Supporting Pattern-Based Dependability Engineering via Model-Driven Development: Approach, tool-support and empirical validation

Brahim Hamid, Jon Perez

Supporting pattern-based dependability engineering via model-driven development: Approach, tool-support and empirical validation

Brahim Hamid [a,*], Jon Perez [b]

[a] IRIT, University of Toulouse, 118 Route de Narbonne, 31062 Toulouse Cedex 9, France
[b] IKERLAN-IK4 Research Centre, Mondragon, Spain

Keywords:
Dependability
Safety
System engineering
Patterns
Meta-modeling
Model driven engineering

A B S T R A C T

Safety-critical systems require a high level of safety and integrity. Therefore, generating such systems involves specific software building processes. Many domains are not traditionally involved in these types of software problems and must adapt their current processes accordingly. Typically, such requirements are developed ad hoc for each system, preventing further reuse beyond the domain-specific boundaries. This paper proposes a solution for software system development based on the reuse of dedicated subsystems, i.e., so-called *dependability patterns* that have been pre-engineered to adapt to a specific domain. We use Model-Driven Engineering (MDE) to describe dependability patterns and a methodology for developing dependable software systems using these patterns. Moreover, we describe an operational architecture for development tools to support the approach. An empirical evaluation of the proposed approach is presented through its practical application to a case study in the railway domain, which has strong dependability requirements, to support a pattern-based development approach. This case study is followed by a survey to better understand the perceptions of practitioners regarding our approach.

1. Introduction

Safety-critical systems require a high level of safety and integrity. Therefore, the generation of such systems involves specific software building processes. These processes are often error-prone because they are not fully auto- mated, even if some level of automatic code generation or model-driven engineering support is applied. Furthermore, many critical systems also have assurance requirements, ranging from very strong levels involving certification (e.g., EN-50,129 (CENELEC, 1999) for railway systems and DO-178B (RTCA, 1992) for airborne systems) to reduced levels based on industry practices. These systems can be found in many application sectors such as automotive, aerospace, and home control, and come with many common characteristics, including real-time and temperature constraints, computational processing, power constraints and/or limited energy and common extra-functional properties such as dependability, security and efficiency (Ravi et al., 2004; Kopetz, 2011).

The integration of various concerns, such as dependability, requires the availability of both application development and expertise. Many domains not traditionally involved in this type of software development and must adapt their current processes accordingly. Typically, such requirements are developed ad hoc for each system, preventing further reuse beyond the domain-specific boundaries. This is especially true for railway systems, as they exist in many use cases. Many of these systems belong to critical infrastructures, where other economic and social aspects are based on. Hence capturing and providing this expertise via *dependability patterns* (Daniels and Vouks, 1997; Powel, 2003; Tichy et al., 2004; Radermacher et al., 2013) has become recently an area of research. Dependability patterns enable the development of dependable applications and liberate the developer from having to address technical details. We believe that the specification and packaging of dependability patterns can provide an efficient means of addressing these problems, improving industrial efficiency and fostering technology *reuse* across domains (the reuse of models at different levels), thus reducing the time and effort required to design a complex system (McClure, 1997; Agresti, 2011). Model-driven engineering (MDE) (Selic, 2003; Atkinson and Ku¨hne, 2003) also provides a very useful contribution to the design of safety-critical systems (Ziani et al., 2012; Panesar-Walawege et al., 2013) because it reduces the time/cost required for understanding and analyzing system artifact descriptions due to the abstraction mechanisms. Moreover, it reduces the cost of the development process thanks to the generation mechanisms. Hence, dependability pattern integration must be considered during the MDE process.

* Corresponding author. Fax: +33 5 6150 4173.
E-mail addresses: hamid@irit.fr (B. Hamid), JPerez@ikerlan.es (J. Perez).

In system and software engineering, design patterns (Gamma et al., 1995; Henninger et al., 2007) are considered effective tools for the reuse of specific information. They are widely used today to provide architects and designers with reusable design knowledge. They are triples that describe *solutions* for commonly occurring problems in specific contexts. Indeed, pattern-based development has recently gained more attention in software engineering by addressing new challenges that had not been targeted in the past (Henninger et al., 2007). In fact, they are applied in modern software architecture for distributed systems, including middleware and real-time embedded systems (Schmidt and Buschmann, 2003). There are patterns for generic architecture problems (Buschmann et al., 1996, 2007), security (Schumacher, 2003; Fernandez, 2013), safety (Alexander et al., 2007; Preschern et al., 2013) and other non-functional requirements (Powel, 2003). The related approaches promote the use of patterns via reusable design artifacts. However, a gap between the development of systems using patterns and the information in the pattern representations remains (Zdun and Avgeriou, 2008). This becomes even more observable when addressing specific concerns, such as dependability.

In this paper, we present a model-based approach for dependability system and software engineering that uses *patterns* to represent dependability solutions and knowledge, which fosters reuse. In such a vision, the dependability patterns derived from (resp. associated with) domain-specific models are designed to assist the application developer integrate application models with dependability building-block solutions. Dependability patterns are defined from a platform-independent perspective (i.e., they are independent of the implementation) and are expressed in a consistent manner with domain-specific dependability models. Consequently, they will be much easier to understand and validate by application designers in a specific area. This work is conducted within the context of a model-based security and dependability research project, and our collaboration with safety-critical system suppliers suggested a need for this work. The dependability solutions used by safety-critical system developers are based on the application domain and occasionally on the software development environment, including the design and coding stages. There is a need to link these concepts to dependability, which will ease the certification process. The lack of appropriate links between application domain concepts and dependability concepts poses three main challenges. First, the dependability engineer must re-engineer its existing solutions. Second, the application designer may not understand domain-specific dependability solutions. