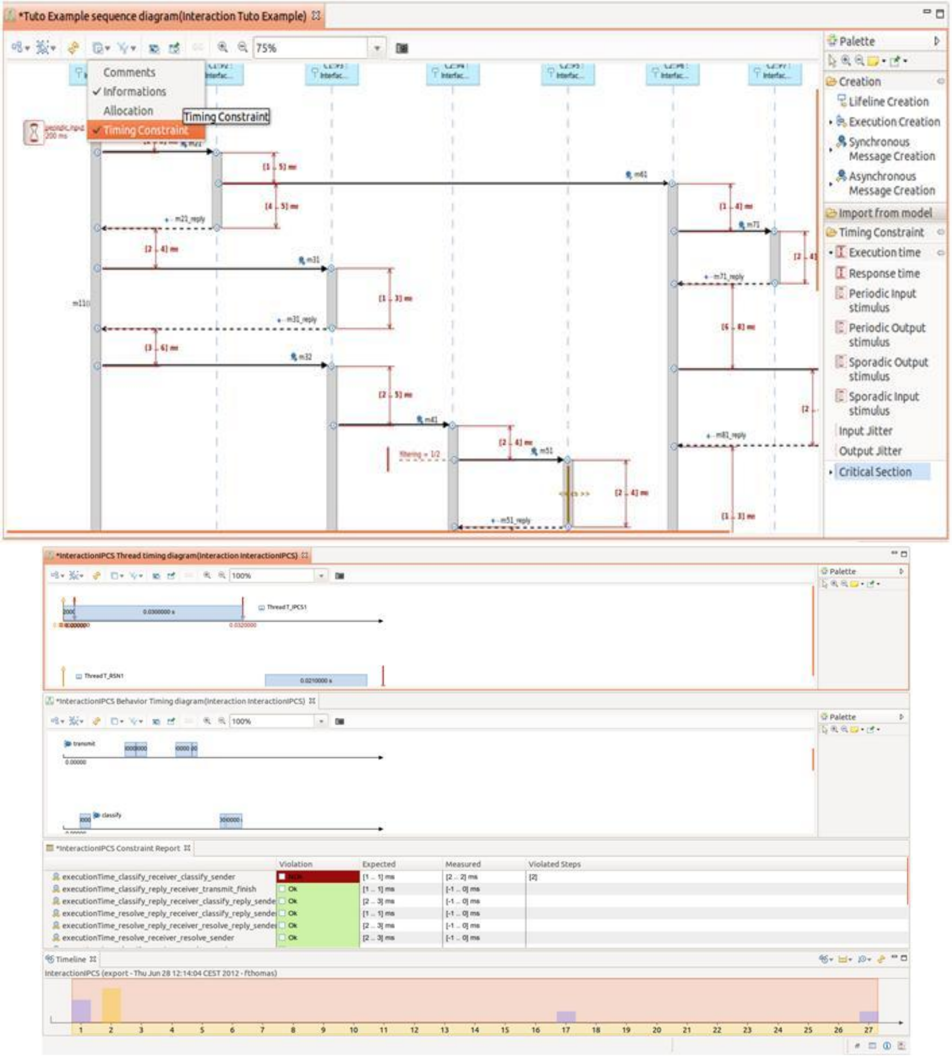
Step 3: Timing Characteristics

The user must launch the performance analysis method in order to get performance estimations.

Example	 

*HwRessources Object Diagram(Package HwRessources)
100%
idealClk : IdealClock
CPU1 : CPU
Palette
Object Diagram
Create Instance
Reference Instance
Set Default Value
Comments
Comment
Link comment
*SwRessources Object Diagram(Package SwRessources)
100%
idealClk : IdealClock
Task1 : Task
Task2 : Task
Task3 : Task
Palette
Object Diagram
Create Instance
Reference Instance
Set Default Value
Comments
Comment
Link comment

*model Allocation Table
Schedulable Resource
HW Computing Resource
MyCCM_Component1
MyApplication
Part : C1
Port : P2
Port : P3
Port : P1
Operation : m11
Task1
CPU1
Part : C2
Port : P4
Operation : m41
Task3
Port : P6
Operation : m61
Task2
Port : P5
Port : P7
Port : P8
Port : Port9
Refresh table
Sort by
Show/Hide
Delete line
Export
Show Properties View
Team
Compare With
Replace With
Checkstyle
OTime
Traceability
Allocation
Software Allocation
Hardware Allocation
model::PlatformDef::HwPlatform::CPU1



*Tuto Example sequence diagram(Interaction Tuto Example)

*InteractionIPCS Thread timing diagram(Interaction InteractionIPCS)

*InteractionIPCS Behavior Timing diagram(Interaction InteractionIPCS)

*InteractionIPCS Constraint Report

	Violation	Expected	Measured	Violated Steps
executionTime_classify_receiver_classify_sender	NOK	[1 .. 1] ms	[2 .. 2] ms	[2]
executionTime_classify_reply_receiver_transmit_finish	Ok	[1 .. 1] ms	[-1 .. 0] ms	
executionTime_classify_reply_receiver_classify_reply_sende	Ok	[2 .. 3] ms	[-1 .. 0] ms	
executionTime_resolve_reply_receiver_classify_reply_sende	Ok	[1 .. 1] ms	[-1 .. 0] ms	
executionTime_resolve_reply_receiver_resolve_reply_sende	Ok	[2 .. 3] ms	[-1 .. 0] ms	
executionTime_resolve_receiver_resolve_sender	Ok	[2 .. 3] ms	[-1 .. 0] ms	

Timeline

InteractionIPCS (export - Thu Jun 28 12:14:04 CEST 2012 - fthomas)

5.4.4 Model Analysis

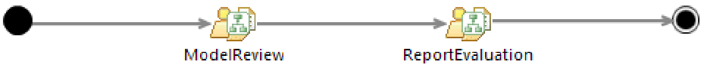


Figure 39: Model Based Analysis Pattern

The ModelAnalysis starts with the definition of the analysis rules. This is dependent on the nature of the tools to be analysed. The rule set could be both generic and predefined or it can be specific for the actual process and project definition.

Model Analysis can be done at multiple places during the development process. Most obvious are places where certain work products are completed and new process steps shall be entered. Model Analysis can help to ensure the quality of the work products.

The analysis of a model can be done in a manual way, in an automatic way or in both ways at the same time. The latter is to be preferred as it combines the benefits of both approaches. So the automatic way can check a high number of rules while the manual analysis can benefit from the experience of the quality engineer who can better adapt to the specifics of the project and the respective work products.

In general the Model Analysis can be separated into ModelReview (either automatic or manual) and ReportEvaluation.

5.4.4.1 ModelReview and ReportEvaluation

The following picture shows the Model Analysis in the Requirements Engineering Phase.

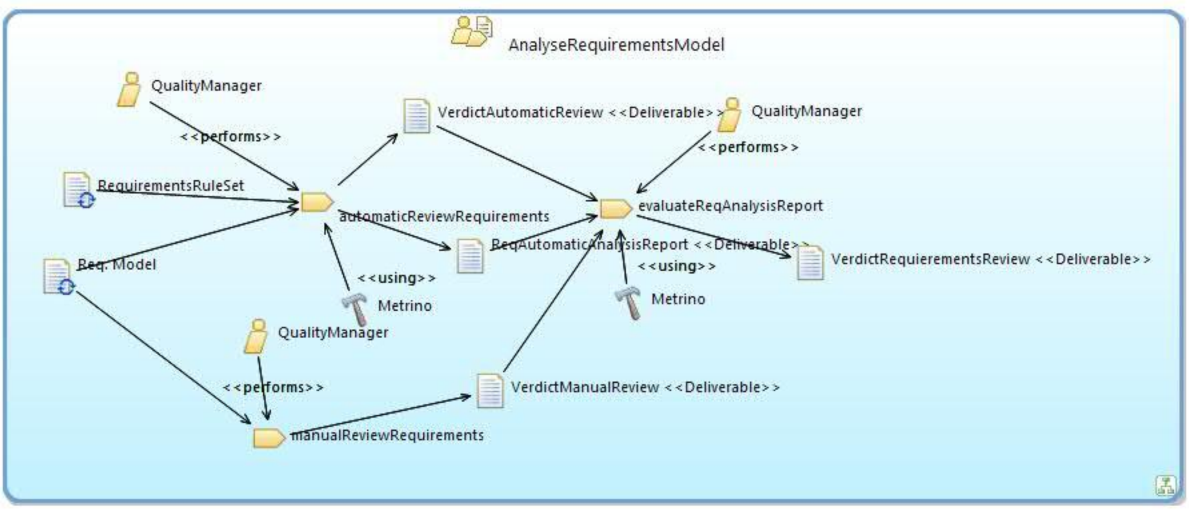
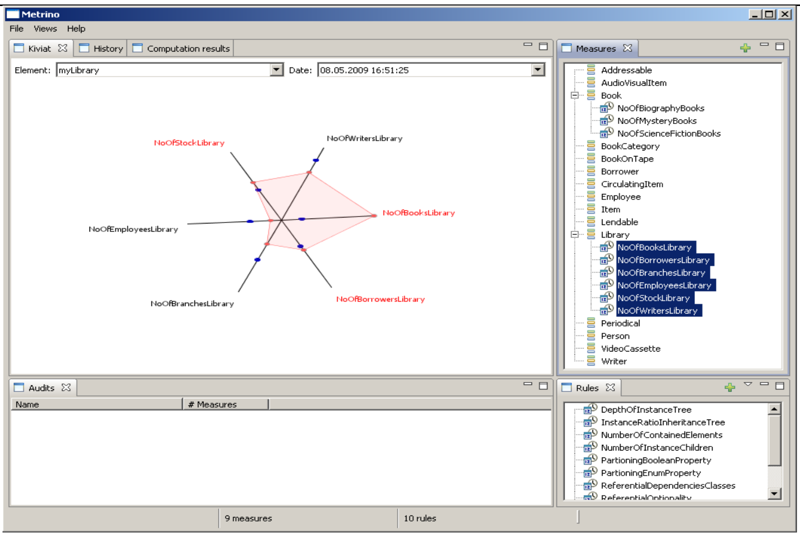

AnalyseRequirementsModel
QualityManager
<<performs>>
RequirementsRuleSet
Req. Model
automaticReviewRequirements
VerdictAutomaticReview <<Deliverable>>
QualityManager
<<performs>>
evaluateReqAnalysisReport
ReqAutomaticAnalysisReport <<Deliverable>>
VerdictRequierementsReview <<Deliverable>>
<<using>>
Metrino
<<using>>
Metrino
QualityManager
<<performs>>
VerdictManualReview <<Deliverable>>
manualReviewRequirements

Figure 40: Model Analysis

The Model Analysis is concerned with several work products.

- Rule set
- Model under analysis
- Automatic Report / Verdict automatic Report
- Manual report / Verdict manual report
- Final verdict

ModelReview and Report Evaluation	
Overview	Analyze Requirements with respect to certain rules
Assigned Tickets	#115, #88, #170,
Language	SMM, OCL, UML for Marte profile
Tools	Metrino
Parameters	n/a
Pre-condition	Analyzed model shall adhere to metamodel definition
Post-Condition	Model is unchanged
Steps	**Automated Review:** This step will use the predefined rules set appropriate for the analyzed model (e.g. RequirementsRuleSet) and a model which is to be analyzed. Both work products have to be loaded into the Metrino tool. After doing so the analysis can be started. As a result the Metrino tool creates a visual output of the analysis in tabular view and in a Kiviat view. The results of the analysis can optionally be persisted and can be used for analysis over a certain period of time. The Metrino tool creates also a report describing the results of the analysis. **Manual Review:** The manual review of a model is done by loading the model in a

	corresponding tool. Optionally the model can be loaded with a reflective editor (e.g. EMF reflective Ecore Editor) which allows browsing through the model as well. Quality Manager performs now a manual review and delivers findings in a report. **Evaluate Analysis Report:** Quality Manager shall now evaluate the automatic and the manual model analysis report and shall give a final verdict whether the analyzed model is conformant to the corresponding rule set.
Example	

Element: myLibrary

Date: 08.05.2009 16:51:25

9 measures

10 rules

5.5 Model based Testing

5.5.1 Modelling of Test Purposes for Black Box Tests

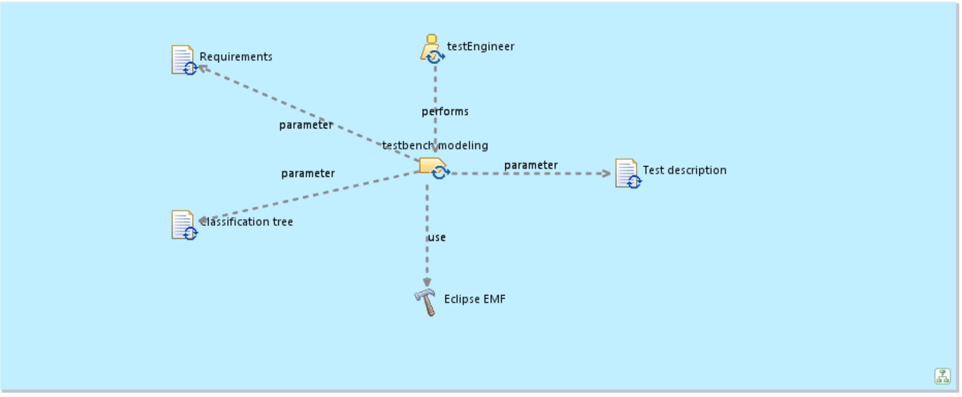
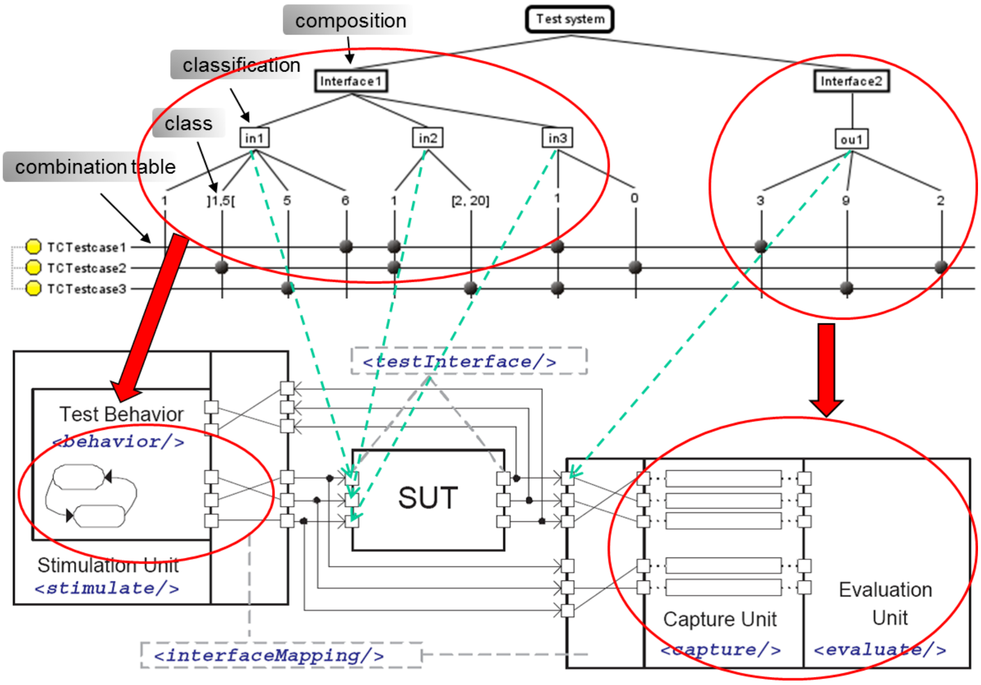


Figure 41: Modelling of Test Purposes for Black Box Tests

Modeling of test purposes for black box tests	
Overview	This pattern describe some of the VERDE modelling approach for black box test by means of classification tree method for embedded systems (CTM/ES) and the use of an interchange format based on TestML. TestML is a tool independent and XML-denoted language that was developed for the interchange of test descriptions. It was originally designed to meet the specific demands of model-based testing of embedded automotive software.
Assigned Tickets	#76, #113, #218, #38, #117, #268, #270-284, #285 #15
Language	XML
Tools	Any tool supporting the standard (full compliance)
Input	Requirements, Classification trees, in A description (.xml)of the test bench including both the System-Under-Test (SUT) and elements referring to the individual components of the test environment, out
Pre-Conditions	
Post-Conditons	
Steps	Test cases (horizontal lines in CTM) are defined in a table of possible combinations based on the classification tree method for embedded systems (CTM/ES) as depicted in Figure 42. Interfaces (compositions) of the SUT are represented as tree, classifications represent the individual inputs/outputs and classes are used for the specification of the value ranges. The VERDE test-bench modelling process covers different test stages from the module to integration and system tests as well as test levels from Model-In-the-Loop (MIL) to Hardware-In-the-Loop (HIL). The basis of the VERDE test-bench interchange format is a self-contained language definition that makes it possible to cover test descriptions at different levels of abstraction independent from the respective tool environment. Such a test system consists of the following components: • **System under test (SUT):** represents the system that is to be tested. Mainly relevant for TestML is its test interface. From the perspective of

	TestML, the SUT itself disappears behind the test interface. - **Stimulation Unit:** This unit is responsible for the generation of test stimuli. The actual test execution takes place here. - **Capture Unit:** This records the system reactions and/or the test stimuli. Therefore this unit can be used for the generation and representation of execution traces and is in the focus of working task 4.5 within the VERDE project. - **Evaluation Unit:** This unit is responsible for the evaluation of test cases. It accesses all data and execution traces recorded by the capture unit and can be operated temporally independent from the stimulation unit. Specification of stimulation, recording, and evaluation is undertaken, with a strict conceptual separation, by the elements stimulate, capture and evaluate. See chapter 5, section 5.1 of deliverable F5.1.1 for further details.
Example	 composition Test system classification Interface1 Interface2 class in1 in2 in3 ou1 combination table 1]1,5[5 6 1]2, 20] 1 0 3 9 2 TCTestcase1 TCTestcase2 TCTestcase3 <testInterface/> Test Behavior <behavior/> SUT Stimulation Unit <stimulate/> Evaluation Unit Capture Unit <capture/> <evaluate/> <interfaceMapping/> **Figure 42: Classification Tree method and TestML**

5.5.2 Modelling of Test Purposes for Compositional Testing

Components involved in the model of a system can be designed and implemented specifically or simply reused out of existing contexts. For designers, bridging the gap between abstract specifications of systems and concrete executions of components can be challenging: choosing or implementing components and communication architectures to design the system requires anticipating the result of all possible component interactions in order to insure they stay within the system requirements. The task is hard because the number of possible interactions is often huge or even infinite in a purely reactive case.

Verde deliverable F5.4.2 proposes compositional testing techniques where such combinatory explosion can be avoided. Test purposes are basically captured by a model of intended interactions between the components of a system, only based on the knowledge of component interfaces and timing constraints expressed at the system level. From such behavioural system specifications, it this then possible to determine how intended interactions constrain the behaviours of components. The result is a set of constraints on component executions elicited from the system intended behaviours. Such constraints can then be used as behavioural specifications to guide the design or the choice of components to be plugged in the system, within a refinement process.

This process pattern focuses on the modelling rules for the specification of the interaction model used by the techniques described in Verde deliverable F5.4.2. This interaction is concretely modelled via a sequence diagram, enriched with timing and data constraints.

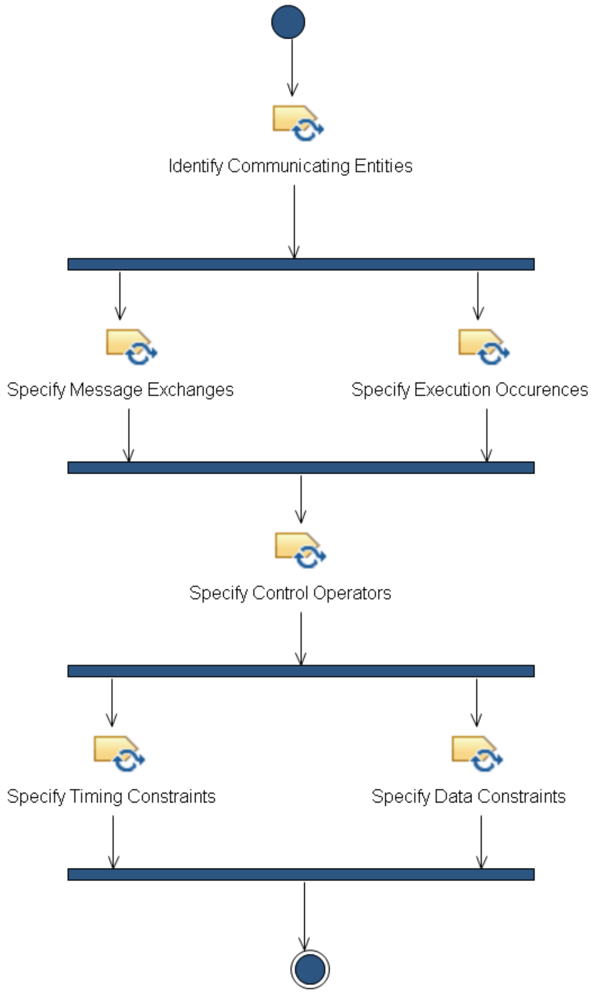

Identify Communicating Entities
Specify Message Exchanges
Specify Execution Occurences
Specify Control Operators
Specify Timing Constraints
Specify Data Constraints

Figure 43: Modelling of Test Purposes for Compositional Testing

Modelling Test Purposes for Compositional Testing	
Overview	Build a UML/MARTE interaction model, suited to compositional testing techniques of deliverable F5.4.2.
Assigned Tickets	#76, #113, #218, #38, #117, #268, #270-284, #285, #15
Language	UML, MARTE
Tools	MDT Papyrus, Optimum
Parameters	- in component : Component [1]

	- out interactionModel : Interaction [1]
Pre-condition	The following hypothesis apply on the input component model: - The input component contains parts. - The components typing these parts (as well as the input component) can only have atomic flow ports (i.e., no client/server ports).
Post-condition	None
Steps	**Identify Communicating Entities:** The purpose of the interaction model is to specify the messages that are expected to flow between the ports of interacting components. In this step, each port involved in the exchange of messages is therefore modeled with a dedicated lifeline. **Specify Message Exchanges:** According to semantic hypothesizes underlying testing techniques described in F5.4.2, only asynchronous message exchanges are supported. This kind of message is capture by tracing a Message between the source and target lifelines, and by setting property “messageSort” to “asynchCall”. Note that the directions associated with atomic flow ports (represented in the interaction by lifelines) constrain possible sources and targets of messages. Messages can flow only from an “out” (or “inout”) atomic flow port towards an “in” (or “inout”) atomic flow port. In addition, the source and target ports must be type compatible. Note that the methodology does not put particular requirements on the presence or absence of connectors in the structure of the input component. In the case where connectors are actually specified, they should however be considered as additional constraints on possible message exchanges: Messages can only flow between interconnected ports. **Specify Execution Occurrences:** As explained in the introduction to this pattern specification, interacting components are considered as black boxes. Nevertheless, in order to model the expected behaviour of the overall system, it may be necessary to capture some abstract representation of the computations that are intended to happen inside each individual component. This kind of computation is represented as a BehaviourExecutionSpecification on a Lifeline. The property “Behaviour” of BehaviourExecutionSpecification can then be used to reference a particular OpaqueBehaviour. This OpaqueBehaviour must be owned by the component typing the port represented by the lifeline. The tool chain described in F5.4.2 supports the following assignment syntax: OPAQUE_BEHAVIOR_BODY = < <PORT_NAME | PROPERTY_NAME> ‘=‘ <VSL_EXPRESSION> > | <NEW_VALUE_EXPRESSION> NEW_VALUE_EXPRESSION = ‘new(‘ <PORT_NAME | PROPERTY_NAME> ‘)’ The usage of operator “new” simply denotes that, during the execution of the behaviour, a new value has been assigned (respectively sent) to the operand property (respectively on the operand port). No particular assumptions are made about the new value (i.e., it can be the same as the former value). Note that, according to step 1 (Identify Communicating Entities), the modelled

interaction only contains lifelines for ports, not for parts associated with these ports. Having the BehaviourExecutionSpecification on lifelines representing ports should be seen as a shorthand notation for an interaction where lifelines associated with parts would also be modelled, and the BehaviourExecutionSpecifications would be put onto these lifelines.

Note also that a BehaviourExecutionSpecification, as a kind of ExecutionSpecification, owns a “start” and “finish” OccurenceSpecification. These occurrence specifications can then be used to put constraints on the denoted executions, like in the two last step of this process pattern.

Specify Control Operators:

In addition to Messages and BehaviourExecutionSpecifications, compositional testing techniques of F5.4.2 also support particular kinds of CombinedFragment. In a UML interaction, a CombinedFragment enables to specify control structures around interaction fragments (Note: BehaviourExecutionSpecification as well as MessageOccurenceSpecification indirectly inherit from InteractionFragment. The metaclassMessageOccurenceSpecification is used to denote emission / reception events underlying a Message).

The precise nature of the control structure represented by a CombinedFragment is denoted by its property “interactionOperator”. Fragments playing the role of operands for the “interactionOperator” are denoted by the property “operand” of CombinedFragment. UML defines several interaction operators. However, according to semantic hypothesis underlying testing techniques of F5.4.2, only the three following ones are supported:

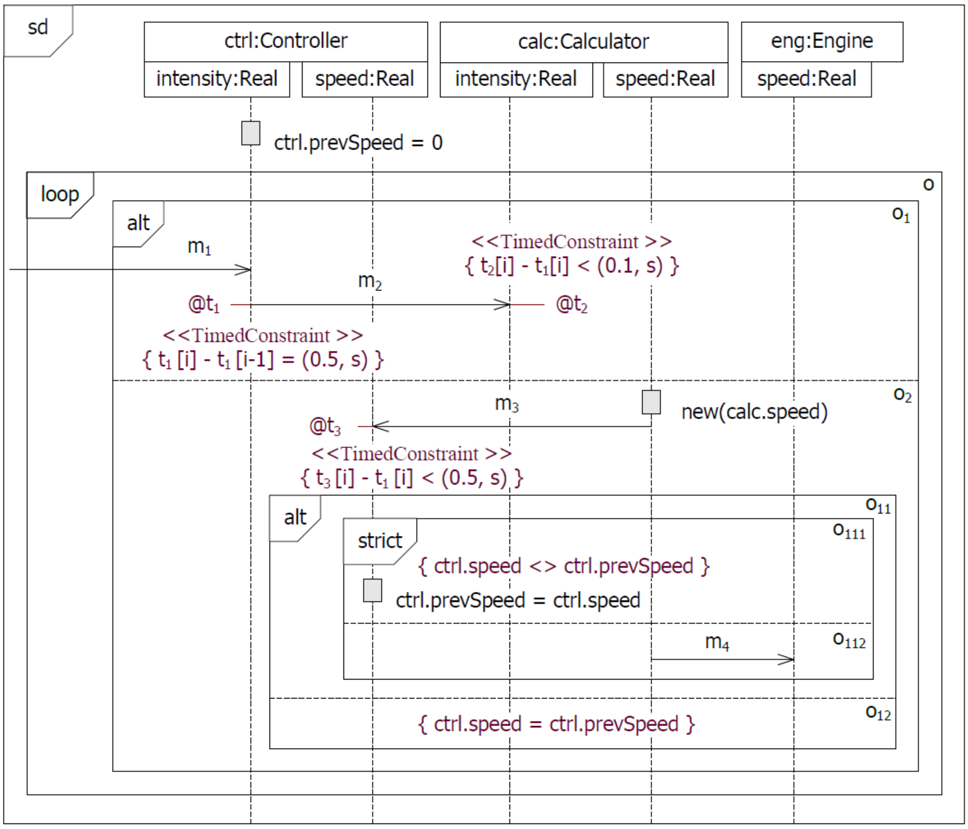
- *loop*, which represents a repetition of an execution scheme (where an execution scheme is composed of a set message exchanges and behaviour executions),
- *alt*, which represents a non-deterministic choice among a set of possible execution schemes,
- *strict*, which can be used to identify instants when all lifeline behaviours are forced to leave execution.

Specify Timing Constraints:

The resulting interaction model can be enriched with explicit timing constraints. These constraints rely on the usage of VSL and time observations, and they must be annotated with stereotype «TimedConstraint» from the MARTE profile. The rules for modelling time constraints using VSL are given in pattern [PUT REF: Expressing timing constraints with time observations]. Note that time constraints can be associated with occurrence specifications or messages. Concerned occurrence specifications or messages must be identified through the property “constrainedElement” of the Constraint. In the case of a TimeConstraint, occurrence specifications are denoted using the name of a TimeObservation referencing the occurrence specification (The reference is given by property *event* of metaclassTimeObservation). In the case where a time constraint relates to occurrence specifications directly or indirectly part of a loop (i.e., combined fragment where *interactionOperator* equals *loop*), these time observations can be indexed (e.g. @*t[i]*), where the index refers to the time observation in particular iteration of the loop.

Specify Data Constraints:

In addition to timing constraints, interaction models can also be enriched with data constraints. The modelling rules for capturing these constraints are similar to those used for timing constraints, except that underlying VSL expressions are not allowed to handle time observations. The only “variable” that can be used are either ports or properties. Note that data constraints can be associated with occurrence specifications. Concerned occurrence specifications must be identified

	through the property “constrainedElement” of the Constraint. Semantically, data constraints associated with reception occurrence specifications are meant to be evaluated after the reception of the message.
Example	The following example illustrates the application of the various modelling steps described in this section. The example is extracted from F5.4.2, and represents the expected behaviour of a rain-sensor wiper controller in a car.  sd ctrl:Controller intensity:Real speed:Real calc:Calculator intensity:Real speed:Real eng:Engine speed:Real ctrl.prevSpeed = 0 loop o alt o_1 m_1 <<TimedConstraint >> { $t_2[i] - t_1[i] < (0.1, s)$ } m_2 @t_1 @t_2 <<TimedConstraint >> { $t_1 [i] - t_1 [i-1] = (0.5, s)$ } o_2 m_3 new(calc.speed) @t_3 <<TimedConstraint >> { $t_3 [i] - t_1 [i] < (0.5, s)$ } alt o_{11} strict o_{111} { ctrl.speed <> ctrl.prevSpeed } ctrl.prevSpeed = ctrl.speed m_4 o_{112} o_{12} { ctrl.speed = ctrl.prevSpeed } The system contains a controller *ctrl* which receives rain intensity values from a sensor (not depicted in the diagram) on its port *intensity* through message *m1*. Every 0.5 seconds, the received value is resent to a calculator component *calc* whose main purpose is to compute a speed for the wiper depending on the rain intensity (message *m2*). The frequency is identified by means of the constraint *t1[i]-t1[i-1]=(0.5,s)* (where *s* means seconds).

5.5.3 Test Generation for Black Box Tests

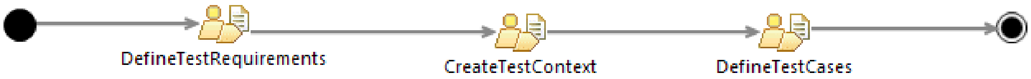


Figure 44: TestGeneration Black Box Testing

The Test Generation BlackBox Testing starts with the definition of the test requirements. These requirements reflect the system requirements in a way which helps to model what should be tested and for what reason. Next step is the creation of a test context. A test context contains the test configuration, test components, system under test. Last step is the definition of test cases within the test context.

5.5.3.1 Define Test Requirements

The following picture shows the Definition of Test Requirements.

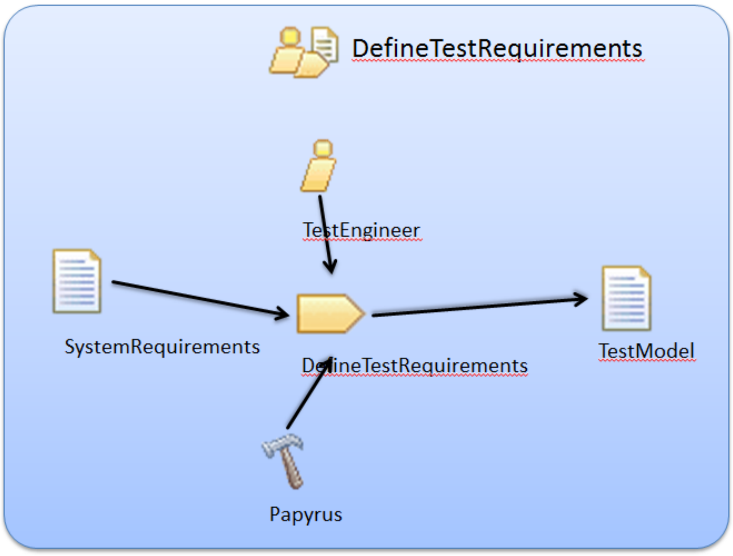
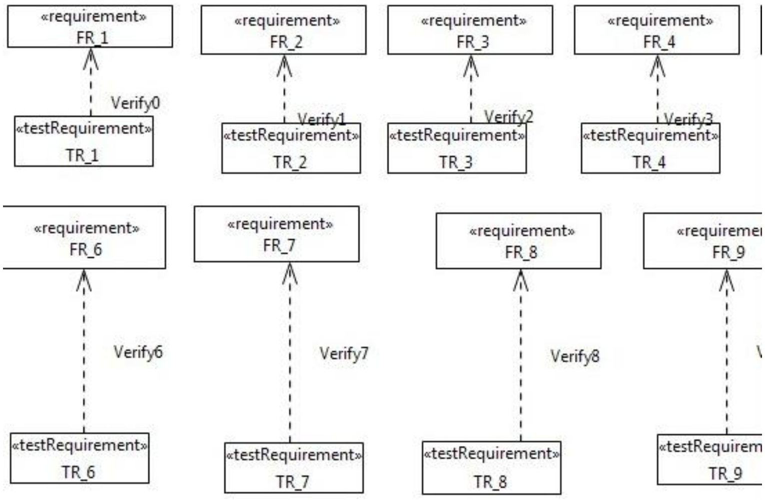


Figure 45: Define Test Requirements

Define Test Requirements is concerned with the following work products.

- System Requirements Model
- Test Model

ModelReview and Report Evaluation	
Overview	Test Engineer starts to work out the test requirements
Assigned Tickets	#47, #69, #113, #160, #163, #222, #15, #38, #40, #117, #242, #243, #268, #270-287, #109, #12, #41, #87, #111, #159, 164, #338, #339, #340, #350,
Language	UML-Profile for Testing, UML for Marte profile, SysML
Tools	Papyrus (or any other UML-Compliant tool)
Parameters	n/a
Pre-condition	Sound system requirements definition
Post-Condition	Testmodel with test requirements
Steps	**Create Test Model:** First of all a test model needs to be created. This test model may optionally refer to the system model in order to reuse the system model type system. Alternatively, the test model can be created inside the system model but in this case it needs to be put into a separated package in order to keep separation between test and system relevant model element. The package needs to be stereotyped with <<testPacakge>>. **Create Test Requirements**: Test requirements needs to be created in order to control the creation and execution of the tests. Test requirements shall refer to the system requirements. The stereotype <<testRequirement>> shall be used.

Example	 «requirement» FR_1 «requirement» FR_2 «requirement» FR_3 «requirement» FR_4 Verify0 Verify1 Verify2 Verify3 «testRequirement» TR_1 «testRequirement» TR_2 «testRequirement» TR_3 «testRequirement» TR_4 «requirement» FR_6 «requirement» FR_7 «requirement» FR_8 «requireme FR_9 Verify6 Verify7 Verify8 «testRequirement» TR_6 «testRequirement» TR_7 «testRequirement» TR_8 «testRequirem TR_9

5.5.3.2 Create Test Context

The following picture shows the creation of Test Context.

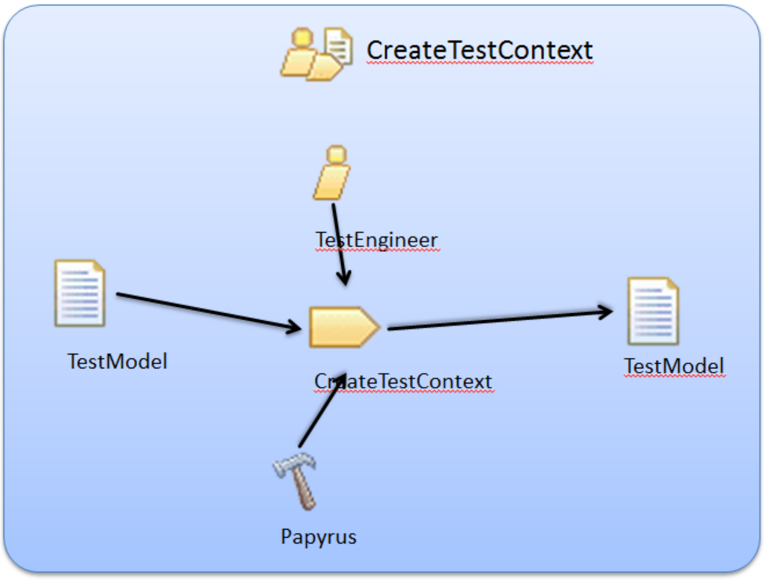
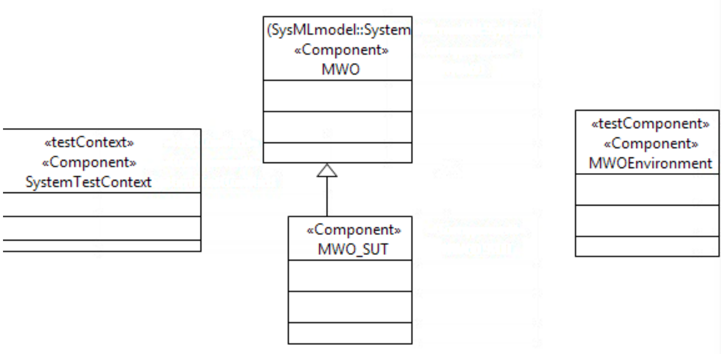


Figure 46: CreateTestContext

Create Test Context is concerned with the following work products.

- Model
- Test Model

CreateTestContext	
Overview	Create the basic elements of a Test Context
Assigned Tickets	#47, #69, #113, #160, #163, #222, #15, #38, #40, #117, #242, #243, #268, #270-287, #109, #12, #41, #87, #111, #159, 164, #338, #339, #340, #350,
Language	UML-Profile for Testing, UML for Marte profile, SysML
Tools	Papyrus (or any other UML-Compliant tool)
Parameters	n/a
Pre-condition	
Post-Condition	Test context created
Steps	**Creation of Test Context:** The test context is a component with stereotype <<testContext>>. It contains test configuration, test components and the system under test. **Creation of SUT:** The SUT (System Under Test) needs to be modeled. It should reflect the parts of the system model that shall be tested. The modelling of the SUT can be either done from scratch or it can be done by creating a new component, which inherits from the relevant system component or components. **Creation of Test Components**: Corresponding to the definition of the SUT the Test components (which acts as environment for the SUT) needs to be modeled. This could be either done from scratch or it can be realized by inheriting from the system components. The Test components needs to be stereotyped <<testComponent>>.
Example	 «testContext» «Component» SystemTestContext (SysMLmodel::System «Component» MWO «Component» MWO_SUT «testComponent» «Component» MWOEnvironment

5.5.3.3 Define Test Cases

The following picture shows the Definition of Test cases.

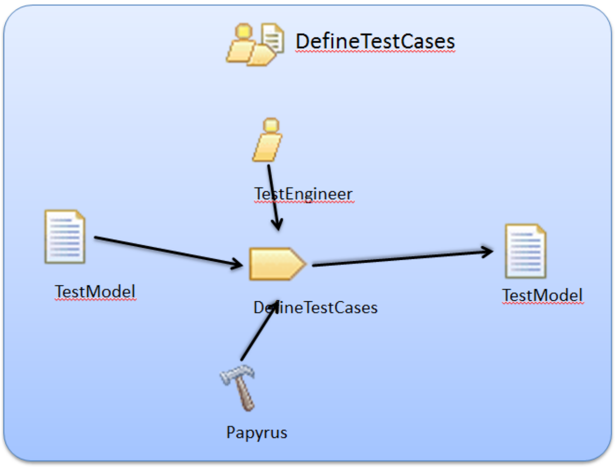
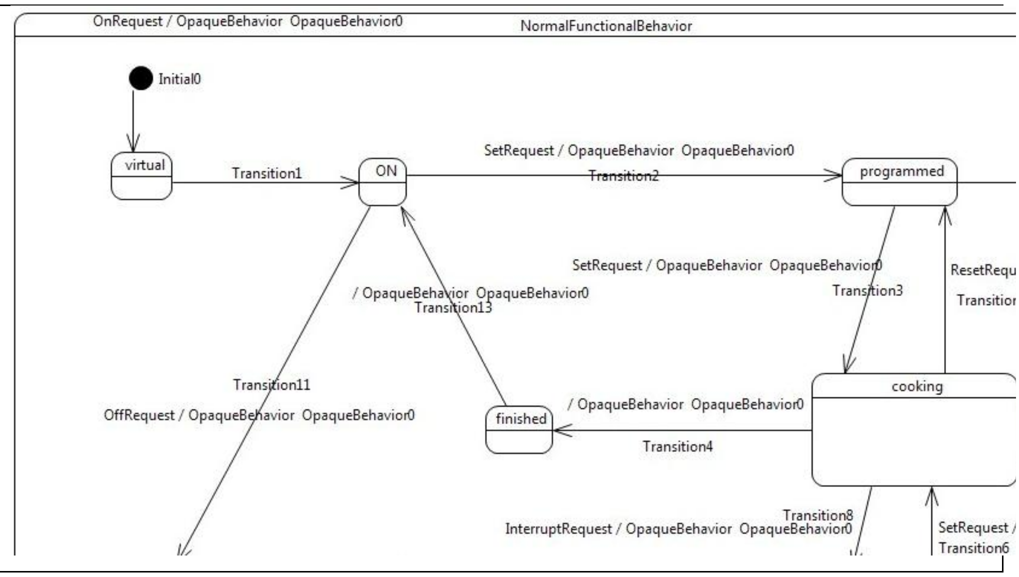


Figure 47: Define Test Cases

Define Test Requirements is concerned with the following work products.

- Test Model

DefineTestCases	
Overview	Definition of Test cases
Assigned Tickets	#47, #69, #113, #160, #163, #222, #15, #38, #40, #117, #242, #243, #268, #270-287, #109, #12, #41, #87, #111, #159, 164, #338, #339, #340, #350,
Language	UML-Profile for Testing, UML for Marte profile, SysML
Tools	Papyrus (or any other UML-Compliant tool)
Parameters	n/a
Pre-condition	
Post-Condition	
Steps	**Manual Definition of Test Cases:** Test cases can be create manually by using sequence diagrams. This is an essential part of the creation of test campaigns as the automatic derivation of test cases shall be supported by the experience and manual investigation of a test engineer. **Definition of Test Behaviour**: In order to derive test cases automatically the behaviour of the system under test needs to be modeled. This shall be done in a manual step and should not depend on the behaviour modeled for the system in the system model. In this way the system model and test model remain separated and systematic errors done in the system model are not propagated to test model.

Example	

OnRequest / OpaqueBehavior OpaqueBehavior0
NormalFunctionalBehavior
Initial0
virtual
Transition1
ON
SetRequest / OpaqueBehavior OpaqueBehavior0
Transition2
programmed
SetRequest / OpaqueBehavior OpaqueBehavior0
Transition3
ResetRequ
Transitior
/ OpaqueBehavior OpaqueBehavior0
Transition13
Transition11
OffRequest / OpaqueBehavior OpaqueBehavior0
finished
/ OpaqueBehavior OpaqueBehavior0
Transition4
cooking
Transition8
InterruptRequest / OpaqueBehavior OpaqueBehavior0
SetRequest /
Transition6

5.6 Implementation / Deployment

5.6.1 Allocation of Functional Components on the Platform

Assigned tickets: #56, #58, #76, #218, #264, #10, #92, #142, #144, #451, #452, #453, #454, #455, #95

The allocation of components relies on UML abstractions with stereotype *allocate* from MARTE. Several ways of allocating components can be considered:

- allocating operations of component ports on execution resources, in case of client server ports,
- allocating component ports on execution resources,
- allocating whole componentinstances on execution resources,
- allocating component instances directly to memory spaces.

In all cases a UML abstraction needs to be defined with the UML property that represents the instance of the execution resource as the target of the abstraction.

