A. D. MCCXXIV

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

A MODEL-DRIVEN METHODOLOGY FOR CRITICAL SYSTEMS ENGINEERING

FABIO SCIPPACERCOLA

Tesi di Dottorato di Ricerca

(XXVIII Ciclo)

MARZO 2016

Il Tutore

Prof. Stefano Russo

Il Coordinatore del Dottorato

Prof. Francesco Garofalo

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

via Claudio, 21- I-80125 Napoli - ☎ *[#39]* (0)81 768 3813 - *[#39]* (0)81 768 3816

A MODEL-DRIVEN METHODOLOGY FOR CRITICAL SYSTEMS ENGINEERING

By
Fabio Scippacercola

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
"FEDERICO II" UNIVERSITY OF NAPLES
VIA CLAUDIO 21, 80125 – NAPOLI, ITALY
MARCH 2016

Table of Contents

List of Tables

List of Figures

Acronyms

ALF OMG Action Language for UML.

BB-PIT Black Box Platform Independent Test Model, Sec. 3.4.2.

BB-PST Black Box Platform Specific Test Model, Sec. 3.5.2.

BDD SysML Block Definition Diagram.

BSP Board Support Package, Sec. 8.1.1.

CAN Controller Area Network (Bus).

CIM Computation Independent Model, Sec. 1.1.2.

CIT Computation Independent Test Model, Sec. 3.4.1.

CIV Computation Independent Viewpoint, Sec. 1.1.2.

CPT Category Partition Testing.

CUA Component Under Analysis, Sec. 5.4.1.

DSL Domain-Specific Language.

ETCS European Train Control System, Sec. 6.2.1.

FMEA Failure Mode and Effects Analysis, Sec. 5.1.

fUML OMG Semantics of a Foundational Subset for Executable UML Models.

GB-PIT Grey Box Platform Independent Test Model, Sec. 3.4.3.

HIL Hardware-in-the-loop.

HMI Human-Machine Interface, Sec. 6.2.1.

IBD SysML Internal Block Diagram.

IS Interlocking System, Sec. 6.2.1.

KB Knowledge Base.

M2M Model-to-Model Transformation, Sec. 1.1.

M2T Model-to-Text Transformation, Sec. 1.1.

MBE Model-Based Engineering, Sec. 1.1.

MDA Model-Driven Architecture, Sec. 1.1.

MDD Model-Driven Development, Sec. 1.1.

MDE Model-Driven Engineering, Sec. 1.1.

MDT Model-Driven Testing, Sec. 1.1.

MIL Model-in-the-loop.

OMG Object Management Group.

OS Operating System.

OXF IBM Rhapsody Object Execution Framework, Sec. 6.2.4.

PB Prolan Block, Sec. 6.1.

PIM Platform Independent Model, Sec. 1.1.2.

PIT Platform Independent Test Model, Sec. 1.1.3.

PIV Platform Independent Viewpoint, Sec. 1.1.2.

PM Prolan Monitor, Sec. 7.1.

PSM Platform Specific Model, Sec. 1.1.2.

PST Platform Specific Test Model, Sec. 1.1.3.

PSV Platform Specific Viewpoint, Sec. 1.1.2.

RAMS Reliability, Availability, Maintainability and Safety.

RBC Radio Block Centre, Sec. 6.2.1.

RTOS Real-Time Operating System.

SDLC Software Development Life Cycle.

SEEA Software Error Effect Analysis, Sec. 5.2.

SIL Safety Integrity Level.

SPLC Software Product Life Cycle.

SUT System Under Test.

SXF IBM Rhapsody Simple Execution Framework, Sec. 6.2.4.

SysML OMG Systems Modeling Language, Sec. 1.1.3.

UML OMG Unified Modeling Language.

UTP UML Testing Profile, Sec. 1.1.3.

V&V Verification and Validation.

WB-PST White Box Platform Specific Test Model, Sec. 3.5.1.

Acknowledgements

I believe that from the eyes of other people I could have given the impression to be a person selfish of words or feelings in the past. But the truth is that I extremely care of who I am fond of, so much that sometimes I was scared about how my words and actions would have been interpreted by them.

A Ph.D. teaches you to expose your opinions, thought and theses. And not only these; I skipped the acknowledgements in my BSc and MSc dissertations, now it is time to move on.

I would like to deeply thank all who have been around me in this academic path: my supervisor, prof. Stefano Russo, and all professors, researchers and colleagues (but I would like to call them friends) of the MOBILAB and CINI. I cannot forget to mention, András Zentai, Nuno Silva, my family, and all my best friends.

Thanks by heart.

Napoli, Italy Fabio
March 31, 2016

Introduction

Software engineering, as most of branches of engineering, has always evolved increasing the level of abstractions. If we just look at one of the most peculiar and fascinating technology of this discipline, the programming languages, we will note how the first abstractions, i.e., the second generation languages – the *assembly languages* –, were born soon after programmers had struggled with machine codes; then came the third generation programming languages, that freed the programmers from low level details of the machine, and finally the fourth generation languages, which added more facilities and masked recurrent problems, such as the representation of data and the interworking between heterogeneous systems. In this perspective, Model-Driven Engineering (MDE) aims at raising the level of abstraction in software design and verification [1], and promises to innovate the traditional methodologies of software development.

Model-driven approaches focus on a *model*, i.e., on descriptions of a system that neglect aspects that are not of interest at the current stage in a software process; the process advances transforming the model in documents, intermediate artifacts, or in the final product. The result is that MDE is going to shift the traditional development paradigm, based on different kinds of artifacts composed by domain experts in multiple formats, to a common formalism – the model –, by which the artifacts are obtained through computer-assisted transformations. This *model-centric* paradigm introduces several benefits into the process, and leads to better productivity and quality of artifacts, shorter development time, and enhanced automation, which includes automatic code generation and support to the activities

of verification and validation.

Due to these benefits, industry is increasing the adoption of MDE. This is shown by recent industrial surveys, which investigated the adoption of MDE methodologies and technologies in practice [2, 3]. In particular, MDE is attractive for the development of critical systems, since it can speed up the activities of Verification and Validation (V&V): model-driven approaches are widely exploited in industry for the early verification of the systems, through techniques such as model reviews, guideline checkers, Rapid Control Prototyping and Model- and Software- in-the-loop Tests. These techniques shift the cost of development from the phases of V&V to the ones of requirement analysis and design, but lead to benefits in terms of residual errors. Companies not performing model-in-the-loop testing find almost 30% more errors during module test [4].

Incorporating model-driven techniques into a legacy well-proven development processes is not simply a matter of placing models and transformations in traditional methodologies: the activities have to be carefully redesigned to exploit the benefits of MDE, and the skills and expertise of the engineers. Indeed, success stories on the adoption of MDE are reported after long time of technological innovation, that required several pilot project experiments, and the rethinking of traditional activities, complemented by custom supporting tools. Motorola could achieve an increase of quality and productivity (ranging from 1.2x to 8x), and an approximately 33% reduction in the effort required to develop test cases, after 15 years of wide spectrum adoption of MDE [5, 6]; the recent case study in [7] reports the successful introduction of a MBE process after four years and three projects had been defined and consolidated.

The problem of defining a model-driven life cycle is exacerbated in critical domains, where the process has to comply to strict requirements to assure high level of quality of the artifacts. In fact, in safety-critical domain, the efforts for verification and validation account for the major part of the costs, while safety standards require accurate assessments

of the system and prescribe qualified tools.

Past studies attempted to apply pre-existing processes to MDE, or to create new ones [8], but these approaches cannot be proposed as a replacement of current industrial practices, especially for the development of certified critical systems. There are still few model-driven methodologies that cover the full system development life cycle, and that are suited to apply MDE on a large scale, in processes shared among more partners.

Indeed, few previous approaches targeted to offer flexible and complete model-driven life cycles that could be customized for industrial needs, and, in particular, designed for critical domains, where is the demand to support a broad range of activities of V&V and comply with standards and open technologies.

The need of a consolidated MDE methodology was also experienced personally by the Ph.D. candidate, during twelve-months of industrial-academic partnership in which he was involved: in the framework of the European project "CErtification of CRItical Systems" (CECRIS, [9]), the candidate participated to a transfer of knowledge of MDE technologies in Prolan Co., a Hungarian company which develops certified products for safety critical process control and rail signalling systems; during this activity, it emerged the lack of well-defined processes for the development of a CENELEC SIL-4 safety critical signalling system that was suited for the real industrial needs.

Thesis contribution

1. **This thesis proposes a novel model-driven life cycle that is tailored to the development of critical systems, and overcomes limitations of previous approaches**:

    - It covers the full software life cycle and is suited to replace current industrial processes, in particular for the development of products that undergo to CENELEC 50128 certification;

- It supports several techniques of Verification and Validation in a conventional V-Model, including techniques of early fault detection, and safety assessment;
- It benefits from OMG standards, adopting Model-Driven Architecture (MDA), SysML, UML and Model-Driven Testing (MDT).

2. **The methodology integrates an original approach for model-driven system validation, based on a new model named *Computation Independent Test model* (CIT)**, that is exploited to perform *Model-, Software-, and Hardware- In-the-loop testing*.

3. **Moreover, the process supports the Failure Modes and Effect Analysis (FMEA), with a novel approach to conduct Model-Driven FMEA, based on custom SysML Diagram, namely the *FMEA Diagram*, and Prolog**.

Since the thesis benefits from the fruitful collaboration with industry, **the approaches have been experimented in multiple real-world case studies, from railway and automative domains**, offered by the industrial partners. Two case studies of CENELEC 50128 SIL-4 signalling systems have been provided by Prolan, whereas one automotive case study has been provided by the Portuguese company Critical Software SA, that develops safety-critical systems and provides consulting and expertise for the certification.

Finally, as an additional contribution of this thesis, the author aims at increasing the knowledge on MDE success and failure factors, since more concrete industrial experiences are necessary to get a clear comprehension of risks and benefits of MDE, especially for critical systems [8].

The work includes material from the following research papers, already accepted or published in peer-reviewed conferences:

- F. Scippacercola, R. Pietrantuono, S. Russo, A. Zentai, "Model-Driven Engineering of a Railway Interlocking System", In: *Proc. of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015)*. 2015, pp. 509-519.

SCITEPRESS. ISBN: 978-989-758-083-3.

- F. Scippacercola, R. Pietrantuono, S. Russo, A. Zentai, "Model-in-the-loop Testing of a Railway Interlocking System", In: *Model-Driven Engineering and Software Development.* 2015, Communications in Computer and Information Science (CCIS), vol. 580, pp. 375-389, Springer International Publishing. DOI: 10.1007/978-3-319-27869-8; ISSN: 1865-0929; ISBN: 978-3-319-27868-1.
- F. Scippacercola, R. Pietrantuono, S. Russo, N. P. Silva, "SysML-based and Prolog-supported FMEA", In: *Proc. of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW).* 2015, pp. 174-181, IEEE. ISBN: 978-1-5090-0406-5.
- F. Scippacercola, R. Pietrantuono, S. Russo, A. Esper, N. Silva, "Integrating FMEA in a Model-Driven Methodology", accepted to *the International Space Sysem Engineering Conference (DASIA).* 2016.

The first study presents the proposed model-driven development life cycle (thesis contribution 1), the second one focuses on the CIT and on the Model-in-the-loop extension (thesis contribution 2), while the third and forth scientific papers introduce the model-driven FMEA and FMEA Diagram (thesis contribution 3).

The dissertation is organized as follows:

Chapter 1 introduces the main concepts of Model-Driven Engineering and Model-Driven Architecture, then presents a discussion on the current state of practice of model-based and model-driven approaches.

Chapter 2 presents a systematic literature review on model-driven processes for critical systems, and discusses the limitations of previous studies.

Chapter 3 introduces the novel model-driven methodology for critical systems engineering, and discusses its benefits.

Chapter 4 presents details the Computation Independent Test Model, and the benefits from environmental modeling in the software development life cycle.

Chapter 5 presents our proposal for a model-driven Failure Mode and Effects Analysis, and the novel SysML FMEA Diagram.

Chapter 6 introduces our experience with the proposed development life cycle on a case study on the Prolan Block, a safety-critical part of an interlocking system provided by

Prolan.

Chapter 7 presents the Prolan Monitor, another safety-critical system provided by Prolan, and the experience with the CIT and in-the-loop testing.

Chapter 8 discusses an automotive safety-critical system provided by Critical Software, on which we performed a FMEA using the proposed approach.

Chapter 1

Model-Driven Engineering

1.1 Basic Concepts and Definitions

Since models have always been applied at different extents in many problems and activities, there are many acronyms with fuzzy borders in the universe of software engineering. In this thesis we refers to the terminology of [10].

When processes exploit models as support for their goals they are part of *Model-Based Engineering* (MBE), and we call the activities *document-centric*, since models are only a means to achieve the targets, but there is not particular emphasis on them. Therefore MBE is the broadest term that cover all the methodologies and activities that employ models (Fig. 1.1).

Model-Driven Engineering (MDE) focuses on the processes where models are key artifacts of the activities (*model-centric*). When we restrict to considering MDE for supporting the development of systems, we can use the more specific term of *Model-Driven Development* (MDD). One approach of MDD is the *Model-Driven Architecture* (MDA), proposed by the Object Management Group (OMG) [11]. The *Model-Driven Testing* (MDT) is a theory of software testing that introduces concepts enabling to transform models in test-cases in order to support verification and validation activities. Even though MDT is not an OMG standard, it uses an OMG's standard profile, the *UML Testing Profile* (UTP).

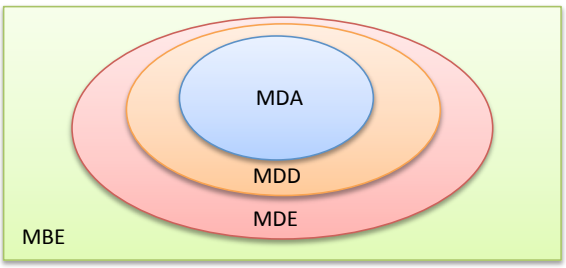


Figure 1.1: The terminology adopted in the thesis.

MDE is founded on concepts of *models* and *transformations*: instead of producing (textual) documents as artifacts – requirements, design, code, test artifacts – engineers, in MDE, focus on models as primary artifacts (Fig. 1.2).

Models are defined in (semi-)formal languages, that are typically machine-understandable and drawn with the support of tools. Other artifacts are derived through defined transformations: *Model-to-Model* transformations (M2M), or *Model-to-Text* transformations (M2T) from models to textual documents, source code or testing artifacts (such as test cases and test scripts).

As argued by Kent [12], MDE approaches, in general, can identify different levels of decomposition and can employ ad hoc or domain-specific languages for models and transformations, whereas MDA is bound to OMG's standards.

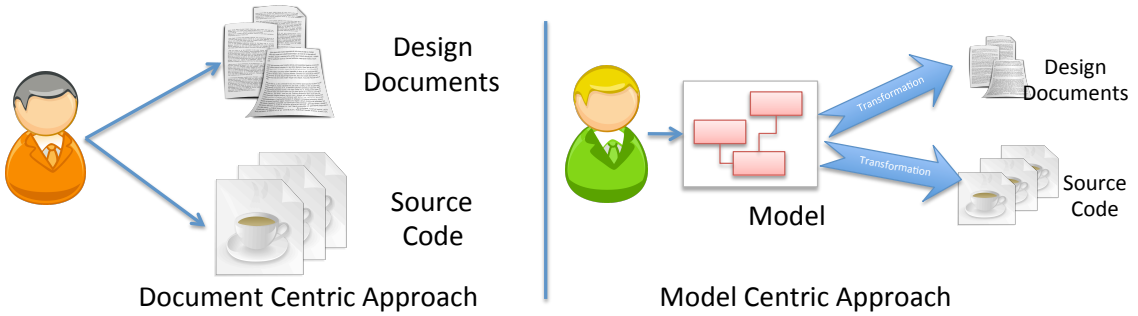


Figure 1.2: Document-centric and Model-centric approaches.

1.1.1 Model-Driven Architecture

The Object Management Group is an international trade association incorporated as a nonprofit corporation in the United States, with affiliate organizations around the globe. In the 90's, OMG standardized the object request broker (ORB) and a suite of objet services. This work was guided by the Object Management Architecture (OMA) that provides a framework for distributed systems and by the Common ORB Architecture, or CORBA, a part of that framework.

OMG first conceived MDA as a technology to overcome the interoperability problems of applications partially addressed by CORBA [13]. Indeed, even if CORBA provided a good solution for the interoperability of applications, it became clear how it is difficult for large enterprises to standardize on different middleware platforms: enterprises have applications on different middleware, that simply have to be integrated even though this process turned out to be expensive and time-consuming. Furthermore, middleware systems continue to evolve and even CORBA could not be a guarantee for next decades. Therefore, MDA was proposed as a better way to reach portability, interoperability and reusability through architectural separation of concerns, in the sight of OMG Vision, that postulates how the myth of a standalone application or standard for developing software as well as for data interchange died.

The recent 2.0 revision of guide of the standard [11] defines MDA as an approach for deriving value from models and architecture in support of the full life cycle of physical, organizational and I.T. systems. MDA became an approach to deal with complexity and interdependence of complex systems to derive value from models and modeling by defining the *structure*, *semantics*, and *notations* of models using industry standards. The value from models is derived from [11]:

- *Using Models as communication vehicles.* MDA facilitates teams and communities

to come to a common understanding and/or consensus. MDA provides well defined elements, foundation for models and libraries of common vocabularies, rules and processes;

- *Derivating artifacts via automated trasformation.* The translation can be partial or full and it reduces the time and cost for realizing a design and for changes and maintenance. Automated derivation involves a platform independent "source model" with some parameters for how that source model is to be interpreted;

- *Performing analysis on models.* Models can be used subject of analysis, including model validation, statistics and metrics;

- *Simulating and executing the model.* Models as data can drive simulation engines that can assist in both analytics and execution of the designs captured in models;

- *Deriving information from models.* Well defined models can be used to derive information (such as documentation, derived insights, process playbooks, etc.), new models and other artifacts;

- *Structuring Unstructured Information.* MDA technologies provide a way to define structured representation of these documents.

These values translate in flexibilities during whole development process, parts of these were discussed in [13]:

Implementation new implementation infrastructure (the "hot new technology" effect) can be integrated or targeted by existing designs.

Integration: since not only the implementation but the design exists at time of integration, we can automate the production of data integration bridges and the connection to new integration infrastructures.

Maintenance: the availability of the design in a machine-readable form gives developers direct access to the specification of the system, making maintenance much simpler.

Testing and Simulation: since the developed models can be used to generate code, they can equally be validated against requirements, tested against various infrastructures and can used to directly simulate the behavior of the system being designed.

MDA focuses on intrinsic value of models to increase the abstraction level when developing software, software will be developed by modeling, and models will become code by applying automatic transformations on models. Transformations are not different from language compiler, but are more robust to the change of technologies, because will change the transformation rules but will no change models.

In order to enable (automatic) transformations of models, it has been necessary to introduce mechanisms to reason on modeling itself: this has been done introducing the concept of meta-modeling, i.e., introducing models for modeling languages. These concepts are commons to MDE, but MDA standardized the formalisms to use, so as to have four layer of abstractions (Fig. 1.3):

M0 is the user data layer, it is the layer at lowest abstraction and the elements are concrete objects of the problem domain.

M1 is the layer of modeling concepts. Here are the UML models of entities that abstract the user data layer, like UML classes or association. At this level are models defined by software engineers to define the requirements or architecture of the system.

M2 is UML Metamodel, i.e., M2 defines, through UML, the syntax of UML models in M1, as well as their semantic. For instance, M2 will constraint you to do not use UML links for connecting classes but UML objects. M1 models can be seen as instances of concepts of M2 layer and, by M2, you can check consistence of your UML models.

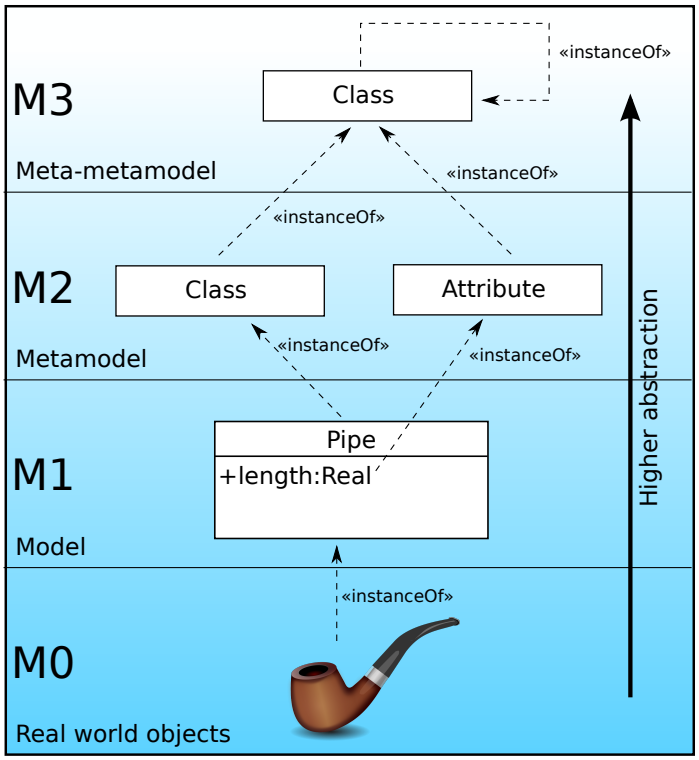


Figure 1.3: The OMG Modeling infrastructure.

M3 is most abstract layer defined by OMG. At this level is Meta-Object Facility (MOF) language. By MOF OMG can define syntax and semantic for meta-languages. In the MDA, MOF enables to define transformation rules among different models (of M1 layers) that are compliant to different meta-models (of M2 layers).

Using its modeling infrastructure, OMG can define rules to transform models to models (M2M) or model to text (M2T). With M2T transformation, MDE refers to that kind of transformation that produces source code (or less structured documents) from models.

MDA also defines more standards that address the complete development life cycle, covering analysis and design, programming, testing, component assembly as well as deployment and maintenance. One of the characteristics of UML is its capability to be easily extended by mechanisms defined in the standard: by UML *Profiles*, *Tagged Values* and *Constraints*,

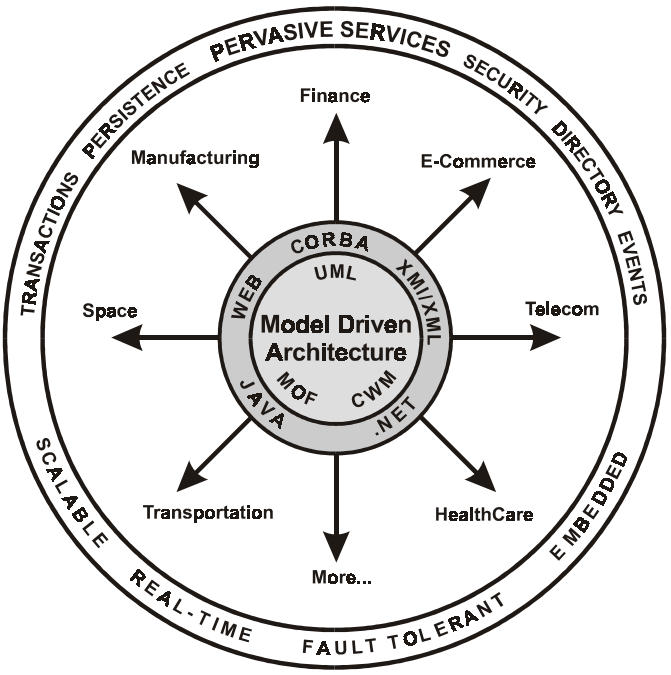


Figure 1.4: Pervasive services and specialized computing environments in the vision of the MDA [14].

custom domain specific languages (DSLs) can be defined, reusing and extending the elements of the UML language.

OMG provided a full set of platform independent pervasive services (Fig. 1.4) that models can exploit. These cover functional and non-functional application requirements ranging from persistence and event handling services to scalability solutions, security and fault tolerance services. Through these, MDA aims at reaching to goal of universal integration. Some of these standards are presented in tab. 1.1. Regarding the inclusion of UML in the standard, the recent revision of MDA, does not require to use UML, but only defined the compliance with the Meta Object Facility (MOF) for the tool compliance*MDA Compliant*.

Finally, MDA models are defined as *information selectively representing some aspect of a system based on a specific set of concerns. The model is related to the system by an explicit or implicit mapping* [11]. Thus, models are often, and preferably, expressed graphically with

Acronym	Standard	Description
MOF	Meta Object Facility	Provides means for defining meta meta-languages.
UML	Unified Modeling Language	Defines a modeling and specification language.
OCL	Object Constraint Language	A declarative language for describing rules.
XMI	XML Metadata Interchange	A standard for exchanging metadata information.
CWM	Common Warehouse Metamodel	Defines specification for modeling metadata in data warehouse.
QVT	Query View Transform	Standard set of languages for model transformation.
MOFM2T	MOF Model-to-Text Transformation Language	Specification for a model transformation language (in particular to text).
UTP	UML Testing Profile	UML Profile enabling to support V&V activities.
SPEM	Software Process Engineering Meta-model	UML Profile to support constructing process models.

Table 1.1: Common OMG standards in MDA.

drawings, but they can also be textual, even in a natural languages, but it can adopts several notations and formats. For example a model of a software system could include a UML class diagram, E/R (Entity-Relationship) diagrams, and images of the user interface, while a model of a physical system could include a representation of the hardware and physical environment, and a performance simulation.

1.1.2 MDA Viewpoints and Views

Model-Driven Architecture starts with the well-known and long-established idea of separating the specification of the operation of a system from the details on how that system uses the capabilities of its platform. MDA enables to specify a system independently from the platform that supports it, and to transform the system specification into one for a particular platform.

A *viewpoint* specifies a reusable set of criteria for the construction, selection, and presentation of a portion of the information about a system, addressing particular stakeholder

concerns [11]; in other words, a viewpoint defines the abstractions to adopt to focus on particular concerns within the system. A *view* is a representation of a particular system that conforms to a viewpoint [11].

In MDA terms, abstraction eliminates certain elements from the defined scope and may result in introducing a higher level viewpoint at the expense of removing detail. A more abstract model encompasses a broader set of systems, whereas a less abstract model is more specific to a single system or restricted set of systems. One important capability of MDA is the automation that provides for the transformation between levels of abstraction by the use of pattern.

MDA specifies three viewpoints, that offers levels of separation of concerns to realize a system. The three viewpoints are the *computation independent viewpoint* (CIV), the *platform independent viewpoint* (PIV) and a the *platform specific viewpoint* (PSV).

Computation Independent Viewpoint The computation independent viewpoint focuses on the environment of the system, and the requirements for the system; the details of the structure and processing of the system are hidden or as yet undetermined.

Platform Independent Viewpoint The platform independent viewpoint focuses on the operation of a system while hiding the details necessary for a particular platform.

Platform Specific Viewpoint The platform specific viewpoint combines the platform independent viewpoint with an additional focus on the detail of the use of a specific platform by a system.

The recent version of MDA standard [11] reduces the emphasis on the CIV, and defines a *platform* as a the set of resources on which a system is realized. This set of resources is used to implement or support the system. For instance, a platform can be the organizational structure or a set of buildings and machines (in case of business or domain platform

View	Textual Model
CIM	Priority must be given to shipping the oldest articles.
PIM	The Articles are ordered according to date for shipment.
PSM/OOD	Articles are sorted for shipment() in increasing order, using their attribute date.
PSM/OOP	Use quicksort() to sort Articles, using Articles. Date as sorting key, before passing them to shipment().

Table 1.2: An example on the MDA views: CIM defines the requirements textually using a vocabulary familiar to domain experts. PIM defines the operations that satisfy the requirement, while PSM refines the PIM adding the messages and the operations to invoke in a Object-Oriented design and implementation.

types); or operating systems, programming libraries, and CPUs (when considering computer hardware and software platform types).

A platform model also specifies requirements on the connection and use of parts of the platform, and the connections of an application to the platform. Example: OMG has specified a model of a portion of the CORBA platform in the UML profile for CORBA. This profile provides a language to use when specifying CORBA systems. The stereotypes of the profile can function as a set of markings. A generic platform model can amount to a specification of a particular architectural style.

Considering the previous views, MDA defines the *Computation independent Model* (CIM), the *Platform Independent Model* (PIM), and the *Platform Specific Model* (PSM). MDA refines CIM in PIM and in PSM using model transformations during development process (Fig. 1.5).

Computation independent Model (CIM)

A Computation Independent Model is a view of a system from the computation independent viewpoint. CIM is at the highest level of abstraction; it neglects the processing and the internal structure of the system and offers a model that is independent by computation details.

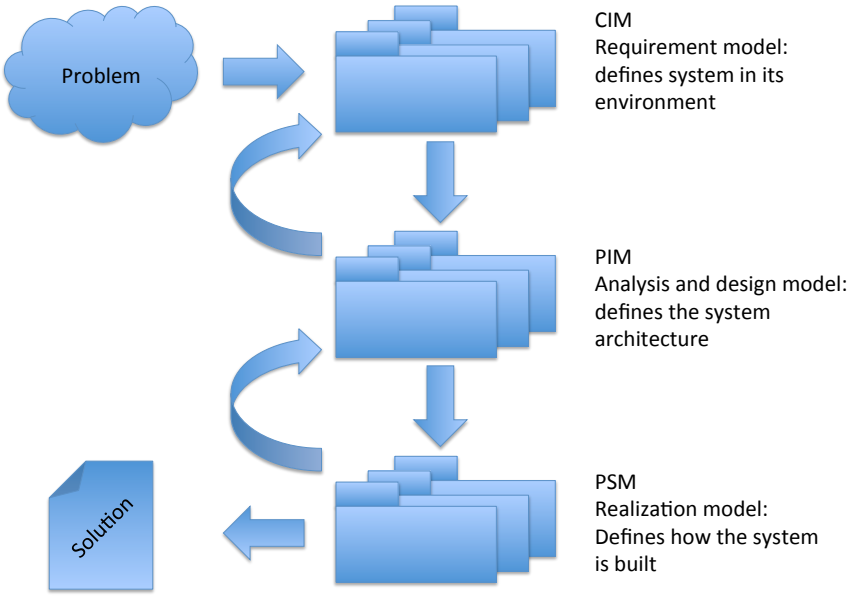


Figure 1.5: The MDA Viewpoints.

CIM is also sometimes called a *domain model*, since it offers a vocabulary that is familiar to the practitioners of the domain and is used for specification. Therefore, CIM plays an important role in bridging the gap between those that are experts about the domain and its requirements on the one hand, and those that are experts of the design and construction of the artifacts that together satisfy the domain requirements, on the other.

Platform Independent Model (PIM)

A Platform Independent Model is a view of a system from the platform independent *viewpoint.* PIM is a model that focuses on the operations of the system, but abstracts the relations that concern a particular technology or execution platform, such as the hardware interface, the programming language and the middleware. Since a PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type.

Platform Specific Model (PSM)

A platform specific model is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform. It can be translated into code to provide a partial or total implementation of the system.

1.1.3 UML extensions of interest

Other UML extensions of interest for the thesis are the *OMG Systems Modeling Language* (SysML) and the *UML Testing Profile* (UTP), which has been proposed in the context of *Model-Driven Testing* (MDT).

SysML is a general-purpose modeling language published by OMG [15] for system engineering, that provides specific support for capturing functional and performance requirements, quantitative constraints, and information flows.

While UML is more software-centric, SysML supports the specification, analysis, design, verification and validation of a wide range of complex systems, that include hardware, software, processes, personal and facilities. SysML partially overlaps with UML 2.0 (Fig. 1.6), it reuses a subset of UML 2 (namely, UML4SysML) but provides additional extensions which have no counterparts in UML or which replace UML constructs as in form of an UML Profile.

One of the main features of SysML is the definition of two new diagram types over UML: *Requirement Diagram* and the *Parametric Diagram*. The first one can be used to model system requirements and their relationships, whereas parametric diagrams can be used for performance and quantitative analysis.

Model-Driven Testing is an MDE activity for V&V [16], it aims at exploiting model-driven approaches also for testing, i.e., test infrastructure and test cases are generated by transformation of models. To this end, an UML standard profile, the UML Testing

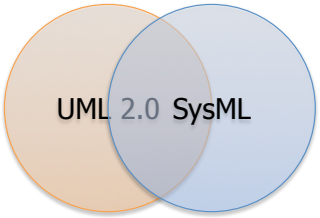


Figure 1.6: The overlap between UML and SysML.

Profile (UTP), can be used to adapt UML as a test specification language: UTP offers the elements to model test architecture, test behavior, test data and test time with UML; by UTP model, test cases and test scripts are derived through transformations, using mapping rules, for instance to Java (using JUnit framework) or TTCN-3. Test cases can be generated automatically starting by behavioral diagrams, such as sequence diagrams, state machines or activity diagrams.

The viewpoints of MDA have also been applied for MDT. The study [17] introduces the concepts of *Platform Independent Test Model* (PIT) and *Platform Specific Test Model* (PST) to separate the test aspects for the general business logic from the platform and technology dependent features of the SUT: this enables to better reuse of testing artifacts for the deployment on multiple target platforms. PIT an PST (skeletons) can be obtained by transformations of PIM and PSM. Test code can be generated by transformation of PIT and PST (Fig. 1.7).

The approach has been extended in [18] with the *Computation Independent Test Model* (CIT): according to the author, the CIT derives from the CIM, and models test objectives and test suite structures combined with overall test strategies used in a specific development process.

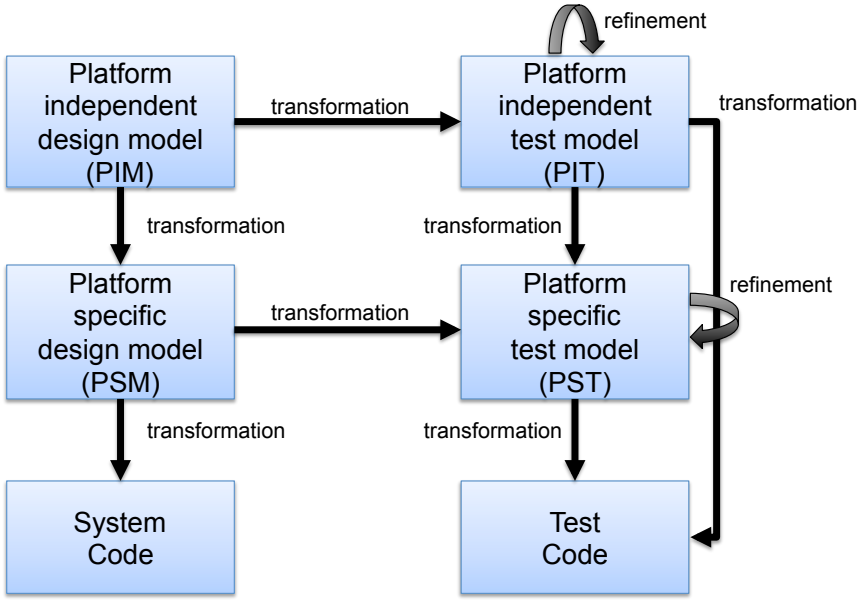


Figure 1.7: The integration of MDA and MDT proposed by Dai [17].

1.2 State of practice of MBE and MDE

Several surveys analyzed the diffusion and the benefits of Model-Based and Model-Driven technologies into industrial practices, after 30 years from the introduction of the first MD tools on the market. However, these analyses are still not enough to get a complete picture about the state of the MD practices. Indeed, an aspect that is often neglected by these surveys is that the utilization of MD techniques is tight dependent on the domain considered in the analysis, which influences the demands of the users as well as the stability of the environment and the availability and maturity of the supporting tools. For instance, if we consider the development of embedded systems, a stabler environment has eased the diffusion of very complex MD tools such as Matlab Simulink or SCADE, that are always evolving in complexity and functionalities since they were introduced on the market, more than twenty and ten years ago. Differently, if we restrict to the general purpose market, the model-driven tools are generally more limited in their scopes, and the unstable environment

requires that they evolve rapidly as new technologies emerge. This observation agrees with the findings of past surveys: the results on the adoption of MB and MD in the industries for the general market differ from the ones in the domain of the embedded system. Even if MDE is always perceived beneficial, the benefits are not so evident like in the domain of the embedded system industry.

MDE in the general industry seems no completely mature yet, and partially debated, with no stable tools and many potentials of model-driven approaches to be exploited. Summarizing multiple observations through surveys of past years in general industry, we can conclude that:

- MDE is spreading with the time in industry, but it is still far to be pervasive. It followed the concurrent evolution of the languages (such as UML) and of the tools: in 2005 practitioners were using MBE for conceptual modeling [19], in 2008 models-centric approaches were perceived better than code-centric ones in most of tasks [20], in 2010 and 2011 MDE has been widely observed in diverse range of industry [21, 22, 23, 24, 25, 26, 27, 28], despite there are many problems and no general and common consensus with these approaches;

- models are mainly used for design and documentation, while the benefits of advanced techniques (such as code generation, test case generation, or model animation) are lowly exploited: models are introduced mostly as an enabling technology inside the process, to enable business that otherwise would not be possible [21, 22, 23];

- UML is gaining the market[1], but the tools are no mature yet, they are considered one of the biggest problem by the industry, that is worried by the easiness of their usage, the vendor lock-in problem, and the interoperability [25, 26, 27, 28];

[1]It is worth to mention, that disagreeing from the previous surveys, the study [29] targeted the use of UML in industry on fifty software designers and found that UML was not used at all or only selectively or informally.

- MDE depends on the business domain and on organizational factors, it need changes inside the personnel, the processes, and company practices: MDE demands for special skills and changes in the importance of developers and software engineers, as retraining software coders to think at a higher level of abstraction which can reveal a difficult task. These aspects have not been well addressed by the MDE, and the current approaches do not adequate to the people, but the people have to adapt to them.

A partially different scenario is observed in the domain of embedded systems, where we can draw the following picture:

- model-based techniques are widely adopted (almost pervasive in automotive domain), and models are used not only for informative and documentation purposes but they were the key artifacts of the development processes [2, 3].

- the needs for introducing models was mainly for shorter development time, and to improve reusability and quality, whereas less than the half had the need to introduce models for exploit formal methods, or because they were required by the standards [2, 3].

- the activities of verification and validation (V&V) had a huge impact by their adoption in the automotive domain [4]: the industry were used to widely exploit model-driven approaches for the early verification of the systems, by techniques such as model reviews, guideline checkers, Rapid Control Prototyping (RCP) and Model- and Software-in-the-Loop Tests, that lead to better quality, reduced development time, due to the shifting of the costs to the phases of requirement analysis and design;

- according to [30] UML is not used widely, due to short lead-time for the software development, or lack of understanding or knowledge of UML models, however this survey, limited to MDE/MDA in Brazilian industry, does not agree with [2, 3] targeting the European industries of embedded systems. These authors found that the

majority of survey participants were using Matlab/Simulink/Stateflow, followed by Eclipse- based tools. The most used modeling languages were the OMGs ones (UML and SysML);

- as for the general-market domain, in the top shortcomings identified there is in the scarce interoperability and the usability issues of the tools, and the high (initial) efforts to train the developers and to adopt these techniques [2, 3] .

Why was the diffusion of MB and MD techniques is different for the general market and for the embedded systems? We claim that this is due to: (i) the different weight of the activities in the development process (more on design and implementation for the general market; and more on analysis and V&V for embedded systems); (ii) the parallel evolution of the code-centric technologies that are available for the development, which raised even more the level of abstraction during the design, and simplified the way the systems are implemented. The hypothesis partially reflects the different focus on the adoption of models in the two domains, since there is more emphasis on design and documentation in the general market, and on the V&V techniques for the embedded systems.

1.3 Benefits and Open Challenges

The previous surveys identify the current state of the adoption of the model-driven approaches into the industry by collecting the opinions of the practitioners on the benefits and drawbacks of model-based and model-driven techniques. However, besides these quantitative data, there is the need of empirical studies that analyze qualitatively and critically the merits and faults of model-driven approaches. Indeed, the success or failing factors of MDE are still unclear, and more research is needed [31].

A systematic review of empirical studies on MDE between the period 2000 to June 2007, has been performed by Mohagheghi and Dehlen [8]. MDE can effectively reduce the cost

and development time, however it depends on the grade of adoption in the development process: a success story is the one of Motorola [5, 6], that used MDE for more than 15 years in a wide spectrum of activities, ranging from protocol implementations up to hand-held devices or network controllers; they experienced an increase in quality and productivity (ranging from 1.2x to 8x) and an approximately 33% reduction in the effort required to develop test cases.

Motorola could achieve these results, only within a mature process that was supported by own-made translators and tools for the model exploitation. Indeed, one common issue of MDE is the absence of well-defined processes [8, 32, 33], as the application of MDE requires changes in the activities, corporate culture and skills of the employees: many software engineering methods are not fitted to use models as main artifacts, and the environments seems not mature enough. Some previous studies attempted to apply pre-existing processes to MDE, or to create own ones, but MDE shifts the importance of many activities to (automatic) transformation rules, and change consolidated development process is not a naive task. The study [7] reports a successful introduction of a MBE process after four years and three projects had been defined and consolidated: there is the need to look beyond the technical benefits of a particular approach to MDE and instead concentrate on social and organizational issues [23].

Moreover, the process becomes a more difficult problem in safety-critical domain, where compliance with certification standards poses additional requirements on the methodologies for product life cycle. For these kind of systems, the major part of costs are for the activities of verification and validation, so rigorous and well-assessed techniques have to be integrated within the development process for the early detection of faults and to guarantee the quality of the product. In addition, non-functional requirements, such as the safety, the reliability and timing requirements, are a primary concern that have to take into account by these processes: current MDE methodologies does not cope well stringent functional requirements

and qualities in current systems, i.e., the ability of these approaches to adapt to rapidly changing hardware and implementation platforms that are highly complex [31].

Parallel to the challenge of the product life cycle, there is the open problem of the supporting tools: they are not mature yet, and influence most of the adoption of MDE. Moreover, the vendor lock-in problem is also perceived as a problem, and the companies prefer to adopt open source solutions or to develop their own tools. Indeed, the tools are not well usable, do not interoperate between themselves, do not keep in synchronization the models at different level of abstractions, are not flexible to collaborative working, and are not suited with the adoption of different models and modeling notations [31]. Thus, model-driven processes have to careful consider the problem of defining the toolchain for supporting the activities.

Chapter 2

Literature Review

2.1 Model-Driven Processes for Critical Systems

This chapter presents a systematic literature review performed with the goal of investigating current methodologies for the engineering of critical systems. As suggested by the studies on the use of MDE in practice, the process is an important aspect to be considered when adopting model-driven approaches, since it influences on the effectiveness of these techniques.

2.1.1 Research questions

The study has been undertaken as a systematic literature review based on the guidelines as proposed by Kitchenham [34]. The focus of our review is on the proposed model-based or model-driven methodologies for the life cycle of critical systems. The research questions addressed by this study are:

RQ1. What is the focus of the studies on MBE/MDE for critical systems, and the relevance of the development methodologies?

RQ2. What are the motivations that lead the research on MBE/MDE development methodologies for critical systems?

RQ3. How does this research benefit from models?

RQ4. What are the current limitations of this research?

2.1.2 Search process

The search process was performed using the digital scientific catalog Elsevier Scopus [35], looking for indexed studies in the database until 2016.

According to our focus, we wanted to select paper that cover model-driven processes in safety- and mission- critical systems. Thus, the query retrieved scientific papers using in the title, abstract or keywords the terms "model-based", and "safety-critical system", including synonyms of both terms. We limited our analysis to the publications in the field of Computer Science and Engineering:

```
TITLE-ABS-KEY
(
   (("Model-based" OR "Model based" or "MBE" )
      OR
     ("MDE" OR "Model-Driven" OR "Model driven" OR "MDA"))
   AND
   (( "safety" OR "critical" or "safety critical"
      OR "safety-critical"
      OR "mission" OR "mission-critical")
      PRE/0 ( "system" OR "systems"
         OR "domain" OR "domains"
         OR "application" OR "applications"))
) AND PUBYEAR < 2016
```

2.1.3 Inclusion and exclusion criteria

Each study was selected manually by the author, based on the analysis of the title and of the paper abstract. We excluded those studies whose focus is not about model-driven or model-based approaches for the engineering of critical systems. In other words, we filtered

out the papers not concerning the life cycle of software (and hardware) systems. Similarly, we discarded retracted articles, and duplicates.

2.1.4 Paper classification

The scientific papers have been manually classified, according to their main contribution, as reported in title and in the abstract. We adopted the following classification:

Typology We differentiated the studies by their main class of focus, according to the categories listed in Tab. 2.1. Since this classification is not orthogonal, we preferred that class more specific for the work. The typologies can be divided into two main branches: *Investigations* and *Contributions*, the latter including *Life Cycles*, *Non-Functional Requirements* and *Infrastructure*.

Modeling Scope We identified if the studies refer to *Model-Based* or *Model-Driven* approaches, and if they are based on OMG standards. We also classified the scientific papers by *Model-Based Testing* or by *Model-Driven Testing*.

Industry Domain We assigned the studies to an industrial domain, according to their scope, i.e., if they focus on one particular type of systems. The Industry domain classes considered are: *Aerospace*, *Automotive*, *Avionics*, *Embedded*, *General-Market* (such as, web applications), *Railway*, *Others*, and *Unspecified*.

2.1.5 Data collection

From each study, we extracted the following data:

- the title, authors, abstract, keywords, and publishing year;
- the source and document type (article, conference paper, etc.);
- the category of the main contribution;

- the kind of techniques based on the models adopted by the study;
- the domain referred by the work.

2.2 Search results

The query provided 858 results: analyzing by authors and titles the results, 16 duplicates were removed. Then, the remaining 842 results were analyzed manually by the author according to the inclusion/exclusion criteria.

The filtering led to the selection of about 50% of the papers, i.e., 423 results were later classified according to the typology categories listed in Table 2.1, as perceived by the reviewer by the analysis of the title and of the abstract.

In order to answer to research questions RQ.2-RQ.4, we reviewed again the studies belonging to the classes of Development Methodologies, Survey, Literature Review and Experiences, selecting and analyzing the relevant ones by the analysis of the full paper. However, we excluded eleven papers by this analysis: the studies written by the author of this thesis, and the papers whose full text could not be retrieved. We could not access to the studies available on SAE International [36], InderScience [37] or AHS [38].

RQ1. What is the focus of the studies on MBE/MDE for critical systems, and the relevance of the development methodologies?

Sorting the typologies of papers by number of studies, the top five typologies (that account for the 51.1% of all the works) are Verification and Validation (15.1%), Safety (12.1%), Development Methodologies (9.2%), Design and Implementation (8.0%), Security (6.6%). The data are tabulated (Tab. 2.1) and represented graphically (Fig. 2.1). These results show how the activities of V&V, Safety and Development Methodologies have deserved primary focus by previous research.

	Typology	Description	Size	%
Investigation	Survey	Investigation conducted by interviews.	2	0,5
	Literature Review	Investigation conducted by analyzing scientific literature.	17	4,0
	Empirical Study	Investigation conducted by experiments.	20	4,7
	Practical Experience	Report on experiences.	22	5,2
Life cycle	Dev. Methodologies	Contribution regarding MB, MD methodologies or on the development process.	39	9,2
	Requirements	Contribution regarding requirements analysis or management, or validation at early stage.	10	2,4
	Design and Impl.	Contribution limited to the design or implementation of the product.	34	8,0
	V&V	Contribution regarding activities of Verification and Validation.	64	15,1
	Certification	Contribution focused on easing or supporting the certification.	21	5,0
	Maintenance	Contribution on the maintenance or post-deploy activities.	11	2,6
	Documentation	Contribution for the documentation of the system.	3	0,7
Non Functional	NF Dependability	Contribution for the dependability of the system.	13	3,1
	NF Security	Contribution for the safety of the system.	28	6,6
	NF Traceability	Contribution on improving the (model) traceability.	4	0,9
	NF Time	Contribution for performance or timing issues in real-time systems.	8	1,9
	NF Safety	Contribution on the safety requirements of the systems.	51	12,1
	Others	Contribution not falling in one of the previous NF typologies.	16	3,8
Infrastructure	Transformation	Contribution on model transformation rules.	20	4,7
	Tools	Contribution on tools for model-based environment.	16	3,8
	Languages	Contribution on languages for modeling.	9	2,1
	Others	Contribution on supporting environments or specific features for the development or V&V.	15	3,5

Table 2.1: The typologies adopted for classifying the studies.

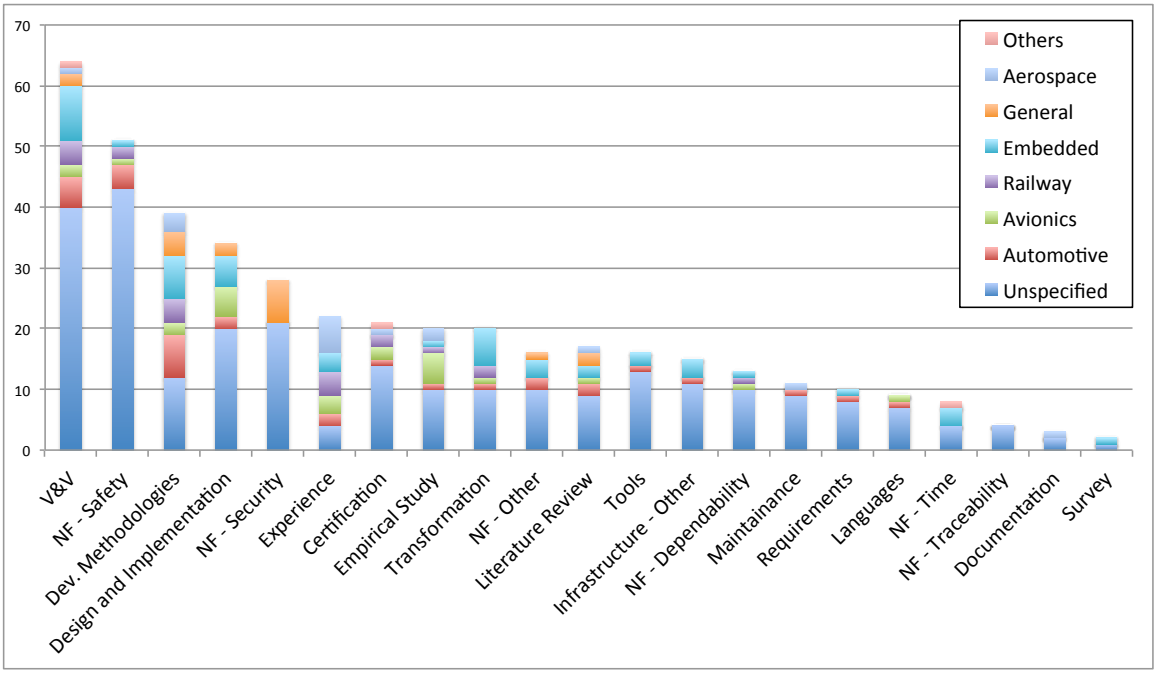


Figure 2.1: Distribution of reviewed papers by Typology and Industry Domain.

By analyzing the industry domains (Tab. 2.2, Fig. 2.2), we can find some interesting points:

- most of papers added contributions considering Embedded and Automotive domains, followed by Avionics and Railway. This is consistent with the fields where MBE techniques are more spread in practice;
- the papers in the class of Security only considered the General Market Industry: this can suggest that security concerns are scarcely addressed in other industrial domains;
- the model transformations have been discussed mainly for domain of embedded systems;
- the studies on the certification encompass all the domains excluding Embedded and General Market domain. This result is consistent with the domains where certification is pursued.

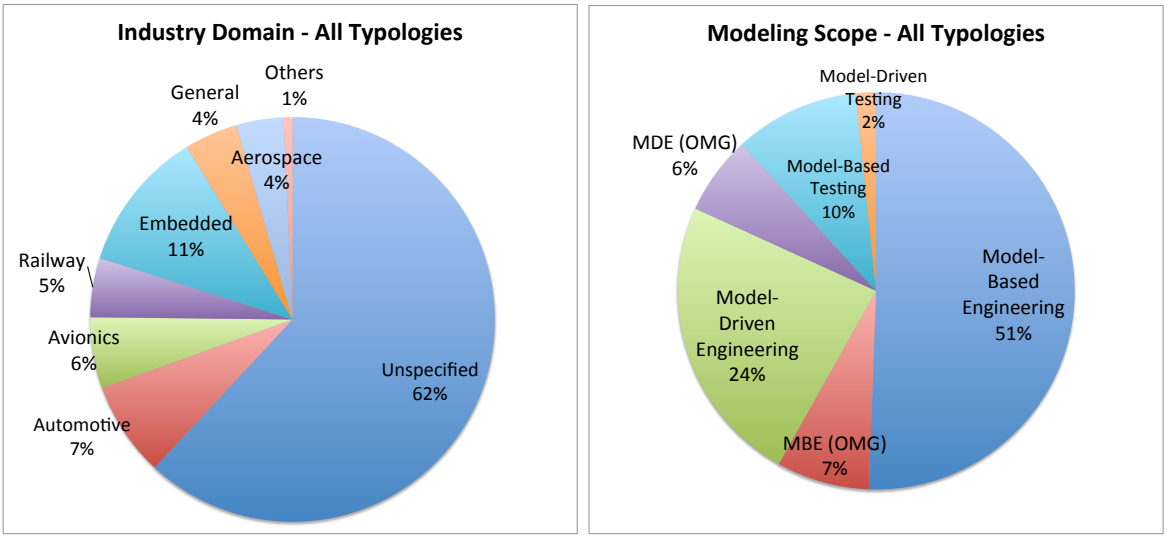
Fig. 2.2 also depicts the Modeling Scope of reviewed studies. Model-Based approaches are more common than model-driven approaches (MBE and MBE (OMG), total 58%, vs. MDE and MDE (OMG), total 30%). OMG standards were adopted by the 13% of the studies. This can be symptom that OMG standards or the OMG supporting tools are not mature yet to be object of more attention by research.

Restricting to the class of studies in Development Methodologies (Fig. 2.3), we observe that they used, as industry domains, Embedded and Automotive Systems, followed by Railway and General Systems, in line with the union of all the classes. However, these studies increase the focus on MDE (MDE and MDE (OMG) 59% vs. MBE and MBE (OMG) 41%) and on OMG standards (24%). Thus more efforts have been spent for model-driven methodologies and compliance with OMG standards.

In this category, the studies can be divided in two branches, according to their objective: in the first branch there are the studies that focused on defining the abstractions and transformations between the models to design, develop and assess the software system. Instead, the second branch includes the studies that proposed software/system life cycle methodologies (exploiting model-based or model-driven approaches). For instance, MDA belongs to the first branch, since it describes a framework of abstractions and transformations for software development. In order to be applied concretely, MDA should be mapped on a complete life cycle process, such as the waterfall model, or the V-model.

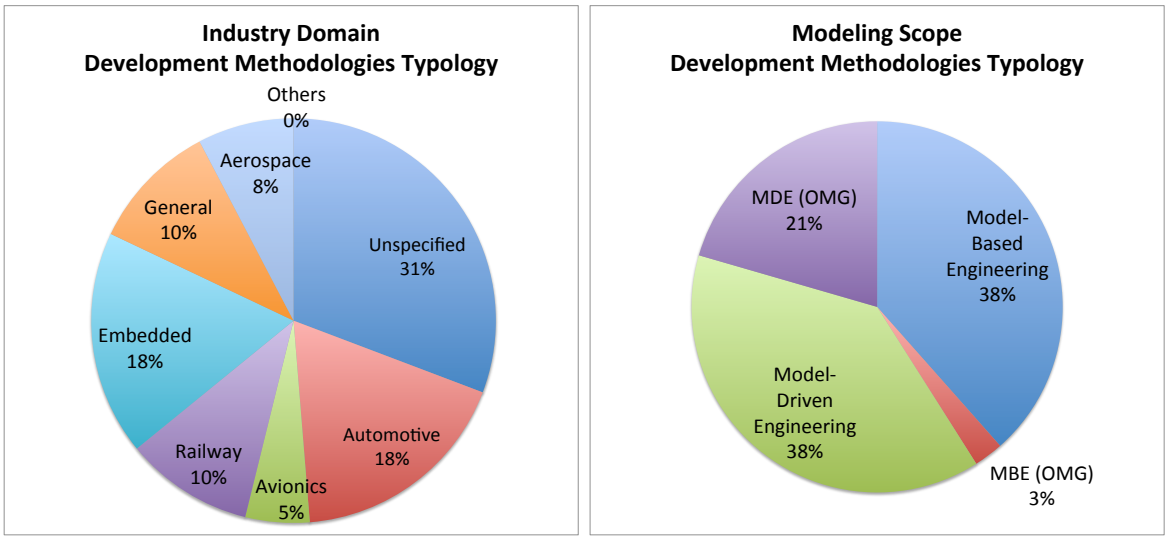
Typology	Unspec.	Automotive	Avionics	Railway	Embedded	General	Aerospace	Others	Total
V&V	40	5	2	4	9	2	1	1	64
NF - Safety	43	4	1	2	1	0	0	0	51
Dev. Method.	12	7	2	4	7	4	3	0	39
Design and Impl.	20	2	5	0	5	2	0	0	34
NF - Security	21	0	0	0	0	7	0	0	28
Experience	4	2	3	4	3	0	6	0	22
Certification	14	1	2	2	0	0	1	1	21
Empirical Study	10	1	5	1	1	0	2	0	20
Transformation	10	1	1	2	6	0	0	0	20
NF - Other	10	2	0	0	3	1	0	0	16
Literature Review	9	2	1	0	2	2	1	0	17
Tools	13	1	0	0	2	0	0	0	16
Infrastr. - Other	11	1	0	0	3	0	0	0	15
NF - Depend.	10	0	1	1	1	0	0	0	13
Maintenance	9	1	0	0	0	0	1	0	11
Requirements	8	1	0	0	1	0	0	0	10
Languages	7	1	1	0	0	0	0	0	9
NF - Time	4	0	0	0	3	0	0	1	8
NF - Traceability	4	0	0	0	0	0	0	0	4
Documentation	2	0	0	0	0	0	1	0	3
Survey	1	0	0	0	1	0	0	0	2
Total	262	32	24	20	48	18	16	3	423

Table 2.2: Typologies of papers by Industry Domain.



(a) (b)

Figure 2.2: Distribution of all reviewed studies by Industry Domain (a) and by Modeling Scope (b).



(a) (b)

Figure 2.3: Distribution of studies in class Development Methodologies by Industry Domain (a) and by Modeling Scope (b).

RQ2. What are the motivations that lead the research on MBE/MDE development methodologies for critical systems?

Hereafter, we restrict to Development Methodologies (39 papers). We can cluster previous research on Development Methodologies into the following classes:

1. studies that targeted to use models as enabling factors for activities not currently performed in the process (such as model checking or requirement validation);

2. works which aim at getting the benefits offered by models for activities generally performed with low assistance (e.g., implementation or automatic test case generation);

3. contributions that ease the procedures related to the certification of critical systems (for instance, using requirement traceability or safety report generation).

Indeed, considering particular enabling factors, new design and development methodologies have been proposed to introduce more rigor in the design [39, 40], to extend MDA

with regards to real-time systems [41], to better support the reuse [42], or to take safety into account [43, 44].

Similarly, several methodologies tried to integrate different models and tools to augment the benefits of model-based approaches [45, 46, 47, 48]. Also, we found applications of model-driven methodologies for particular purposes, such as to enable model-driven hardware [49] and hardware-software generation [50], for Autonomic Network Design [51], to integrate stakeholder quality requirements on dependable systems' information sharing middleware [52], or to introduce a new component-based method for developing masking fault-tolerant systems [53].

Many development methodologies have also been proposed to introduce additional activities of verification and validation, particularly relevant in critical domain. For instance, to enable the assessment of functional and RAMS (Reliability Availability Maintainability Safety) requirements, validation and model-checking [54], or requirement validation [55, 56].

The support of traditional activities, such as code generation or automatic test case generation, also accounts for several works that want to reduce cost and time or improve the quality of the development [40, 57, 42, 58, 59, 60, 61].

Other studies present model-based and model-driven methodologies for the development of certified systems for safety-critical products. In this case, the methodologies have to comply to the requirements prescribed by the standards and must consider particular requirements in the system life cycle.

Grant elaborates in [62] a model-driven software development methodology based on UML and compliant with RCTA DO-178C specification for airborne software systems.

In railway domain, the standard CENELEC EN50128 defines a V-Model software development life cycle: a model-based process compliant with this standard and based on SCADE environment is proposed in [43], whereas [32] discusses a life cycle derived from the OpRail project, that includes UML and model-based techniques during the development

and V&V. Also [63] present a methodology based on Simulink adopted for the development of a SIL-4 safety-critical railway product.

Another methodology based on a V-Model and compliant to the MIL-STD-498 is proposed in [60].

RQ3. How does this research benefit from models?

The research on MBE/MDE dev. methodologies for critical systems essentially benefits from models:

1. to utilize the higher level of abstraction and the automation offered by the models for supporting design, implementation and reuse;

2. to introduce formal methods for supporting V&V activities, such as automatic test case generation, timing analysis, model checking, formal proof, and validation of formal requirements;

3. to model particular concepts (for safety or certification) in order to trace and control elements along whole product Life Cycle, enabling to perform assessments or reporting.

Models have been used in several ways by researchers. The first group tend to exploit available tools, thus they focus on integrating models and languages. Indeed, for critical systems, there is the demand for mature and stable tools: past studies adopted SCADE [43], Matlab/Simulink [45, 55, 56, 63], or IBM Rhapsody [60, 32], or custom solutions, including the adoption of SPIN model checker [59] and OpenModelica [64].

In the first group we cite the study in [45] that proposes a methodology to integrate application models (such as Simulink and Lustre) with architecture models (AADL), in order to analyze functional model and assess the architecture; and the ones in [46, 47] which discuss a framework for the seamless integration of special purpose software and hardware

development tools in one holistic tool-chain, enabling to adapt the model with the most proper tools during the life cycle. In the same group there is the ASSERT process [39], which proposes a methodology composed by a modeling phase, a model transformation and verification phase (that verifies the feasibility of the system), and one of automatic code generation: it adopts multiple notations according to four views of the model (ANS.1, SDL and SCADE and AADL).

The second group of studies use less formal models, such as requirements in natural language or UML 2.0, and discuss different approaches to get the benefits of automation. Part of these studies introduced language dialects, or domain-specific languages (including profiled UML/SysML), to combine the semi-formal language with the benefits of a formal specification. It is worth to note that OMG defined recently new standards to make UML 2.0 formal and to ease the model specification by textual representation: the Semantics of a Foundational Subset for Executable UML Models (fUML) [65], and the Action Language For UML (ALF) [66]. However, these standards seem to have not been exploited yet for critical systems.

Informal requirements have been addressed by [55], using specifications based on Simulink models: the author presents its approach to complete the Twin Peaks process adopted at NASA formal requirements specification and automated test generation, using Simulink/-Matlab development environment; to the same end, Z notation has been adopted by avionic company Rockwell Collins [56] in a sort of model-driven waterfall model for embedded system controllers, that integrate several support tools for automatic code generation (Matlab/Simulink) and for the activities of V&V (SCADE and Reactis). Z notation has been also adopted by [62] to enrich the semantic of a set of UML models which are produced through a series of iterative refinements and transformational processes. Safety requirements have also been assessed in a inspired ISO-26262 process by mapping SysML/UML elements to models in Modelica language and performing simulation [64].

The authors of [58] propose an iterative refinement process augmented with different validation and verification methods to finally generate a correct Java implementation from (Abstract) State Machines: along the process simulation, model review and model checking activities are performed on the model. The study [59] provides a tool-chain for software development integrated with advanced techniques of V&V, such as model-checking and model-based testing for state based event driven systems described with UML 2.0 state-charts whose semantic was completely specified by the authors in a MDA framework. A methodology with contracts is described in [40] based on a meta-model named Common System Meta-model (CSM) which features concepts to support component-based design, and to specify formal and non-formal requirements, supporting whole product life cycle.

An approach focusing on formal methods and based on transformations for model-driven analysis and model-driven testing to assess functional and RAMS (Reliability Availability Maintainability Safety) requirements is presented in [54]: it exploits domain-independent (MARTE-DAM and UTP) and domain-specific languages, to generate formal models, such as Generalized Stochastic Petri Nets. Specialized profiles to exploit UML has been also used in [42], that adopt xUML profile in order to offer an execution semantic to PIM models and to ensure that verification begins in the modeling phase.

The study in [60] proposes to use UML models together with the UTP profile for a Model-Driven Testing methodology, for V&V and better reuse of testing artifacts.

Models have also been exploited for Safety: the authors of [43], propose a methodology based on SCADE Suite integrating a Safety-Process: a model-based validation approach of safety requirements is performed using a safety model in series with a function model that checks the system output; the CORAS project [44] describes a framework that tracks the evolution of the correlation between risk management and viewpoint oriented modeling throughout the life cycle, for model-based risk management of security critical systems adapted with a Rational Unified Process for development.

Model-driven approaches are also being used for generating software-hardware, through transformation of models in hardware description languages (e.g., VHDL): the MADES approach [50] offers a process based on MARTE and SysML to model-driven generate hardware and software for embedded systems, including techniques of verification and traceability.

RQ4. What are the current limitations of this research?

There are still few model-driven methodologies that cover the full software development life cycle, and that are suited to be replace current industrial practices and organizations, especially where certification is pursued and processes must comply with standards.

Indeed, to effectively apply model-driven approaches on a large scale (and hopefully shared among more partners), it is necessary to define strict processes that prescribe the activities and the artifacts to be produced, as well as the methods to evaluate the quality of the output. The most of past studies limited to describe the workflow of their approaches, neglecting the problem of integrating them within industrial product life cycle processes: only the 30% of the studies in the class of development methodologies discussed their approaches within life cycle processes.

However, proposing a methodology inclusive of the whole product life cycle does not mean to prescribe fixed processes: the idea of exploiting one formalism, or monolithic processes suited for all industry needs, is not realistic. We believe that the observations made in the seminal work of Brooks in 1986 [67] can be extended to Model-Driven Engineering: MDE is a sort of one of the *silver bullets* analyzed by the author, since it really can address the essential difficulties of building software systems. Indeed, MDE effectively masks the essential complexity of software and it offers powerful forms of graphic representations of the software (i.e., MDE tackles the essential difficulties of the complexity and invisibility of software). Moreover, MDE automates the artifacts (and code) generation, which is one of the promising features discussed skeptically by Brooks. MDE provides these benefits

because of the formalisms that exploits, which can hide (and not remove!) part of the essential difficulties through the specific abstractions and notations introduced to face the problem. However, no one model can deal with all the systems: according to the specific instance of the problems, one formalism can result more suited than others.

Therefore, model-driven life cycle methodologies should be open not only to multiple formalisms and tools – and be flexible to their seamless usage – but should also be open to a broad range of activities that a company can choose to use, to skip, or to perform in a manual way, according to its needs and business. For instance, one of the most emphasized benefits, the automatic code generation, can be refused if it translates in additional costs for the certification: the study [24] reports a case study about an increase of eight times for the costs for certification because of the lack of readability and inefficiency of code generated; this, in turn, does not mean that a company cannot benefit from model-driven approaches to support other parts of product life cycle.

The approaches [45, 46, 47, 48] moves in this way, trying to better integrate multiple formalisms in MDE, but they lack of a flexible methodology that focuses on the organizational factor.

Indeed, another limitation that has been observed, is that the methodologies are generally too tied to the technology: tools have to support the process, and they should not force the activities. In addition, a company should not rely on the long-term support of the tools that has adopted in own business, and standard languages and open source solutions should be preferred.

By analysis of the full texts we found that around the 2/3 of the studies uses an open language at least for a part of the methodology, i.e., OMG standards, AADL, or Modelica Language [64]. Even if these values are encouraging, all new methodologies should dedicate more focus on open languages, standards and interoperability.

Similarly, MDE should take serious concern of all the forms of verification and validation,

especially for development of critical systems; we believe that as for the implementation as for the V&V there is too much expectation on a *push-button solution*: even if model-checking and formal proof are of utmost importance for the verification, they should not be the main objective of a new methodology, but they should consider a broader set of techniques of V&V, such as equivalence partitioning, boundary value analysis, interface testing, simulation, or model-in-the-loop tests, which we did not find integrated in one methodology for critical systems.

This consideration also applies when considering certification. Indeed, for the production of a safety critical system, such as a SIL-4 system as prescribed by railway standard CENELEC 50128, it is generally possible to customize the development process with a subset of techniques of verification and validation selected among multiple alternatives: model-checking can figure among these options, but a company could prefer to adopt a cheaper or a more easy-to-perform solution. Therefore, a methodology that integrates a wide range of techniques of V&V increases its usefulness and interest on the market.

Summarizing the previous discussion, we can state that:

1. there are still few methodologies that cover full software development life cycle and are flexible but complete to replace current industrial processes, especially for the development of certified systems;
2. there is the need of methodologies that do not limit to one particular formalism or tool, but be open to different models and standards, focusing on the activities, rather than on the technologies;
3. a wide range of V&V activities, even if exploited in current model-driven practices, have not been offered integrated within methodologies. This limits the flexibility of the model-driven processes for industrial needs, even in critical domains.

We found that our observations regarding the limitations of the model-driven engineering for critical systems are in line with the analysis in [33].

Chapter 3

A Model-Driven Methodology for Critical Systems Engineering

3.1 Overview

In this chapter we present a novel model-driven software development life cycle for critical systems based on the V-model suggested by the CENELEC EN 50128 (Fig. 3.1), the European standard for software for railway applications. The activities of the CENELEC V-Model process can be grouped in those concerning development, that are on the left side of the 'V', and those focusing on verification and validation (V&V), that are on the opposite side. The activities of V&V require planning stages that are performed before their actual execution: these planning stages are carried out during design in Fig. 3.1.

Besides the V-Model, CENELEC EN 50128 also prescribes requirements on the artifacts produced at each stage, on the activities to be performed, as well as on the people that have to execute the tasks. For instance, if we consider the highest integrity level (SIL-4), distinct people have to test, verify and validate the product, in order to cross-check their work. The phases adjacent to the 'V', the *Software Planning* and *Software Assessment*, aim at tuning and assessing the activities of the life cycle, defining the tasks to be performed during the process and checking that the product and all artifacts satisfy the requirements and comply with the standard.

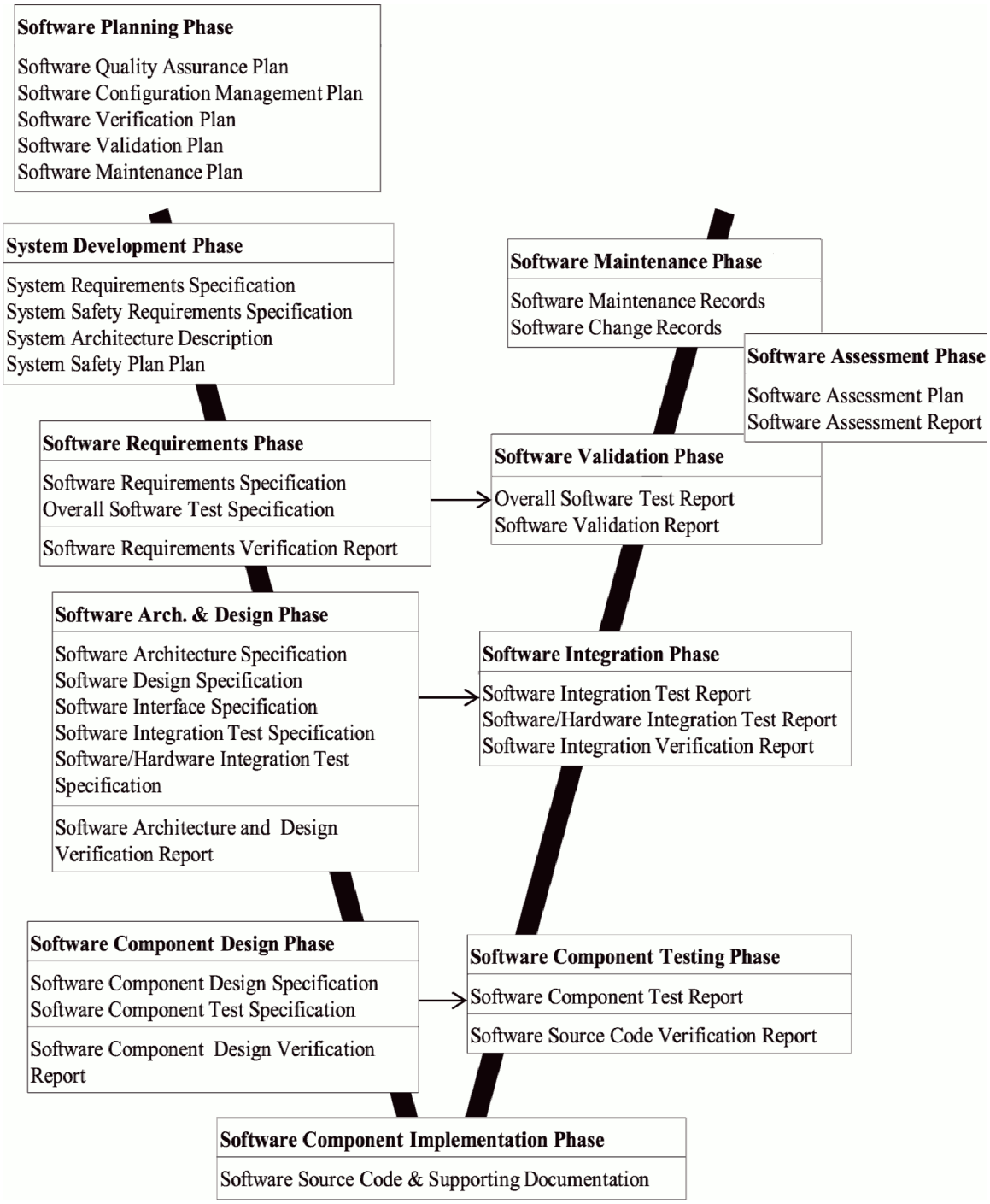


Figure 3.1: EN 50128 Software Development Life cycle

We focused on the core phases of the CENELEC V-Model, on those phases that lie on the 'V' and range from the *System Development Phase* to the *Software Validation Phase* (Fig. 3.1), i,e., on the Software Development Life Cycle (SDLC). This core can easily be integrated in all current industrial processes which comply with CENELEC EN 50128.

The life cycle that we propose is shown in Fig. 3.2: it is divided into a left, central, and right parts. The activities on the left are of forward engineering (i.e., system analysis, design and implementation), the phases in the center are of V&V planning, while on the right side there are the activities of V&V planning execution. Our process integrates in a novel way three different abstractions along the vertical and horizontal progressions of the phases (Fig. 3.3):

1. the abstractions of the Model-Driven Architecture (i.e., the Computation Independent, Platform Independent and Platform Specific Viewpoints, see Sec. 1.1.2), that have been adopted for the development, i.e., for the phases on the left side of the 'V';

2. the abstractions of a Model-Driven Testing Methodology based again on MDA viewpoints, which have been used for the V&V planning and execution, where we define first a *Platform Independent Test Model* and then a *Platform Specific Test Model*, moving from the central to the right side of the 'V';

3. the abstractions implicit to the CENELEC V-Model, i.e., the different focuses at system level, integration-level and component-level, that in a novel way we have combined with the design and V&V viewpoints, for each level of the 'V'.

Indeed, the CENELEC V-Model life cycle adopts implicitly different viewpoints on the system for each level of the 'V': the top level focuses on the system as a whole, the level below uses a viewpoint on the system architecture, then it considers the components and their internal design; finally, the lowest level of the 'V' sees source code details. These abstractions are used on the both sides of the V-Model, for development and V&V.

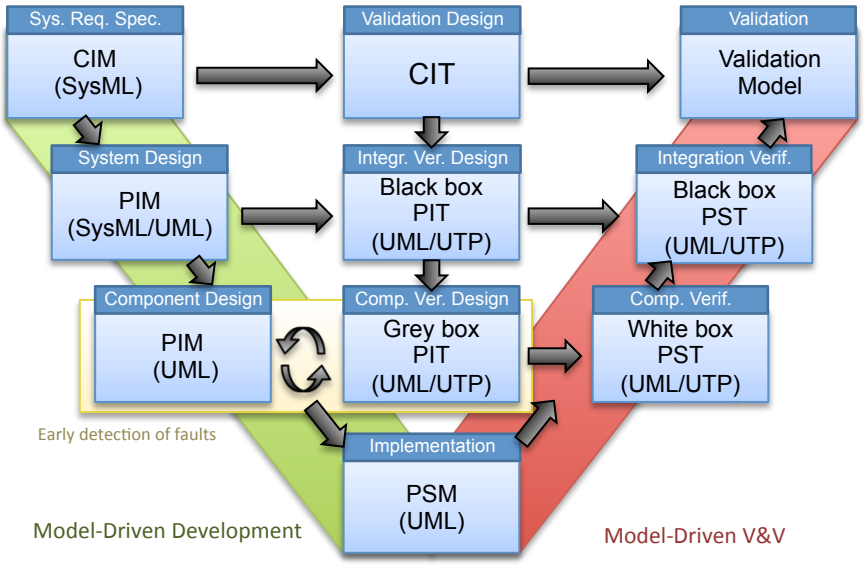


Figure 3.2: The proposed model-driven V-Model life cycle. Boxes show the phases, the models produced, and the formalisms used. The arrows represent dependency between artifacts. The *Component Design* has a dependency with the *Component Verification Design* if it exploits the test model to early detect faults.

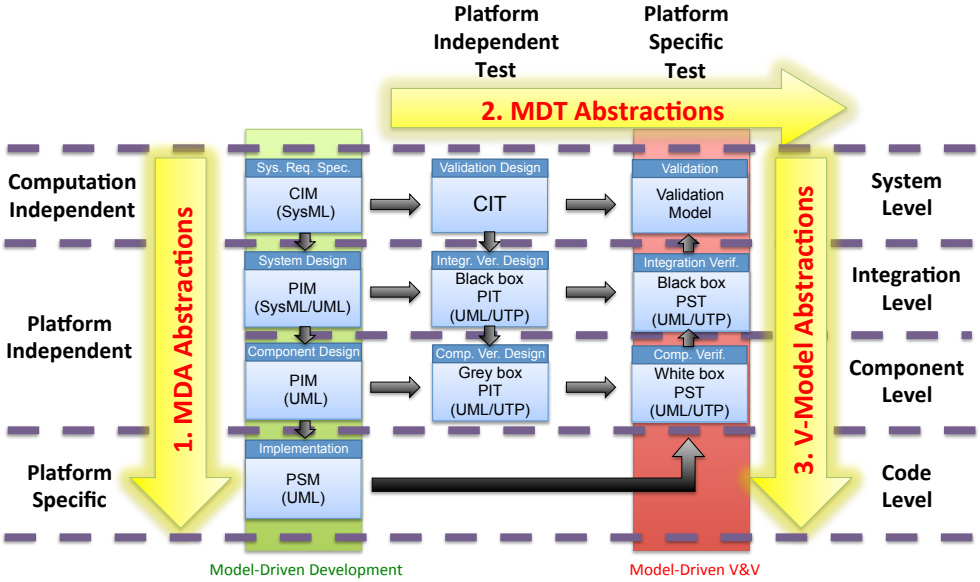


Figure 3.3: The three main abstractions exploited in the proposed model-driven life cycle. Black arrows represent dependency between artifacts. Yellows arrows represents the abstractions adopted and follow their progressions.

The process starts with *System Requirements Specification*, by defining the system environment and software requirements. Then, *System Design* and *Component Design* are carried out. The former defines a high level system architecture, identifying the hardware-software interface, and the components interfaces. Requirements are then allocated to components, and the Designer specifies their responsibilities and expected interactions. Finally, in *Component Design* the Designer completes the components with the internal design, and the *Implementation* concludes the development. For the forward engineering, we define a Computation Independent Model (CIM), a Platform Independent Model (PIM) and a Platform Specific Model (PSM), following the MDA principles.

The V&V planning activities have been isolated in the phases at the center of the V-Model, in *Validation Design*, *Integration Verification Design*, and *Component Verification Design*. These phases are followed by the ones of V&V execution that are performed on the right side of the 'V', i.e., *Validation*, *Integration Verification* and *Component Verification*. For instance, Validation Design produces the Overall Software Test Specification after system requirements have been specified by System Requirement Specification. Then, the actual activity of validation, is performed in *Validation*. at the end of the 'V', after *Integration Verification*, to assess the product conformance to requirements.

For the phases of V&V, we propose a model-driven methodology based on the MDA abstractions: the planning phases use Platform Independent Test Models, whereas the execution phases build Platform Specific Test Models. In fact, the V&V execution phases on the right side of the 'V' benefit from the availability of the implementation, that constraint the technological platform.

Using this methodology we improve the reuse of artifacts of design and V&V, supporting most of activities of the life cycle with model-driven approaches and making the process high suitable for different industrial needs; indeed, the three abstractions act like three degrees of freedom during the process and enhance the reuse at different extents.

The methodology is not tied to a particular formalism, but we propose to adopt OMG standards, i.e., SysML and UML, to be open to multiple tools and promote the interoperability of the models. It is worth to note that custom profiles can be introduced in the process to potentiate the automatic generation of artifacts throughout the whole SDLC, thus reducing the manual efforts.

We defined our process taking as reference the railway standard CENELEC EN 50128 standard, that will be considered throughout the following of this chapter. Since CENELEC is a standard for safety-critical systems, our process is a good candidate to be adopted also in other domains under other standards.

3.2 Roles and Responsibilities

The activities of our life cycle are assigned to a limited number of roles, that comply with the CENELEC EN 50128 standard. We identified the following role and responsibilities:

Requirement Manager is responsible for specifying the software requirements. (S)he shall be competent in requirements engineering and be experienced in application's domain (as well as in safety attributes);

Designer transforms software requirements into a solution, defining the system architecture and developing component specifications. (S)he has be competent in engineering of the application area, and safety design principles;

Implementer transforms design solutions into data, source code or other representations to create the product software artifacts. (S)he has to be competent in engineering of the application area and implementation languages and supporting tools;

Tester develops the test specifications, and handles the test implementation and execution. (S)he has to be competent in the domain where testing is carried out;

Integrator manages the integration process using the software baselines, developing the integration test specification. (S)he has to be competent in the domain where component integration is carried out;

To all of these roles it is required skills with modeling, and experience with model-driven engineering, as well as with the adopted formalisms and tools.

3.3 Model-Driven Development

For the forward engineering, we exploit the abstractions of Model-Driven Architecture integrated with the abstractions of the CENELEC V-Model. The integration of both abstractions leads to split the Platform Independent Viewpoint into two Viewpoints, a PIV at integration level, and a PIV at component level; therefore, from the two PIV we design two different Platform Independent Models (Fig. 3.3).

3.3.1 System Requirements Specification

Phase	System Requirements Specification.
Goal	Define the system and software requirements specification.
Main Role	Requirement Manager.
Input	The specification of the system to realize with a high level description of the system architecture and safety requirements, as well as the plan of the safety plan.
Model	Computation Independent Model (CIM).
Viewpoint	Computation Independent at System Level.
Diagrams	Requirement Diagram, Use Case Diagram, Block Definition Diagram, and behavioral diagrams, such as State Machine Diagram, Sequence Diagram, Activity Diagram, and Timing Diagram.
Output Model	CIM modeling software requirements and their relations.
M2T Output	CIM – Software Requirements Specification.

System Requirements Specification targets to define the system and the specification of software requirements. According to CENELEC, Requirement Manager executes the activities in this phase.

In a model-driven fashion, the Requirement Manager aims at defining a *Computation Independent Model* starting from a high level specification of the system to realize, that is the input of the process. According to the CENELEC standard, this phase also receives, as input, a high level description of safety requirements as well as a high level plan of the safety plan. If safety requirements are also modeled in the CIM, they can be object of automatic analysis in the further stages of the development.

The CIM is essential to model the system requirements, and the relations between them, such as the dependencies and containment relationships. We suggest to define the CIM using SysML, because it turns out particularly suited in this phase due to the *Requirement diagram*, the *Use Case Diagram*, and the *Block Definition Diagram*.

Requirement Diagram is one of the improvements over UML offered by SysML. It can be used to display textual requirements, and to represent the relationships between them, such as containment, derive requirement and copy. Requirement Diagram can also be used to model the relationships between requirements and the other model elements, that can satisfy requirements, verify and refine them.

UML Use Case Diagram models the use cases of the system and the actor. The use cases are specifications of set of actions performed by the system, that yields observable results of interest for actor or system stakeholders. They are generally associated with the main functionalities offered by the system. The actors are roles of users or external systems that interact with the subject (i.e., the system).

Block Definition Diagram is similar to the Class Diagram of UML, but display elements that are not of software domain, thus can reveal more suited than UML for modeling of hardware and software systems.

As an example, a CIM can be composed by a requirement diagram to model the system requirements; by an Use Case Diagram to represent the functionalities and the entities that interact with the system; and by a Block Definition Diagram, to represent the relations (such as information flows) between the system and the other entities, as well as between couples of external actors.

The CIM constitutes one of the output of the phase. Through a transformation it should be possible to derive from the CIM the Software Requirements Specification document.

3.3.2 System Design

Phase	System Design.
Goal	To develop a software architecture that achieves the requirements of the software.
Main Role	Designer.
Input	CIM modeling software requirements and their relations.
Model	Platform Independent Model (PIM).
Viewpoint	Platform Independent at Integration Level.
Diagrams	Mainly Structural diagrams: Component Diagram, Class Diagram, (Protocol) State Machine Diagram.
Output Model	PIM modeling software architecture, components' interfaces and their relations.
M2T Output	PIM – Software Architecture Specification, Software Design Specification, Software Interface Specification.

The System Design phase receives as input the CIM from System Requirements Specification, and the Software Designer targets to refine this model into a *Platform Independent Model* that defines the software architecture, and the interfaces between the components, and between the components and the overall software. The architecture is specified by the means of structural diagrams, such as Component and Class Diagrams. System requirements are assigned to the system components, and the model keeps record of the

traceability.

In addition, the Designer has to model the interfaces between the hardware and the software. Since the viewpoint is platform independent, these interfaces should abstract away as much as possible from one particular technological platform.

PIM describes, for each component, its requirements, interfaces, and, when necessary, its expected protocol, i.e., the expected I/O relations at components' interfaces. The protocol can be modeled with behavioral diagrams. In particular, UML offers to this end the *Protocol State Machines* which models the expected I/O relations of an element without specifying how it realizes the behavior.

At this stage, from the CIM it should be possible to partially derive several artifacts required by the CENELEC standard, in particular: the Software Architecture Specification, the Software Design Specification and the Software Interface Specification document. Among other requirements, the Software Architecture Specification document identifies all software components and, for all of them, the subset of requirements that they cover, as well as if they are new or exiting, their safety integrity level, and if they have been previously validated. The Software Design Specification document mainly addresses the software components and relates them with the architecture, specifies their interfaces with the environment and with other software components, as well as their data structure and main algorithms. Finally, the main concern of the Software Interface Specification document is to specify the required sequences of input and output as well as their valid range of values (that shall be the basis to build the test cases).

Of course, the previous artifacts can be used in the following platform independent activities (such as for the generation of abstract test cases during the platform independent V&V), but it may be requested to adapt these artifacts for a specific platform (as well as all the artifacts derived by these ones) for certification.

3.3.3 Component Design

Phase	Component Design.
Goal	To develop a component design that meets the software design specification.
Main Role	Designer.
Input	PIM modeling software architecture, components' interfaces and their relations.
Model	Platform Independent Model (PIM).
Viewpoint	Platform Independent at Component Level.
Diagrams	Mainly Behavioral diagrams: (Behavioral) State Machine Diagram, Sequence Diagrams, Activity Diagrams.
Output Model	PIM modeling also components' internal design, and behavior.
M2T Output	PIM – Software Component Design Specification.

During Component Design the Designer refines the PIM, to specify the internal design of the components. The Designer identifies all lowest software units, that has to fully detail with the input and output of the interfaces, specifying their algorithms and data structure. To this end, the Designer can use Internal Structure Diagrams, to specify the internal design of each component; and Behavioral State Machines, or Activity Diagrams to complement the structural description with the behavior.

At this stage the PIM is complete, and offers a complete structural and behavioral description of the system. Depending on the formalism adopted during development, the PIM can be also runnable and object of simulation.

The enriched model can be translated into the a Software Component Design Specification, according to the CENELEC terminology.

3.3.4 Implementation

Phase	Implementation.
Goal	To produce software that is correct and verifiable.
Main Role	Implementer.
Input	PIM modeling components' internal design, their relations and behavior.
Model	Platform Specific Model (PSM).
Viewpoint	Platform Specific at Code Level.
Diagrams	Mainly Behavioral Diagrams.
Output Model	PSM modeling all software details specific for the platform including the code level.
M2T Output	PSM – Software Source Code and Supporting Documentation.

Implementation phase deals with the production of software that is analyzable, testable, verifiable and maintainable. Before implementing the code, following a model-driven approach, the PIM is refined into one or more *Platform Specific Model*s, each one bound to a target platform. A PSM adds low level details to the PIM which concern the implementation. For instance, a PSM can adapt the generic types of the variables with the actual ones provided by a particular programming language, or can bind the data and function calls to the specific interfaces offered by a middleware or by the OS that have been chosen for the instantiation of the PIM.

If we consider UML, at this phase the specification of a component's behavior could require the annotation of the model with the syntax of a programming language (e.g., C/C++ or ADA).

The PSM(s) can be at various degree of automation translated into code. to provide a partial or a total implementation of the system. When automatic translators are employed, the settings of the code generators have to be intended as part of the PSM.

3.3.5 Early Fault Detection Techniques in Development

Before the implementation phase, in Component Design phase the PIM already offers a *behavior-complete* view of the system at component level, thus it can be object of simulation or animation. Model animation can be used in the process as a form of early fault detection: animated behavioral diagrams (such as state machines or activity diagrams) support the engineer in assessing the quality and correctness of the design.

However, depending on the actual formalism adopted, it could be difficult to build a runnable PIM without completing the specification with a programming language. These extra annotations can be reused for a PSM, or only be used for making a system prototype.

Considering OMG standards, we can also employ the Action Language for UML (ALF) [66], that enables to complete the model specification with a textual representation not bound to a particular programming language.

Techniques of prototyping/animation during modeling are also recommended by the CENELEC standard.

3.4 Model-Driven Verification and Validation Design

For the activities of V&V, we designed our process based on UML and UTP profile, in order to adopt OMG standards also for the Test Models. However, depending on system's domain and on the particular kind of verification to perform, domain-specific languages could be easier to employ for these activities.

3.4.1 Validation Design

Phase	Validation Design.
Goal	To analyze the actors' behavior to design validation tests.
Main Role	Tester.
Input	CIM modeling software requirements and their relations.
Model	Computation Independent Test Model (CIT).
Viewpoint	Platform Independent Test at System Level.
Diagrams	Structural and behavioral diagrams of the system environment.
Output Model	CIT offering a model of the environment and expected system interaction.
M2T Output	CIT – Overall Software Test Specification.

The activities in *Validation Design* focus is on the actors, rather than on the system, and the goal of the phase is to design overall system (validation) tests.

Here, the Tester models the behavior of the actors and of the environment in the *Computation Independent Test Model* (CIT), starting from the requirements modeled in the CIM. It uses behavioral diagrams, such as Sequence Diagrams, or State Machines Diagrams, or Activity Diagram, to model the expected interactions and behavior of system actors.

CIT is exploited to analyze the actors' behavior, and to design validation tests, e.g., in form of UTP sequence diagrams, that model the actors' interactions with the system. For instance, considering a railway interlocking system, a CIT could model the behavior of trains and station operators, based on the historical/expected interactions they exchange with the system.

Moreover, since CIT interfaces are the complement of those of the system, if the Tester creates a runnable model, then in-the-loop tests can be performed on the system during next phases. By simulation of the CIT, it could be also possible to derive automatically validation tests, based on the modeled operational profile.

By model-to-text transformation of the CIT, the Overall Software Test Specification document can be partially derived. According to CENELEC EN 50128, this artifact identifies for each required system function test cases to be performed on the completed software.

3.4.2 Integration Verification Design

Phase	Integration Verification Design.
Goal	To design integration tests to show that components behave correctly integrated together.
Main Role	Integrator.
Input	PIM modeling software architecture, components' interfaces and their relations. CIT offering a model of the environment and expected system interaction.
Model	Black Box Platform Independent Model (BB-PIT).
Viewpoint	Platform Independent Test at Integration Level.
Diagrams	Static diagrams and behavioral diagrams for the test harness and test case specification.
Output Model	BB-PIT modeling the testing infrastructure and the integration level test cases.
M2T Output	BB-PIT – Software Integration Test Specification.

In Integration Verification Design, the Integrator has to design integration tests to show that components behave correctly when integrated together. To this end, (s)he defines a model of the expected behavior of the system's components, which is independent from their inner design. Indeed, at this stage the viewpoint is at Integration Level, and the internal architecture of the components was not modeled in the input model, i.e., in the PIM at Integration level. Therefore, we refer to the model defined during this phase as *Black Box Platform Independent Test Model* (BB-PIT).

BB-PIT provides static and dynamic views of the system's components, and it is used to support functional testing in the unit/integration/system verification (Fig. 3.4). The static

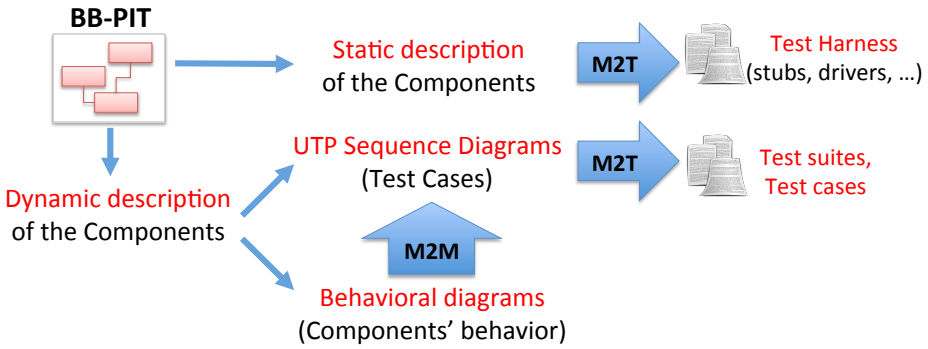


Figure 3.4: The transformations of the BB-PIT.

description supports the generation of test harness, such as stubs and drivers for unit and integration testing. The dynamic description is composed by:

- UTP Sequence Diagrams, by these diagrams is possible to generate test suites or test cases;
- other behavioral diagrams, such as Behavioral State Machines or Activity Diagrams, defined starting by requirements assigned to each component in the PIM model. By these diagrams through transformations, it is possibile to derive the additional test cases (i.e., UTP Sequence Diagrams).

Differently from design model, the BB-PIT can model the behavior of one component with more than one state machines. In fact, the purpose of behavior modeling is different between the PIM and the BB-PIT: the PIM specifies how to build the system, and represents the specification that an actual implementation must comply with; a BB-PIT describes the expected behavior in a way to verify its correspondence between requirements and implementation (e.g., by using the BB-PIT for test case generation, the description represents the specification that test cases must comply with). Since test suites can be composed by grouping test cases derived by several state machines, test cases can be generated by multiple (and easier) state machines which focus on different subsets of component

functionalities.

By the BB-PIT, a preliminary Software Integration Test Specification document is derivable using using a model-to-text transformation. Depending on the particular platform specific transformations of the PIM, this document can be refined with additional details during Integration Verification.

It is finally worth noting that, since BB-PIT derives from requirements and is barely influenced by design details, it can support validation too.

3.4.3 Component Verification Design

Phase	Component Verification Design.
Goal	Design tests to confirm that components perform their intended functions.
Main Role	Tester.
Input	PIM modeling components' internal design, their relations and behavior. BB-PIT modeling the testing infrastructure and the integration level test cases. CIT offering a model of the environment and expected system interaction.
Model	Grey Box Platform Independent Model (GB-PIT).
Viewpoint	Platform Independent Test at Component Level.
Diagrams	Behavioral diagrams for the test case specification.
Output Model	GB-PIT modeling the testing harness and component test cases.
M2T Output	GB-PIT – Software Component Test Specification.

Component Verification Design targets to design tests to confirm that components perform their intended functions, checking how they interact to perform the function. Another activity of this phase is to confirm that all parts of the system are tested.

Following a model-driven approach, the Tester refines the BB-PIT defining the *Grey Box Platform Independent Test Model* (GB-PIT). The GB-PIT starts from the functional black box tests defined by the BB-PIT and refines the model, exploiting the PIM at Component Design level, which offers an internal view of the system. Similarly to BB-PIT, also the

GB-PIT uses mainly behavioral diagrams, for test case generation and specification.

Following this flow, engineers focus on a functional V&V modeling (at unit/integration/system level) in the *Integration Verification Design* (e.g., in a bottom up testing approach during integration testing), and then move to functional and structural V&V modeling in this phase.

Using a model-to-text transformation, by the GB-PIT it should be possibile to derive a Software Component Test Specification document that defines the tests to show that each component performs its intended functions and the degree of test coverage (of the model elements) reached and required. This document has to be refined with in Component Verification, considering a specific target code.

3.4.4 Early Fault Detection Techniques in V&V Design

When the activities of V&V design are completed, we reach a stage where a PIM and the CIT are available, and abstract test cases have been defined in the GB-PIT. Multiple techniques of early fault detection can be used depending on the details of the models (Fig. 3.2):

1. the test cases defined in the PIT can be executed on the PIM, to perform a preliminary verification of the design model;

2. if the PIT is runnable, the environment model defined in the CIT can used to perform model-in-the-loop testing;

3. if CIT and PIM models can be subject of model checking, in the model-in-the-loop configuration we can assess the absence of any undesired condition in operation.

The early detection of faults in the requirements or in the design model leads to a reduction of overall costs of development. The study in [4] reports that companies not

performing the model-in-the-loop testing were finding almost 30% more errors in module test, where their correction is more expensive.

3.5 Model-Driven Verification and Validation Execution

The activities of V&V execution, i.e., the ones on the right side of the V-Model, refine the models created by the Design phases, adapting them to the specific platforms on which the software has been implemented.

3.5.1 Component Verification

Phase	Component Verification.
Goal	To assess the correctness of system components.
Main Role	Tester.
Input	Software Source Code and Supporting Documentation. GB-PIT modeling the testing infrastructure and the component test cases. PSM modeling all software details specific for the platform to code level.
Model	White Box Platform Specific Test Model (WB-PST).
Viewpoint	Platform Specific Test at Component Level.
Diagrams	Refinement of behavioral diagrams UTP Profiled and of the model.
Output Model	WB-PST defining the component test cases adapted for the specific platform.
M2T Output	WB-PST – Software Specific Component Test Specification, Test Source Code (Test Harness, Test Cases, etc.), Software Component Test Report.

Component Verification aims at assessing the correctness of system components. In a model-driven approach, before performing the actual verification, the Tester focuses on a model.

Indeed, the Tester refines the GB-PIT with additional details that derive from the target platform considered for PSM and implementation. This can lead to define new test cases or to adapt for a specific platform the abstract test cases and behavior of GB-PIT. Considering

the level code awareness of this model and the component level viewpoint, we named it as *White Box Platform Specific Model* (WB-PST).

For instance, the WB-PST can calculate now the test coverage on the basis of the final system code. Together with the model, also previous artifacts can now be refined at this stage: the Software Component Test Specification generated by the GB-PIT can be refined in the Software Specific Component Test Specification that includes details deriving by the implementation.

It worths to notice that the WB-PST shall not be conceived to assist only testing, but the model can be exploited to support any kind of verification. As an example, a WB-PST can derive consistent and efficient code review plans by considering the component software metrics and implementation details.

By the WB-PST it is possibile to apply model-to-text transformation to obtain Test Source Code, e.g., the JUnit/TTCN-3 Test Harness with test cases. After the execution of the test cases, a Software Component Test Report can be generated, that includes the test results and the achieved degree of coverage.

The separation between PITs and PSTs raises another discussion: in order to perform testing on the actual implementation, a SUT Adapter is generally needed as part of the testing harness. A SUT Adapter can translate high level interactions to low level messages which are actually exchanged with the SUT. As we have seen, the refinement PIT-PST enriches the model with additional tests and details, that derive by the platform. Depending on the SUT, we can include into this definition also the transformation of a generic SUT Adapter (defined in the PIT) to a specific one in the PST.

3.5.2 Integration Verification

Phase	Integration Verification.
Goal	To assess the correctness of the interactions of system components.
Main Role	Integrator.
Input	Software Source Code and Supporting Documentation. BB-PIT modeling the testing infrastructure and the integration level test cases. PSM modeling all software details specific for the platform to code level.
Model	Black box Platform Specific Test Model (BB-PST).
Viewpoint	Platform Specific Test at Integration Level.
Diagrams	Refinement of behavioral diagrams UTP Profiled and of the model.
Output Model	BB-PST modeling integration level test cases with platform specific details.
M2T Output	Software Specific Integration Test Specification, Software Integration Test Report.

Integration Verification phase targets to perform integration test and verify the interactions among the components, when integrated together. Here, the Integrator refines the BB-PIT model, defining the *Black Box Platform Specific Test Model* (BB-PST). Similarly to the WB-PST, the BB-PST can exploit platform specific details to complete the integration test specification of the BB-PIT. Therefore, the BB-PST refines the existing model and can add new behavioral diagrams (UTP Profiled) to define test cases.

For instance, the BB-PST can be exploited to perform interface testing, a technique that is highly recommend by CENELEC EN 50128: interface testing is executed knowing the actual domain of all interface variables, and selecting particular input to assess the behavior of the (integrated) components (e.g., at their normal, boundary, or invalid values).

By the BB-PST it is possible to generate integration test cases, that are executed on the implementation. In addition, by model-to-text transformation it is also possible to derive a Software Specific Integration Test Specification document (that complements the

Software Integration Test Specification document with the new information of the PST) and a Software Integration Test Report, which describes the results of the integration testing.

3.5.3 Validation

Phase	Validation.
Goal	To assess that the system meets its system and software requirements.
Main Role	Tester.
Input	Software Source Code and Supporting Documentation. CIT offering a model of the environment and expected system interaction.
Model	Validation Model.
Viewpoint	Platform Specific Test at System Level.
Diagrams	Structural diagrams for modeling the adapters for the software and hardware specific interfaces.
Output Model	Validation Model with CIT in-the-loop configurations.
M2T Output	Overall Software Test Report, Software Validation Report.

In Validation phase, the Tester has to assess that system and software requirements are met. Therefore, (s)he executes the overall system tests defined in the CIT. Moreover, if the CIT is executable, the Tester can put the CIT and the software in-a-loop, to perform software-in-the-loop and hardware-in-the-loop testing.

SIL and HIL testing can be used for validation testing, since the CIT is a model of the environment (i.e., of the system's actors); as well as for performance testing, such as stress and load testing, by generating representative operational profile for the system. It is worth to note that performance testing is highly recommended by the CENELEC EN 50128 standard for the verification of SIL-4 systems.

At this stage it is possible to partially generate the Overall Software Test Report and Software Validation Report with the results of testing on the system.

3.6 Discussion

In this chapter we have presented a novel model-driven software development life cycle suited for critical systems. The methodology is in agree with the requirements of CENELEC EN 50128 standard; it uses a compliant V-Model with the required activities and roles, enabling to partially generate several artifacts prescribed by the standard. The process supports many V&V techniques mandatory or highly recommended for the certification of SIL-4 products, and guarantees the independence of models used for design and V&V, as required by EN 50128

The process combines three levels of abstractions in the 'V', derived by MDA, MDT and CENELEC V-Model. The V-Model abstractions have been originally integrated with the MDA viewpoints of development and V&V phases, to design or assess particular aspects of the system:

- at System Level, the design focuses on system's requirements and actors, while the V&V phases focus on the environment and on the system validation tests;
- at Integration Level, the design phase models the system architecture (i.e., the components and their interfaces), and the V&V activities focus on *functional testing*, since the viewpoint offers a black box view of components;
- at Component Level, the development completes the design with the internal details of the components, and the V&V concentrate on *structural testing*.

Using these abstractions, the engineers benefit from the different viewpoints of model to better focus on the relevant aspects of the system during the various phases of development and V&V.

We proposed to use OMG standards for development and V&V; this enable to employ commercial and open source tools to support the process, and increases the interoperability

of the models. However, a certification standard can limit the tools that can be used in the process, since the qualification can request tool developed with the same rigor as the software under development itself. In particular, CENELEC EN 50128 defines three classes of tools for the compliance:

T1 are tools that generate no output that can directly or indirectly contribute to the executable code (including data) of the software;

T2 are tools that supports test or verification of the design or executable code, where errors in the tool can fail to reveal defects but cannot directly create errors in the executable software;

T3 are tools that generate output which can directly or indirectly contribute to the executable code (including data) of the safety related systems.

Tools in class T3 need strong evidence of their quality, based on strict validation process, or on other arguments, such as a suitable combination of history of successful use (as in [63]). Tools in class T2 require less efforts: if we consider a tool of automatic test case generation, a manual review of the test suites can be enough to detect and recover the malfunctioning of a test generator. Therefore, depending on the specific needs and availability of supporting tools, other formalisms can selected in our process, as long as they do not interfere with the viewpoints uses for each phase.

The process does not natively support model checking or formal proof during software requirements specification, design and implementation, but these techniques could be introduced in the process adopting other formalisms. Even if these techniques can be highly recommended by the standards they can generally be replaced by other alternatives. For instance, to certify a EN 50128 SIL-4 product, formal methods are not mandatory, and modeling approaches can be preferred to the former for software design and implementation, and V&V. As an example, an approved combination of activities for verification and testing

is dynamic analysis and testing, traceability, test coverage for code, functional testing and static analysis (or software error and effect analysis).

Thus, considering the tool qualification and task selection, model-driven methodologies for safety-critical system should not fix the techniques to use in the process, but they should support a wide range of different tasks and verification approaches, in order to be flexible and customizable according to the company's needs: depending on the product under development, code generation techniques or model-checking could be not convenient for high safety-critical systems, whereas other techniques can turn out more cost-effective.

The proposed life cycle supports a wide range of V&V activities, through the multiple abstractions used for modeling. Indeed, the methodology natively supports techniques of functional testing, structural testing, interface testing, validation testing, system simulation/animation, model-/software-/hardware- in-the-loop-testing, load, and stress testing. Most of these techniques are mandatory or highly recommended by CENELEC EN 50128.

Moreover, we support approaches of early fault detection during the development, through animation/simulation; and during the V&V design phases, by executing test cases on the PIM or performing model-in-the-loop testing or other forms of assessments. In particular, model-in-the-loop tests seem particularly effective to reduce costs of development, since companies not performing such testing were finding almost 30% more errors in module test [4].

Considering the scientific literature, other authors proposed model-driven V-Model life cycles in agree with CENELEC EN 50128 [7, 32, 43], however our methodology possesses particular peculiarities that distinguish our proposal from past studies: in particular, none of these previous studies used the abstractions that we have proposed, which support a broad ranges of V&V activities.

The experience of introducing formal model-based design and code generation in the development process of a railway signaling manufacturer is reported in [7]. The authors

propose a V-based development model that split into two verification branches, one for the activities performed on the models, and other one for the tasks concerning source code and system. Differently from our proposal, their process is tightly integrated with Simulink/Stateflow platform, and is not based on the MDA and MDT viewpoints; thus their methodology is more rigid (for the supporting tools and prescribed activities) and limit the reuse on multiple target platforms.

In [43] the authors essentially propose a model-based SDLC bound on SCADE, and integrating in the 'V' additional safety-oriented activities exploiting the models. They also introduce a *verification model*, however this is not conceived to perform in-the-loop or performance testing.

The study in [32] performs a deep analysis on the methodological aspects of supporting a certification-oriented process with UML and model-driven tools. They introduce as reference a CENELEC EN 50128 V-Model process and discussed the benefits and drawbacks of using UML in the process throughout the life cycle. Their discussion involves much of the support that we exploit in our process, but they mainly analyze the use of UML in the life cycle, without proposing one particular model-driven SDLC: they conclude that no mature and consistent methodology has been found yet, despite UML can improve the development of safety-critical systems in practice today.

Considering the abstractions that we adopted in our process, a study is presented in [68], which integrates MDA and MDT OMG approaches in a V-Model process based on MIL-STD-498. However, they did not consider the CENELEC V-Model abstractions, and supported less activities of V&V.

Summarizing, we can state that our life cycle addresses the limitation of the previous research (Sec. 2.2):

1. our model-driven process originally exploits three different kinds of abstractions to support the software development life cycle, it is based on the safety-critical standard

CENELEC EN 50128 and aims at being suited to be integrated into current industrial practices for the development of high critical certified systems;

2. we conceived our life cycle to adopt OMG standards, in order to be supported by commercial or open source compliant tools; however, the process is not tied to one specific technology, and it can be easily adapted to other formalisms and supporting tools, that can work with the key viewpoints of the methodology;

3. the process exploits MDA viewpoints to enhance the reuse of all project artifacts, and integrates a wide range of V&V activities, through the multiple abstractions used for modeling. This enables the industries to adapt the process to their needs, choosing the most cost-effective combination of alternatives for the engineering of critical system.

Chapter 4

Model-Driven In-the-Loop Testing

4.1 Introduction

In this chapter, we focus on the topmost part of the proposed life cycle, and we detail how our methodology exploits model-driven techniques to define an environmental model during the phase of *Validation Design*, and describe the particular benefits that this model can provide to the product life cycle.

4.2 The Computational Independent Test Model

The goal of the *Validation Design* phase (Sec. 3.4.1) is to analyze the actors' behavior to design validation test. To this end the Tester models the *Computation Independent Test Model* (CIT) that, by means of behavioral diagrams (e.g., Sequence, State Machine or Activity diagrams), specifies the expected behavior of the environment when interacting with the system. As for the other Test Models, such as the BB-PIT (Sec. 3.4.2), these behavioral diagrams can be used to derive validation test cases, e.g., in form of UTP sequence diagrams (see Fig. 3.4), which are executed during the *Validation* stage during validation testing.

We highlight that the definition of CIT also appears in previous studies. Indeed, the authors in [18] define the CIT as a model derived from the CIM, which contains test objectives and test structures from the business objectives with overall test strategies used in

a specific development process. However, our definition is different from their one, since we use the CIT as a model for the actual design of validation tests that abstract from the computation details of the system under analysis (SUT).

Besides to develop validation tests for the SUT, we also propose to develop the CIT as *an executable model of the environment*, i.e., the CIT becomes a *symmetric SUT*, with an interface that is the complement of the one of the PIM. This peculiarity enables to be introduce multiple forms of V&V during the product life cycle, which are discussed in the following.

Model-in-the-loop testing

Since the CIT has an interface complementary to the one of the PIM, we can put the two models in-a-loop, as shown in Fig. 4.1, and the CIT can be used to perform model-in-the-loop test (since it is runnable), enabling to:

- validate the system against its expected interactions with external actors;
- create a simulated environment to reason about the operational aspects of the system in its environment (also through model animation).

Model-in-the-loop (MIL) testing can be executed as soon as the PIM is available, i.e., during the *Component Design*. Therefore, the CIT enables to a form of early fault detection

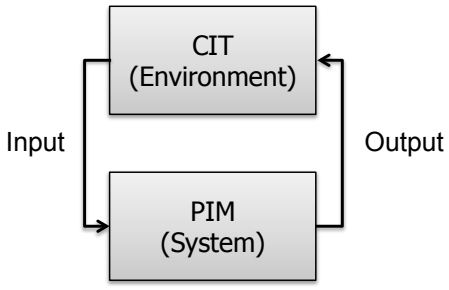


Figure 4.1: The CIT and the PIM in configuration *in-the-loop*.

during the development (Fig. 3.2).

In particular, if the Tester can use additional/external sources to model the actors' behavior (such as domain knowledge or historical data), then the CIT can also identify missed or wrong software requirements, by assessing the actual behavior of the system in a simulated environment. The CIT becomes in this way a mean for cross-checking the input requirements, as specified in the CIM. For instance, considering the railway domain, a CIT could model the behavior of the trains according to the signals provided on the railway by the signalling system: the correct interlocking can be verified by the simulations performed through in-the-loop tests.

The CIT has an interface symmetric to the PIT: when the PIM is refined in a PSM, the interfaces of the two models do not match anymore. In order to execute MIL tests we propose to develop a software adapter, to convert the interface of the PIM with the one of the PSM. Exploiting adapters, the Tester can perform MIL tests also on PSMs.

Software- and Hardware-in-the-loop Testing

The idea of using an adapter for linking the CIT with a PSM, can be applied also for the system implementation: when a system implementation is available, the Tester can build an adapter to allow the CIT to interact with the actual SUT.

This solution allows to perform Software- and Hardware-in-the-loop Testing on the SUT.

Performance Testing

CIT also enables to performance testing, a technique generally adopted during the development of critical systems, and recommended by the certification standards. Indeed, since the CIT models the actors' behavior, it can be used to generate system inputs representative of the operational profile. Parametrizing the number of agents (i.e., the number of instances of the actors) we can perform load and stress testing.

For instance, if we consider a mission critical web service, a CIT can model the typical

sequences of requests that the clients perform on the SUT. Instantiating multiple models of the clients, we can assess the response time, and other performance metrics of the PIM, while it is subject to multiple concurrent requests.

Model-checking

Depending on the particular formalism adopted for modeling, we can also introduce techniques of model-checking on the *in-the-loop model*: indeed the Tester can assess the absence of any undesired condition during the system operation, through the analysis of the combination of states of the PIM and of the CIT together.

Back-to-back testing

A particular case is when the CIT coincide with the PIM, i.e., when the PIM itself possess symmetric interfaces: for instance, this happens with *peer-to-peer* systems, i.e., when the system communicates with other instances of itself, behaving as client and as server at the same time. For this kind of system, it is possible to instantiate two PIMs, and to put in-a-loop each other, in order to perform *back-to-back testing*.

For instance, the *Inter-Vessel Traffic System Exchange Format* (IVEF) service is a common framework for the exchange of maritime information between shore-based e-Navigation systems. These systems connect each other in order to exchange data about the vessels, behaving as server for other IVEF clients, and acting as client to get information from other IVEF servers. By linking crossed the interfaces of two IVEFs, we can execute back-to-back testing (Fig. 4.2).

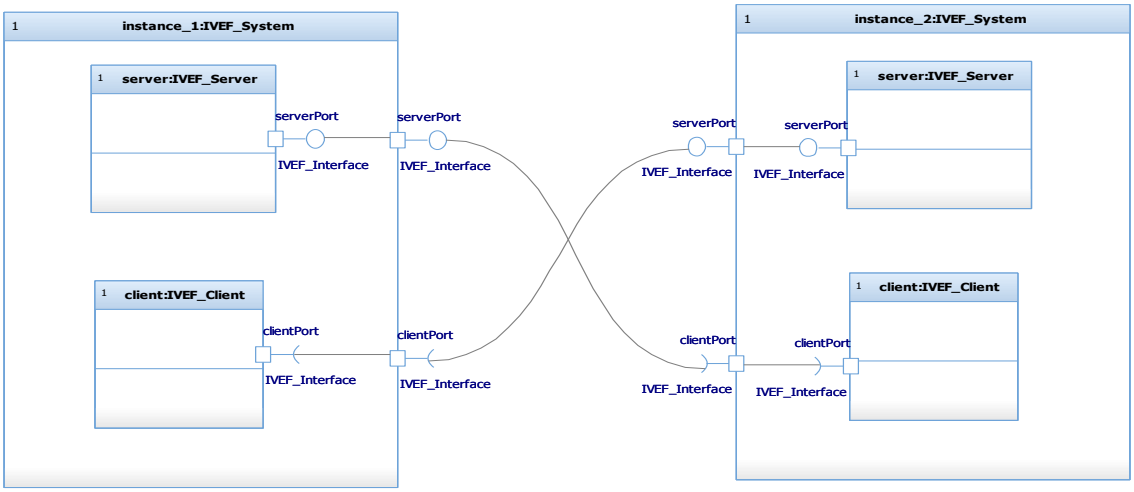


Figure 4.2: Back-to-back testing

4.3 Discussion

We have presented in detail the benefits of the CIT: it offers an *executable* model of the environment, which captures the behavior of the system's context and possesses interfaces *complementary* to those of the SUT.

CIT turns out to be useful to create an environment-aware specification of the system, supporting the definition of a validation test plan, and to set-up a framework to perform model-, software- and hardware-in-the-loop tests.

These kinds of tests enable to detect design flaws at early stages of the development life cycle, shifting the cost of the development from the phases of V&V to the ones of requirement analysis and design, where is cheaper to correct the faults; and leading to benefits in terms of residual errors. The survey [4] reports that companies not performing MIL testing were finding almost 30% more errors in module test, where their correction is more expensive.

Thus, through the integration of the CIT within the proposed life cycle, we increase the ranges of techniques of V&V supported by the methodology, making it more flexible to the multiple industrial needs for the development and assessment of critical systems.

Chapter 5

Model-Driven Failure Mode and Effects Analysis

5.1 Background

Failure Mode and Effects Analysis (FMEA) is an engineering technique for evaluating the effects of potential *failure modes* of parts of a system[1]. In the critical systems domain, it is widely used to systematically identify the potential failures of components and analyze their effects on the system, which could adversely affect its overall reliability or safety.

The analysis considers the system decomposition down to basic components: each component is analyzed, identifying all its potential failure modes; for each mode, the propagation of the effects up to the system as a whole is studied. For quantitative analysis, e.g., for reliability assessment, FMEA includes estimates of quantitative parameters, such as expected failure rates. Failure modes and rates may derive from components' technical specifications, historical data, or further appropriate information, such as handbooks of reliability prediction models. Finally, an evaluation of severity and/or probability of failure modes provides a prioritized list for corrective actions and design improvements.

Conceptually, FMEA can be performed using different criteria for the system decomposition. It can adopt: (i) a *functional approach*, when a functional FMEA is performed on

[1] Parts may be for instance subsystems, assemblies, components, functions. Here, we will generically refer to them as *components*.

the functions and focuses on ways in which functional objectives of a system go unsatisfied or are erroneous; (ii) a *structural approach*, when a FMEA is performed on the system architecture and focuses on the failure modes of the components; (iii) a *hybrid approach*, when a mix of the previous approaches is used, and the analysis moves from the system's components to their functions or vice-versa.

The main artifact used in FMEA is a *worksheet*, providing guidance for conducting a structured analysis, for checking consistency, and for documentation. A worksheet should report the following minimal information for every failure mode of a component:

- Failure mode;
- Component-level effects resulting from failure;
- System-level effects resulting from failure;
- Failure mode causal factors;
- Recommendations.

Different worksheet types lead to a different amount and type of information to be derived from the analysis. The specific form to adopt depends on the customer, the system safety working group, the safety manager, the reliability group, or the reliability/safety analyst.

Nowadays, several ad hoc tools are available on the market. They provide guidance, help to reduce mistakes, and offer useful features, e.g. for consistency checking, traceability and documentation. However, their support for the analysis is limited, and particular tasks are not performed, e.g., analysis of faults propagation among components and the effectiveness of fault barriers on the system safety.

Indeed, FMEA is still a systematic reasoning, time-consuming technique, generally performed manually, by analyzing the system's components and writing the worksheet tables:

1. the support provided by tools is limited to specific tasks and more complex tasks, such as the analysis of the effects of multiple failures are often neglected by the analyst;

2. the needed knowledge to perform FMEA is typically spread in many design documents, in different formats, that depend on multiple teams. This problem is exacerbated when FMEA is outsourced to external companies;

3. the reuse of FMEA artifact is very low, since the information is not managed and engineered;

4. there is no standard methodology that prescribes how to perform FMEA that is shared among multiple teams and companies.

This lack of support, along with the increasing complexity of systems, leads either to expensive and error-prone (manual) analyses or to approximate results. Therefore, performing FMEA is still is a repetitive and time consuming task and there is the demand for new approaches and tools to improve and support the analysis.

Indeed, FMEA is particularly relevant for the development of critical systems, since it can be used for risk and hazard analysis, as well as it can used for evaluating the quality of the system architecture. For instance, the standard CENELEC EN 50128 highly recommend[2] the *Software Error Effect Analysis* (SEEA) for SIL-3 and SIL-4 systems, to assess the software architecture, as well as a technique of verification: according to EN 50128 terminology, SEEA is a form of FMEA analysis conducted on the software.

5.2 Overview of Model-Driven FMEA

In this chapter we present a model-driven approach for FMEA that can extend the software development life cycle for critical systems presented in Chap. 3 to integrate FMEA analysis

[2] According to the standard the requirements can be *mandatory*, *highly recommended*, *recommended*, indifferent, or *not recommended*.

of software systems among the techniques of validation and verification supported by the methodology.

The proposed model-driven FMEA approach adopts SysML, and uses the model for enabling formal knowledge representation, thus automated reasoning on propagation of (single or multiple) failures on their interferences and their effects. The approach eases the FMEA tasks in that it supports reasoning in the same conceptual framework of a model-driven design methodology, favoring communication among the designer and the analyst, early exploitation of design artifacts for FMEA, and automating inductive reasoning steps about fault propagation under single as well as multiple failures.

The process is outlined in Fig. 5.1. It is divided in three phases, *System Modeling*, *FMEA Modeling* and *Model Analysis*:

System Modeling System Modeling targets to formalize in a model the knowledge of the system required as input for a FMEA analysis. To this end the Designer uses a SysML model to specify functional and non-functional system requirements, the system architecture, the system components, and their interfaces, assigned requirements and behavior at the required level of abstraction for the scope of the FMEA analysis;

FMEA Modeling. This phase aims at enriching design model with *FMEA-oriented information* exploitable for automatic FMEA analysis. Using a *FMEA Profile*, the FMEA Analysts conducts a *model-centric FMEA* and refines the SysML Design Model with failure modes, internal faults, failure propagation, and other information. The model is then transformed into a *Prolog Knowledge Base* (KB) of the System, that can related to KBs of common domain libraries or past projects. The KB enables to automatic FMEA analyses and to the reuse of shared knowledge;

Model Analysis. During Model Analysis the FMEA Analyst uses the Prolog Knowledge Base of the system to perform the actual FMEA analysis, by making queries on the

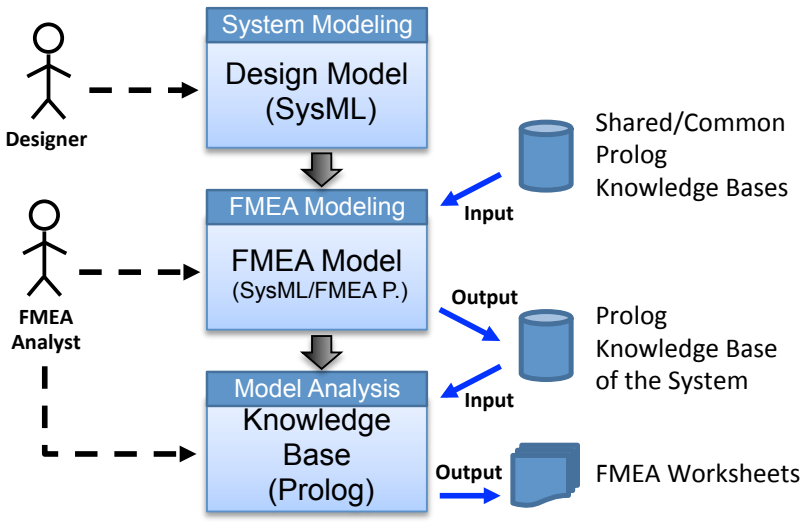


Figure 5.1: Overview of the proposed model-driven FMEA process. Boxes show the phases, the models produced, and the formalisms used. Dashed lines indicate the roles of the personnel, black filled arrows represent dependency between artifacts, blue arrows the artifacts in input or output.

model. The results are then used to derive FMEA output and worksheets by model-to-text transformations.

Our approach aims at being highly integrable and tightly coupled with model-driven product and software development life cycles: it exploits SysML to support FMEA analyses of any kind of system, including hardware, software systems or systems of systems; moreover, it is conceived to benefit from the availability of system models derived from design activities.

Considering the integration with the software development life cycle proposed in Chap. 3, the design models built during the development subprocess of the 'V' satisfy the output criteria of the FMEA System Modeling phase, therefore they can be directly reused as input for FMEA Modeling phase (Fig. 5.2). Indeed, the PIMs and the PSM specify the system design as defined by the Designer since System Design phase (Sec. 3.3.3), and include the specification of the system requirements modeled by the Requirement Manager during

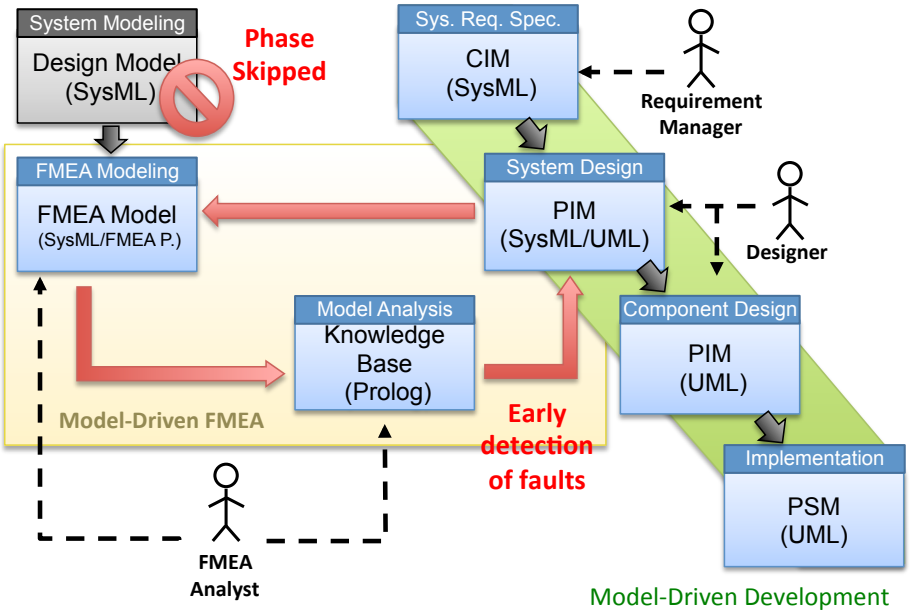


Figure 5.2: Integration of the FMEA approach in the life cycle proposed in Chap. 3. Filled arrows represent dependency between artifacts, dashed lines indicate the roles of the personnel. Depending on the scope of the analysis, FMEA Modeling starts from the proper design model (i.e., from PIM at integration or component level, or from the PSM).

the phase of System Requirement Specification (Sec. 3.3.1). Depending on the scope of FMEA Analysis, FMEA Modeling phase starts from the design model at the proper level of abstraction, i.e., from the PIM at integration level, component level or from the PSM.

The integration of the Model-Driven FMEA approach with the development life cycle creates a feedback for the Designer on the quality of the system architecture, enabling to an early detection of faults in the system architecture (Fig. 5.2).

The details of *System Modeling*, *FMEA Modeling* and *Model Analysis* phases are detailed in the following.

5.3 System Modeling

Phase	System Modeling.
Goal	To formalize in a model the knowledge of the system required as input for a FMEA analysis, according to its scope.
Main Role	Designer.
Input	Any source of information by which understand the system requirements, system architecture and safety requirements, systems components, their interfaces, structure and behavior. The inputs have to be selected according to the scope of the FMEA analysis.
Model	Design Model.
Viewpoint	Dependent on the scope of FMEA.
Diagrams	SysML Requirement Diagram, Use Case Diagram, Block Definition Diagram, and behavioral diagrams, such as State Machine Diagram, Sequence Diagram, Activity Diagram, and Timing Diagram.
Output Model	Design Model with information required as input for a FMEA analysis.
M2T Output	–

System modeling aims at collecting all information preliminary to a FMEA analysis in a model that formalizes the knowledge of the system required as input for a FMEA analysis. The viewpoint of this model is strictly dependent on the scope of the FMEA worksheets that are requested as output of the process, but at least identifies the system architecture and specifies the requirements.

The SysML Design Model is built by the Designer, which specifies the system requirements, the system architecture, the components, their interfaces, assigned requirements and their internal structure and behavior at the proper level of abstraction. System requirements are specified by means of Use Cases and Requirement Diagrams; the designer has to carefully model the relations among the requirements, such as dependencies and containments. The system architecture is represented mainly with Block Definition Diagrams and

Internal Block Diagrams, and the model has to specify how requirements are allocated to the components. Finally, operational aspects are specified with SysML behavioral diagrams, e.g., Activity diagrams.

We opt for SysML as the approach is meant to be integrated into standard-based model-driven development processes, where such models are already provided by design engineers; In such cases, System Modeling stage can also be skipped, and the design models can be used directly for the next phase (Fig. 5.2).

Indeed, the usage of the Design Model fosters communication between the Designer and the FMEA analysts with a common and standard language, and allows to model component failure behaviors in an incremental way, as low-level system design proceeds. Also, it integrates FMEA at process level, in a model-driven development life cycle.

When performing FMEA for software systems the approach is suited to use a mixed UML/SysML model.

5.4 FMEA Modeling

Phase	FMEA Modeling.
Goal	To enrich design model with *FMEA-oriented information* exploitable for automatic FMEA analysis.
Main Role	FMEA Engineer.
Input	Design Model with information required as input for a FMEA analysis. Shared Prolog Knowledge Bases.
Model	FMEA Model.
Viewpoint	Oriented by the scope of FMEA.
Diagrams	SysML Use Cases, SysML FMEA Diagrams.
Output Model	FMEA Model exploitable for automatic FMEA analysis.
M2T Output	FMEA Model – Prolog Knowledge Base of the system.

FMEA Modeling is executed by FMEA Analyst, that refines the Design Model with

the FMEA-oriented information required for automatic FMEA analysis of the model. This activity reduces to perform a sort of *model-centric FMEA* on the model, since the analyst has to analyze the model and identify the components' failure modes, internal fault, failure propagation logic and so forth.

For supporting the analyst in this activity, we propose an original approach that exploits our custom defined *SysML FMEA Diagrams*, that are provided by a custom SysML Profiles including FMEA-oriented modeling elements. FMEA Diagrams are presented in detail in the next section.

To enrich the model with FMEA-oriented information, the FMEA Analysis performs the following tasks *for each system component*, with a bottom-up approach:

1. identifies its functionalities with a *FMEA viewpoint*, i.e., at the desired level of abstraction for the scope of FMEA analysis. The functionalities are derivable by behavioral diagrams and components' requirements, as specified in the design model;

2. models these functionalities by means of a use cases (having as subject the component), and identifies the requirements that these functionalities cover, associating the requirements with the component's use cases;

3. specifies the FMEA-oriented information (failure modes, internal faults, etc.) for the component on the basis its functionalities, developing one (or more) FMEA Diagram.

Steps (1) and (2) can reuse information passed by the Designer in the model. Indeed, these tasks could be already be performed during the development life cycle of critical systems, to enforce the traceability of requirements.

For the step (3), the FMEA Analyst can proceed in two iterations. In the first one, (s)he creates one FMEA Diagram for each component and identifies the component failure modes, internal fault and failure effects. The second round starts after all components failure modes have been specified, and the engineer re-examines all diagrams and completes

the model with the remaining FMEA-oriented details (e.g., the activation conditions of the failures). This two-round method eases the bottom-up enrichment of the model through FMEA Diagrams.

The scope of the FMEA analysis influences the level of abstraction of the identified functionalities. Indeed, the components' failure modes turn out to be the deviations from the expected behavior of the component functionalities. For instance, considering a software component, FMEA can be performed at high level on the component's functionalities, or at low level observing the lines of code that implement the component.

Once the FMEA Analysts has augmented models with the FMEA-oriented information, the model can be model-to-text transformed into a Prolog Knowledge Base.

5.4.1 FMEA Profile: the SysML FMEA Diagram

Differently from past proposals, our approach for enriching the model with *FMEA-oriented information* supports the reasoning of the FMEA Analyst on design model, to analyze – by means of the model itself as primary source of knowledge – the failure modes, their propagation and the effects. In other words, the actual activity of conducting a FMEA analysis in a traditional document-centric FMEA process translates into the model-centric activity of refining the design model with FMEA-oriented information.

To support the FMEA-oriented modeling, we propose a structured approach that is driven by new custom defined SysML Diagrams, the *FMEA Diagrams*:

> The FMEA Diagram is a SysML structural diagram, that offers a synoptical view of the functionalities, internal faults, failure modes, requirements and structural dependencies of one component – the *Component Under Analysis* (CUA) – that is the object of the analysis.

A FMEA diagram is split in five logical sections (Fig. 5.3). The diagram places the Component Under Analysis and its failure modes in the middle, using the graphical notation

of Use Cases. Then, the diagram groups the events that are involved in the activation conditions of the failure modes (i.e., the *causes*), and the requirements and functionalities that are affected by the failure modes (i.e., the *effects*): the causes are shown in the left and lower sides of the diagram, whereas the effects appear in the two parts on the right.

The failure modes involving the CUA may be activated by interactions among its internal faults and external failures that propagated from components connected to the CUA. Therefore, the left side the Diagram shows the failure modes of the components connected to the CUA, these elements are inferred from structural relations of the design model; whereas, the lower side represent the CUA's internal faults, that are modeled by the analyst as *behaviors*.

The activation conditions of a failure mode are modeled by the FMEA Analyst associating the external failures and internal faults – which appear, respectively, on the left and lower side of the FMEA Diagram – with a CUA's failure mode, and specifying the logical condition that activate the failure. Additional information can be added to the diagram, such as the probability and severity of the events.

The right side of the Diagram shows the effects of a failure, namely, which use cases are

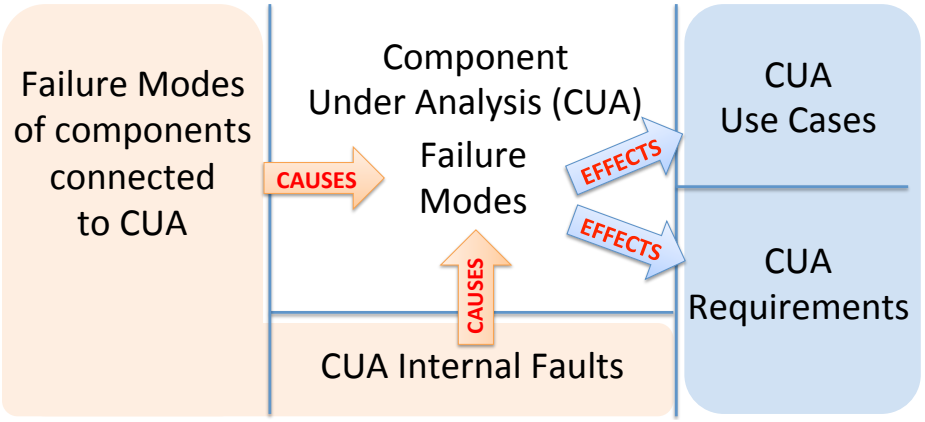


Figure 5.3: Logical layout of a *FMEA diagram*.

affected by the failure mode and which requirements, among those allocated to the CUA, are violated by the failure. Therefore, the top right part of the diagram repeat the CUA's functionalities as Use Cases (as modeled by the FMEA Analysts in the previous step), while the lower right part recalls the requirements assigned to the CUA. By linking the failure modes with the CUA's Use Cases and CUA's requirements, the FMEA Analyst defines the effects of the failures.

Summarizing the tasks to build a FMEA diagram, the FMEA Analyst:

1. places the CUA at the centre of the diagram and specifies its *failure modes* on the basis of the Use Cases on the right side, that model the CUA functionalities defined in the previous step. To reduce the complexity of a FMEA diagram in case of too many failures modes, the analyst can group failure modes into subsets and use multiple diagrams;

2. defines the *internal faults* of the CUA, as one or more behaviors, adding them in the lower part of the diagram;

3. models the *activation conditions* of the failure modes, linking, by associations, external failures and internal faults with the failure modes, and specifying logical conditions, probability and severity;

4. models the *effects* of a failure, linking, by relations, the failure modes with use cases and requirements that have been assigned to the CUA, as listed in the right side of the diagram.

FMEA Diagrams and FMEA-oriented modeling elements are defined in a custom SysML Profile (FMEA Profile). Besides the definition of the semantic and properties of the elements used for FMEA Modeling, the extensions enable to link the components with information already available in form of facts and rules in shared Prolog Knowledge Base (e.g., domain libraries or past projects knowledge bases), to promote the reuse.

5.4.2 MT2 transformation in Prolog

FMEA Modeling ends with the model-to-text transformation of the FMEA-oriented SysML model into a Prolog Knowledge Base (KB). The translation consists in the conversion of the FMEA-oriented information specified by means of the FMEA Profile into facts and predicates in Prolog language.

The FMEA Profile empowers the automatic transformation rules, and enables to link the model elements with additional knowledge bases to support the reuse of domain concepts or of knowledge from past projects.

A Prolog Knowledge Base contains relations, defined by means principally of Horn clauses[3]. These clauses, at the basis of Prolog language, are suited to be employed for the analysis of single and multiple failures. Indeed, the FMEA-oriented information added in the model in the previous steps can be represented as sets of *Horn clauses*.

For instance, let us consider a simple alarm system: the system is connected to an external power source, and possesses a backup battery that can mask short electric outages. A failure in the system occurs when there is no electric energy, and the backup battery is low: therefore, one failure mode of the alarm system is *system stops to work when there is an electric outage and the battery is low*. Considering the three events:

$$p = \text{"Electric outage"},$$
$$q = \text{"Low backup battery"},$$
$$t = \text{"System stops to work"},$$

then, the failure mode can be formalized by the following Horn clause:

$$\neg p \vee \neg q \vee t \iff (p \wedge q) \Rightarrow t$$

Other alternatives have been considered for enhancing the reuse by means of system

[3]A Horn clause is a *clause* (i.e., an expression formed by the disjunction of a finite collection of literals) with *at most* one positive (i.e., non-negated) literal. A Horn clause with no negative literals is sometimes called as *fact*, otherwise as rules. A Prolog Knowledge Base consists of more set of rules, that are named Prolog predicates.

KB, including *ontologies*[4]. However, Prolog has an easier syntax and reveals suited for our purposes without burden the engineers with more complex formalisms. Prolog replicates the inductive-deductive mindset of the FMEA Analyst: rules and predicates follow an inductive process, whereas queries use deductive algorithms.

5.5 Model Analysis

Phase	Model Analysis.
Goal	Perform actual FMEA on the system and generate results.
Main Role	FMEA Analyst.
Input	Prolog Knowledge Base of the system.
Model	Knowledge Base.
Viewpoint	Rules and predicates on the failure logic of the system.
Diagrams	–
Output Model	–
M2T Output	KB – FMEA results and worksheets.

During Model Analysis, the FMEA Analyst uses Prolog to make queries on the system knowledge base produced in the previous phase. The queries translate into the execution of the actual FMEA analysis by the Prolog inference engine. Indeed, Prolog discovers information on the KB, especially knowledge that is hard to extract by a manual or a pure model-based analysis. For instance, the inference engine enables:

- to follow the propagation of failures inside the system;
- to identify root causes of a component's failure;
- to compute failures derived from multiple errors;
- to study the effectiveness of fault tolerance mechanisms.

[4]An ontology is an explicit specification of a conceptualization. That is, an ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents.

To mask the complexity of writing the queries in Prolog, the most common queries performed during a FMEA analysis can be offered as *push-button* analyses. When needed, more complex queries can be performed directly in Prolog.

By the execution of the queries, it is possible to derive results and write FMEA worksheets. Moreover, for queries like the analysis of propagation of failures, it is possible to represent the result graphically directly on the model.

5.6 An Eclipse-Based Support Tool

We designed the architecture of an open source environment that supports the approach (Fig. 5.4). To benefits from the Eclipse Foundation's infrastructure for developing model-driven engineering tools, our solution is a based on an Eclipse plug-in [69], the *FMEA plug-in*. *FMEA plug-in* enables the interworking between:

Papyrus an Eclipse plug-in [70] that provides an integrated environment for editing models in SysML and UML 2. It offers advanced support for UML profiles, and enables to create editors for Domain Specific Languages;

SWI-Prolog an efficient C implementation of Prolog [71], that offers a rich set of application interface libraries.

By this solution, the Designer can use Papyrus for modeling the System Model, and then the FMEA Analyst can refine it with the FMEA-oriented information, using a specific FMEA profile (Fig. 5.5).

FMEA plug-in keeps synchronized the model elements with their description in the knowledge base, and enables the Analyst to query the model. The results of the query can finally be used for the automated generation of FMEA documents (worksheets or documentation).

5.7 Discussion

In this chapter we have presented a model-driven approach for FMEA that can extend the support provided by the V-Model life cycle to this additional form of verification and safety assessment. The key concepts at the basis on our approach are:

Process Integration we aim at integrating FMEA into product development life cycles, using a model-centric/model-driven approach to reduce time and effort for the analysis, and to enable to early exploit FMEA artifacts into the process;

FMEA-oriented Modeling we adopt SysML and propose to extend the model with a FMEA profile to specify FMEA-oriented information, as OMG standards are increasingly being used for critical systems in industries and compatible within our V-Model methodology (Chap. 3). Moreover, the profile originally offers a new custom SysML FMEA Diagram to support the Analyst in modeling the FMEA-oriented information with a specialized synoptical view on the elements needed to conduct the analysis;

Prolog-based Analysis we use Prolog to analyze the knowledge base derived by the

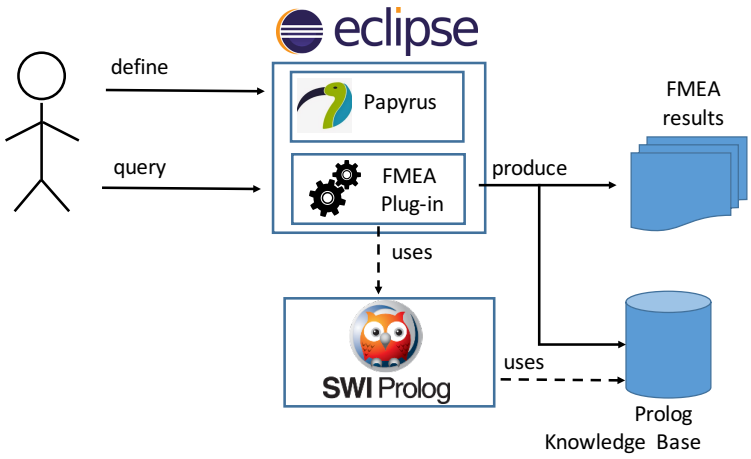


Figure 5.4: An Eclipse-based architecture for model-driven FMEA approach.

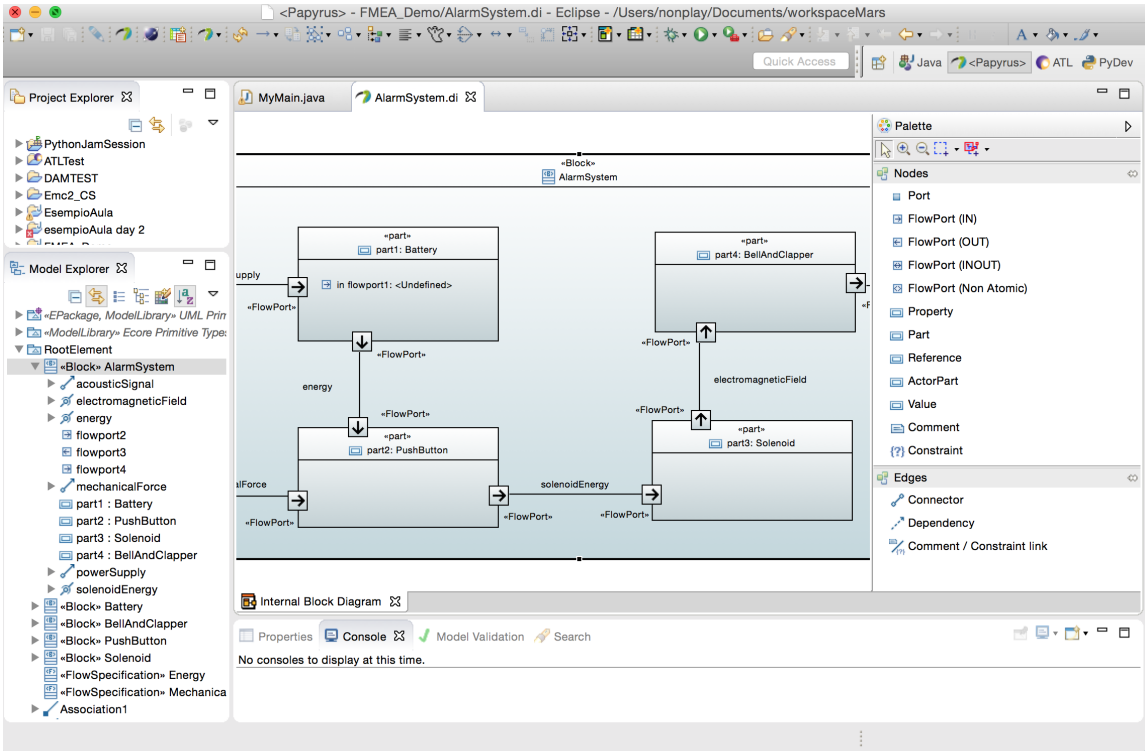


Figure 5.5: SysML modeling with Papyrus.

FMEA-oriented model, which can be algorithmically generated through a M2T transformation; queries expressible in the form of Horn clauses are performed on the KB, replicating the typical inductive-deductive mindset of the FMEA Analyst. The KB is also meant to act as shared repository of knowledge about components' failure modes, so as to favour reuse across multiple projects in an organization.

The integration of the FMEA analysis in a model-driven development life cycle based on SysML provides several benefits:

- it bridges the gap between the Designer and the FMEA analyst, getting their views closer to each other: indeed, the FMEA Analyst can *reason in the same conceptual framework of the design models* to analyze threats to non-functional requirements;

- a relevant *initial effort of FMEA is saved*. FMEA is typically performed by a RAMS[5] team, who receives the input documentation (requirements, design) and needs to extract the system components and their functionalities and interactions, normally through the creation of diagrams. A great advantage is that the RAMS team may receive all this ready information as an input.

- the support for automation provided by model-driven techniques and tools can significantly reduce the effort and cost of FMEA; the worksheet and other *artifacts may be derived through model transformations*.

- since the integration is based on standard languages and defines the model-centric activities and the information to add to the model, it reduces the fragmentation on how FMEA is performed by different teams;

- it favors better consideration of RAMS requirements in the first stages, by allowing early feedback from the FMEA team, and enabling to an early verification and validation of the design. Such process-level improvements reduce the risk of major redesign due to threats to RAMS requirements.

These advantages are clearly inter-related, and the relative importance depend on the context. For instance, when FMEA is outsourced to independent companies, early FMEA feedback may be of greater interest for the outsourcing organization than easing the task of the outsourced team.

The idea of deriving FMEA artifacts from SysML models is not new. Hecht et al. showed how to automatically generate the worksheet from SyML BDDs, IBDs and STDs (State Transition Diagrams) [72], through an intermediate transformation into an AltaRica state machine model. Before them, David et al. derived FMEA from IBDs, SDs (Sequence Diagrams) and the AltaRica language [73], while Xiang et al. from IBDs and the algebraic

[5] Acronym for Reliability, Availability, Maintainability and Safety.

specification language Maude [74]. Less emphasis has been put at the process level, namely on how FMEA can be integrated in a model-driven methodology.

Our approach differs from past proposals in that SysML models are transformed into a Prolog knowledge base. Moreover, *we concerned specifically with the problem of supporting the FMEA engineer in reasoning on design models* to analyze failure modes, propagation and effects, proposing a structured approach driven by FMEA Diagrams.

This idea of representing failure modes by the means of extensions of use cases in FMEA Diagrams has some similarities with *misuse cases* [75], which have been proposed as a means to analyze security threats and applied for safety analysis [76, 77]. The difference is that we are modeling directly the failure modes of a component in relation to the use cases it is involved in, rather than the sequence of actions that the component can perform, whose failure modes can cause harm to some stakeholder. As for use cases, their exploitation for guiding FMEA has been initially proposed by Allenby et *al.* [78]. We leverage the possibility to derive some failure modes by translating the use cases for the CUA considering common deviations from the expected service (e.g. in content and/or timing).

The idea of the exploiting a knowledge base to promote reuse is present in some past studies: [79, 80] used external repository or meta-model as external KB; other studies adopted custom profiles [73], or ontologies [81] for automatic FMEA generation, or knowledge modeling [82]. Our external KB in Prolog aims at promoting the reuse and supporting FMEA by formal queries, without burdening engineers with formalisms, like ontologies, they may be less familiar with.

The use of Prolog for supporting FMEA is envisaged in very few works, but with no link to system design models. In [83], authors report an application of Prolog III for an FMEA case study. In [84] the author proposes, as future work, to exploit pattern analysis in Prolog to develop automated ways to apply model annotations for FMEA.

Chapter 6

Case study 1: Model-Driven Engineering of a Railway Interlocking System

6.1 The Prolan Block Case Study

In this chapter we present an experience report developed in the industrial-academic collaboration within the framework of the European project "CErtification of CRItical Systems" (CECRIS, [9]). In the context of CECRIS, the candidate participated to a transfer of knowledge of model-driven technologies in Prolan Co., an Hungarian company manufacturing certified products for safety critical process control and rail signalling systems.

By the collaboration, it emerged how small and medium size enterprises like Prolan are interested in model-driven technologies but that there are barriers to their introduction, for the deep changes that these require into the organization and in the current industrial practices. Indeed, the adoption of MDE needs that Prolan carefully rethinks and redesigns its current product development life cycle, that currently complies with the railway standards CENELEC EN 50126, EN 50128 and EN 50129: no *proven-in-use* model-driven life cycle for this domain is available and supported by long-term evidence.

Therefore, to gain additional experience on the application of model-driven approaches industry, we set up a pilot project in the company to evaluate the proposed V-Model software

development life cycle (Chap. 3) and the supporting tools with respect to a real industrial context.

As case study, we selected a subset of requirements for the *Prolan Block* (PB), a safety-critical system for railway interlocking system that must be CENELEC EN 50126, EN 50128 and EN 50129 SIL-4 certified.

The system is deployed alongside railway segments, which are named *blocks*. Each block is equipped with a PB, with sensors for detecting incoming and outgoing trains (these sensors are the *axle counters*), and with semaphores[1] that are part of the signalling system. The PB manages the block (Fig. 6.1), receiving data from sensors, and properly setting the semaphores according to its internal state.

The interlocking is realized by the overall distributed system that consists of interacting PBs, which must ensure that no collision will happen on the railway, directing the train

[1] in railway domain semaphores can use optical or mechanical systems to signal. In the following we consider optical semaphores, whose aspects are dependent on the configurations of its lamps.

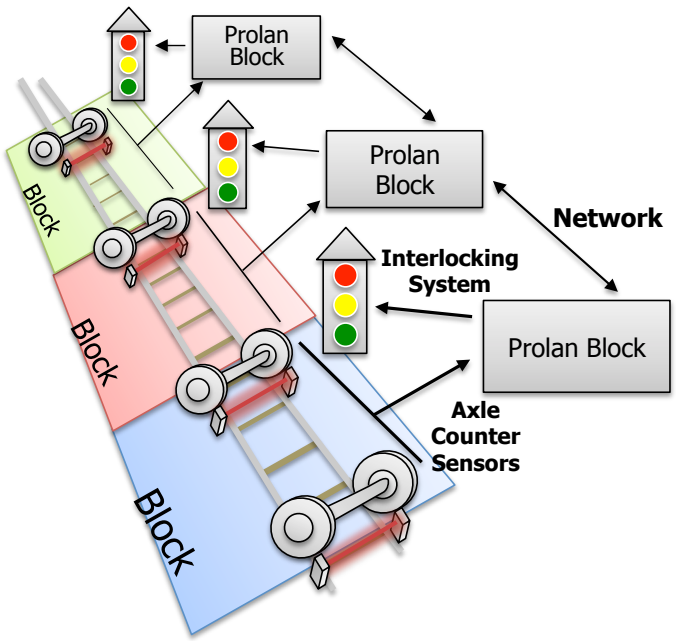


Figure 6.1: A high-level representation of the Prolan Block and its operating environment.

Figure 6.2: Short railway blocks on the Toronto Transit Commission subway system. Copyright Kevin Hadley licensed under CC BY-SA 3.0.

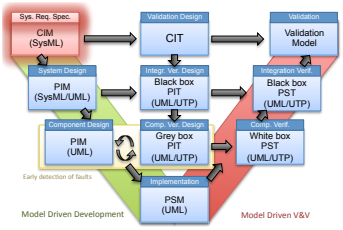
movements by proper sequences of signals. For instance, Figure 6.2 shows the proper sequence of semaphore aspects on short blocks of a segment of the subway of Toronto: according to the specific regulations, the yellow lamps can indicate that the next block's semaphore is red because there is an obstacle (e.g., a train) in the block after the next (e.g., there is a train two semaphores ahead).

6.2 Experimentation

In this section, we report our experience in using the proposed V-Model life cycle for the development and assessment of the Prolan Block.

6.2.1 System Requirements Specification

In *System Requirements Specification*, we develop a CIM that models the system's context (i.e., its environment) and the system requirements. In the case of Prolan most of requirements derives from railway standards and regulations.



As modeling tool Prolan adopted *No Magic MagicDraw* [85], mainly for licensing reasons, since they already had bought one license and had little experience with the tool.

Since we use SysML for the CIM, we modeled the system requirements using SysML Requirement diagrams, and the context, using SysML Block Definition Diagrams (Fig. 6.4). Requirement Diagrams resulted useful to describe functional and non-functional requirements and to define relationships among them, such as the ones of containment and refinement: requirements were groped in hierarchies, and non functional requirements have been isolated and emphasized.

We used the context BDD diagram to show the relations among the system and the actors, i.e., the environment, as well as the flows (of data, matter or energy) at the system interface. The BDD in figure 6.4 represents how the temperature and the external electromagnetic waves have to be considered during system design for safety concerns; moreover, the same diagram includes the actors, i.e., the RBC (Radio Block Centre, a radio signaling system defined by the European Train Control System), the track occupancy detectors (such as the axle counters), the interlocking systems alongside the railway, and a human-machine interface for locally controlling the PB.

Then, actors have been associated with the Prolan Block use cases (Fig. 6.5), to show in which system functionalities they are involved. For instance, the RBC interacts with the system to provide block information, while the track occupancy detectors take part in the reception of block occupancy information.

Functional requirements and use cases were detailed using behavioral diagrams. The diagram in figure 6.6 specifies the requirements on the aspects of the semaphore's lamps, by means of a state machine. Activity digram in figure 6.7 details the activities to be performed for processing the block status information. The use of behavioral diagrams for the requirements specification were perceived to be particularly useful by the requirement engineer for specifying the operational requirements.

In total, the CIM has been modeled using:

- 6 SysML Requirement diagrams;
- 12 SysML Block Definition diagrams;
- 41 Use Cases diagrams;
- 6 State Machines diagrams;
- 29 Activity diagrams;
- 33 Sequence diagrams.

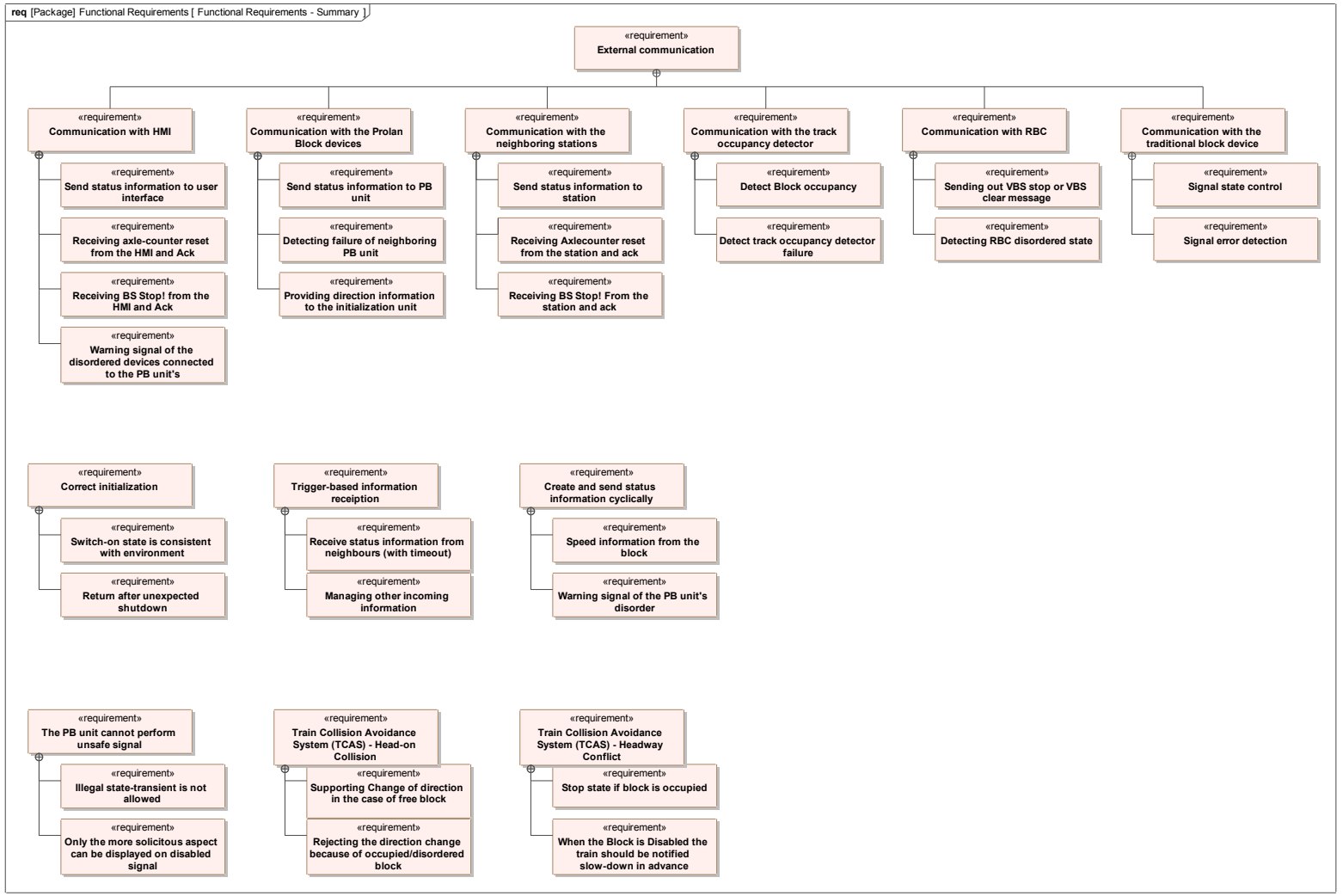


Figure 6.3: CIM SysML Requirement Diagram showing system functional requirements.

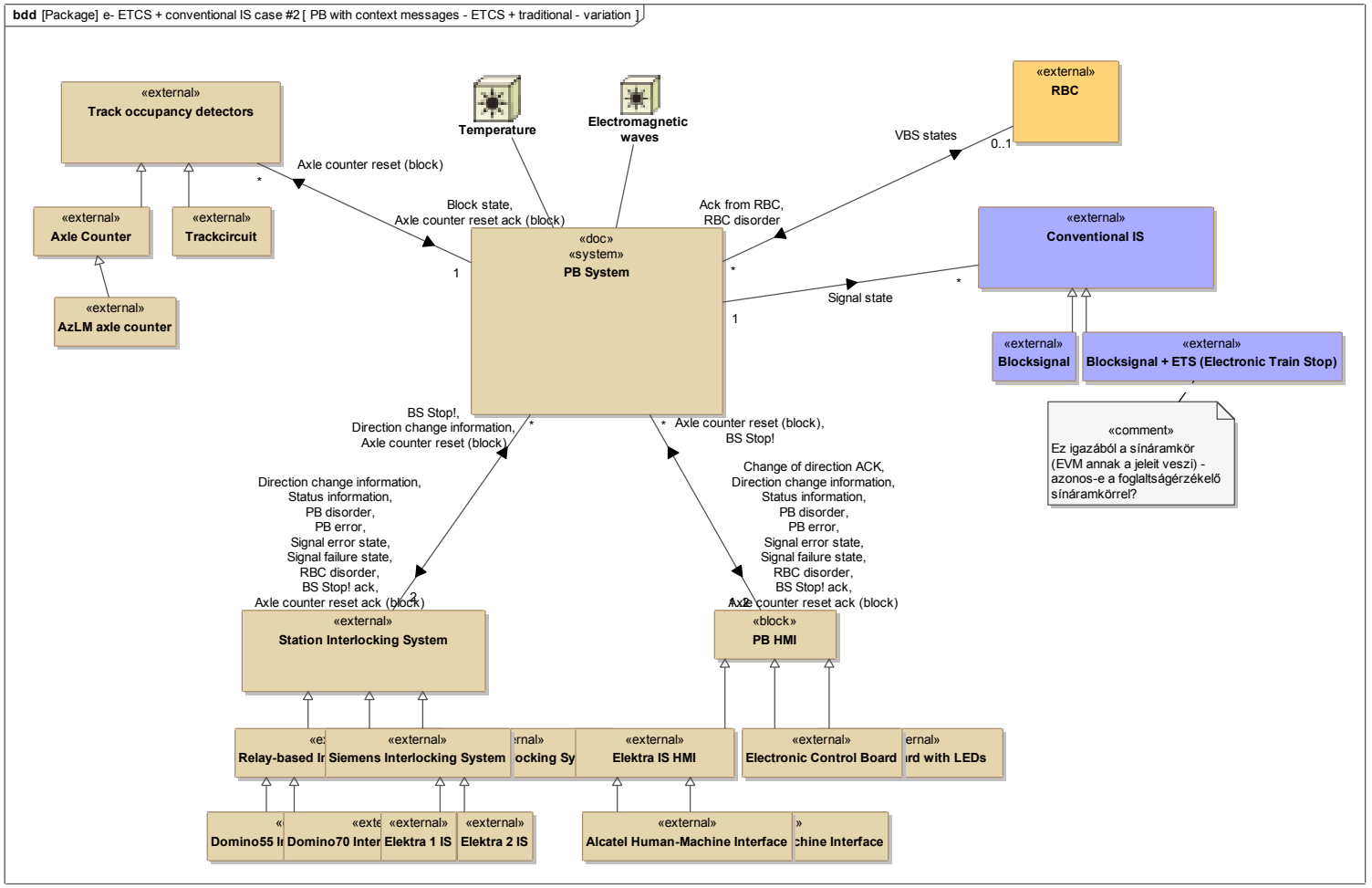


Figure 6.4: CIM SysML BDD showing the PB within its environment.