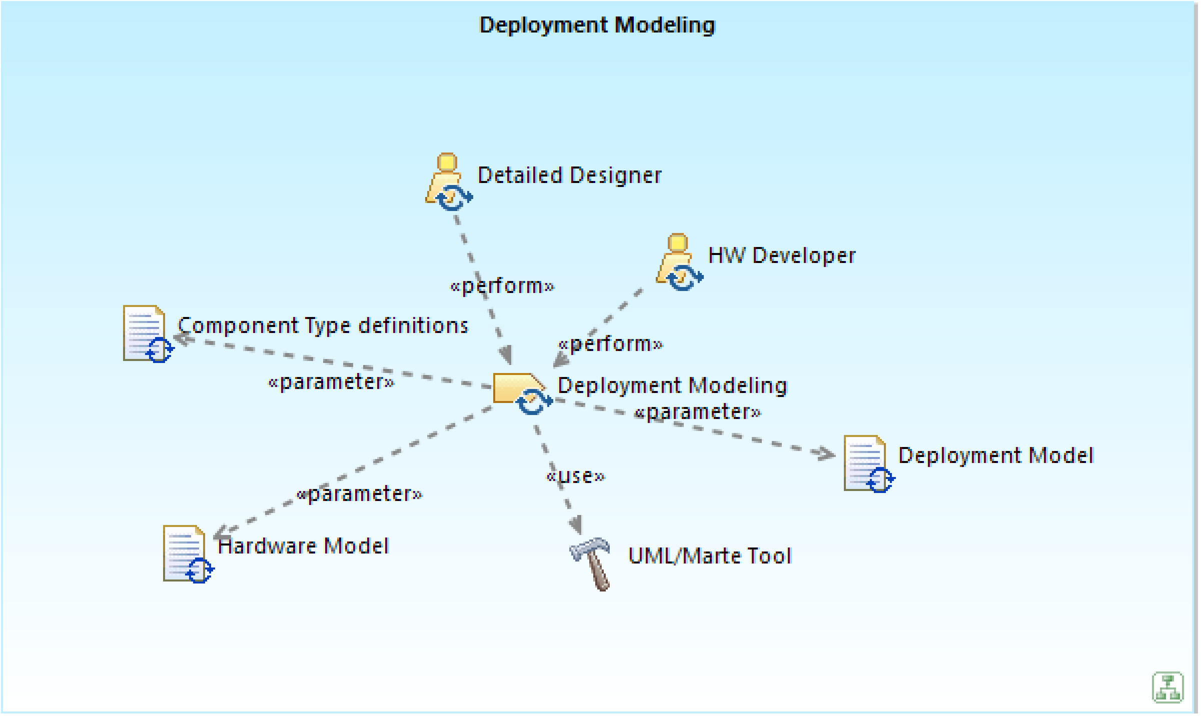


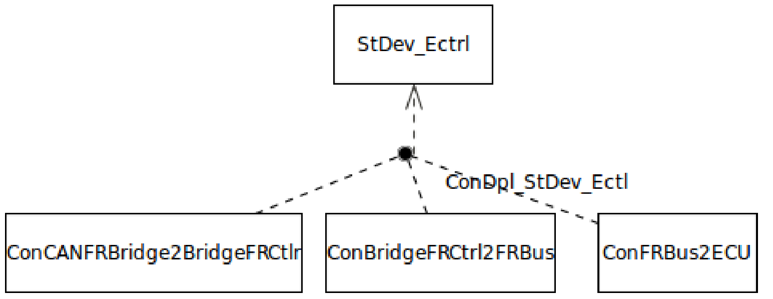
Figure 48: Deployment of Components

Deployment of Components	
Overview	Definition on how functional components are deployed on Hardware
Assigned Tickets	#56, VERDE shall support partitioning (hardware / software and digital / analog) #58, hardware / software integration to one system #95VERDE shall offer NF properties comparison with respect to several different Sw/Hw allocations #451, VERDE shall offer performance properties comparison with respect to several different Sw/Hw allocations #452, VERDE shall offer power/energy properties comparison with respect to several different Sw/Hw allocations

	#453,VERDE shall offer reliabiliry properties comparison with respect to several different Sw/Hw allocations #454, VERDE shall offer NF properties optimization with respect to several different Sw/Hw allocations. #455, VERDE shall offer execution time and power properties comparison with respect to several different Sw/Hw allocations.
Language	UML for Marte profile, VERDE profile definition
Tools	Papyrus MDT or any tool supporting the standard (full compliance)
Parameter	UML Class Diagram with Component Type definitions, in SysML Block Diagram describing the Hardware Baseline, in UML Composite Structure Diagram showing the assignment of functional components to hardware nodes, out
Pre-Conditions	Functional components and Hardware Baseline is defined
Post-Conditions	Composite Structure Diagrams showing the deployment
Steps	**Allocation of a client server port**: Several elements must be set as the sources of the abstraction: the component port, the operation (defined in a UML interface) and the UML property that corresponds to the component instance. It is mandatory to include the component instance in the sources of the abstraction, as UML ports are associated with component declarations, not with component instances. Without specifying the component instance would then lead to allocate the port of all component instances to the specified execution resource. **Allocation of a component port**: Set the UML port and the UML property as sources of the abstraction. For a client server port, this will mean that all of the operations provided by the port will be controlled by the execution resource. For a flow port, this will mean that the reception of messages in this port is controlled by the execution resource. **Allocation of a component instance**: Draw an abstraction arrow from the UML property of the component instance to the UML property of the execution resource instance. In this situation, all ports of the components will be controlled by the execution resource, i.e., users are not allowed to allocate components to more than one execution resource. **Allocation of a component instance directly to memory spaces**: The allocation of a component instance to a memory partition does not imply any execution resource (thread). Thus, it is to be used if a given component instance is passive (i.e. not driven by any thread), or if the definition of execution resources is performed in a separate process.
Example	

5.6.2 Functional Connector Deployment

Deployment of Functional Connectors onto Hardware Connectors	
Overview	If a dedicated model of the hardware (without managing middelware) is used in the model, UML connectors in the functional software component model (in the following called “logical connector”) have to be deployed on UML connectors in the hardware model (in the following called “physical connector”). This is done in a

	UML class digram.
Assigned Tickets	hardware / software integration to one system (Ticket 58) VERDE shall offer different views (Ticket 76) VERDE shall support partitioning (Ticket 56)
Language	UML for Marte profile, SysML
Tools	Any tool supporting the standard (full compliance)
Parameters	Hardware model in SysML, in Functional model in UML/MARTE, in Deployed hardware/software model, out
Pre-Conditions	Functional components and Hardware Baseline is defined, functional components are deployed onto hardware components (SysML blocks)
Post-Conditions	Composite Structure Diagrams showing the deployment of functional components and functional connectors onto one or more physical connectors.
Steps	**Identify logical connectors:** Since a logical connector usually connects software components which are deployed on different hardware components and which are not directly connected in the hardware model, a logical connector must usually refined by multiple physical connectors. **Logical connector deployment:** A dependency relation is used between first physical connector (starting from the hardware component on which the software component is deployed) and logical connector. For the following physical connectors on the path from the source component to the target component (on which the target software component of the logical connector is deployed) in the hardware model, UML/MARTE dependency branches are used to refer to same dependency. Note that the order of dependency branches reflects the connector path in the hardware model between source and target hardware component.
Example	 StDev_Ectrl ConDpl_StDev_Ectl ConCANFRBridge2BridgeFRCtlr ConBridgeFRCtrl2FRBus ConFRBus2ECU

5.6.3 Code Generation for Component Deployment

This methodological step consists in applying code generation tools to the deployment information. Deployment model typically address the following elements: component instances, component connections, execution resource allocations. Component port definitions are not in the scope of the deployment model; however, they are necessary to correctly generate the deployment code.

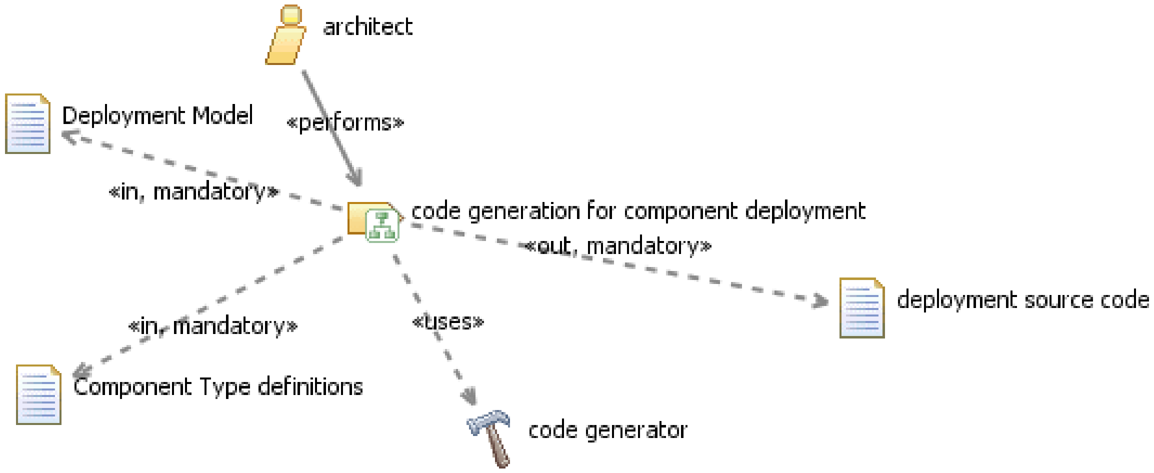


Figure 49: Code Generation for Component Deployment

Code generation for component deployment	
Overview	Generation of the source code that manages the component deployment.
Assigned Tickets	#84 VERDE shall support compilation, debugging and analysis #60 support functional and non functional aspects #64 different task priorities and scheduling schemes
Language	Verde modelling language (corresponding to the Marte profile)
Tools	Code generator, e.g. eC3M, MyCCM
Parameter	in: UML composite structure diagrams for the deployment information in: class diagrams for component definitions out: source code for component deployment
Pre-Conditions	the deployment of the components is defined (connections, resource allocations). This implies that the components are defined also.
Post-Conditions	the deployment source code is generated
Steps	There is one single step that consists in launching the deployment code generator.
Example	

5.6.4 Code Generation for Component Implementation

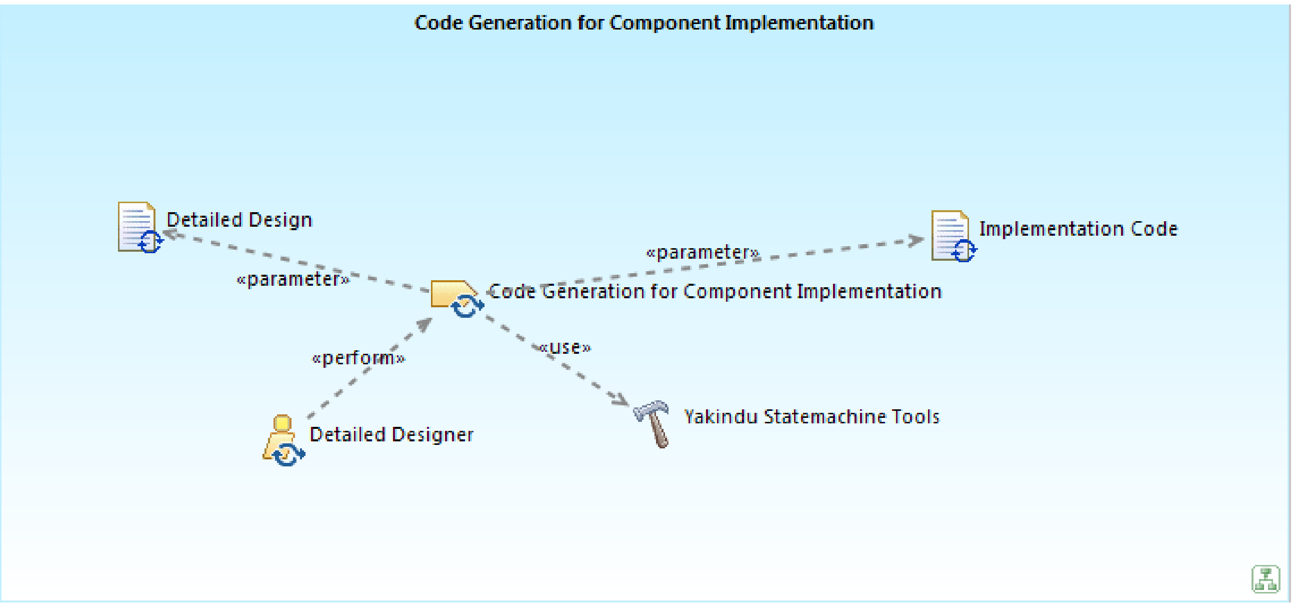
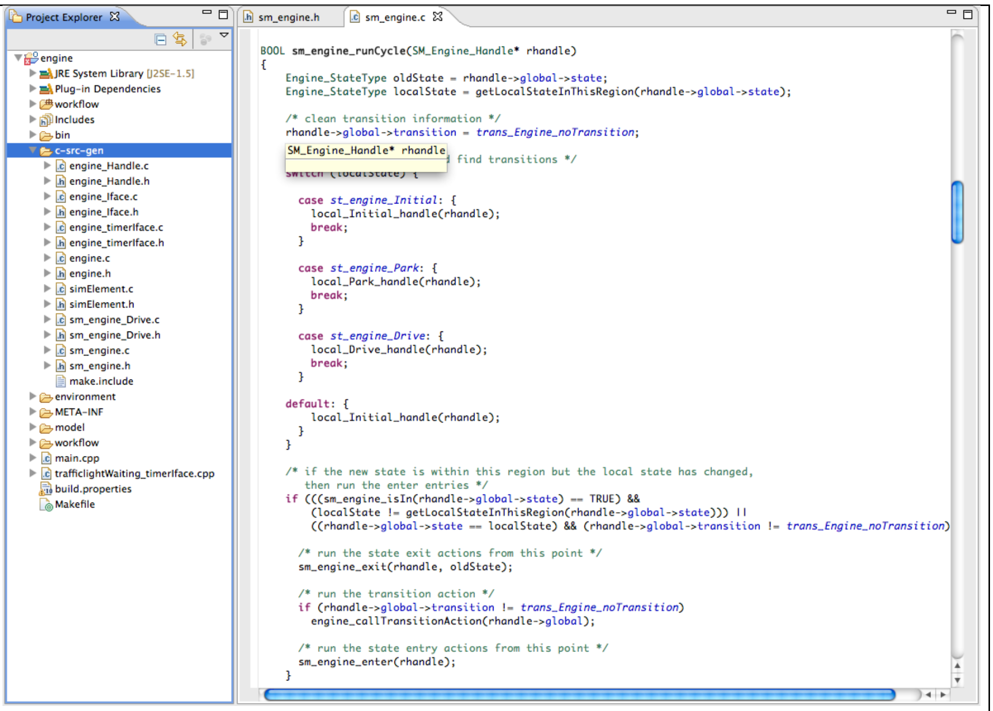

Code Generation for Component Implementation
Detailed Design
«parameter»
Implementation Code
«parameter»
Code Generation for Component Implementation
«perform»
«use»
Detailed Designer
Yakindu Statemachine Tools

Figure 50: Code Generation for Component Implementation

Code Generation for Component Implementation	
Overview	The detailed design model is used to simulate the system. The next step is to convert with a linear sequence of instructions to source code.
Assigned Tickets	
Language	Yakindu SM Metamodel, Simulink Metamodel
Tools	Yakindu Statemachine Tools, Matlab/Simulink
Parameters	Detailed Design Model, in Implementation Code, out
Pre-Conditions	
Post-Conditions	
Steps	**Generate Code:** While generating source code we use the code generators shipped with the modelling tools we are using. Please look into the manuals of the YakinduStatechart Tools and Matlab/Simulink to get detailed instructions on how to generate code.
Example	**Generate Code:** The following screenshot shows the Eclipse Project Explorer with the c-src-gen folder including the c files generated from a YakinduStatemachine and the source code editor including a state machine implementation.

```
BOOL sm_engine_runCycle(SM_Engine_Handle* rhandle)
{
    Engine_StateType oldState = rhandle->global->state;
    Engine_StateType localState = getLocalStateInThisRegion(rhandle->global->state);

    /* clean transition information */
    rhandle->global->transition = trans_Engine_noTransition;

                             find transitions */
    switch (localState) {

      case st_engine_Initial: {
        local_Initial_handle(rhandle);
        break;
      }

      case st_engine_Park: {
        local_Park_handle(rhandle);
        break;
      }

      case st_engine_Drive: {
        local_Drive_handle(rhandle);
        break;
      }

    default: {
        local_Initial_handle(rhandle);
      }
    }

    /* if the new state is within this region but the local state has changed,
       then run the enter entries */
    if (((sm_engine_isIn(rhandle->global->state) == TRUE) &&
        (localState != getLocalStateInThisRegion(rhandle->global->state))) ||
        ((rhandle->global->state == localState) && (rhandle->global->transition != trans_Engine_noTransition)

      /* run the state exit actions from this point */
      sm_engine_exit(rhandle, oldState);

      /* run the transition action */
      if (rhandle->global->transition != trans_Engine_noTransition)
        engine_callTransitionAction(rhandle->global);

      /* run the state entry actions from this point */
      sm_engine_enter(rhandle);
    }
```

5.7 Execution and Simulation

5.7.1 Execution of Tests on the Platform

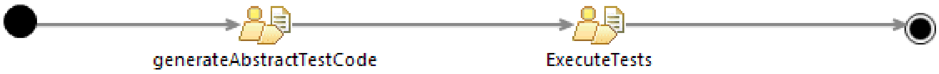


Figure 51: Execution of Tests

The execution of tests starts with the generation of abstract test code, which is in principle independent of the test execution platform. This test code can then be compiled and executed within a test execution platform.

5.7.1.1 GenerateAbstractTestCode

The following picture shows the Definition of GenerateAbstractTestCode.

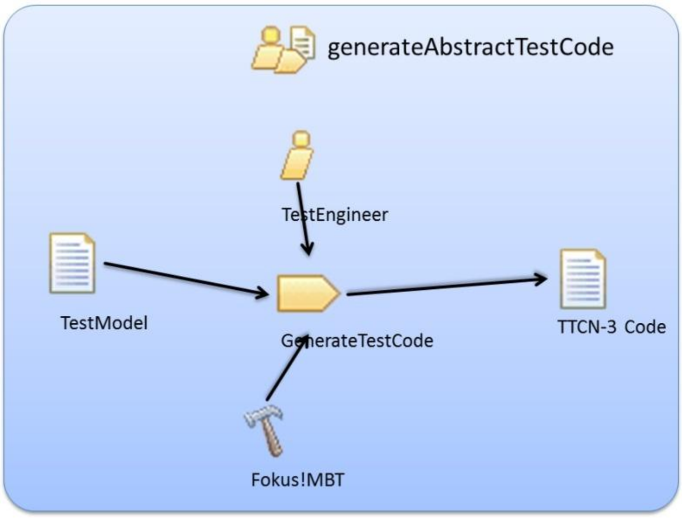


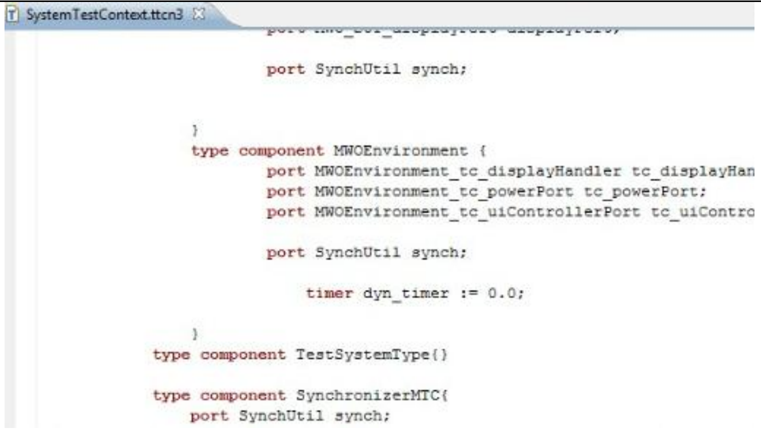
Figure 52: GenerateAbstractTestCode

GenerateTestCode is concerned with the following work products.

- TestModel
- TTCN-3 Code

Each step of the activity is described in the following tables.

GenerateAbstractTestCode	
Overview	Test Engineer generates abstract test code which can be later executed in a test execution environment.
Assigned Tickets	#47, #69, #113, #160, #163, #222, #15, #38, #40, #117, #242, #243, #268, #270-287, #109, #12, #41, #87, #111, #159, 164, #338, #339, #340, #350,
Language	UML-Profile for Testing, UML for Marte profile, SysML, TTCN-3
Tools	Fokus!MBT
Parameters	n/a
Pre-condition	Completed test model containing test case definitions
Post-Condition	Abstract Test code generated for each test case
Steps	**Load Test model:** The test model containing the test cases needs to be loaded into the Fokus!MBT tool chain. This is usually achieved by adding such a test

	model to a project context in Fokus!MBT. **Generate Test Code**: Fokus!MBT offers an export wizard, which allows to export a test model in to TTCN-3 code as an abstract test notation. After executing the export wizard the corresponding TTCN-3 code is added to the project where the test model is contained. The code can now further processed in subsequent steps or even manually modified.
Example	

SystemTestContext.ttcn3

```
                    port SynchUtil synch;


                }
                type component MWOEnvironment {
                    port MWOEnvironment_tc_displayHandler tc_displayHan
                    port MWOEnvironment_tc_powerPort tc_powerPort;
                    port MWOEnvironment_tc_uiControllerPort tc_uiContro

                    port SynchUtil synch;

                        timer dyn_timer := 0.0;

                }
            type component TestSystemType{}

            type component SynchronizerMTC{
                port SynchUtil synch;
```

5.7.1.2 ExecuteTests

The following picture shows the execution of tests.

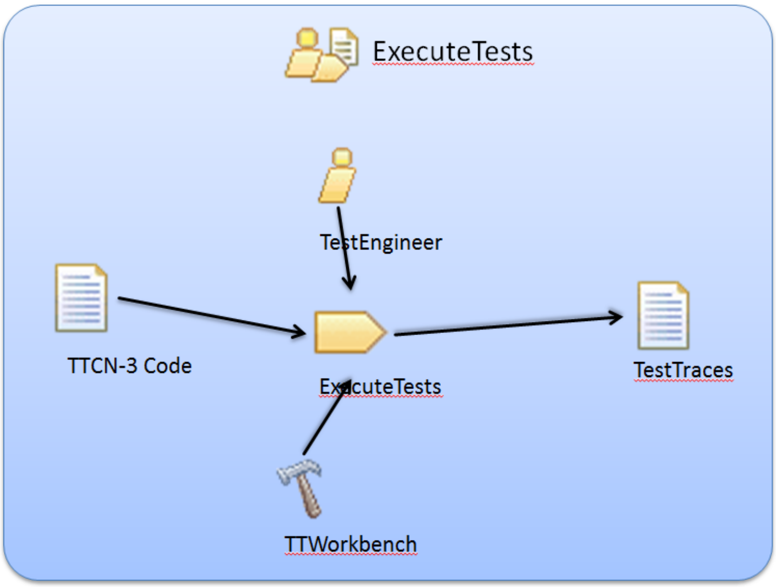
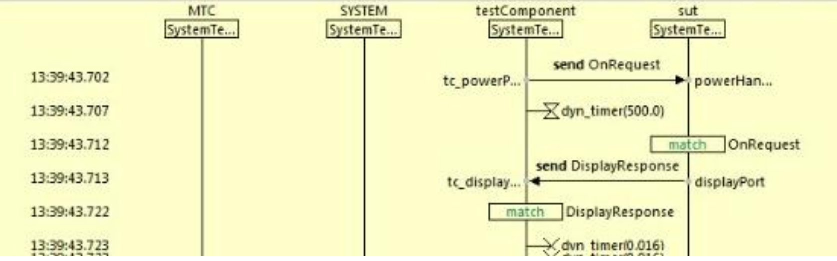


Figure 53: ExecuteTests

ExecuteTestsis concerned with the following work products.

- TTCN-3 Code
- Test Traces

Each step of the activity is described in the following tables.

ExecuteTests	
Overview	Execute the abstract test code in a test execution environment.
Assigned Tickets	#47, #69, #113, #160, #163, #222, #15, #38, #40, #117, #242, #243, #268, #270-287, #109, #12, #41, #87, #111, #159, 164, #338, #339, #340, #350,
Language	UML-Profile for Testing, TTCN-3
Tools	TTWorkbench
Parameters	n/a
Pre-condition	
Post-Condition	
Steps	**Load Test Code:** TTWorkbench can load TTCN-3 files. This is done by adding such a file to a TTWorkbench project. **Creation of Test Harnish:** Before Executing test cases a test harnish needs to be configured. In this step the configuration and the connection to a real system under test (SUT) needs to be established. **Run Tests**: The test cases contained in the TTCN-3 files can be executed by starting the execution in TTWorkbench. TTWorkbench now completes all test cases and stores the test traces.
Example	

5.7.2 Simulation

The virtual prototype is simulated to gain insight into its non-functional (timing, power consumption, communication) and functional behaviour (values of variables respectively signals).