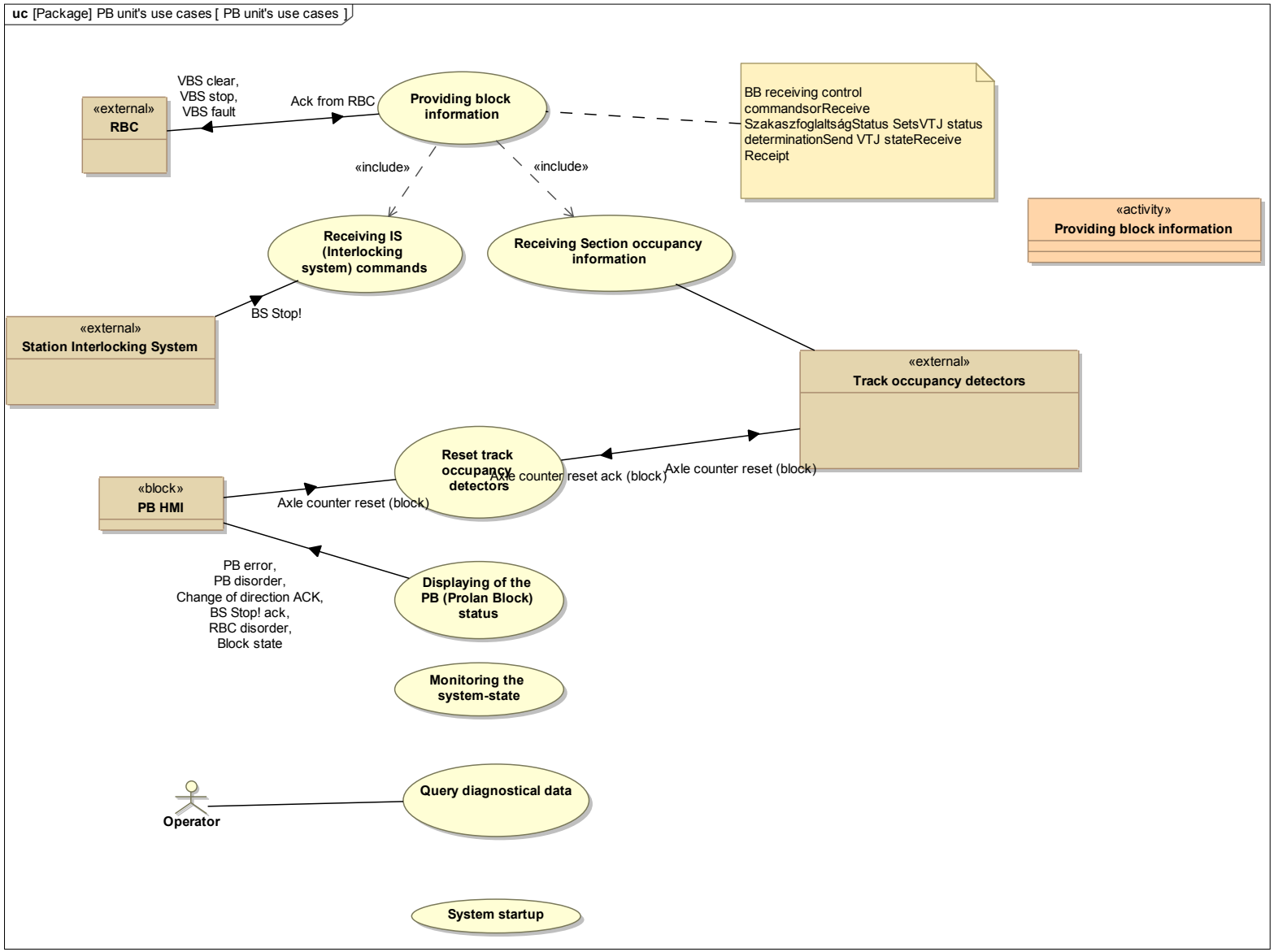


Figure 6.5: CIM Use Case Diagram for the Prolan Block.

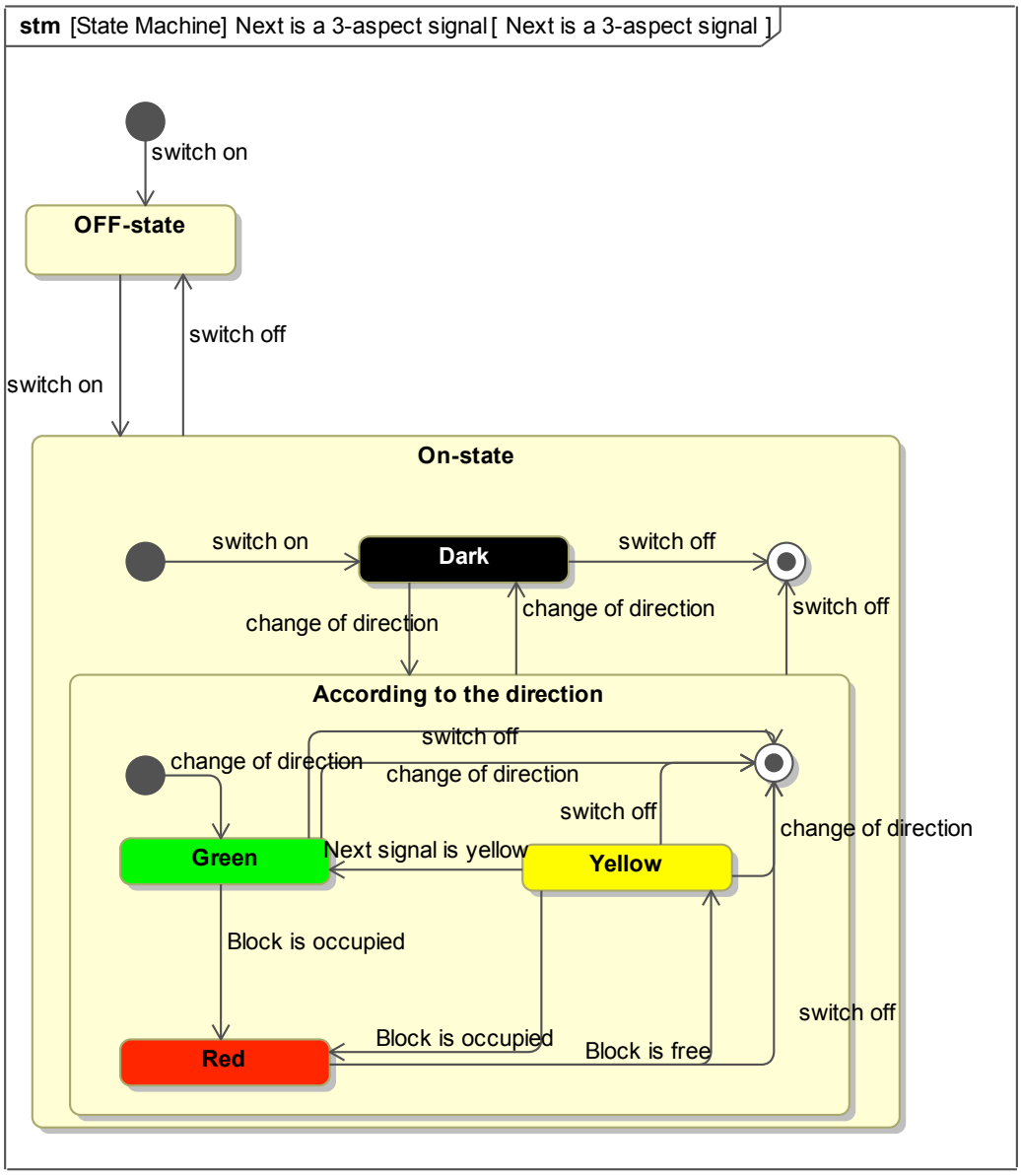


Figure 6.6: CIM SysML State Machine Diagram specifying the semaphore's behavior.

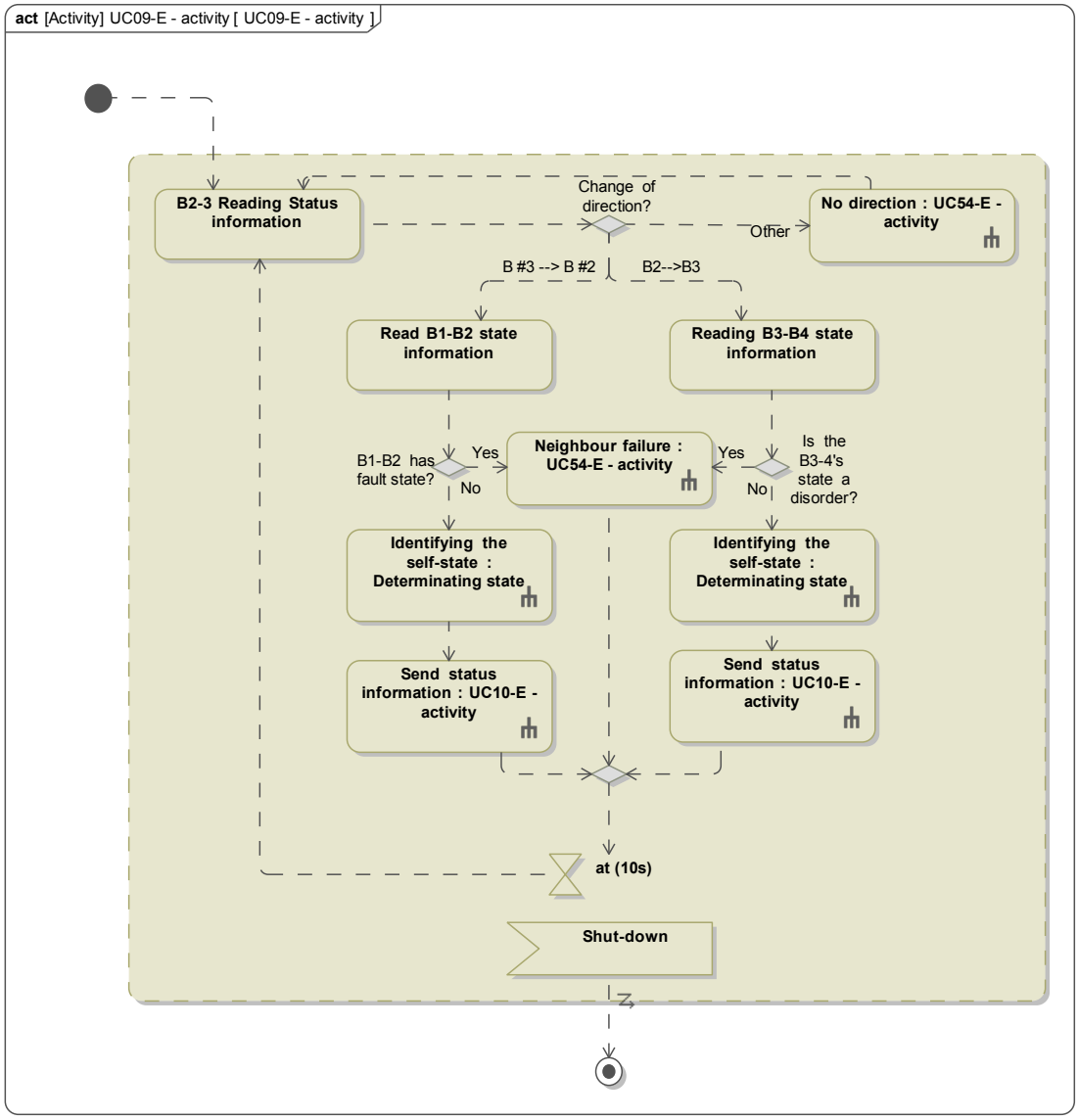
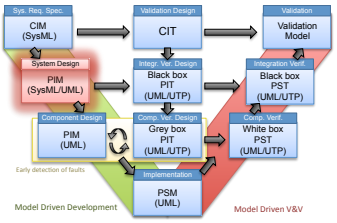


Figure 6.7: CIM SysML Activity Diagram specifying the use case *Processing Status Information*.

6.2.2 System Design



The *System Design* phase defines the high-level architecture of the system, refining the CIM into the PIM with viewpoint at integration level. The architecture was specified by means of UML structural diagrams, namely by component diagrams, and class diagrams, using again MagicDraw.

We identified five components, depicted into the object diagram in figure 6.8:

1. the TrackOccupancyDetector,
2. the NetworkCommunicator,
3. the ISController,
4. the HMIController, and
5. the ProlanBlockCoreLogic.

The first four components have been designed to mask the complexity of interacting with hardware, by offering high-level interfaces, instead of low-level and device-dependent interfaces. The most of the interlocking logic lies into the *ProlanBlockCoreLogic*.

The *TrackOccupancyDetector* interacts with the axle counters and converts low level I/O (like axle-in detected) in high-level domain specific events, such as “train entered in the block” or “train has left the block”. It also handles device failures, notifying with special events the occurrence of exceptional conditions.

ISController has the responsibilities of managing the interactions with the semaphore: it sets its aspect when requested by the *ProlanBlockCoreLogic*, and copes with device failures (such as, the burnout of lamps).

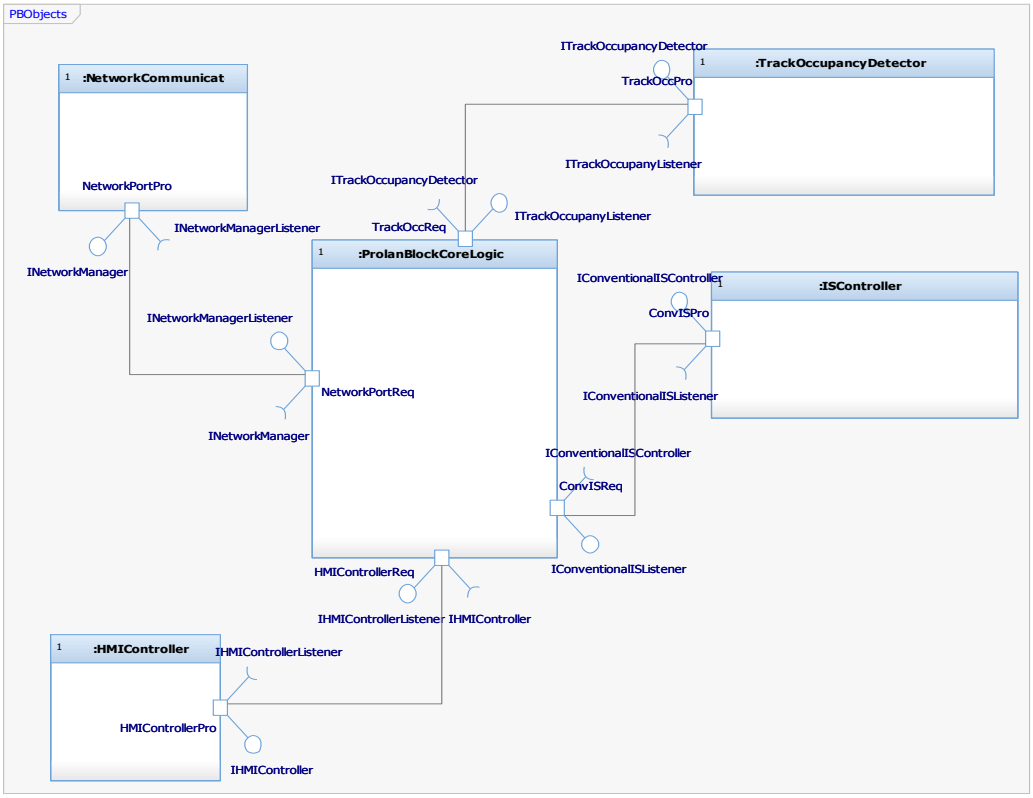


Figure 6.8: High-level system architecture.

NetworkCommunicator is in charge to send and receive messages on the network, from and to the adjacent PBs, while the *HMIController* manages the human-machine interface.

Finally, the ProlanBlockCoreLogic implements the logic for properly set the interlocking systems connected to the PB, according to its internal status and collaborating with the other components.

The PIM also assigns requirements to components (to support requirement traceability) and specifies the components' interfaces, and the behavior, i.e., the expected I/O at the boundaries.

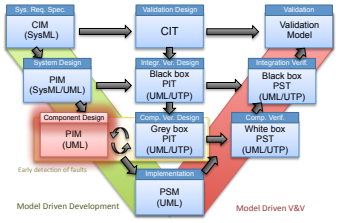
The components' interfaces were defined using standard and platform-independent UML syntax, to abstract away from platform-specific details at this stage. Therefore we followed

guidelines such as:

- the services of any middleware or library shall be defined in terms of abstract interfaces;
- the datatypes of the variables shall be independent of a specific programming language.

6.2.3 Component Design

In *Component Design* we enriched the PIM with the internal design of the components.



The PIM was defined using *IBM Rhapsody Developer* [86] (hereinafter: Rhapsody), and we continued to follow guidelines to keep the model as much as possible platform-independent. Indeed, the tool does not offer a clear separation between a platform-independent and a platform-specific model. For instance, for specifying the behavior of the state machines, we avoided to insert C++ code, and preferred to adopt the UML compliant syntax of *send signal action* and *receive signal action*. Furthermore, we adopted Rhapsody datatypes in lieu of target language datatypes when declaring variables, and their actual type is handled by the code generator.

At this stage we completed the behavioral and internal description of the component, using UML Behavioral State Machines. The PIM state machine diagram for the *TrackOccupancyDetector* is shown in Fig. 6.9: the component counts the axles detected in the block, incrementing the counter when one axle is detected entering the block, and decrementing the counter when one axle is detected exiting the block; if the counter is not zero, then a train must be crossing the block. The component also detects anomalous conditions, such as when the counter does not sum to zero, or when a train takes extraordinary time to leave the block.

The PIM turned out to be almost completely defined, and only few parts could not be

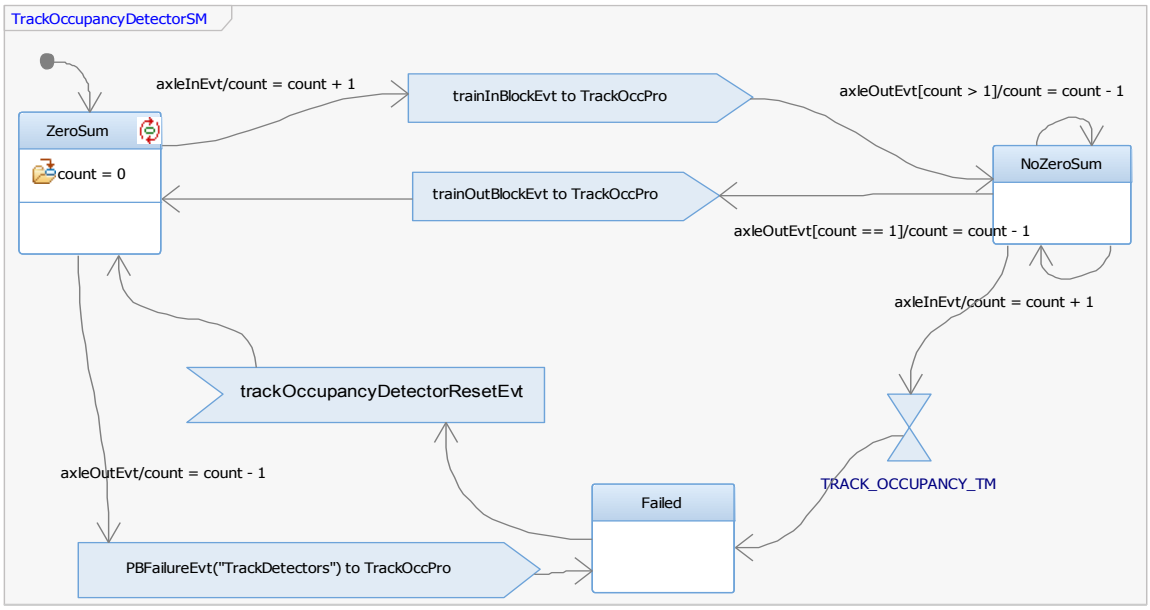


Figure 6.9: The PIM state diagram of the behavior of the *TrackOccupancyDetector*.

specified in a platform-independent style. If supported by the tool, we can define these detail with ALF (Action Language for Foundational UML), or otherwise in pseudo-code. However, since ALF is a recent feature of Rhapsody, in practice we needed to complete the behavior of the component with *prototypes* of the actual code in C++, in order to exploit the model *animation* in Rhapsody.

Indeed, Prolan was interested in assessing the model simulation/animation feature of the model-driven life cycle as a technique of early fault detection. Rhapsody does not actually simulate the PIM, but *animates* it, i.e., it generates an instrumented implementation of the model that enables to track the program execution. This is a feature that Prolan engineers found useful and valuable for having an immediate feedback on the program behaviour.

More in detail, by means of the *Rhapsody Panel Diagrams*, we created an user interface bound to the model, which allowed to send events to the model at runtime and observe its execution (Fig. 6.10). Of course, the execution can be also followed on behavioral diagrams,

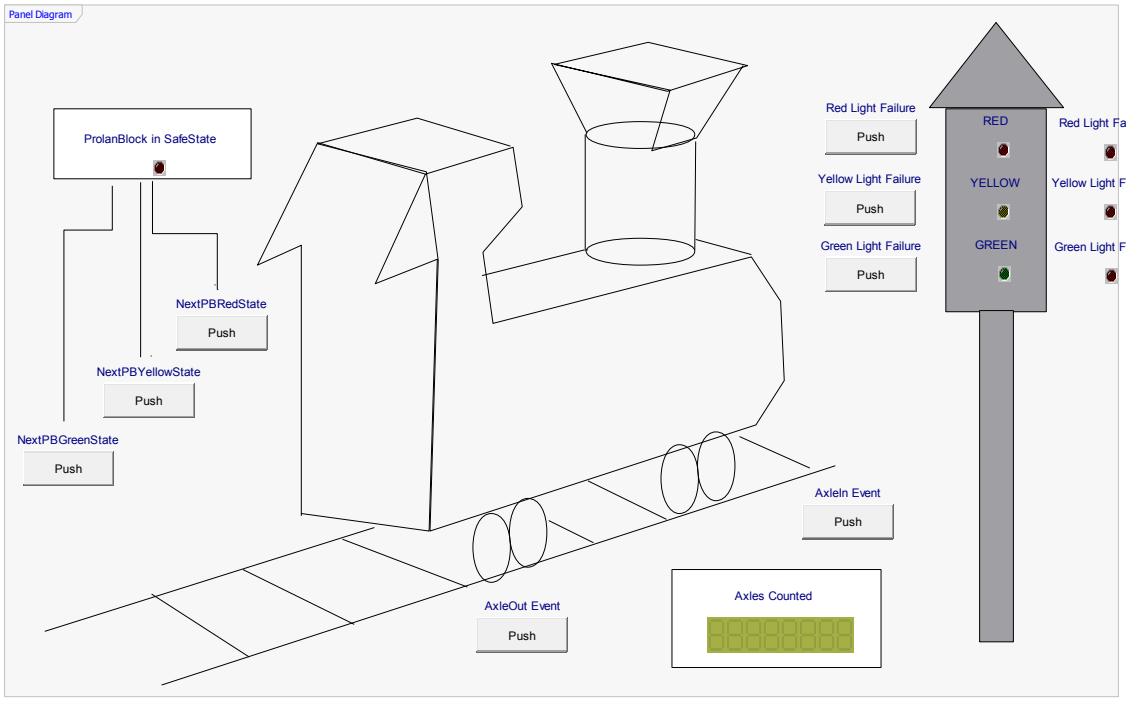


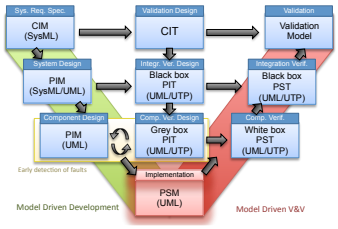
Figure 6.10: The IBM Rhapsody© Panel diagram for the Prolan Block.

e.g., state machines or sequence diagrams.

By model animation, engineers are allowed to run test cases on a prototypal/intermediate PSM, observing the effects on the model and enabling an early detection of design faults.

6.2.4 Implementation

In *Implementation*, we bind the PIM to the target platforms, adding low level details concerning the implementation. For instance, a PSM defines the mapping between the generic datatype with the actual ones offered by a specific programming language, and binds data and function calls to the interfaces of middleware or library chosen for the instantiation.



The PSM can be translated into the final target code to provide a partial or full implementation of the system.

Using IBM Rhapsody, Prolan had to set several tagged values and tool-dependent parameters to enrich the PIM with information platform-specific (Fig. 6.11). For instance, they include how to realize the association (e.g., the datatypes of the collections of elements) and the time resolution for the scheduler of the state machines' event queues. These parameters are used by Rhapsody for the automatic translation of PSM into code. Considering the deployment of the PB on *Prosigma*, the Prolan SIL-4 target platform, we had to specify that the generated code cannot use dynamic memory and that the variables have to be initialized at runtime, due to the lack of memory isolation on *Prosigma*.

Then, the Implementer adds platform specific code in the PSM. However, since it is not efficient to write source code directly in the model, the Implementer can exploit *code round-trip* features of most of modern MDE tools, to update the source code keeping the model synchronized:

1. by automatic code generation, packages, code skeletons, makefiles and other artifacts are automatically model-to-text produced;

2. the Implementer fills the code skeletons with platform-specific details using the support of modern development environments (such as Eclipse);

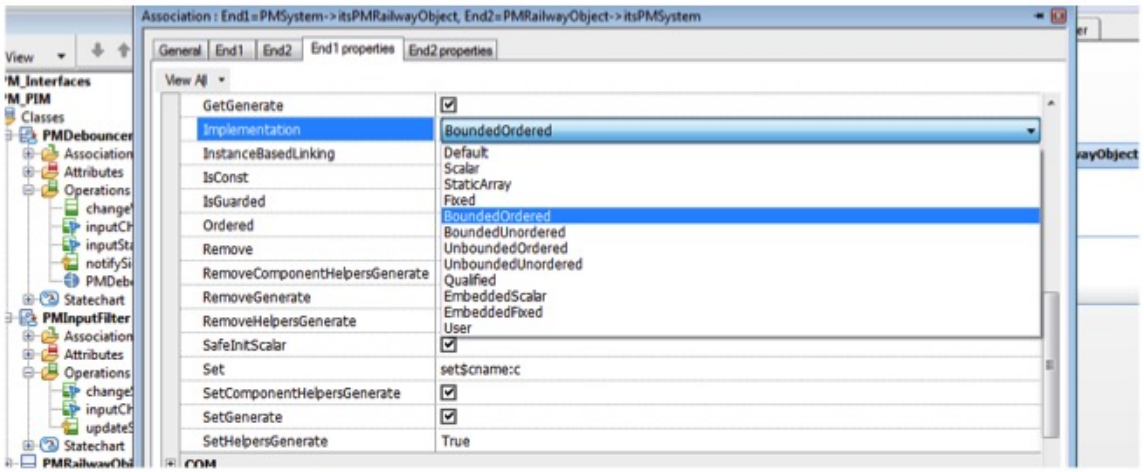


Figure 6.11: Specification of platform specific properties in IBM Rhapsody©.

3. by code round-trip, the model is automatically augmented with the information written manually by the Implementer in the source code.

The translator can be chosen according to source code requirements, and it incldues the supporting execution framework into the source code. For instance, Rhapsody offers two C/C++ frameworks: *IBM Rhapsody Object Execution Framework* (OXF), and *IBM Rhapsody Simple Execution Framework* (SXF) (Tab. 6.1). The latter is dedicated to embedded systems and safety-related development: qualification kits support the certification of the automatic generated code for several standard (including ISO 26262, EN 50128 and recently DO-178B).

The translation of the PSM in C++ source code generates around 7.5 thousands of lines of code for a platform using a conventional OS, 7.3 thousands for target platform using a commercial Real Time Operating System (VxWorks), and 5.9 thousands C lines of code for an embedded systems not using an OS. For the last code generation we used the SXF framework. The code is readable, understandable, and almost ready for the deployment: figure 6.12 shows an excerpt of the automatically generated code for the state machine of

IBM Rhapsody Simple Execution Framework (SXF)	**IBM Rhapsody Object Execution Framework (OXF)**
Static architecture	Dynamic allocation
MISRA C++ 2008 compliant with modeling checks	Not validated for MISRA
No animation/tracing	Animation/Tracing
Only Real Time mode	Real Time/Simulated Time modes
No containers (can be added)	Containers
Static memory manager (only BaseNumberOfInstances)	Static memory manager
Flat statecharts	Flat or reusable statecharts
No Multi-core, No Interfaces, No ports.	Multi-core, Interface-based, With Ports.

Table 6.1: Differences between the frameworks SXF and OXF.

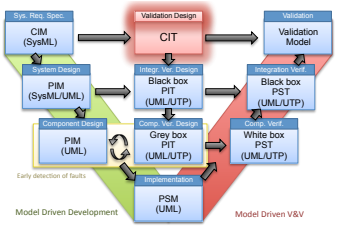
the *TrackOccupancyDetector*, in particular the reception of the event *axleOutEvent* in the state *NoZeroSum* (Fig. 6.9).

```
IOxfReactive::TakeEventStatus
        TrackOccupancyDetectorController::NoZeroSum_handleEvent() {
    IOxfReactive::TakeEventStatus res = eventNotConsumed;
    if(IS_EVENT_TYPE_OF(OMTimeoutEventId)) {
        if(getCurrentEvent() == rootState_timeout) {
         // ...
        }
    } else if(IS_EVENT_TYPE_OF(axleOutEvt_ProlanBlockPkg_id)) {
        //## transition 3
        if(count > 1) {
            NOTIFY_TRANSITION_STARTED("3");
            cancel(rootState_timeout);
            NOTIFY_STATE_EXITED("ROOT.NoZeroSum");
            //#[ transition 3
            count = count - 1;
            //#]
            NOTIFY_STATE_ENTERED("ROOT.NoZeroSum");
            rootState_subState = NoZeroSum;
            rootState_active = NoZeroSum;
            rootState_timeout =
                    scheduleTimeout(TRACK_OCCUPANCY_TM, "ROOT.NoZeroSum");
            NOTIFY_TRANSITION_TERMINATED("3");
           res = eventConsumed;
        } else {
            //## transition 4
            if(count == 1) {
                // ...
            }
        }
    } else if(IS_EVENT_TYPE_OF(axleInEvt_ProlanBlockPkg_id)) {
                //...
    }
    return res;
}
```

Figure 6.12: Fragment of the code automatically generated for the state *NoZeroSum* of the state machine shown in Fig. 6.9 (OXF Framework). In particular, the code focuses on the reception of the event *axleOutEvt*.

6.2.5 Validation Design

Validation Design exploits the CIM to define the CIT. Considering Fig. 6.4, the CIT can focus on modeling the behavior of *Station Interlocking Systems*, of the track occupancy detectors, of the RBC and conventional IS to generate concrete operational profiles.

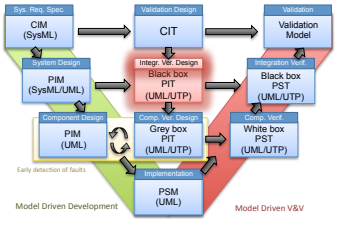


Linking the CIT in an in-the-loop configuration with the system, it enables to validate the system against its expected usage by external actors, and to create a simulated environment in which engineers can assess the system's behavior. Moreover, the operational profile derivable from the CIT can be used to performance (and stress) testing.

Prolan decided to does not build a CIT model for the Prolan Block, and the benefits of CIT modeling have been assessed on another system, the *Prolan Monitor*, that is discussed in the next chapter (Chap. 7).

6.2.6 Integration Verification Design

In *Integration Verification Design* phase we develop the BB-PIT to model the expected behavior of the system's components, independently from their inner design.



The BB-PIT provides structural and behavioral views of the system's components: structural views supports the generation of the testing infrastructure, whereas dynamic description supports functional V&V techniques.

Test cases can be added to the model using UML-UTP Sequence, Activity or State Machine diagrams. The graphical representation of the model is less error-prone than the textual specification of the test cases, and, by model-to-text transformations, we can

translate the test cases to multiple target testing platforms (such as TTCN-3 and JUnit), enhancing readability, reusability, and maintainability.

To support functional testing, we enriched the BB-PIT with the behavioral description of each component, modeling them by state machines. The state machines have been then exploited to automatically generate test cases, adopting as test adequacy criteria, as the coverage of structural elements of the model (e.g., full coverage of the states and of the transitions).

In this phase we adopted *Conformiq Designer*™ [87] (from here onward: Conformiq) that enabled us to generate automatically test cases for the BB-PIT. However, since the tool is not fully UML compliant, we had to specify the components' behavior in *QML*, the specific language used by Conformiq: it is based on a subset of UML State Machines with a Java-like action language (Fig. 6.13).

Using Conformiq, we also generated test cases with an adequacy criterion based on requirement coverage: by means of QML annotations on model transitions or events, we could relate the coverage of the model elements with the requirements, thus Conformiq could compute the coverage of the test cases in front of the system requirements.

During the pilot project, we generated 21 test cases for the *ProlanBlockCoreLogic*, achieving full coverage of requirements, as well as of all states and transitions. As output, Conformiq provided us with the test cases sequence diagrams (Fig. 6.14), that we could export in several target languages[2], such as JUnit and TTCN-3; and the traceability matrix (Fig. 6.15) that correlates the test cases with the features they cover (states, transitions or requirements).

Regarding the exploitation of the BB-PIT structural description, Conformiq automatically generates the test harness, and the Implementer only has to code the SUT adapter, to let the testing framework interact with the system. However Rhapsody also offers a plug-in,

[2]Even though Conformiq can be extended with plug-ins for test scripts generation, at time we were able to import the test cases in Rhapsody only manually.

namely the *Rhapsody TestConductor Add On* [88], for generating automatically the testing harness (including the drivers and stubs, Fig. 6.16), starting by the model design diagrams. Within TestConductor the test cases are directly executed in Rhapsody, and it is possible to observe their effect, following the behavior of an instrumented SUT by means of sequence diagrams. ATG produced around 3.5 thousands of lines of code to provide the system with the testing harness.

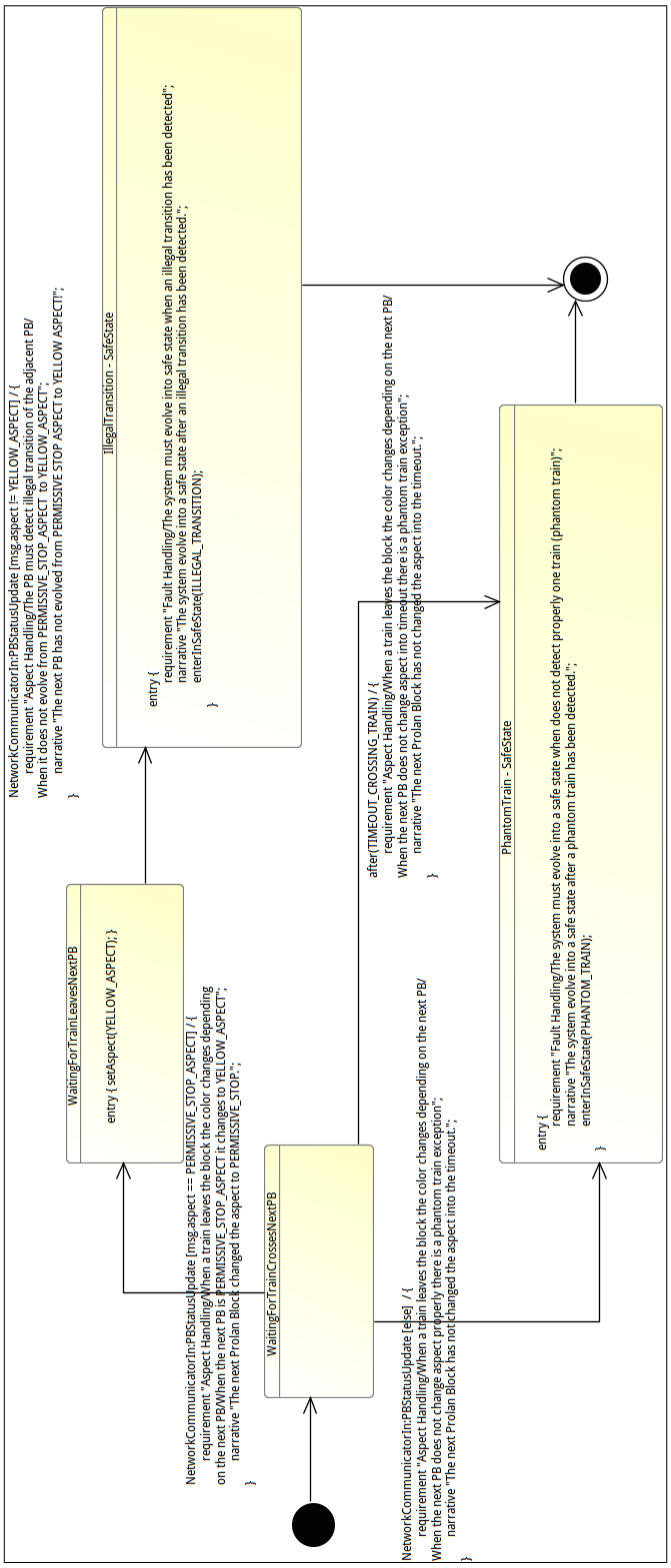


Figure 6.13: A BB-PIT QML State Machine used for test case generation in Conformiq™.

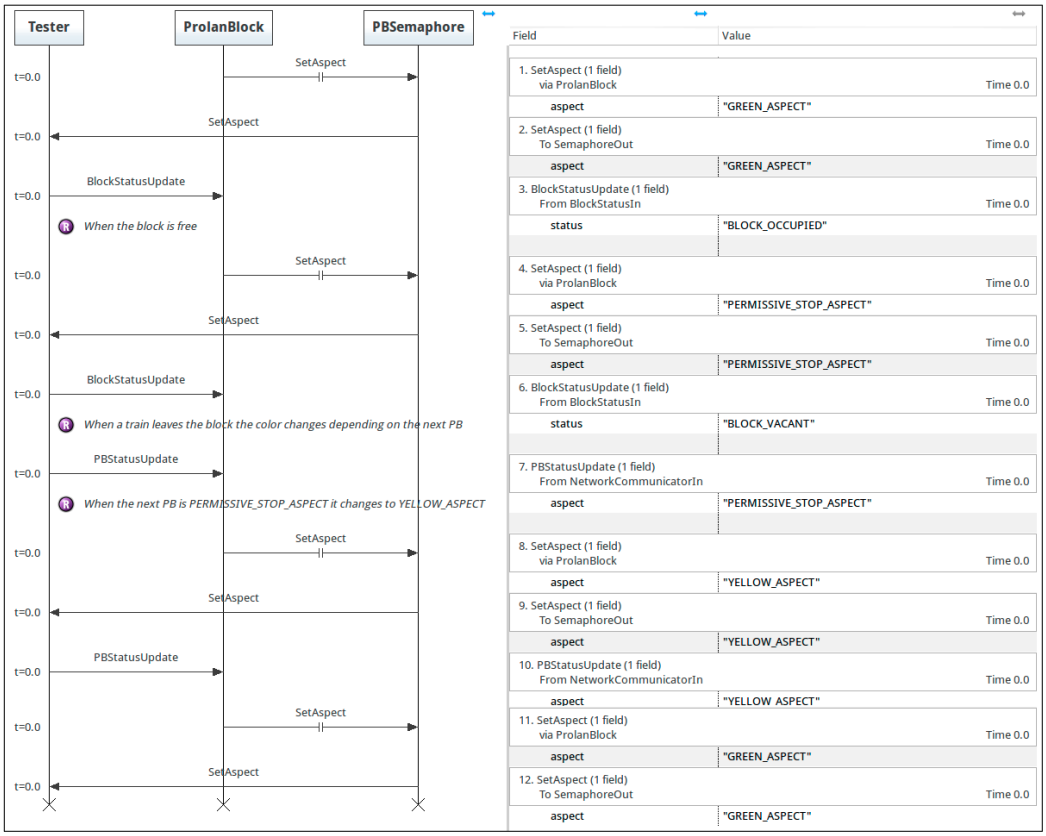


Figure 6.14: A test case automatically generated from the BB-PIT by Conformiq™.

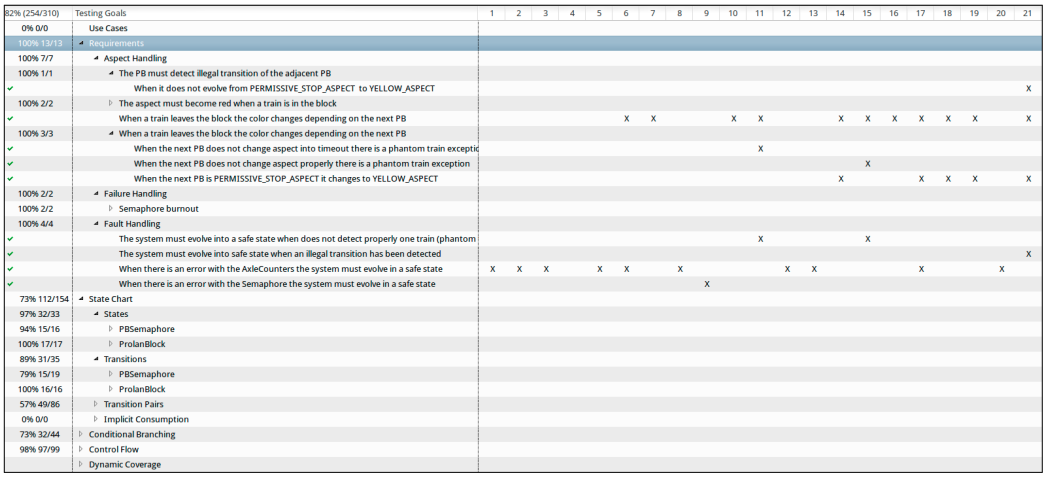


Figure 6.15: The Traceability Matrix automatically generated by Conformiq™.

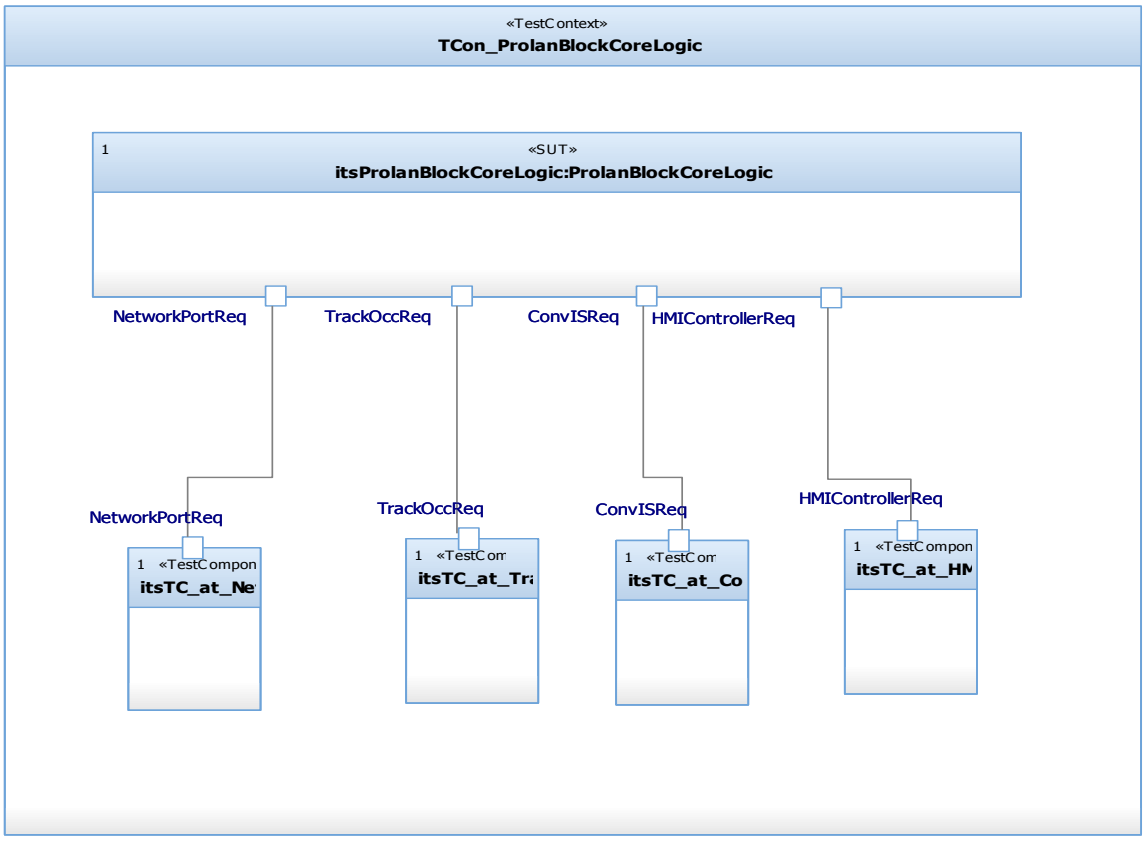
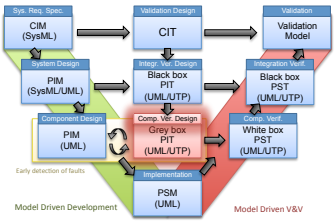


Figure 6.16: The testing harness automatically generated by TestConductor for the *ProlanBlockCoreLogic*.

6.2.7 Component Verification Design

Component Verification Design refines the BB-PIT defining a GB-PIT, that benefits from the PIM at component level viewpoint, which offers a gray box view of the system. The GB-PIT enables additional verification techniques that can exploit structural features for assessing correctness.

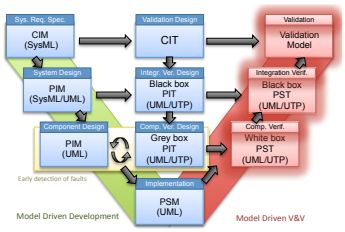


In the Pilot Project, we exploited the GB-PIT to generate structural test cases to cover the design model. To this aim, we used a Rhapsody plug-in, the *Automatic Test Generator* (ATG) [89], to automatically generate additional test cases.

ATG generated ten test cases for the ProlanBlockCoreLogic, modeled as *Rhapsody UTP Sequence Diagrams*[3]. By these test cases we achieved a 91% coverage of PIM structural model features, with 19/21 states and 22/24 transitions covered. However, we were not able to configure the ATG to reach the full model coverage: the few elements not covered are related to a branch for timeout exception handling, and the generator did not find a proper input to trigger the timeout event. Thus, we covered the remaining model elements with one manually-written test case.

6.2.8 Model-Driven V&V Subprocess

The V&V activities of the right side of V-Model refine the PITs considering new details deriving from the target platform, from the code implementation, and from the PSM, and actual perform the V&V activities on the system implementation.



In the pilot project, we limited to investigate on the automatic support for generating test cases using adequacy criteria on the source code, instead of on the model.

We found that the ATG enabled also to generate test cases based on the source code coverage, using adequacy criteria based on the statement and the MC/DC coverage. However, although this feature offers a form of automation for the white box testing, it turned out to be not satisfying in our case, since we were able to cover only 2/9 MC/DC test obligation by automatic test generation. This calls for better solutions for supporting white box test by means of model-driven approaches.

Finally, by M2T transformation, it is possible to derive from Rhapsody detailed test case reports.

[3]Rhapsody UTP is an IBM Profile based on UTP

6.3 Discussion

In this pilot project, we assessed the feasibility, the advantages, and the drawbacks of the proposed model-driven software development life cycle for the development of a real industrial interlocking system that must be CENELEC EN 50126, EN 50128 and EN 50129 SIL-4 certified

The proposed process led to an improvement in the development and testing practices. Requirements engineers found the usage of model-driven approach important to produce better specifications, and to detect more incongruences and missing specifications, than using the previous document-centric process. Same feeling had the system and test designers, who, exploiting models, built quickly prototypes and test models, and benefits from the tool utilities, such as the model animation, to cross-check their design. A gain of productivity and quality was also perceived in testing, because of the automatic generation of testing plans and test cases, and because of a more structured testing design process, that better exploits the interplay between developer and tester views.

Regarding the implementation, the benefits were not immediately evident, since the certification for railway standards limits the advantages of code generation. Therefore, Prolan is still undecided if the efforts for certifying automatically generated code is worth the costs for code development. On the other hand, we recognized MDE fruitful for the new capabilities that models introduce in the overall development process, and for the better quality of produced artifacts. The automatic generation of the code shall not be considered as a crucial factor to adopt model-driven approaches. Similar results were observed in [24], where code generation was not the key factor that justified the introduction of MDE.

Indeed, model-driven approaches enable engineers to work on a more abstract level than the document-centric approaches, focusing on the problem and leaving other artifacts to be derived through transformations. Models enable to introduce new techniques in the

development, such as simulation (model animation) and early fault detection, as well as in V&V activities, e.g., automatic test case generation and model checking. Furthermore, requirements between the model's elements and the artifacts were traced accurately, easing the generation of traceability reports useful for the certification.

However, despite these advantages, the industrial adoption of model-driven approaches present a number of still open issues, that we experienced in our pilot project. We proposed a general framework for a model-driven 'V' process based on MDA and MDT. Our experience shows that each activity must be adapted to the industrial context, and to its domain-specific needs; there is no instantiation that can fit all domains and applications. This is also due to the limited support provided by tools, which can adopt DSLs and do not provide full compatibility and transformations with other languages. For instance, in the activity of *Integration Verification Design* we needed to use QML to automatize the generation of test cases using *Conformiq*. Analogously, Markov chain models could be suited for modeling the CIT, without burden the modelers with more complex formalisms. Another example is provided by temporal logic formula, which could complement the CIM specification to introduce particular model checking techniques. Anyway, UML 2.0 turned out to be suited for our purposes, despite we needed to exploit advanced features of the language (such as the *Connection Points* in the *State Machines*) that are often unknown to less experienced modelers or to engineers using previous versions of UML.

Three commercial technologies have been adopted in this study as support tools for the defined process. We experienced little integrations among the tools, and not full compliance with OMG's standards. Import and export of models among the tools led to several problems, including the lack of support for keeping the models consistent in the various tools. Keeping consistency manually should be avoided as it is error-prone. Moreover, since the adopted tools are closed source and uncertified, their adoption in safety-critical contexts can pose problems, if products must undergo certification. Therefore, we still noticed an

immaturity of software for MDE: there is a need of better integration among tools, and more flexibility is required to support a wider range of activities. MDE tools should not limit the activities of engineers, but should support and adapt to them. It is interesting to note how similar issues were identified in [90] and they are still open after about eight years.

Besides tools interoperability and integration, other big issues to face concern skills and organization: MDE innovation requires engineers with new skills and strong modeling abilities, and companies have to consider re-organizing their structure to better fit the deep changes brought by model-driven approaches. Indeed, the management is required to re-arrange the forces inside the company in order to adapt consolidated practices to the transformation. The importance of developers and testers will change, and analogously the roles assigned to requirements engineers and designers will become more relevant. These variations impact deeply on human-organizational factors; they translate in a severe managerial issue that must be coped with in industries.

Overall, our development life cycle turned out to be able to support, in a CENELEC EN 50128 V-Model, a broad range of engineering development and verification activities, enabling to detecting faults at an early stage of development, and exploiting the automation offered by multiple support tools. These benefits come from the multiple and novel viewpoints exploited by the methodology alongside the software life cycle, that favor a well-defined exploitation of the models used in the V&V activities.

Our experience showed how MDE is a mature technology that can provide fruitful results, not limited to code generation. However, we still experienced open challenges that must be properly addressed when integrating these approaches into the development process, involving the supporting tools, and current team skills and organization. Nevertheless, the experience demonstrated that the life cycle is suited to be gracefully introduced into current industrial processes, for its high flexibility to tools, activities and industrial practices.

Chapter 7

Case study 2: Model-Driven In-the-Loop Testing in Railway Domain

7.1 The Prolan Monitor Case Study

In our previous case study, we experimented the proposed model-driven development life cycle (Chap. 3) for the development of the *Prolan Block* (PB), a railway interlocking system under development by Prolan. In this chapter we focus on the environment modeling, considering the *Prolan Monitor* (PM), another part of the interlocking system, which must be CENELEC EN 50126, EN 50128 and EN 50129 SIL-4 certified. The PB shares with the PM the same hardware and middleware platform (Prosigma), which is the basis of the next generation of the company's products: according to the CENELEC terminology, *Prosigma* is a *generic product*, the PB and the PM are *generic applications*, and their installations are *specific applications*.

Similarly to the Prolan Block, the PM is deployed alongside railway segments, named *blocks*. The purpose of the PM is to send to modern interlocking systems that communicate status signals, generated by legacy interlocking systems physically connected to the PM, via protocols based on IP networks (such as via X.25 over TCP/IP).

In particular, the elements monitored by PM are named *railway objects*: to a railway

object is associated one bit of information coded by one couple of valent and antivalent physical signal values. The PM transmits the binary information associated to a railway object to other devices, and detects invalid states when the couples of electric signals are not consistent.

Indeed, since the physical inputs of PM are physical signals that derive from mechanical switches, they can suffer of special unstable states during which the signals quickly alternate in their value for a transient time, called *bounce time*: the PM has to properly filter bouncing signals separating transient noise from invalid inputs.

7.2 Experimentation

In this section, we report our experience in showing the benefits of environment modeling, i.e., the benefits derived from the CIT, by performing an early validation test of the PIM model through Model-in-the-loop tests (Fig. 4.1), focused on assessing the *debouncing functionality*.

7.2.1 Model-driven development

We followed the development subprocess of the proposed life cycle (Chap. 3) to build an implementation of the Prolan Monitor. Therefore, we executed the phases of System Requirements Specification, System Design, Component Design and Implementation.

System Requirement Specification

In the System Requirements Specification, we define a CIM using SysML, that focuses on the requirements and on the context of the PM. The CIM developed for the PM includes: 27 Activity Diagrams, 4 Block Definition Diagrams, 5 Internal Block Diagrams, 1 Package Diagram, 9 Sequence Diagrams, 7 State Machine Diagrams, and 1 Timing Diagram.

The environment of the PM is shown in the BDD diagram in Fig. 7.1: the PM receives as inputs the signals sent by the (legacy) interlocking system, and communicates with a

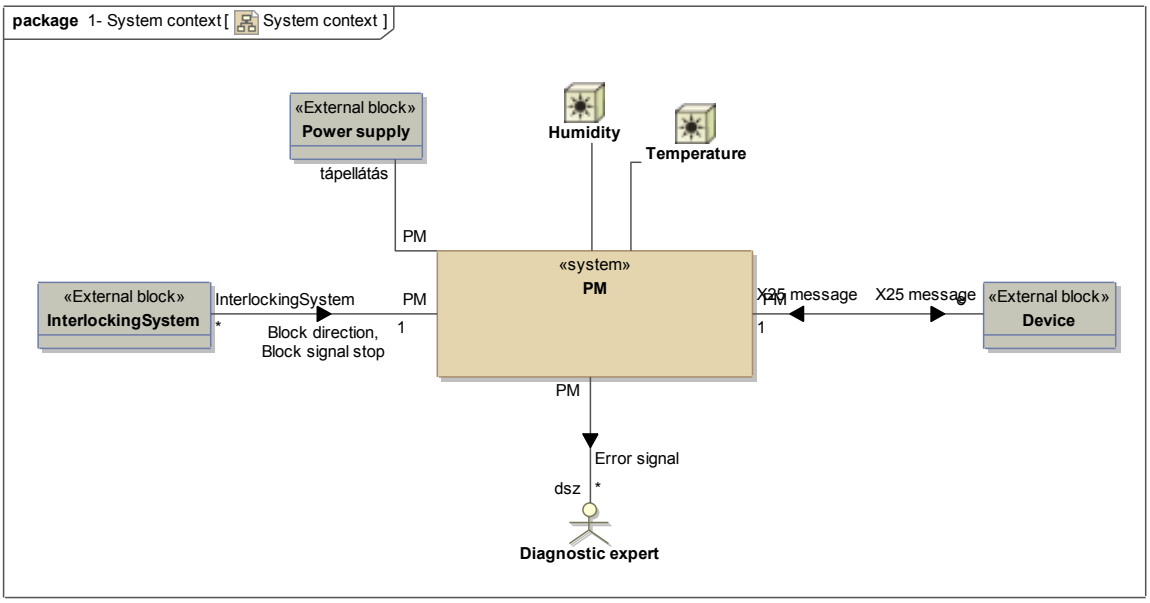


Figure 7.1: A SysML Block Definition Diagram of the CIM, showing the PM in its context.

remote device using X.25 messages.

The timing diagram in Fig. 7.2 specifies the functionality of signal filtering. The PM must sample the input with a period T_{sample}; since the input can suffer from transient states, a filtering solution must be implemented. By the valent and antivalent signals, two *debounced* signals are derived, which filter out the input variations shorter than $messageFilterTime$; then, the invalid values of the couple of debounced signals (i.e., $(0, 0)$ and $(1, 1)$) are masked if they last less than $maxTransientTime$. The railway objects assume invalid state if the signals bounce more than $maxBouncingTime$ or if the transient state lasts more than $maxTransientTime$.

System Design

In the System Design phase, the PIM was modeled, by defining the high level architecture of the system, including components' interfaces and their specification. In this pilot project, we design the PM system as composed of two instances of *PMRailwayObject*s, each of those in charge of managing a couple of input binary signals assigned to a physical railway

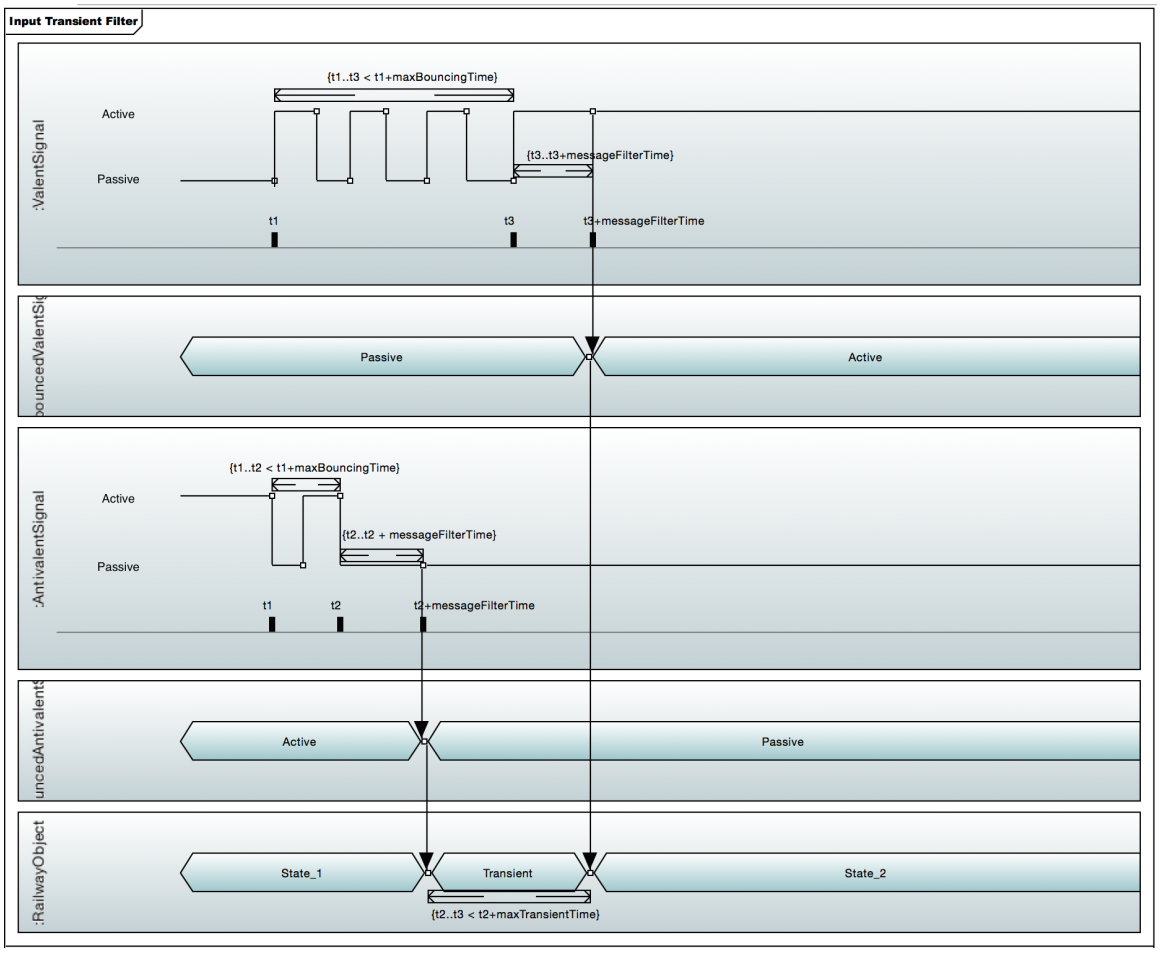


Figure 7.2: A UML Timing Diagram included in the CIM, representing the requirements for the functionality of signal debouncing.

object (fig. 7.3). In order to be platform-independent, the logic for accessing the hardware resources is masked by the *PMInterface*, required by the *PMSystem*.

Component Design

In the Component Design phase, the PIM is refined with the system internal design. The *PMRailwayObject* consists of one *Sampler*, two *PMDebouncer*s and one *PMInputFilter* (Fig. 7.4). The sampler reads from the input channels and notifies the new values to the two *PMDebouncer*s, that filter the bounces of the valent and antivalent signals. Finally,

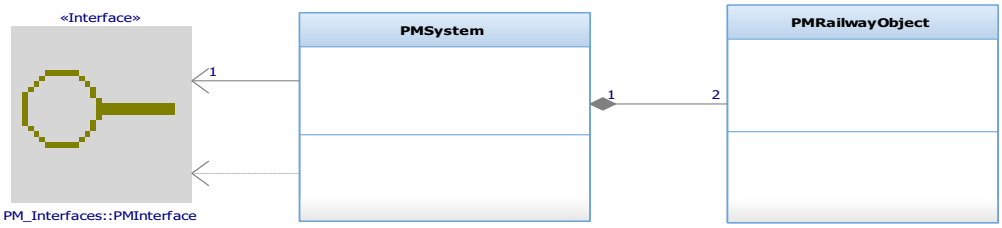


Figure 7.3: High level architecture of the Prolan Monitor.

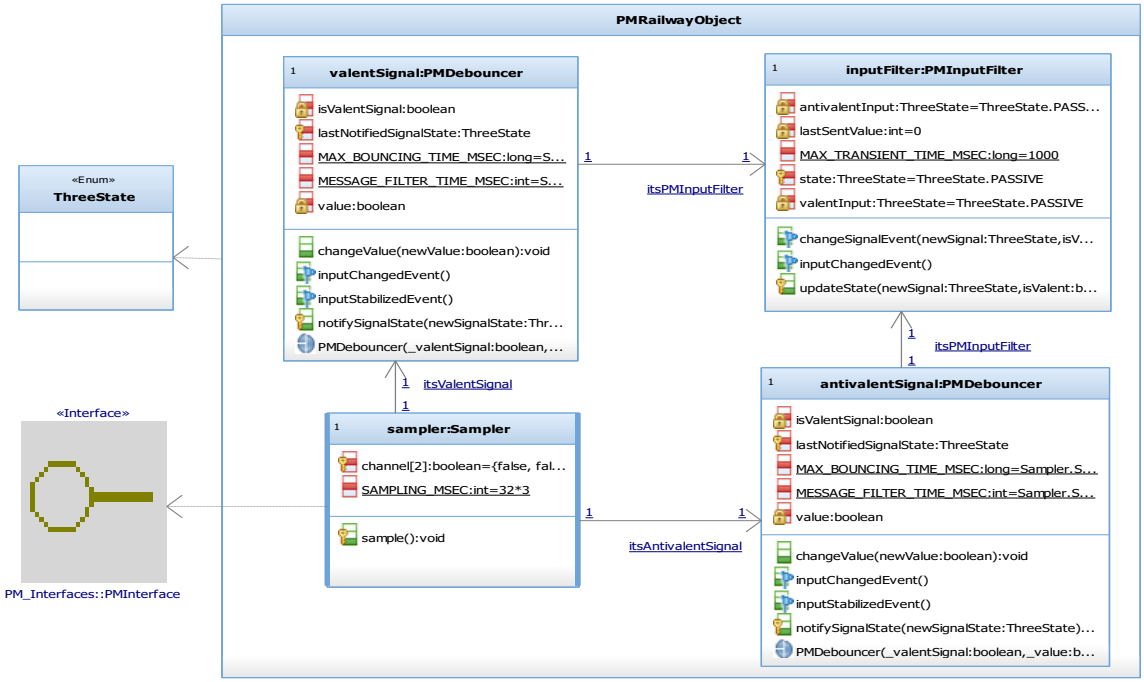


Figure 7.4: The UML Internal Structure Diagram of the *PMRailwayObject*, defined in the PIM during Component Design.

the *PMDebouncer*s propagate the signals to the *PMInputFilter* that filters out transient invalid states, as specified in Fig. 7.2.

At this stage, we also define the behavior of the components, by using UML State Machines or Activity Diagrams. The state machine in Fig. 7.5 models the behavior of the *PMDebouncer*s as a two-orthogonal composite state: the state machine in the left monitors

if the input is stable and sends *inputStabilizedEvent*s to the region in the right. The latter determines if the input is bouncing for a time longer than the maximum allowed.

The PIM is defined using IBM Rhapsody Developer (hereinafter: Rhapsody) [86], following guidelines to let the model be platform-independent. As for the case study of the Prolan Monitor, we avoided to insert target source code, and we preferred to define the behavior using UML. However, the tool partially constrained us to use the target programming language (i.e., Java) to specify the low-level components' behavior.

As the derived model is executable, it can be animated to observe the running program and the system's interactions. This is a feature that we exploited to get an immediate feedback on program behaviour. Moreover, we can easily create a prototype of user interface for interacting with the model: by means of the Rhapsody Panel Diagrams, we can set the inputs and observe the system output (Fig. 7.6).

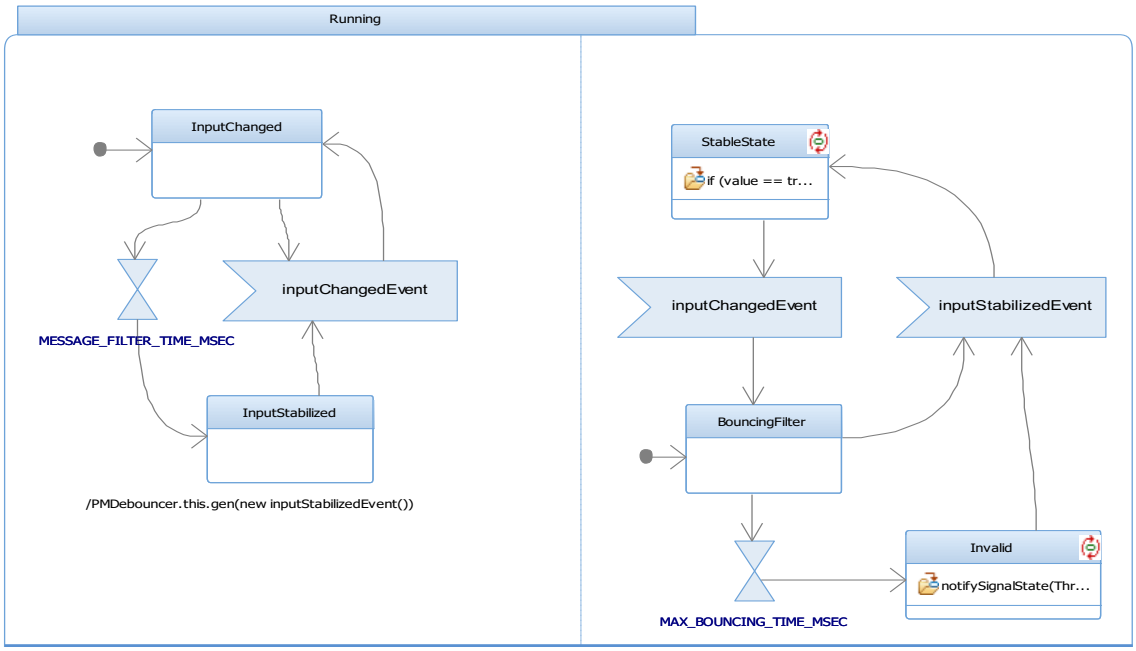


Figure 7.5: The UML Behavioral State Machine of the class *PMDebouncer*, defined in the PIM during Component Design.

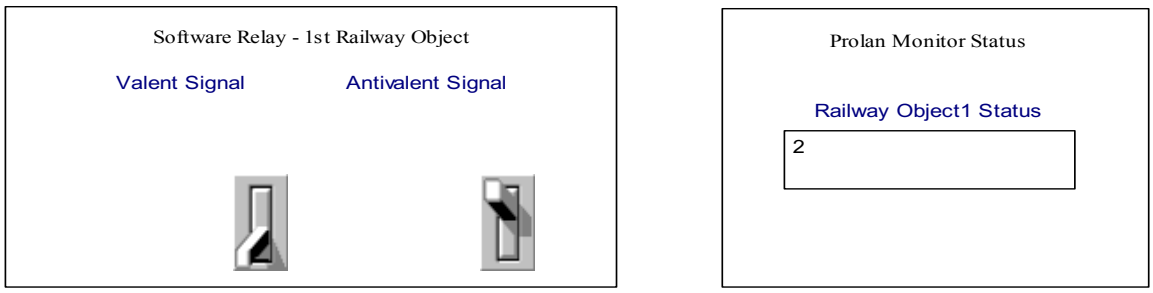


Figure 7.6: Rhapsody Panel Diagram associated to the PIM.

Implementation

In Implementation we define the PSM. Since we are using Rhapsody, this translates in setting several tagged values and tool-dependent parameters to enrich the PIM; then, the additional information is exploited for translating the model into code. The automatic translation generated around 4.3 thousands of lines of Java code; the source code is readable, understandable and ready to run.

7.2.2 The Computational Independent Test Model

The Validation Design phase aims at creating an executable model of the environment. Since we focused on assessing the debouncing features of the PM, our architecture of the CIT model is formed by two *CITRailwayObject*s: each *CITRailwayObject* controls the couple of logical signals associated with the binary information that they encapsulate; from the CIT *point of view*, the PM is an actor (Fig. 7.7).

The *CITRailwayObject*s are modeled as composed of one *SignalGenerator* and one *EventGenerator* (Fig. 7.8): the *EventGenerator* determines the next output to be triggered, as specified by a user-defined operational profile. The *EventGenerator* generates the following events:

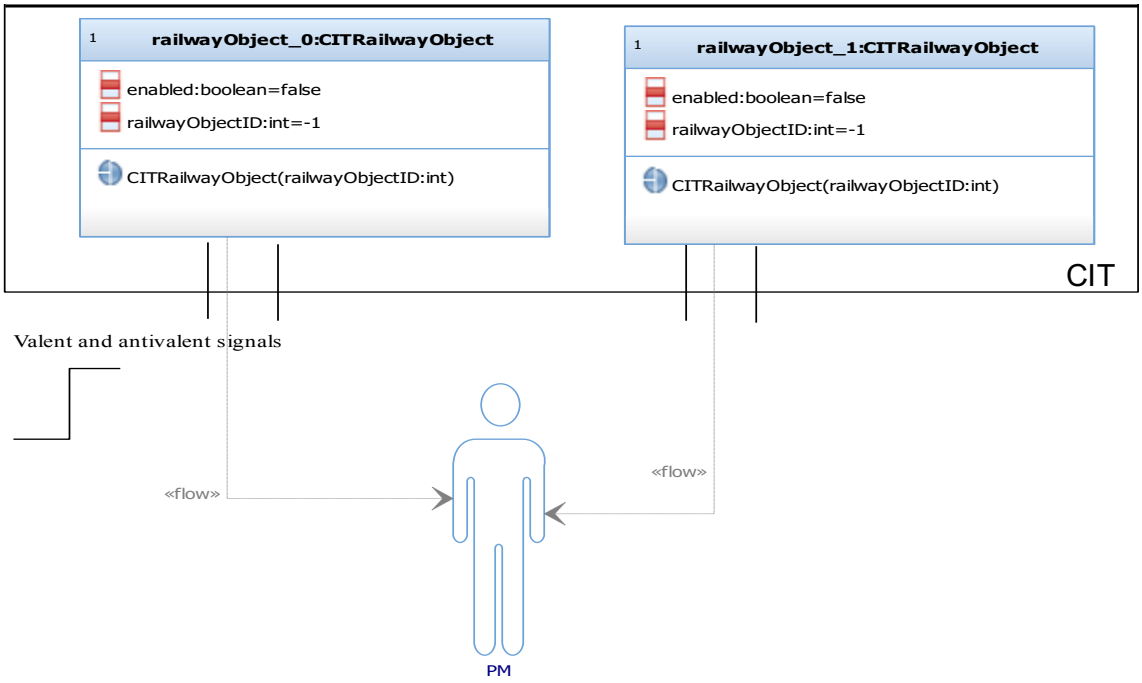


Figure 7.7: The architecture of the CIT.

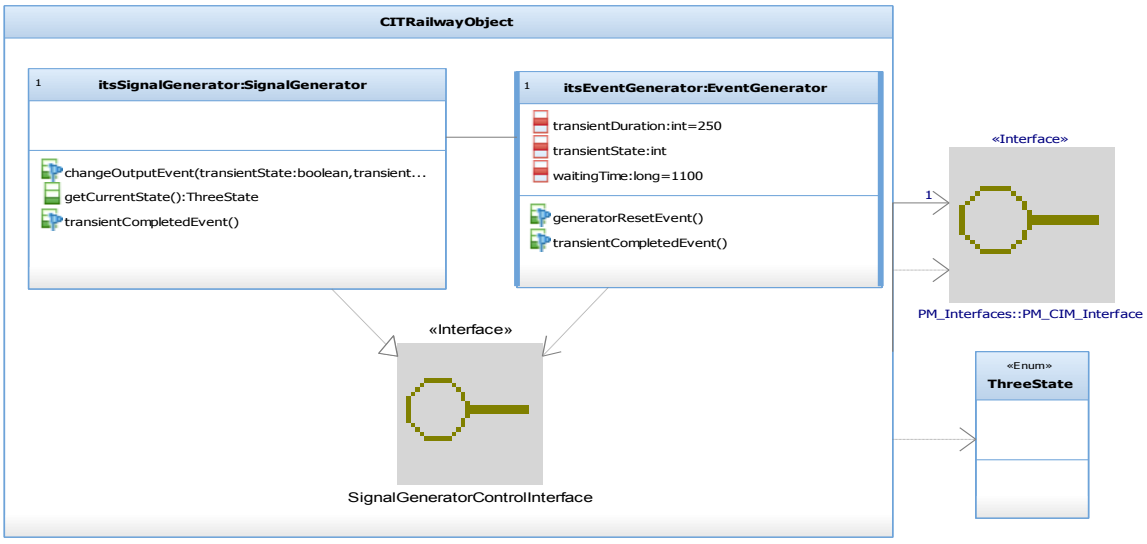


Figure 7.8: The internal design of the CITRailwayObject.

NoAction: the output is not altered in the next event generation loop;

ChangeStableState: the railway object switches between valid stable states;

CreateSpike: the output moves to an invalid state and then back to the previous stable state in order to simulate a transient in the electric signals;

Fail: the railway object moves to an invalid state and then fails.

According to the events sent by the *EventGenerator*, the *SignalGenerator* properly sets the couple of output signals and manages the duration of the transients. The behavior of the *EventGenerator* is modeled by an activity diagram (Fig. 7.9), while the behavior of the *SignalGenerator* by a state machine (Fig. 7.10): the *SignalGenerator* can generate sets of valid or invalid signals; when a *changeOutputEvent* is triggered by the *EventGenerator*, it evolves to the next stable state passing through invalid signals (i.e., $(1, 1)$ or $(0, 0)$) for a time equals to *transientDuration*. Then, if the *CITRailwayObject* is not failed, the *SignalGenerator* notifies the *EventGenerator* that the next stable state has been reached, and it starts to wait for the next event.

A panel diagram allows to interact with the CIT (Fig. 7.11): it offers a couple of knobs to set the period of event generation as well as the duration of transient states, and shows the output generated by the railway object.

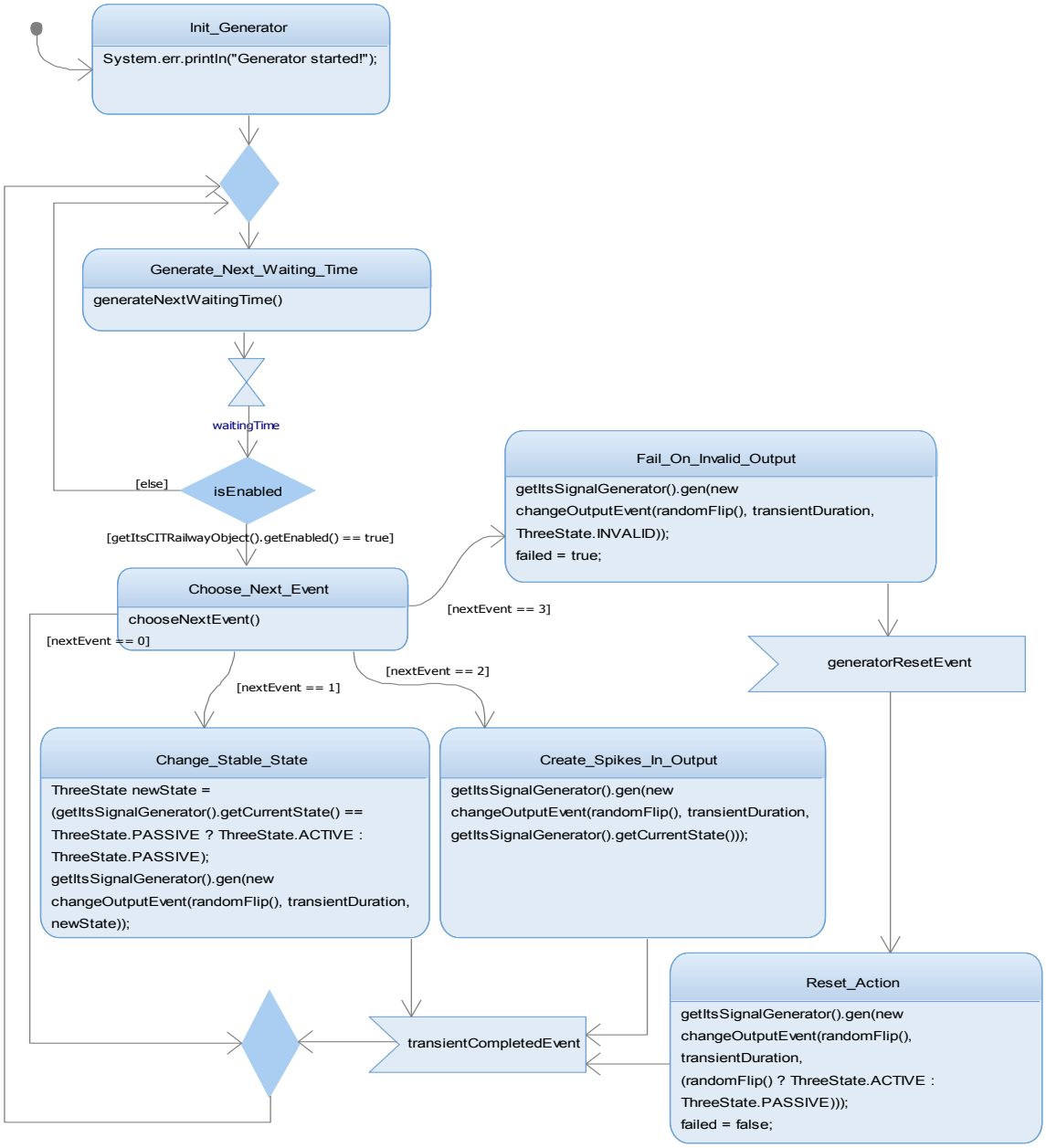


Figure 7.9: The UML Activity diagram that models the *EventGenerator*, defined in the CIT during Validation Design.