The simulation activity processes a set of manually coded SystemC source files and generated ones. The sources are compiled, the system configuration is completed and the resulting binary code is then run inside the Verde Eclipse Platform. The resulting products are the log and trace files which are then subject to further analysis. The generated binary is discarded upon any change to the input source code.

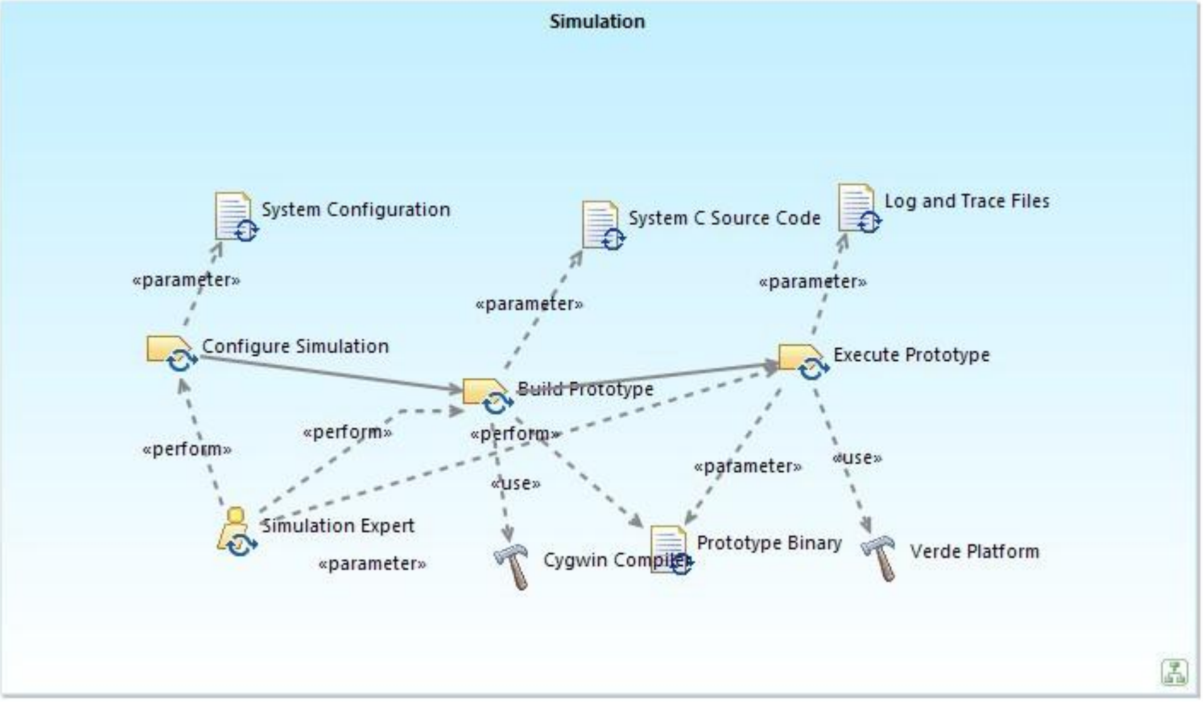


Figure 54: Simulation Tasks

Simulation	
Overview	Simulation of systems composed of hardware and software is the final step for validating and verifying functional and non-functional properties.
Assigned Tickets	Support functional and non functional aspects (Ticket 60) VERDE shall allow the verification and validation of the entire system and even only parts of it. The partwise verification / validation is necessary to reduce complexity, increase simulation performance and allow investigations even if not all components are available. (Ticket 59) Which communication architecture is the optimal solution for the aimed system / product? What's the difference in performance using a slower memory? (Ticket 54) VERDE shall be able to run software on the hardware model. This may be realized using a processor model (ISS) and SystemC-based connected peripherals. (Ticket 57) VERDE shall allow to configure, start and execute simulations out of the VERDE framework, including external simulators, co-simulation and co-emulation (this includes necessary modifications in XML, defines, etc.). (Ticket 129) VERDE should support both the simulation of the entire system and the simulation of subsystems isolated from the rest of the system. (Ticket 130) VERDE should be able to represent and regard operation system aspects during modelling and simulation. This include various schedule strategies, resource management, interrupt handling and so on. (Ticket 65) VERDE should allow to parameterize suitable parameters without necessary recompile and in some cases even the change of parameters during simulation. (Ticket 70) VERDE or the model consisting of component models shall be executable in all environments for which adequate interfaces were provided (e.g. to allow distributed simulations (e.g. cluster/cloud, HW in the loop, Debugger, etc.)). (Ticket

	124)
Language	C++, esp. SystemC, TLM xml for configuration of model parameters
Tools	Eclipse Helios Release and Verde Platform Cygwin tool chain with compiler OSCI SystemC Library version 2.2.0 OSCI TLM Library version 2.0.1
Parameters	Software and Hardware models in SystemC, i.e. the Virtual Prototype (In) Any logging information, i.e. traces, test results, power consumption and timing information (Out)
Pre-condition	All SystemC code has been generated. AND Configuration is complete. AND Virtual Prototype is compileable.
Post-condition	Execution of simulation has finished. AND Trace file has been written.
Steps	In order to set up and run a simulation, the following steps need to be done. **Preparation:** Download and install the open source framework Cygwin that provides a C++ compiler for Windows. In addition download, configure and install necessary libraries (OSCI TLM 2.0.1 and SystemC 2.2.0). Afterwards run Eclipse.exe that is provided as a part of the Verde Platform. **Import prototype into Verde Platform**: If not yet done the virtual prototype has to be imported into the Verde platform. The substeps are: 1. Import the prototype as an Eclipse C++ project. 2. Set all necessary project properties (paths to include directories SystemC/include and TLM/include), paths to SystemC library, select the Cygwin tool chain). **(Optional) Generate missing code fragments:** If configuration or model data has been changed, code generation might need to be restarted. **Configure the prototype:** The prototype may be configurable according to model parameters that can be set using xml configuration files. Some configuration data is stored in the IP-XACT XML format (see example). One example for configurable data in the showcase is the size and segmentation of the address space and its memory maps. The path to the xml file needs to be specified in the C++ File that contains the SC_MAIN method (usually named main.cpp). **Compile the prototype:** All modules of the simulated system have to be compiled using the above mentioned tool chain. In the Verde Eclipse Platform this is started via a menu entry “Project->Build”. **Link prototype to SystemC simulator and other libraries**: In order to be able to run a simulation, the compiled models have to be linked against a OSCI SystemC compliant simulator kernel. In addition more libraries might be linked to the executable model (e.g. a TLM library for higher level modelling of communication). This step is automatically executed by the “Project->Build” command from the last step. **Run simulation**: Once executable model has been created it can be run directly from the Verde platform. In Verde this can be done by right-clicking on the

<table>
<tr>
<td></td>
<td>

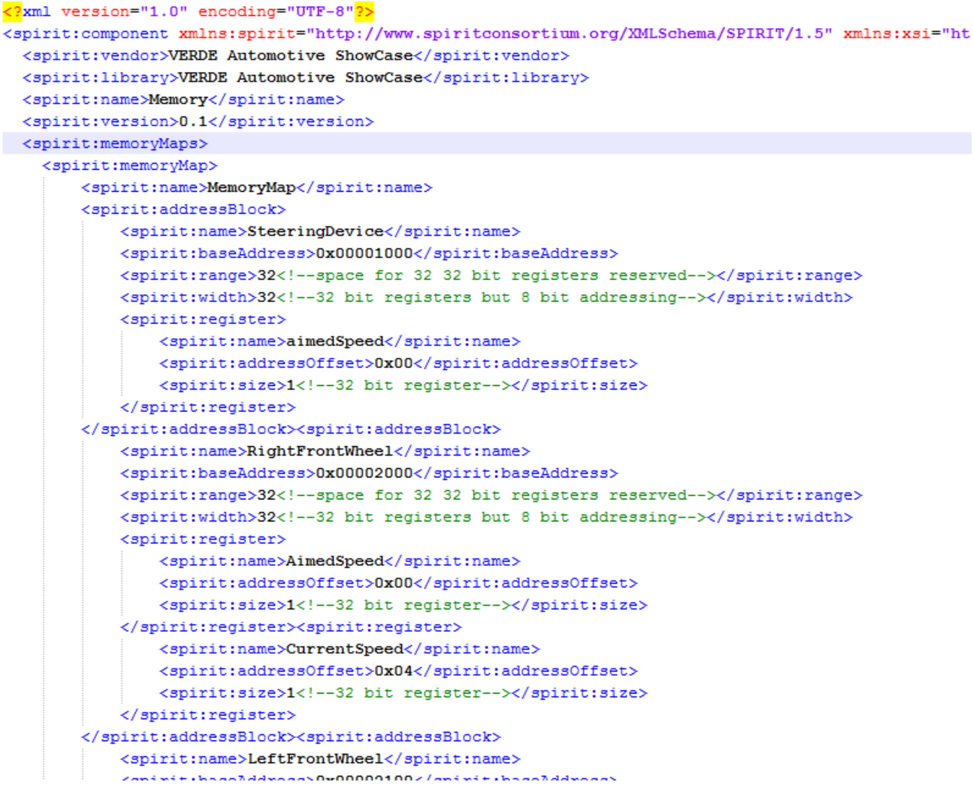
compilation result, i.e. an executable file. The context menu on this file shows the “Run As…-> Local C++ Application” command which triggers the execution to be started.

(Optional) Debug the simulation: Instead of a normal execution, the simulation can also run in debugging mode via “Debug As…-> Local C++ Application”.

Execute post-simulation steps: The simulation produces large amounts of logged data to trace files. These trace files can be analyzed manually in the process step “Trace Visualization” or automatically in the process step “Trace Abstraction”.