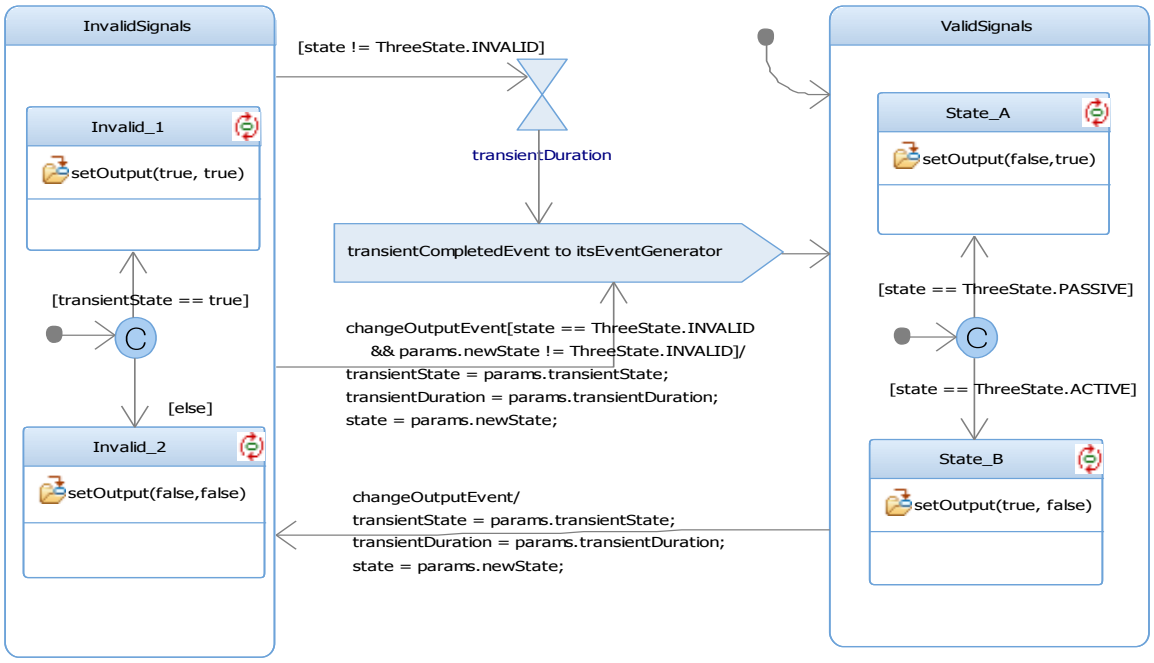


Figure 7.10: The UML Behavioral State Machine model of the *SignalGenerator*, defined in the CIT during Validation Design.

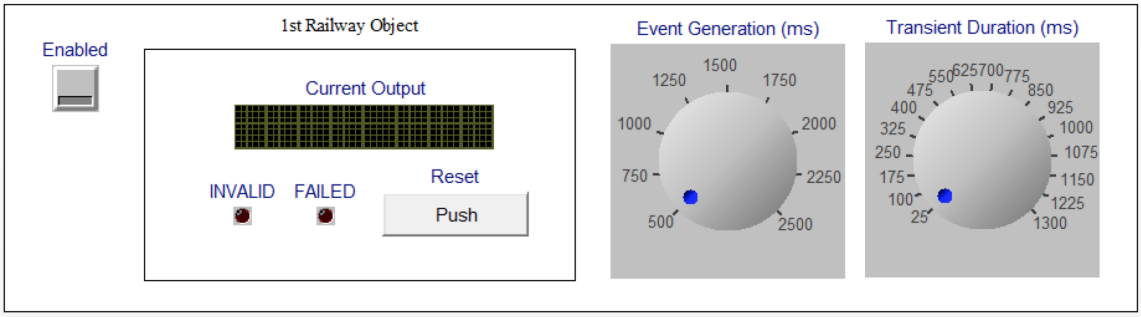


Figure 7.11: Rhapsody Panel Diagram of the CITRailwayObject.

7.2.3 Model-in-the-loop testing

We integrate the CIT and the PIM in order to perform the Model-in-the-loop testing. Since their interfaces are complementary, we link the two models by an adapter that simulates a physical relay, the *VirtualRelay* (Fig. 7.12): the CIT sends commands to switch the virtual relays, while the PIM reads their status.

Once the CIT and the PIM are linked (Fig. 7.13), we can execute the whole model and examine its evolution by means of the output console and of the panel diagrams, as shown in Fig. 7.14.

To assess the fulfillment of the requirements for the functionality of signal filtering, we have designed a test plan to assess if the events received by the *External Device* are the expected ones, according to the behavior of the *Interlocking System* and of the input signals (Figs. 7.1, 7.2). We apply category partition testing (CPT) on the CIT's interface (Table 7.1), deriving six test case obligations: in this type of testing, categories are configurations of the environment (i.e., of the CIT) that lead to the generation of different sequences of effective stimuli to the SUT. The test case specification is summarized in Tables 7.2 and 7.3. As test oracle, we implemented a script to analyze the execution traces of the actors and of the PIM in order to detect any undesired behavior.

To execute the tests, the code of the model (in configuration *in-the-loop*) is generated without the instrumentation needed for animating the model in Rhapsody, so as to avoid

Figure 7.12: The software adapter linking the interfaces required by the CIT and the PIM.

Parameter	**Categories**		**Constraints**	**Test case ID**
Input domain	(1.1)	Valid values and invalid transients		TC1-4
	(1.2)	Valid and invalid values	[ERROR]	TC5
Input frequency	(2.1)	Low frequency		TC1-2, TC4-6
	(2.2)	High frequency	[SINGLE]	TC3
Duration of transients	(3.1)	Undetectable by the SUT	[SINGLE]	TC3, TC6
	(3.2)	Detectable by the SUT		TC1-2, TC5
	(3.3)	Erroneous for the SUT	[ERROR]	TC4
Signal fluctuations	(4.1)	Low probability		TC1, TC3-6
	(4.2)	High probability	[SINGLE]	TC2

Table 7.1: CPT test categories.

TC ID	**Categories covered**	**Event Generation Period**	**Transient Duration**	**Probability of Fluctuations**
TC1	1.1, 2.1, 3.2, 4.1	2.5 s	100 ms	1%
TC2	1.1, 2.1, 3.2, 4.2	2.5 s	100 ms	40%
TC3	1.1, 2.2, 3.2, 4.1	65 ms	10 ms	1%
TC4	1.1, 2.1, 3.3, 4.1	2.5 s	5,000 ms	1%
TC5	1.2, 2.1, 3.2, 4.1	2.5 s	100 ms	1%
TC6	1.1, 2.1, 3.1, 4.1	2.5 s	10 ms	1%

Table 7.2: Specification of the MIL test cases. All test cases except TC5 send to the SUT valid input values ($(1, 0)$ and $(0, 1)$) and invalid transient values ($(0, 0)$ and $(1, 1)$).

Parameter	**Value**
T_{sample}	35 ms
messageFilterTime	$3 * T_{sample} = 105$ ms
maxBouncingTime	$5 * T_{sample} = 175$ ms
maxTransientTime	1,000 ms
Time tick	5 ms
Execution time	300 s

Table 7.3: Configuration of the SUT for the experiments.

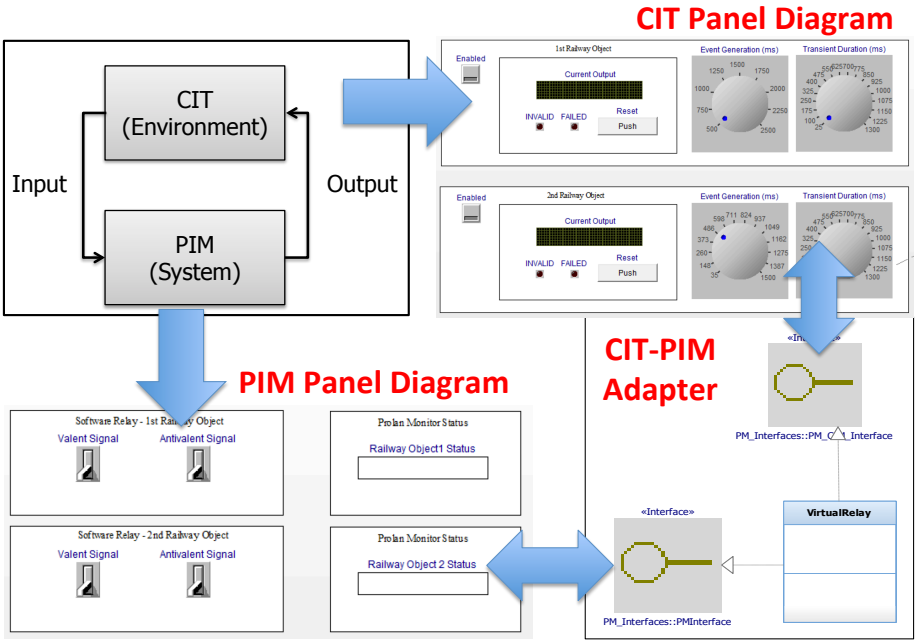


Figure 7.13: The configuration of the PM for MIL Testing.

slowing down the execution. Since the tests TC3 and TC6 require a time granularity of 10 ms, we tuned the *time tick* to 5 ms: this parameter specifies the time resolution to be used to poll the time events of the state machines and of the activities. By analyzing the execution traces during testing, we assured that the hardware was adequate to meet the timing constraints on the event queues' schedulers.

Note that, as the events are sent to a running software system, we are actually performing a form of Software in-the-loop testing. However, tests are not executed on the final software code, but on the instantiation of the PIM generated by Rhapsody that we are adopting for animation and testing purposes.

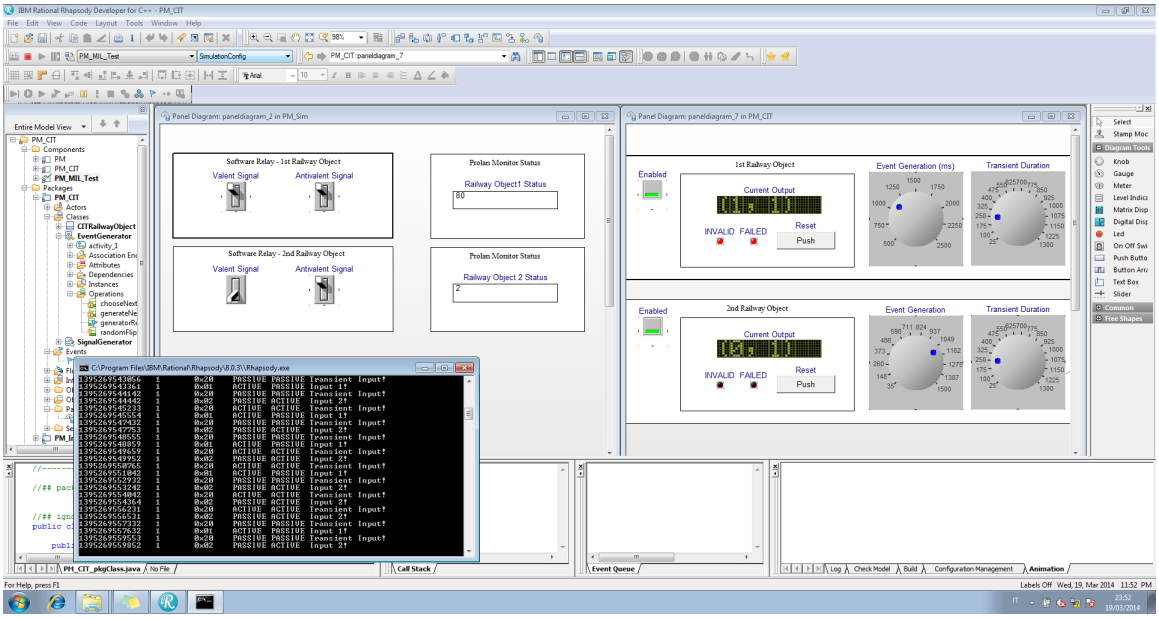


Figure 7.14: A screenshot of Rhapsody showing two panel diagrams and an output console: the panel diagram on the left is linked with the PIM, whereas the panel diagram on the right is connected to the CIT. In Model in-the-loop configuration, both are linked through the *VirtualRelay*, and are part of the animation that produces output in the console.

7.2.4 Software- and Hardware-in-the-loop testing

To use the CIT for Software- and Hardware in-the-loop testing, we only need to change the adapter in figure 7.12, with another adapter that forwards the events to the actual SUT.

Specifically, to run Hardware-in-the-loop tests in our case study, we replace the *VirtualRelay* with an interface that sends the events to a physical relay card connected to *Prosigma*.

The HIL configuration is shown in Figure 7.15:

1. the CIT is connected to a relay card through an Ethernet interface. Interacting with the interface, the CIT controls the relays, thus the input signals for the PM;

2. the relay card is physically connected to *Prosigma*, that is the target hardware on which the Prolan Monitor runs. The PM reads the inputs and filters the signals;

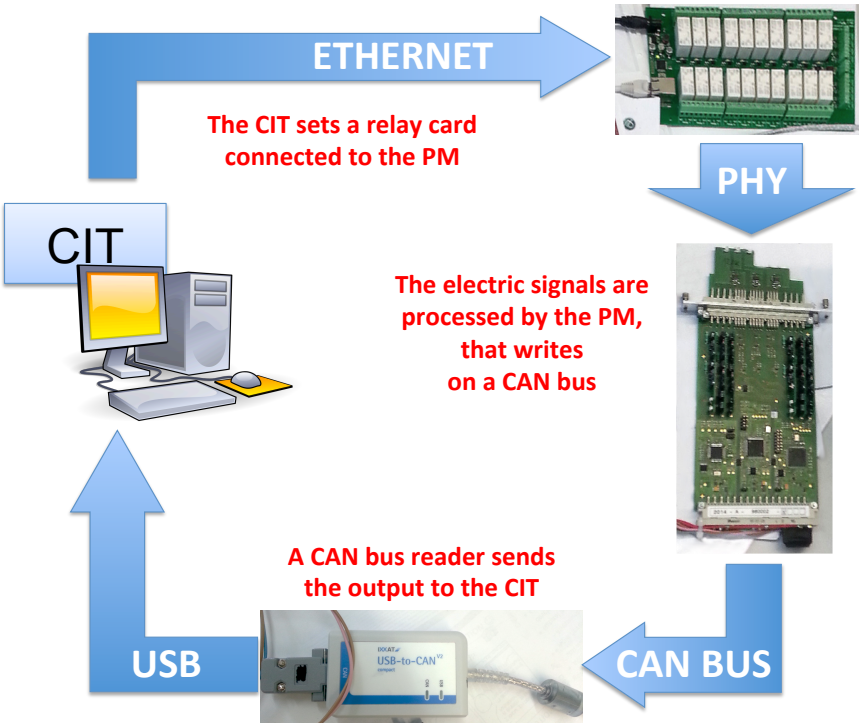


Figure 7.15: The configuration of the PM for HIL Testing.

3. since the PM sends the filtered inputs to a network card connected on a CAN bus, these messages are sniffed by a USB-to-CAN device that is connected to the CIT through USB;

4. the CIT receives the filtered signals through an interface that interact with the USB-to-CAN device, verifying that the PM behaves as expected.

7.3 Discussion

The SUT passed all the model-in-the-loop tests, behaving correctly. The CIT enabled an early detection of design fault in the life cycle, because we exercised the design model in its context, before a complete implementation was available.

Moreover, focusing on modeling the environment, we are suggesting a form of separation of concerns to build the validation test plan, that guides the design of test cases considering the conditions that affect the behavior of the environment, reasoning on the modalities with which it interacts with the SUT.

The CIT also enables to perform SIL and HIL testing, that provide multiple benefits to companies that develops critical systems, like Prolan:

- the company can validate the multiple instances of the system by executing tests on the final products (i.e., the Prolan Monitor on top of *Prosigma* configured for a specific installations – what is called a *specific application*), assessing their behavior in a simulated environment. Indeed, CIT enables to perform load and stress testing, that are high recommended by CENELEC EN 50128 during V&V activities;

- moreover, since the CIT can assess any implementation – not only the one developed using the proposed model-driven methodology –, it is reusable to assess the implementation developed in *diverse programming*. In fact, high safety-critical systems like *Prosigma* adopts a *lockstep fault-tolerant approach*, i.e., it is composed of three computer systems with diverse architectures that run the same functions in parallel. The applications the run on these platform are generally developed with diverse programming, thus the company has to validate three different implementations that comply to the same requirements, but that are developed by separate teams. The use of diverse programming is common for safety-critical applications, and high recommended by CENELEC EN 50128;

Finally, we highlight that the CIT can be easily reused for future projects or be integrated as a part of more complex CITs. In this way, it becomes a valuable company assets for validating domain specific systems.

Chapter 8

Case study 3: Model-Driven FMEA in Automotive Domain

8.1 Experimentation

In Chapter 3 we presented a model-driven methodology to support Failure Mode and Effects Analysis (FMEA), that can be integrated with the proposed software development life cycle. The approach enables formal knowledge representation – thus automated reasoning, making FMEA part of the model-driven design of critical systems.

This work was born from the fruitful industrial-academic collaboration in the context of European project CECRIS [9], indeed the research on FMEA was favored during six month of secondment of the Ph.D. candidate in Critical Software S.A., a multinational company with headquarters in Portugal that develops safety-critical systems and provides consulting and expertise for the certification. Critical Software performs FMEA analysis for its customers, and a model-driven approach for FMEA is useful to support its business, saving efforts and time to conduct the analysis.

The following case study has been provided by the ARTEMIS Joint Undertaking project *Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environment* (EMC^2) [91], that aims at providing cost efficient solutions for integrating applications with different levels of safety and security on single computing platforms

in open contexts, including multi-core and many-core computing hardware.

In particular, the use case is a demonstrator of a modern embedded system for mixed criticality car applications under development by Critical Software S.A., that offers a solution to run concurrently software with distinct safety-critical requirements on a multi-core platform.

In this case study, we report our experience on how to conduct the analysis of failure modes, propagation and effects from SysML design models, using our approach, including FMEA Diagrams.

8.1.1 System Modeling

In System Modeling, the Designer targets to formalize the knowledge of the system by means of the model. Therefore, (s)he starts to specify the system requirements through use cases (Fig. 8.1). The system is an info-entertainment software which runs concurrently with a safety-critical in-car emergency call (*eCall*) application. The latter activates assistance in case of accidents, automatically sending relevant information (e.g., position) to rescue services.

Then, the Designer models the requirements and their dependencies, by means of SysML Requirement Diagrams: the requirements can be refined with a top-down approach, using relations of containment; and defining dependencies between couples of requirements. For instance, figure. 8.2 includes the non-functional requirement *EMC2-REQ-0305* that specifies the constraints on the execution of real-time tasks: it belongs to the class of requirements named *ECM2-REQ-0300*, and depends on the functional requirement *EMC2-REQ-0121*, which requires a scheduler with a preemptive queue for the proper handling real-time tasks.

After the requirements have been specified using SysML Requirement and Use Case Diagrams, we move to model the system architecture, using Internal Block Diagrams. The top-level consists of a real-time operating system (RTOS) that manages safety-critical tasks

and resources running concurrently on distinct CPU cores, with a commercial off-the-shelf operating system (Android OS), contained within a virtualized environment managed by the Hypervisor (Fig. 8.3). The less critical tasks run in user space atop the RTOS. Finally, the Hypervisor and the RTOS interact with a Board Support Package (BSP) abstracting hardware-specific services. The RTOS internal structure (Fig. 8.4) comprises a Scheduler, and other components managing Clock, Devices, Resources, and System Calls. Requirements are allocated to components at this stage.

In addition, the Designer specifies by behavioral diagrams the realization of the use cases and the components's behavior, according at an abstraction level suited for the scope of the FMEA analysis. For instance, we modeled the Activity Diagram in Fig. 8.5 to show how the internal parts of the system collaborate to realize the use case *Perform eCall* (Fig. 8.1).

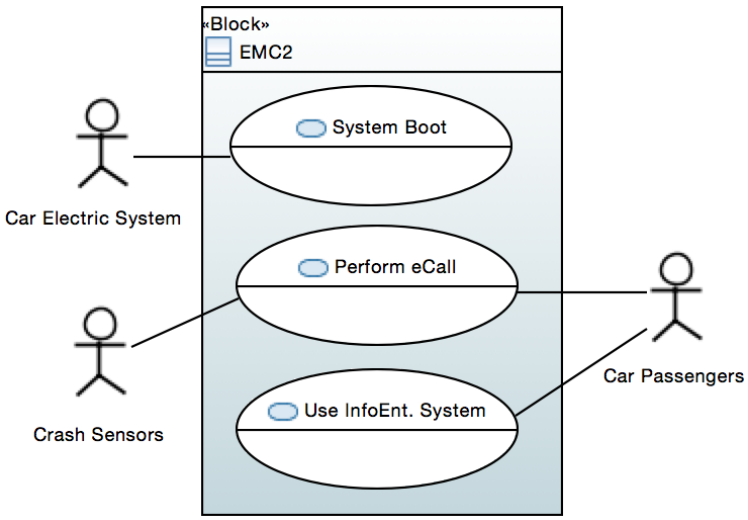


Figure 8.1: Excerpt of Use Cases Diagram of EMC^2 prototype.

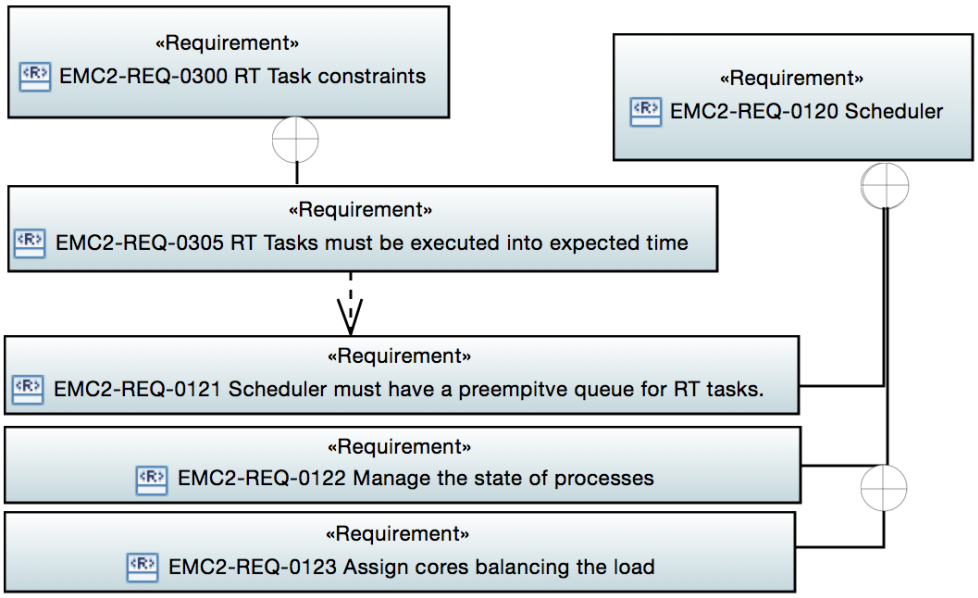


Figure 8.2: Excerpt of SysML Requirements Diagram of EMC^2.

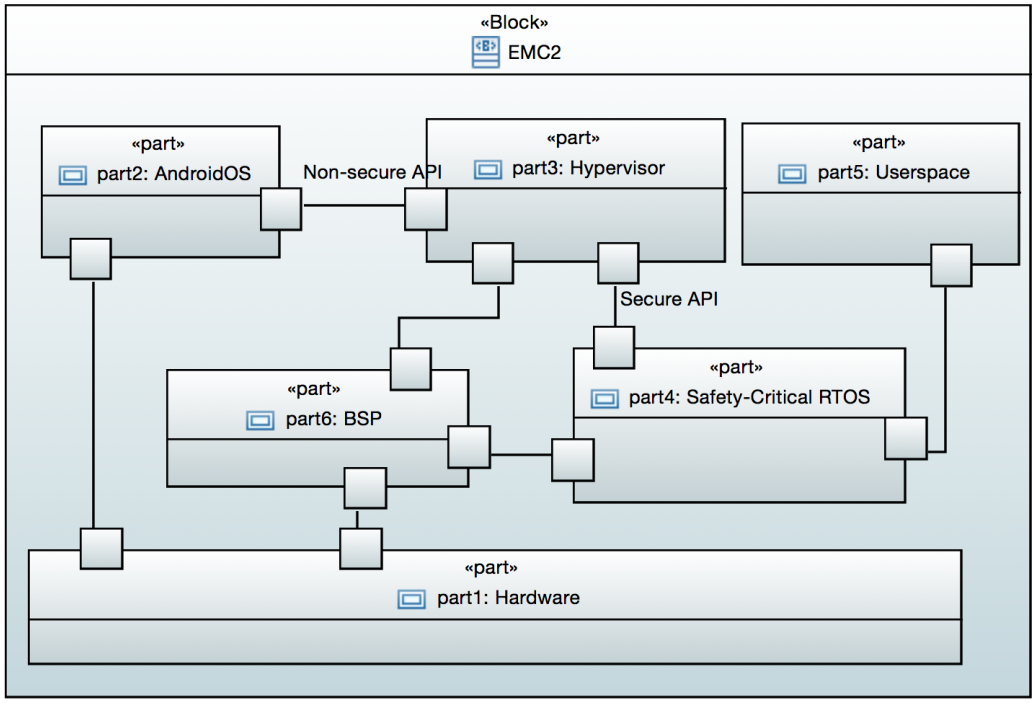


Figure 8.3: SysML Internal Block Diagram of the EMC^2 prototype.

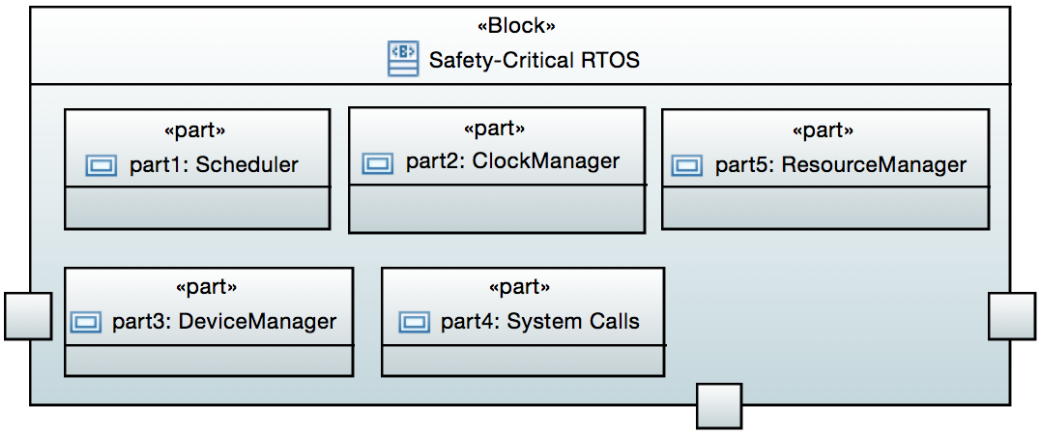


Figure 8.4: Internal Block Diagram of the Safety-Critical RTOS Component.

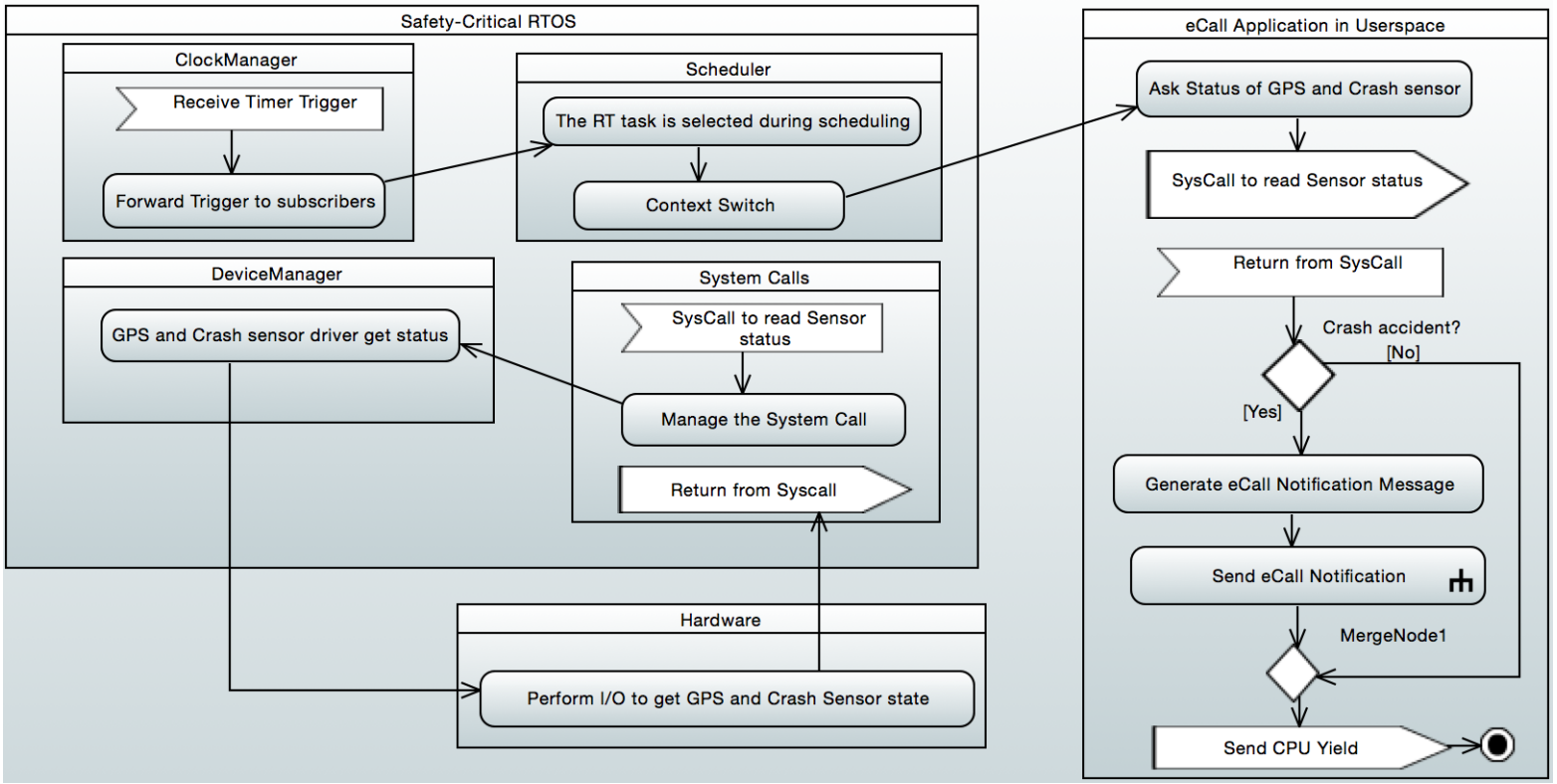


Figure 8.5: Activity Diagram of the use case *Perform eCall*.

8.1.2 FMEA Modeling

FMEA Modeling aims at refining the design model with *FMEA-oriented information* exploitable for automatic FMEA analysis in the next phase. It starts by refining the design model: the FMEA Analyst identifies the components' functionalities with a *FMEA viewpoint*, i.e., abstracting the component functionalities according to the FMEA scope, and on the basis of the behavioral diagrams modeled by the Designer, such as the Activity Diagram of Fig. 8.5.

The FMEA Analysts specifies the component functionalities by means of use cases. For instance, figure 8.6 shows the high-level functionalities assigned to the *Safety-Critical RTOS* and to the *Scheduler*: the *Scheduler* creates new processes, modifies their status, and performs their scheduling to the processors.

Now come into play the FMEA Profile and the FMEA Diagrams for conducting the

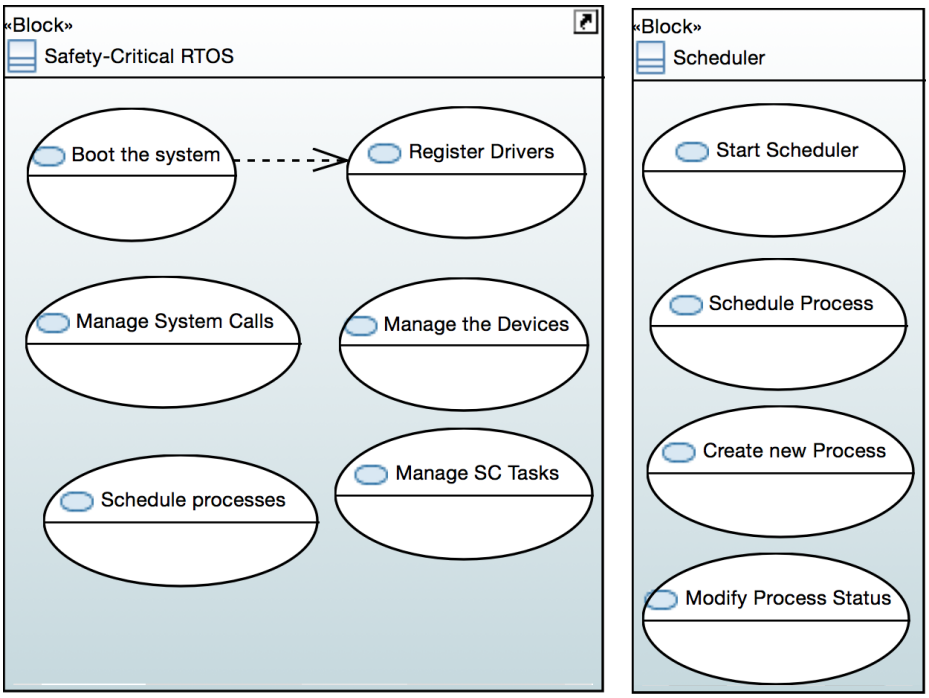


Figure 8.6: Use Cases for the SC-RTOS and Scheduler components.

analysis of failure modes, propagation and effects in a systematic way. The engineer proceeds in two iteration with a bottom-up approach.

In the first iteration, (s)he creates one FMEA Diagram for each component: (s)he places the CUA in the middle, its functionalities in the top-right side of the diagram, and CUA's requirements in the lower-right; then, looking at the CUA functionalities, the analyst defines the CUA failure modes. The failure modes are represented as use cases with the stereotype *FailureMode*.

A tool should support the automatic generation of FMEA Diagrams. Indeed, an automatic model-to-model translator can create one FMEA Diagram for each CUA. Then, the translator can also identify the CUA's adjacent components, its assigned requirements and functionalities, and synthesize these information in the FMEA Diagram. Therefore, a simple translator can provide the Analyst with an environment ready for adding the valuable information in the FMEA model, avoiding redundant and pedantic tasks.

In the second iteration, once all components' failure modes have been specified, the engineer re-examines all diagrams: since the FMEA Diagram now have been populated with the failure modes of the adjacent components, it is easy for the Analyst to connect the external failure modes and internal faults with the CUA's failure modes, and these with their effects, by linking failure modes with the requirements.

Figure 8.7 shows one FMEA Diagram of EMC^2, having the *Scheduler* component as CUA. We can observe how the diagram offers a synoptical view on the CUAs functionalities and requirements, internal faults, and failure propagation. For instance, the *ClockManager*s failure mode *Timer Callbacks are not handled properly* can propagate to the *Scheduler*, causing the failure mode *Priority of Real-time Processes not respected*, that violates the requirement *EMC2-REQ-0121* and affects the functionality *Schedule Process*.

Besides the information added with the graphical support, the Analyst has to enrich the model with textual properties made available by the FMEA Profile, e.g., to specify the

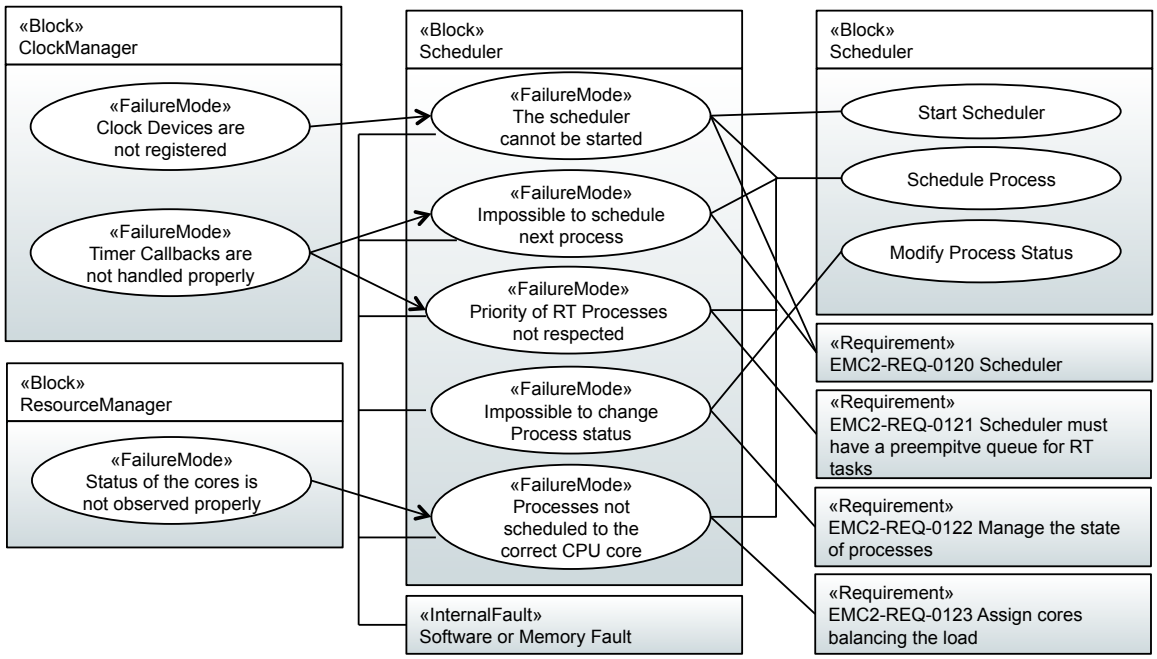


Figure 8.7: FMEA Diagram for the EMC^2 *Scheduler* component.

logical conditions that determine the propagation of a failure, or the reuse of past projects or domain knowledge. For instance, we adopted the stereotypes *FailureMode* and *InternalFault* to customize the semantic of the UML Use Case and Behavior for the purposes of FMEA (Fig. 8.7).

8.1.3 M2T trasformation

FMEA Diagrams augment the model with the information needed to perform a FMEA Analysis. Adopting a model-driven approach, the FMEA model is model-to-text translated into a Prolog Knowledge base, which is suited to perform the next phase of Model Analysis. The FMEA Profile empowers the automatic transformation rules, and helps to define the mapping between the model elements with the Prolog facts and predicates.

In this case study, we manually performed a preliminary translation of a portion of the model, converting in Prolog the information about the requirements, use cases, fault and

failure propagation. The knowledge base defines in Prolog the facts regarding the elements (e.g., the blocks, the internal fault and the component's functionalities), as well as the relations between the component and the failure mode, and between the functionalities and the requirements, through the predicates *failureMode*, *impactUC* and *impactReq*; then, the predicate *doesFail* relates the components failure modes with the faults and failures occurred in the system.

Figure 8.8 shows a Prolog code excerpt relative to the *Scheduler*. The predicates defined in the figure are the following:

isBlock(Component)

Component is a block.

isInternalFault(Component, Fault)

Fault is an internal fault of *Component*.

isUC(Component, UC)

UC is an use case of *Component*.

failureMode(Component, FailureMode)

FailureMode is a failure mode of *Component*.

impactUC(Component, FailureMode, UC)

FailureMode is a failure mode of *Component* that impacts the use case *UC*.

impactReq(Component, FailureMode, Req)

FailureMode is a failure mode of *Component* that violates the requirement *Req*.

doesFail(Component, FailureMode, FailureModeList, FaultList)

Component fails due to failure mode *FailureMode*, given the list of activated faults *FaultList* and the failures in the system *FailureModeList*.

```
/* Scheduler */
isBlock(scheduler).
isInternalFault(scheduler, softwareOrMemoryFault).
isUC(scheduler, startScheduler).
isUC(scheduler, scheduleProcess).
isUC(scheduler, modifyProcStatus).

/* ---- */
failureMode(scheduler, schedulerCannotBeStarted).
impactUC(scheduler, schedulerCannotBeStarted, startScheduler).
impactUC(scheduler, schedulerCannotBeStarted, scheduleProcess).
impactReq(scheduler, schedulerCannotBeStarted, req_0120).

doesFail(scheduler, schedulerCannotBeStarted, FailureModeList, FaultList) :-
    once(member([scheduler, schedulerCannotBeStarted], FailureModeList)),
(
    once(member([scheduler, softwareOrMemoryFault], FaultList));
    doesFail(clockManager, clockDevicesNotRegistered, FailureModeList, FaultList)
).

/* ---- */
failureMode(scheduler, impossibleToScheduleNextProcess).
impactUC(scheduler, impossibleToScheduleNextProcess, scheduleProcess).
impactReq(scheduler, impossibleToScheduleNextProcess, req_0120).

doesFail(scheduler, impossibleToScheduleNextProcess, FailureModeList, FaultList) :-
    once(member([scheduler, impossibleToScheduleNextProcess], FailureModeList)),
(
    once(member([scheduler, softwareOrMemoryFault], FaultList));
    doesFail(clockManager, timerCallbacksNotHandled, FailureModeList, FaultList)
).
```

Figure 8.8: Fragment of the Prolog Knowledge Base relative to the *Scheduler*.

8.1.4 Model Analysis

After the knowledge base has been generated, the analyst can pose query to the Prolog inference engine to actually perform the FMEA analysis, and to generate the FMEA worksheets. The queries are based on the predicates which have been defined in shared knowledge base, and that can be reused across multiple projects. The complexity of using Prolog for querying the system can be masked using simpler graphic user interfaces that abstract away from the underlying syntax of Prolog.

As an example of predicates that are present in a shared knowledge base to support the

analyses, we discuss the following:

shallowReqViolated(Component, LocalFault, LocalReqViolated, FailureMode)
true if the activation of *LocalFault* in *Component* causes *FailureMode* that causes the violation of the requirement *LocalReqViolated* which has been directly assigned to the component.

shallowAllReqViolated(Component, LocalFault, LocalReqViolatedSortedList)
true if and only if the activation of *LocalFault* in *Component* causes the violation of all the component requirements that appear sorted lexicographically in *LocalReqViolatedSortedList*.

deepReqViolated(Component, LocalFault, ReqViolated, FailureModeList)
true if the activation of *LocalFault* in *Component* causes directly or indirectly the violation of the requirement *ReqViolated*, due to the propagation of a failure of *Component* (listed in *FailureModeList*) which is generated by *LocalFault*, and considering all the requirements that are reachable in the graph of the requirements.

deepAllReqViolated (Component, LocalFault, ReqViolatedSortedList, FailureMode)
true if and only if the activation of *LocalFault* in *Component* causes directly or indirectly the violation of all the requirements ordered lexicographically in *ReqViolatedSortedList*, due to the propagation of the *FailureMode* of *Component* which is generated by *LocalFault*, and considering all the requirements that are reachable in the graph of the requirements.

All *deep-predicates* are also offered in form of *SysReqViolated*, that filter out the violation of requirements that are not at system level. An excerpt of the shared Prolog knowledge base containing the predicates for the model analysis is shown in figure 8.9.

```
% Search for a requirement violated without considering propagations
% (in adjacent components and in the requirement graph).
shallowReqViolated(Comp, LocalFault, LocalReqViolated, FailureMode) :-
    isInternalFault(Comp, LocalFault),
    doesFail(Comp, FailureMode, _, [[Comp, LocalFault]]),
    impactReq(Comp, FailureMode, LocalReqViolated).

shallowReqViolated(Comp, LocalFault, LocalReqViolated) :-
    shallowReqViolated(Comp, LocalFault, LocalReqViolated, _).

shallowAllReqViolated(Comp, LocalFault, LocalReqViolatedSortedList) :-
    isInternalFault(Comp, LocalFault), % Needed to print the LocalFault unification
    findall(LocalReqViolated,
        shallowReqViolated(Comp, LocalFault, LocalReqViolated),
        Z),
    sort(Z, LocalReqViolatedSortedList).

% Search for a requirement violated considering propagations
% (in adjacent components and in the requirement graph).
deepReqViolated(Comp, LocalFault, ReqViolated, FailureModeList) :-
    isInternalFault(Comp, LocalFault),
    doesFail(C, F, FailureModeList, [[Comp, LocalFault]]),
    (impactReq(C, F, ReqViolated);
     impactReq(C, F, LocalReqViolated), reqFailure(LocalReqViolated, ReqViolated)).

deepReqViolated(Comp, LocalFault, ReqViolated) :-
    deepReqViolated(Comp, LocalFault, ReqViolated, _).

deepSysReqViolated(Comp, LocalFault, SysReqViolated, FailureModeList) :-
    sysReq(SysReqViolated),
    deepReqViolated(Comp, LocalFault, SysReqViolated, FailureModeList).

deepSysReqViolated(Comp, LocalFault, SysReqViolated) :-
    deepSysReqViolated(Comp, LocalFault, SysReqViolated, _).

deepAllReqViolated(Comp, LocalFault, ReqViolatedSortedList) :-
    isInternalFault(Comp, LocalFault),
    findall(ReqViolated, deepReqViolated(Comp, LocalFault, ReqViolated), Z),
    sort(Z, ReqViolatedSortedList).

deepAllReqViolated(Comp, LocalFault, ReqViolatedSortedList, FailureMode) :-
    isInternalFault(Comp, LocalFault),
    failureMode(Comp, FailureMode),
    findall(ReqViolated,
        (deepReqViolated(Comp, LocalFault, ReqViolated, FailureModeList),
                once(member([Comp, H], FailureModeList)), H = FailureMode), Z),
    sort(Z, ReqViolatedSortedList).
```

Figure 8.9: Fragment of the Prolog shared Knowledge Base defining predicates for model analysis.

1st Query – Local Effects of Failure Modes

In the first query we asked to the inference engine to provide all the local effects of the *Scheduler*'s failure modes that are due to a single fault.

As shown in Fig. 8.10, Prolog correctly reported the five failure modes modeled in the FMEA Diagram in Fig. 8.7, and for each of them Prolog identified the affected requirements.

```
?- shallowReqViolated(scheduler, LocalFault, LocalReqViolated, FailureMode).

LocalFault = softwareOrMemoryFault,
LocalReqViolated = req_0120,
FailureMode = schedulerCannotBeStarted

LocalFault = softwareOrMemoryFault,
LocalReqViolated = req_0120,
FailureMode = impossibleToScheduleNextProcess

LocalFault = softwareOrMemoryFault,
LocalReqViolated = req_0121,
FailureMode = priorityOfRTNotRespected

LocalFault = softwareOrMemoryFault,
LocalReqViolated = req_0122,
FailureMode = impossibleToChangeProcStatus

LocalFault = softwareOrMemoryFault,
LocalReqViolated = req_0123,
FailureMode = procNotScheduledToCorrectCore
```

Figure 8.10: Results of the execution of the query 1 on the EMC^2 knowledge base.

2nd Query – End Effects of Failure Modes

In the second query, we asked Prolog to identify all the system level effects for each failure mode of the *Scheduler*.

As reported in Fig. 8.11, Prolog identified two system level requirement violations, *EMC2-REQ-0120* and *EMC2-REQ-0120*. The former is caused by any *Scheduler*'s failure modes, since it is a requirement associated to the functionalities offered by a scheduler in the system; while the latter is caused by the failure mode *Priority of RT Processed not*

respected: this failure mode leads to the violation of the local requirement *EMC2-REQ-0121* (as shown in the FMEA Diagram in Fig. 8.7) that through *EMC2-REQ-0305* propagates to the requirement *EMC2-REQ-0300* (as represented in Fig. 8.2).

```
?- deepAllSysReqViolated(scheduler, LocalFault, SysReqViolatedList, FailureMode).

LocalFault = softwareOrMemoryFault,
SysReqViolatedList = [req_0120],
FailureMode = schedulerCannotBeStarted

LocalFault = softwareOrMemoryFault,
SysReqViolatedList = [req_0120],
FailureMode = impossibleToScheduleNextProcess

LocalFault = softwareOrMemoryFault,
SysReqViolatedList = [req_0120, req_0300],
FailureMode = priorityOfRTNotRespected

LocalFault = softwareOrMemoryFault,
SysReqViolatedList = [req_0120],
FailureMode = impossibleToChangeProcStatus

LocalFault = softwareOrMemoryFault,
SysReqViolatedList = [req_0120],
FailureMode = procNotScheduledToCorrectCore.
```

Figure 8.11: Results of the execution of the query 2 on the EMC^2 knowledge base.

3rd Query – Causes of a Requirement Violation

In the third query we *reversed* the previous query, since we investigated all the failure modes that cause the violation of the requirement *EMC2-REQ-0300*. This query is relevant to check what failure modes must be managed to reduce the risk of critical system failure.

For this query, Prolog identified two failure modes: the first one is due to the *Scheduler*'s failure mode *Priority of RT Processed not respected* (i.e., the case spotted in the previous query); whereas the second one is associated to the failure mode *Timer Callbacks are not handled properly* of the *ClockManager*. In fact, analyzing the second failure mode (Fig. 8.7), we observe that this failure mode propagates again to the Schedulers failure mode *Priority*

of RT Processed not respected, that we know that affects *EMC2-REQ-0300*.

```
?- deepSysReqViolated(Component, LocalFault, req_0300, FailureModeList).

Component = scheduler,
LocalFault = softwareOrMemoryFault,
FailureModeList = [ [scheduler, priorityOfRTNotRespected] | _G45]

Component = clockManager,
LocalFault = softwareOrMemoryFault,
FailureModeList = [ [scheduler, priorityOfRTNotRespected],
      [clockManager, timerCallbacksNotHandled] | _G48]
```

Figure 8.12: Results of the execution of the query 3 on the EMC^2 knowledge base.

4th Query – Worksheet Generation

As fourth query, we wanted to assess the ability of the knowledge base to provide all the data necessary to generate a generic FMEA Worksheet for the *Scheduler*.

To build the column of the local effects we can use the output of the first query, which determined the local effects of all Scheduler's failure modes. Instead, to get the root causes and the end effects of each failure mode, we execute on the knowledge base the following query:

deepSysReqViolated(Component, LocalFault, SysReq, FailureModeList), once(member([scheduler, SchedulerFailure], FailureModeList)).

The query returns *true* if *LocalFault* in *Component* propagates according to the *FailureModeList* and causes a violation of the system requirement *SysReq*. For each *Component*, *LocalFault* and *SysReq*, the query gives only one answer per failure mode of the *scheduler* (i.e., only one unification between *SchedulerFailure* in *FailureModeList*).

By executing the query, we extracted all the information needed to complete the FMEA Worksheet: the data have been transformed and tabulated in Tab. 8.1.

Failure Mode	**Causes**	**Local Effects**	**End Effects**
The scheduler cannot be started	ClockManager due to software or memory fault that causes timer callbacks not properly handled	EMC2-REQ-0120	EMC2-REQ-0120
	ClockManager due to software or memory fault that inhibits the registration of the clock devices		
	Scheduler due to internal software or memory fault.		
Impossible to schedule next process	ClockManager due to software or memory fault that causes timer callbacks not properly handled	EMC2-REQ-0120	EMC2-REQ-0120
	Scheduler due to internal software or memory fault.		
Priority of the RT Processes not respected	ClockManager due to software or memory fault that causes timer callbacks not properly handled	EMC2-REQ-0121	EMC-REQ-0300
	Scheduler due to internal software or memory fault.		EMC2-REQ-0120
Impossible to change Process status	Scheduler due to internal software or memory fault.	EMC2-REQ-0122	EMC2-REQ-0120
Processes not scheduled to the correct core	Resource Manager due to software or memory fault that causes a status of the cores not observed properly.	EMC2-REQ-0123	EMC2-REQ-0120
	Scheduler due to internal software or memory fault.		

Table 8.1: FMEA Worksheet for the *Scheduler* generated automatically by queries on the knowledge base.

8.2 Discussion

In this case study we reported our experience with conducting the proposed model-driven FMEA in Chap. 5.

The approach tightly integrates the work of the Designer with the FMEA Analyst, that can start the analysis by reusing the knowledge provided by the former with a common language based on the standard SysML. How it was experienced in Critical Software, this solution provides high benefits for the FMEA, since the FMEA Analysts face the problem of manually extracting the information from multiple sources, in an error-prone way and typically using diagrams to synthesize the knowledge.

The novel FMEA Diagram enables the analyst to reason directly on the model to identify failure modes, propagation and effects, exploiting it as primary source of knowledge of the system:

- FMEA Diagrams offer a synoptical view on the structure and behavior of the components, thus help the Analyst in reasoning on the failure modes and failure propagation in a model-centric FMEA approach;
- by M2M transformations, the FMEA Diagrams concentrate the time of the FMEA Analyst in enriching the model with the valuable FMEA-oriented information;
- FMEA Diagram aims at becoming a common language between multiple RAMS teams to share the knowledge about the failure behavior of the system.

The analyses performed in Prolog revealed the suitability of the language to analyze the FMEA-oriented model for conducting the typical queries addressed in a FMEA analysis.

The use of a Prolog makes the methodology more flexible, since the analysts can reuse the knowledge derived by domain libraries or past projects: the Analysts can easily add new types of predicates in the knowledge base to support more specific and complex queries.

Indeed, Prolog can support particular kind of analysis typically not currently addressed during a FMEA Analysis due to their complexity, such as the analysis of multiple faults, or the effects of fault barriers in the system. The engine also supports custom queries sent directly from the command-line interface.

Finally, during Model Analysis we were able to generate a detailed FMEA worksheet for the component under analysis. Applying the methodology within model-driven life cycles, we can effectively reduce time and cost of development, enabling to early verification of RAMS requirements and support to FMEA analysis and documentation.

Conclusion

Model-driven techniques are appealing for industries, as they can reduce development costs, time and guarantee better quality of the products. However, the introduction of model-driven engineering into traditional development process is not a matter of using models and supporting tools to gain the benefits of the models, they require strong adaptation of current industrial practices, and have an influence on the software development process.

Indeed, MDE influences the organization and involves the skills of the engineers, the roles, the communication between the teams, the tools and the stakeholders' responsibilities. Companies that aim at innovating their process face the problem on how they should change long established methodologies: traditional development processes need to carefully reengineered for the MDE, and it is not clear the impact of the integration of MDE into development life cycles [92]; despite the great advances in the field of MDE, there are still few studies that focus on complete model-driven life cycle processes that are suited for the mainstream adoption. The lack of consolidated model-driven life cycles has become a serious concern of the scientific community, that has recently started a new series of conferences dedicated to the topic: the Model-Driven Development Processes and Practices workshop (MD^2P^2) [93].

This dissertation aims at filling this gap, by proposing a novel software development process that uses model-driven approaches during the whole life cycle, addressing the limitations of past proposals:

- it covers the full software development life cycle, and merges into the 'V' the abstractions of MDA, MDT and CENELEC V-Model, to favor the reuse and improve the exploitation of models, by focusing on the relevant aspects of the system during all phases of the process. The methodology agrees with the CENELEC EN 50128 standard, as it uses a compliant V-Model and includes the activities and roles required for the certification, supporting the automatic generation of the artifacts;

- the life cycle is flexible to multiple applications and domain, since it uses OMG standard and general-purpose languages. Nevertheless, the methodology is founded on *viewpoint*s rather than on *model*s, thus it can be instantiated with domain-specific formalisms and particular tools to meet the industrial needs;

- the process supports numerous techniques of model-driven V&V, including functional, structural, interface and validation testing. This enables to choose the most cost-effective combination of methods during the V&V. Moreover, it includes a new definition of the CIT, that enables to perform model-, software-, and hardware- in-the-loop testing: these kinds of tests are valuable for the assessment of critical systems, because allow to the detection of faults at an early stage, reducing the overall costs of development;

- the process includes a model-driven SysML FMEA, which supports reliability and safety analysis. The integration favors better consideration of RAMS requirements in the first stages, by allowing early feedback from the FMEA team, that enables to an early verification and validation of the design. Differently from past studies, the proposal exploits custom SysML FMEA Diagram, that speeds up the activity of FMEA Modeling, by supporting the reasoning of the FMEA Analyst in the same conceptual framework of the Designer. Moreover, Prolog follows the same inductive-deductive mindset of the analyst and opens to custom queries, that enable to a wide

range of FMEA analyses, also enhancing the reuse of the domain knowledge by means of separate Prolog libraries.

Concluding, this work has been strongly influenced by the fruitful collaboration with the industry conducted by the Ph.D. candidate, in the framework of the EU Project CECRIS. Industry has still no clear understanding on how to adapt their processes to exploit MDE, and most of past research focused on particular phases or subprocesses of the software development life cycle, creating multiple isolated techniques. This thesis engineers the model-driven development of critical systems, and composes, with new ideas, the multiple fragments of research to create one picture of the MDE, ready for the industrial application and the development of critical systems.

Glossary

Activity the execution of a step of a Software (sub)process in the context of a SDLC.

Artifact is an outcome of a software process.

Bounce time is the transient time during which an unstable signals quickly alternates in its value.

CAN Bus is a high reliable serial bus designed to allow microcontrollers to communicate with each other. It was born in automotive domain and now adopted for many kinds of embedded systems.

Diverse programming is an approach for detecting and masking residual design (and software) faults by developing redundant versions of the system.

Generic application is a system that can be re-used for a class/type of application with common functions (CENELEC EN 50129).

Generic product is a system that can be re-used for different independent applications (CENELEC EN 50129).

Markov chain is a discrete time stochastic process that evolves through states on a state space: informally, at each step the process is in a state s and can evolve to s' with a probability only depending on the current state and not on the sequence of events that preceded it..

Markov decision process is a discrete time stochastic control process: informally, at each step the process is in a state s and randomly can evolve to a state s' by choosing an action a. The probability of choosing a is not influenced by previous states and actions, and the action determines a reward for the decision maker.

Model a description of system that throught abstractions neglects all the aspects that are not of interest.

Phase used as synonym of subprocess in the context of a Software Process.

Safety Function a function that implements a part or whole of a safety requirements (CENELEC EN 50128).

Safety Integrity the likelihood of a system satisfactorily performing the required safety functions under all the stated condition within a stated period of time (CENELEC EN 50126).

Safety Integrity Level one of a number of defined discrete levels for specifying the safety integrity requirements of the safety functions to be allocated to the safety related systems. Safety Integrity Level with the highest figure has the highest level of safety integrity (CENELEC EN 50126).

Safety-related software software which performs safety functions (CENELEC EN 50128).

Signal aspect is the appearance of a lineside signal, as viewed from the direction of an approaching train, or the appearance of a cab signal [94].

Software Development Life Cycle includes the software processes used to specify and transform software requirements into a deliverable software product.

Software Process is a set of interrelated activities and tasks that transform input work products into output work products; it includes the definition of inputs, transforming

work activities, and outputs generated, can also include input and exit criteria, decomposition of the work activities into tasks, personnel, roles and responsibilities and tools. May include subprocesses.

Software Process Model is a simplified description of a software process that presents one view of that process.

Software Product Life Cycle includes a SDLC with additional software processes for all other inception-to-retirement processes for a software product.

Specific application is a system used for only one particular installation (CENELEC EN 50129).

Task is the smallest units of work subject to management accountability.

UML Profile is a collection of extensions to customize UML (and SysML) for a specific purpose.

View a representation of a particular system that conforms to a viewpoint [11], Sec. 1.1.2.

Viewpoint a reusable set of criteria for the construction, selection, and presentation of a portion of the information about a system, addressing particular stakeholder concerns [11], Sec. 1.1.2.

Bibliography

[1] D. C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2 (Feb. 2006), pp. 25–31.

[2] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. "Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain". In: *Proc. of the 7th International Conference on Model-Driven Engineering Languages and Systems (MODELS)*. Ed. by J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran. Springer International Publishing, 2014, pp. 166–182.

[3] N. Marko, G. Liebel, D. Sauter, A. Lodwich, M. Tichy, A. Leitner, and J. Hansson. *Model-Based Engineering for Embedded Systems in Practice, Research Reports in Software Engineering and Management*. Tech. rep. University of Gothenburg, 2014.

[4] M. Broy, S. Kirstan, H. Krcmar, B. Schätz, and J. Zimmermann. "What is the benefit of a model-based design of embedded software systems in the car industry?" In: *Emerging Technologies for the Evolution and Maintenance of Software Models*. IGI Global, 2013, pp. 343–369.

[5] P. Baker, S. Loh, and F. Weil. "Model-Driven Engineering in a Large Industrial Context — Motorola Case Study". In: *Proc. of the 8th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ed. by L. Briand and C. Williams. Springer Berlin Heidelberg, 2005, pp. 476–491.

[6] T. Weigert and F. Weil. "Practical experiences in using model-driven engineering to develop trustworthy computing systems". In: *Proc. of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*. Vol. 1. IEEE, 2006, pp. 208–215.

[7] A. Ferrari, A. Fantechi, and S. Gnesi. "Lessons Learnt from the Adoption of Formal Model-Based Development". In: *Proc. of 4th International Symposium on the NASA Formal Methods (NFM)*. Ed. by A. E. Goodloe and S. Person. Springer Berlin Heidelberg, 2012, pp. 24–38.

[8] P. Mohagheghi and V. Dehlen. "Where Is the Proof? - A Review of Experiences from Applying MDE in Industry". In: *Proc. of 4th European Conference on the Model Driven Architecture – Foundations and Applications (ECMDA-FA)*. Ed. by I. Schieferdecker and A. Hartman. Vol. 5095. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 432–443.

[9] EU Project CECRIS, CErtification of CRItical Systems. http://www.cecris-project-.eu, visited on 2016-03.

[10] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. 1st. Morgan & Claypool Publishers, 2012.

[11] Object Management Group (OMG). *MDA Guide*. http://www.omg.org/cgi-bin/doc?-ormsc/14-06-01, visited on 2016-03. Version 2.0. 2014.

[12] S. Kent. "Model Driven Engineering". In: *Proc. of the Third International Conference on Integrated Formal Methods (IFM)*. Springer-Verlag, 2002, pp. 286–298.

[13] Object Management Group (OMG). *MDA Guide*. http://www.omg.org/cgi-bin/doc?omg/-03-06-01, visited on 2016-03. Version 1.0.1. 2003.

[14] R. Soley et al. "Model driven architecture". In: *OMG white paper* (2000).

[15] Object Management Group (OMG). *Systems Modeling Language (SysML)*. http://-www.omg.org/docs/formal/08-11-02.pdf, visited on 2016-03. Version 1.1. 2008.

[16] P. Baker, Z. R. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, and C. Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer-Verlag New York, Inc., 2007.

[17] Z. R. Dai. "Model-driven testing with UML 2.0". In: *Proc. of the 2nd European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA)*. Ed. by D. Akehurst. Tech. rep. 17-04. University of Kent. 2004, pp. 179–187.

[18] I. Schieferdecker. "The UML 2.0 Test Profile as a Basis for Integrated System and Test Development". In: *Proc. of Köllen Druck+Verlag GmbH, Jahrestagung der Gesellschaft für Informatik*. Vol. 35. 2005, pp. 395–399.

[19] I. Davies, P. Green, M. Rosemann, M. Indulska, and S. Gallo. "How do practitioners use conceptual modeling in practice?" In: *Data & Knowledge Engineering* 58.3 (2006), pp. 358–380.

[20] A. Forward and T. C. Lethbridge. "Problems and Opportunities for Model-centric Versus Code-centric Software Development: A Survey of Software Professionals". In: *Proc. of the International Workshop on Models in Software Engineering (MISE)*. ACM, 2008, pp. 27–32.

[21] J. Hutchinson, M. Rouncefield, and J. Whittle. "Model-driven engineering practices in industry". In: *Proc. of the 33rd International Conference on Software Engineering (ICSE)*. 2011, pp. 633–642.

[22] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. "Empirical Assessment of MDE in Industry". In: *Proc. of the 33rd International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 471–480.

[23] J. Hutchinson, J. Whittle, and M. Rouncefield. "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure". In: *Science of Computer Programming* 89, Part B (2014), pp. 144–161.

[24] J. Whittle, J. Hutchinson, and M. Rouncefield. "The State of Practice in Model-Driven Engineering". In: *IEEE Software* 31.3 (2014), pp. 79–85.

[25] F. Tomassetti, M. Torchiano, A. Tiso, F. Ricca, and G. Reggio. "Maturity of software modelling and model driven engineering: A survey in the Italian industry". In: *Proc. of the 16th International Conference on Evaluation Assessment in Software Engineering (EASE)*. 2012, pp. 91–100.

[26] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio. "Preliminary Findings from a Survey on the MD State of the Practice". In: *Proc. of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2011, pp. 372–375.

[27] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio. "Benefits from Modelling and MDD Adoption: Expectations and Achievements". In: *Proc. of the 2nd International Workshop on Experiences and Empirical Studies in Software Modelling (EESSMod)*. ACM, 2012, pp. 1–6.

[28] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio. "Relevance, benefits, and problems of software modelling and model driven techniques – A survey in the Italian industry". In: *Journal of Systems and Software* 86.8 (2013), pp. 2110–2126.

[29] M. Petre. "UML in Practice". In: *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, pp. 722–731.

[30] L. T. W. Agner, I. W. Soares, P. C. Stadzisz, and J. M. Simão. "A Brazilian survey on UML and model-driven practices for embedded software development". In: *Journal of Systems and Software* 86.4 (2013), pp. 997–1005.

[31] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. C. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill, J. Kienzle, M. Schöttle, F. Steimann, D. Stikkolorum, and J. Whittle. "The Relevance of Model-Driven Engineering Thirty Years from Now". In: *Proc. of the 17th International Conference on Model-Driven Engineering Languages and Systems (MODELS)*. Ed. by J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran. Springer International Publishing, 2014, pp. 183–200.

[32] M. Huhn and H. Hungar. "8 UML for Software Safety and Certification". In: *Model-Based Engineering of Embedded Real-Time Systems: International Dagstuhl Workshop. Revised Selected Papers*. Ed. by H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz. Springer Berlin Heidelberg, 2010, pp. 201–237.

[33] R. Pettit, N. Mezcciani, and J. Fant. "On the needs and challenges of model-based engineering for spaceflight software systems". In: *Proc. of the IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. 2014, pp. 25–31.

[34] B. Kitchenham. *Guidelines for performing Systematic Literature Reviews in Software Engineering, Version 2.3*. Tech. rep. EBSE-2007-01. Keele University and University of Durham, 2007.

[35] Elsevier. *Scopus*. http://www.scopus.com, visited on 2016-03.

[36] SAE International. *SAE Homepage*. http://www.sae.org, visited on 2016-03.

[37] Inderscience Enterprises. *Inderscience Online*. http://www.inderscienceonline.com, visited on 2016-03.

[38] AHS International. *AHS Homepage*. https://vtol.org/store/index.cfm, visited on 2016-03.

[39] J. Hugues, M. Perrotin, and T. Tsiodras. "Using MDE for the rapid prototyping of space critical systems". In: *Proc. of the 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP) - Shortening the Path from Specification to Prototype*. 2008, pp. 10–16.

[40] A. Baumgart, P. Reinkemeier, A. Rettberg, I. Stierand, E. Thaden, and R. Weber. "A model-based design methodology with contracts to enhance the development process of safety-critical systems". In: *Proc. of the 8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*. Vol. 6399. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 59–70.

[41] P. Sanchez, J. Barreda, and J. Ocon. "Integration of domain-specific models into a MDA framework for time-critical embedded systems". In: *Proc. of the International Workshop on Intelligent Solutions in Embedded Systems*. IEEE, 2008, pp. 1–15.

[42] L. Burgareli, S. Melnikoff, and M. Ferreira. "A software modeling approach based on MDA for the Brazilian Satellite Launcher". In: *Proc. of the SpaceOps Conference*. IEEE Computer Society, 2008, pp. 627–632.

[43] H. Wang, C. Gao, and S. Liu. "Model-based software development for automatic train protection system". In: *Proc. of the 2nd Asia-Pacific Conference on Computational Intelligence and Industrial Applications (PACIIA)*. Vol. 1. IEEE, 2009, pp. 463–466.

[44] I. Djordjevic, C. Gan, E. Scharf, R. Mondragon, B. Gran, M. Kristiansen, T. Dimitrakos, K. Stølen, and T. Opperud. "Model based risk management of security critical systems". In: *Proc. of the Risk Analysis III*. Vol. 5. Management Information Systems. WIT Press, 2002, pp. 253–264.

[45] J. Delange, L. Pautet, J. Hugues, and D. De Niz. "An MDE-based process for the design, implementation and validation of safety-critical systems". In: *Proc. of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2010, pp. 319–324.

[46] G. Macher, M. Stolz, E. Armengaud, and C. Kreiner. "Filling the gap between automotive systems, safety, and software engineering". In: *Elektrotechnik und Informationstechnik* 132.3 (2015), pp. 142–148.

[47] H. Sporer, G. Macher, E. Armengaud, and C. Kreiner. "Incorporation of Model-Based System and Software Development Environments". In: *Proc. of the 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2015, pp. 177–180.

[48] M. Bordin, T. Tsiodras, and M. Perrotin. “Experience in the Integration of Heterogeneous Models in the Model-driven Engineering of High-Integrity Systems”. In: *Proc. of the 13th Ada-Europe International Conference on Reliable Software Technologies. Proceedings*. Ed. by F. Kordon and T. Vardanega. Springer Berlin Heidelberg, 2008, pp. 171–184.

[49] Z. Wang, A. Herkersdorf, S. Merenda, and M. Tautschnig. “A model driven development approach for implementing reactive systems in hardware”. In: *Proc. of the Forum on Specification, Verification and Design Languages (FDL)*. IEEE, 2008, pp. 197–202.

[50] I. Gray, N. Matragkas, N. Audsley, L. Indrusiak, D. Kolovos, and R. Paige. “Model-based hardware generation and programming - The MADES approach”. In: *Proc. of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*. IEEE, 2011, pp. 88–96.

[51] A. Prakash, I. Schieferdecker, M. Wagner, and C. Hein. “Rotary dial model - A model-driven methodology for autonomic network design”. In: *Proc. of the IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2012, pp. 891–896.

[52] S. Biffl, R. Mordinyi, and A. Schatten. “A model-driven architecture approach using explicit stakeholder quality requirement models for building dependable information systems”. In: *Proc. of the ICSE 2007 Workshops: 5th International Workshop on Software Quality (WoSQ)*. IEEE, 2007.

[53] R. Jeffords, C. Heitmeyer, M. Archer, and E. Leonard. “Model-based construction and verification of critical systems using composition and partial refinement”. In: *Formal Methods in System Design* 37.2-3 (2010), pp. 265–294.

[54] F. Flammini, S. Marrone, N. Mazzocca, R. Nardone, and V. Vittorini. “Model-Driven V&V Processes for Computer Based Control Systems: A Unifying Perspective”. In: *Proc. of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies (ISoLA)*. Ed. by T. Margaria and B. Steffen. Springer Berlin Heidelberg, 2012, pp. 190–204.

[55] N. Rungta, O. Tkachuk, S. Person, J. Biatek, M. Whalen, J. Castle, and K. Gundy-Burlet. “Helping system engineers bridge the peaks”. In: *Proc. of the 4th International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)*. ACM, 2014, pp. 9–13.

[56] D. Hardin, T. Hiratzka, D. Johnson, L. Wagner, and M. Whalen. “Development of security software: A high assurance methodology”. In: *Formal Methods and Software Engineering. Proc. of the 11th International Conference on Formal Engineering Methods (ICFEM)*. Vol. 5885. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 266–285.

[57] D. Méry and N. K. Singh. “A generic framework: from modeling to code”. In: *Innovations in Systems and Software Engineering* 7.4 (2011), pp. 227–235.

[58] P. Arcaini, A. Gargantini, and E. Riccobene. "Rigorous development process of a safety-critical system: from ASM models to Java code". In: *International Journal on Software Tools for Technology Transfer* (2015), pp. 1–23.

[59] G. Pintér, I. Majzik, and Z. Micskei. "Supporting Design and Development of Safety Critical Applications". In: *Proc. of the 10th Symposium on Programming Languages and Tools, (SPLST)*. Ed. by Z. Horváth, L. Kozma, and V. Zsók. Eotvos University Press, 2007, pp. 61–75.

[60] G. Carrozza, M. Faella, F. Fucci, R. Pietrantuono, and S. Russo. "Integrating MDT in an industrial process in the Air Traffic Control domain". In: *Proc. of the 23rd IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2012, pp. 225–230.

[61] M. Borek, N. Moebius, K. Stenzel, and W. Reif. "Model-driven development of secure service applications". In: *Proc. of the IEEE 35th Software Engineering Workshop (SEW)*. IEEE, 2012, pp. 62–71.

[62] E. Grant and T. Datta. "Roadmap to a DO-178C formal model-based software engineering methodology". In: *Proc. of the International MultiConference of Engineers and Computer Scientists (IMECS)*. Vol. 1. IAENG, 2015, pp. 460–465.

[63] A. Ferrari, M. Papini, A. Fantechi, and D. Grasso. "An industrial application of formal model based development: The metrô rio ATP case". In: *Proc. of the 2nd International Workshop on Software Engineering for Resilient Systems (SERENE)*. ACM, 2010, pp. 71–76.

[64] A Garro, A Tundis, L Rogovchenko-Buffoni, and P Fritzson. "From Safety Requirements to Simulation-driven Design of Safe Systems". In: *Proc. of the 12th International Conference on Modeling and Applied Simulation (MAS)*. Curran Associates, 2013, pp. 40–49.

[65] Object Management Group (OMG). *Semantics of a Foundational Subset for Executable UML Models (fUML)*. http://www.omg.org/cgi-bin/doc?formal/2013-08-06.pdf, visited on 2016-03. Version 1.1. 2013.

[66] Object Management Group (OMG). *Concrete Syntax for a UML Action Language: Action Language for Foundational UML (ALF)*. http://www.omg.org/spec/ALF, visited on 2016-03.

[67] F. P. J. Brooks. "No Silver Bullet Essence and Accidents of Software Engineering". In: *Computer* 20.4 (1987), pp. 10–19.

[68] G. Carrozza, M. Faella, F. Fucci, R. Pietrantuono, and S. Russo. "Engineering Air Traffic Control Systems with a Model-Driven Approach". In: *IEEE Software* 30.3 (2013), pp. 42–48.

[69] Eclipse Foundation. *Eclipse*. http://www.eclipse.org, visited on 2016-03.

[70] Commissariat à l'Énergie Atomique, Atos, Cedric Dumoulin. *Papyrus*. http://www.eclipse.org/papyrus, visited on 2016-03.

[71] Wielemaker, Jan and alia. *SWI-Prolog*. http://www.swi-prolog.org, visited on 2016-03.

[72] M. Hecht, E. Dimpfl, and J. Pinchak. "Automated Generation of Failure Modes and Effects Analysis from SysML Models". In: *Proc. on the 2014 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2014, pp. 62–65.

[73] P. David, V. Idasiak, and F. Kratz. "Reliability study of complex physical systems using SysML". In: *Reliability Engineering & System Safety* 95.4 (2010), pp. 431–450.

[74] J. Xiang, K. Yanoo, Y. Maeno, and K. Tadano. "Automatic synthesis of static fault trees from system models". In: *Proc. of the 5th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*. IEEE, 2011, pp. 127–136.

[75] G. Sindre and A. L. Opdahl. "Eliciting security requirements with misuse cases". In: *Requirements Engineering* 10.1 (2004), pp. 34–44.

[76] G. Sindre. "A look at misuse cases for safety concerns". In: *Situational Method Engineering: Fundamentals and Experiences*. Springer, 2007, pp. 252–266.

[77] T. Stålhane and G. Sindre. "A Comparison of Two Approaches to Safety Analysis Based on Use Cases". In: *Conceptual Modeling (ER)*. Ed. by C. Parent, K.-D. Schewe, V. Storey, and B. Thalheim. Vol. 4801. Lecture Notes in Computer Science. Springer, 2007, pp. 423–437.

[78] K. Allenby and T. Kelly. "Deriving safety requirements using scenarios". In: *Proc. of the 5th IEEE Int. Symp. on Requirements Engineering*. IEEE, 2001, pp. 228–235.

[79] C. Picardi, L. Console, F. Berger, J. Breeman, T. Kanakis, J. Moelands, S. Collas, E. Arbaretier, N. De Domenico, E. Girardelli, O. Dressler, P. Struss, and B. Zilbermann. "AUTAS: a tool for supporting FMECA generation in aeronautic systems". In: *Proc. of the 16th European Conference on Artificial Intelligence (ECAI)*. IOS Press, 2004, pp. 750–754.

[80] M. Molhanec. "Model based FMEA method for solar modules". In: *Proc. of the 36th International Spring Seminar on the Electronics Technology (ISSE)*. IEEE. 2013, pp. 183–188.

[81] Y. Kitamura, N. Washio, Y. Koji, M. Sasajima, S. Takafuji, and R. Mizoguchi. "An ontology-based annotation framework for representing the functionality of engineering devices". In: *Proc. of the ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers. 2006, pp. 125–134.

[82] V. Ebrahimipour, K. Rezaie, and S. Shokravi. "An ontology approach to support FMEA studies". In: *Expert Systems with Applications* 37.1 (2010), pp. 671–677.

[83] P.-J. Gailly, W. Krautter, C. Bisière, and S. Bescos. "The Prince project and its applications". English. In: *Logic Programming in Action*. Vol. 636. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 54–63.

[84] P. Schmidt. "An ontology-based annotation framework for representing the functionality of engineering devices". In: *Proc. of the Workshop on Spacecraft Flight Software (FSW)*. ASME, 2012, pp. 125–134.

[85] No Magic, Inc., Magic Draw. *MagicDraw*. http://www.nomagic.com/products/magicdraw.html, visited on 2016-03.

[86] IBM Corp. *Rational Rhapsody Developer*. http://www-03.ibm.com/software/products/-it/ratirhap, visited on 2016-03.

[87] Conformiq Inc. *Conformiq Designer*. http://www.conformiq.com/products/conformiq-designer, visited on 2016-03.

[88] IBM Corp. *Rational Rhapsody TestConductor Add On, User Guide*. http://pic.dhe-.ibm.com/infocenter/rhaphlp/v7r6/topic/com.ibm.rhp.oem.pdf.doc/pdf/RTC_User_Guide-.pdf, visited on 2016-03.

[89] IBM Corp. *Rational Rhapsody Automatic Test Generator Add On, User Guide*. http://pic-.dhe.ibm.com/infocenter/rhaphlp/v7r5/topic/com.ibm.rhapsody.oem.pdf.doc/pdf/ATG_-User_Guide.pdf, visited on 2016-03.

[90] M. Staron. "Adopting Model Driven Software Development in Industry – A Case Study at Two Companies". English. In: *Model Driven Engineering Languages and Systems*. Vol. 4199. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 57–72.

[91] Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environment (EMC^2). http://www.artemis-emc2.eu, visited on 2016-03.

[92] R. Hebig and R. Bendraou. "On the Need to Study the Impact of Model Driven Engineering on Software Processes". In: *Proc. of the International Conference on Software and System Process (ICSSP)*. ACM, 2014, pp. 164–168.

[93] R. Hebig, R. Bendraou, M. Völter, and M. Chaudron. "Model-Driven Development Processes and Practices: Foundations and Research Perspectives". In: *Proc. of the 1st International Workshop on Model-Driven Development Processes and Practices*. CEUR, 2014, pp. 2–6.

[94] European Railway Agency (ERA). *Glossary of railway terms*. http://www.era.europa-.eu/document-register/pages/glossary-of-railway-terms.aspx, visited on 2016-03. 2010.