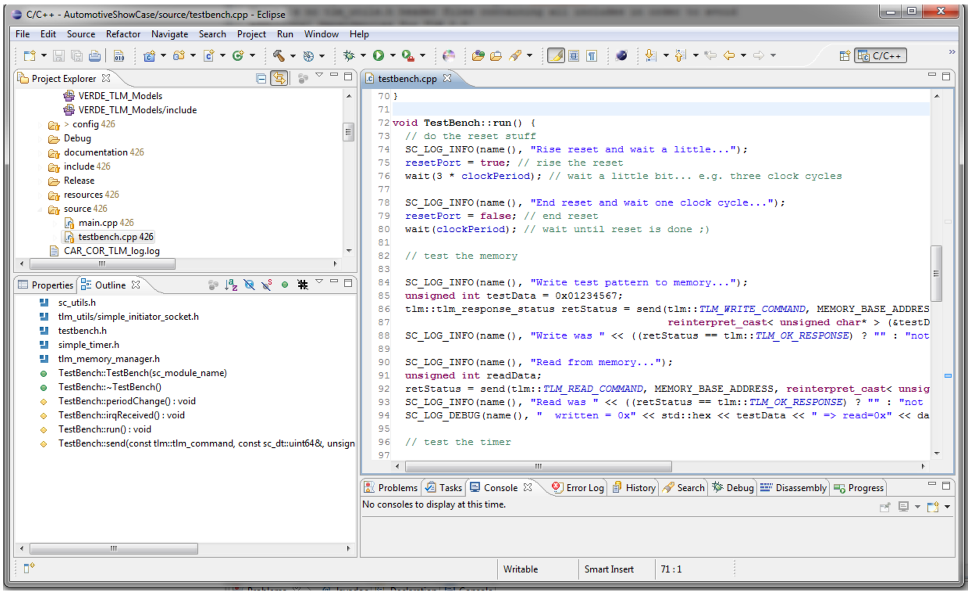
</td>
</tr>
<tr>
<td>Example</td>
<td>

Example configuration data (IP-XACT xml):



```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:component xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5" xmlns:xsi="ht
  <spirit:vendor>VERDE Automotive ShowCase</spirit:vendor>
  <spirit:library>VERDE Automotive ShowCase</spirit:library>
  <spirit:name>Memory</spirit:name>
  <spirit:version>0.1</spirit:version>
  <spirit:memoryMaps>
    <spirit:memoryMap>
        <spirit:name>MemoryMap</spirit:name>
        <spirit:addressBlock>
            <spirit:name>SteeringDevice</spirit:name>
            <spirit:baseAddress>0x00001000</spirit:baseAddress>
            <spirit:range>32<!--space for 32 32 bit registers reserved--></spirit:range>
            <spirit:width>32<!--32 bit registers but 8 bit addressing--></spirit:width>
            <spirit:register>
                <spirit:name>aimedSpeed</spirit:name>
                <spirit:addressOffset>0x00</spirit:addressOffset>
                <spirit:size>1<!--32 bit register--></spirit:size>
            </spirit:register>
        </spirit:addressBlock><spirit:addressBlock>
            <spirit:name>RightFrontWheel</spirit:name>
            <spirit:baseAddress>0x00002000</spirit:baseAddress>
            <spirit:range>32<!--space for 32 32 bit registers reserved--></spirit:range>
            <spirit:width>32<!--32 bit registers but 8 bit addressing--></spirit:width>
            <spirit:register>
                <spirit:name>AimedSpeed</spirit:name>
                <spirit:addressOffset>0x00</spirit:addressOffset>
                <spirit:size>1<!--32 bit register--></spirit:size>
            </spirit:register><spirit:register>
                <spirit:name>CurrentSpeed</spirit:name>
                <spirit:addressOffset>0x04</spirit:addressOffset>
                <spirit:size>1<!--32 bit register--></spirit:size>
            </spirit:register>
        </spirit:addressBlock><spirit:addressBlock>
            <spirit:name>LeftFrontWheel</spirit:name>
            <spirit:baseAddress>0x00002100</spirit:baseAddress>
```

Example screenshot from SystemC simulation inside Verde Platform:



```
sc_utils.h
tlm_utils/simple_initiator_socket.h
testbench.h
simple_timer.h
tlm_memory_manager.h
TestBench::TestBench(sc_module_name)
TestBench::~TestBench()
TestBench::periodChange() : void
TestBench::irqReceived() : void
TestBench::run() : void
TestBench::send(const tlm::tlm_command, const sc_dt::uint64&, unsign
```

```
}

void TestBench::run() {
  // do the reset stuff
  SC_LOG_INFO(name(), "Rise reset and wait a little...");
  resetPort = true; // rise the reset
  wait(3 * clockPeriod); // wait a little bit... e.g. three clock cycles

  SC_LOG_INFO(name(), "End reset and wait one clock cycle...");
  resetPort = false; // end reset
  wait(clockPeriod); // wait until reset is done ;)

  // test the memory

  SC_LOG_INFO(name(), "Write test pattern to memory...");
  unsigned int testData = 0x01234567;
  tlm::tlm_response_status retStatus = send(tlm::TLM_WRITE_COMMAND, MEMORY_BASE_ADDRES
                                            reinterpret_cast< unsigned char* > (&testD
  SC_LOG_INFO(name(), "Write was " << ((retStatus == tlm::TLM_OK_RESPONSE) ? "" : "not

  SC_LOG_INFO(name(), "Read from memory...");
  unsigned int readData;
  retStatus = send(tlm::TLM_READ_COMMAND, MEMORY_BASE_ADDRESS, reinterpret_cast< unsig
  SC_LOG_INFO(name(), "Read was " << ((retStatus == tlm::TLM_OK_RESPONSE) ? "" : "not
  SC_LOG_DEBUG(name(), "  written = 0x" << std::hex << testData << " => read=0x" << da

  // test the timer

```

No consoles to display at this time.

</td>
</tr>
</table>

5.7.3 Trace Abstraction

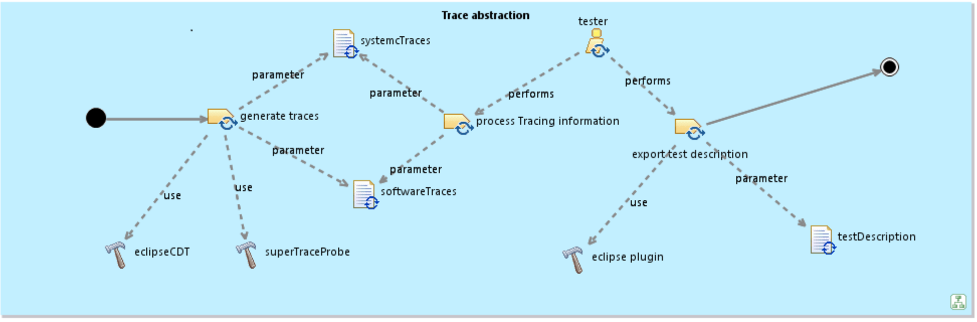
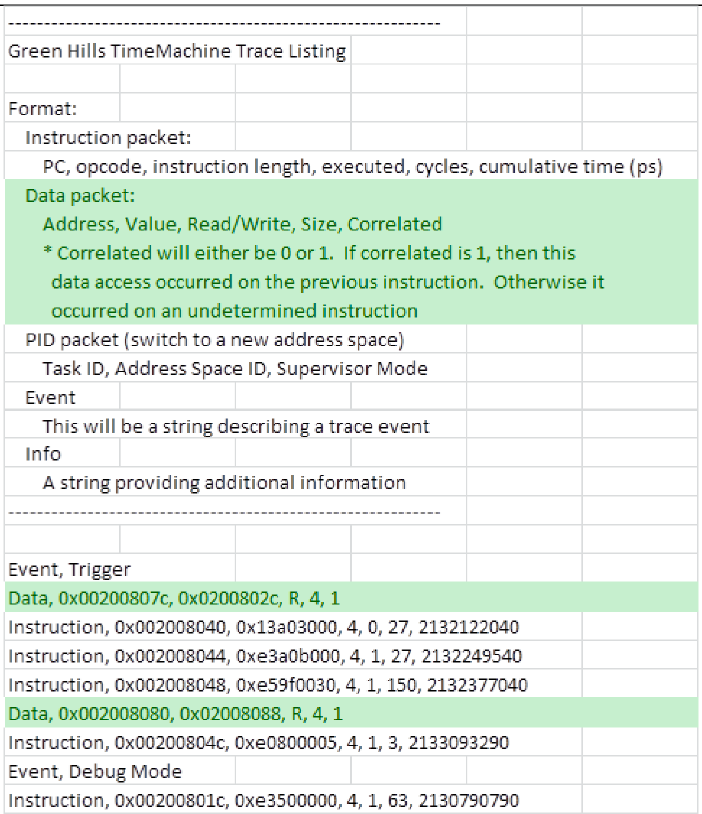


Figure 55: Trace Abstraction

Trace Abstraction	
Overview	This pattern describes the use of execution traces for the derivation/generation ofstimulus pattern and acceptance criteria evaluator for testing purposes.
Assigned Tickets	Support functional and non-functional aspects (Ticket 60) VERDE should support both the simulation of the entire system and the simulation of subsystems isolated from the rest of the system. (Ticket 130) Abstraction of legacy code to components. (Ticket 404) NFP analysis for functionality implemented in C/C++. (Ticket 440) VERDE shall offer execution time and power properties comparison with respect to several different Sw/Hw allocations. (Ticket 455) Execution time and power properties validation/extraction at early stage (on host). (Ticket 462)
Language	C++, SystemC, SystemVerilog, TLM. XML: for the representation of the testdescription. vcd, csv:format for the representation of traces.
Tools	Any tool supporting the standard (full compliance)
Parameters	• Software traces generated by tracing hardware like the *SuperTrace Probe* tracing hardware from Green Hills Systems which records tracing information from executed programs at runtime(.*csv* file), in • Traces generated during SystemC simulation (.*vcd* file), in • Test description file needed for further code-generation of stimulus pattern generator module and corresponding acceptance evaluator module, out
Pre-Conditons	
Post-Conditions	
Steps	• Import tracesfromcsv or vcd files • Abstraction (transformation) of tracing information. ➢ Derive test behaviour for stimulation unit of testbench ➢ Derive evaluation criteria • Export test description in an interchange format (.xml)

<table>
<tr>
<td>Example</td>
<td>


```
-------------------------------------------------------------
Green Hills TimeMachine Trace Listing

Format:
  Instruction packet:
    PC, opcode, instruction length, executed, cycles, cumulative time (ps)
  Data packet:
    Address, Value, Read/Write, Size, Correlated
    * Correlated will either be 0 or 1.  If correlated is 1, then this
     data access occurred on the previous instruction.  Otherwise it
     occurred on an undetermined instruction
  PID packet (switch to a new address space)
    Task ID, Address Space ID, Supervisor Mode
  Event
    This will be a string describing a trace event
  Info
    A string providing additional information
-------------------------------------------------------------

Event, Trigger
Data, 0x00200807c, 0x0200802c, R, 4, 1
Instruction, 0x002008040, 0x13a03000, 4, 0, 27, 2132122040
Instruction, 0x002008044, 0xe3a0b000, 4, 1, 27, 2132249540
Instruction, 0x002008048, 0xe59f0030, 4, 1, 150, 2132377040
Data, 0x002008080, 0x02008088, R, 4, 1
Instruction, 0x00200804c, 0xe0800005, 4, 1, 3, 2133093290
Event, Debug Mode
Instruction, 0x00200801c, 0xe3500000, 4, 1, 63, 2130790790
```

Figure 56: Example of trace file (.csv)

- Generated csv-file contains online traces of instructions and data operations with needed cycles and cumulated times from which time stamps can be derived for each tracing step.
</td>
</tr>
</table>

6. Conclusion

Goal of this task in the VERDE project was to cope with the iterative, incremental and validation-driven design of component based real-time and embedded systems. In a first step the current practices used by the industrial partners from various domains have been described to indentify expected improvements within the development process.

It has been pointed out, that all industrial partners have well established development processes in place. On one hand, the used processes provide some similarities for activities like requirements capturing and modelling of a system architecture. On the other hand, they are also specialized for a certain domain, especially for activities like verification and testing. It was not the intention within the VERDE project to define a unified process for all domains. Nevertheless a common VERDE spiral process was described. The description of such a spiral process is nothing new, but within the context of VERDE precise guidelines needed in such a process are described

Therefore modelling steps that are needed to address the requirement tickets defined by industrial partners have been identified and described. Methodological patterns are described for each step and provide practical and “easy to put in practice” modeling solutions for concrete modeling issues. Such patterns can be selected by end users when needed and integrated in domain specific processes.

References

[1] AUTOSAR 4.0 Specification on www.autosar.de

[2] AUTOSAR 4.0 Software Component Templatehttp://www.autosar.org/download/R4.0/AUTOSAR_TPS_SoftwareComponentTemplate.pdf

[3] (OSCI), Open SystemC Initiative.TLM 2.0 Language Reference Manual. [Online] http://www.systemc.org/downloads/.

[4] Object Management Group (OMG), Unified Modelling Language: Superstructure, Version 2.3, OMG Document formal/2010-05-05, 2010

[5] Object Management Group (OMG), UML Profile for MARTE: Modelling and Analysis of Real-Time Embedded systems, Version 1.0, OMG document ptc/2009-11-02, 2010

[6] Baker, P., Dai, Z.R., Grabowski, J., Haugen, Ø., Schieferdecker, I. Williams, C.:Model-driven testing – using the UML testing profile. Springer (2007)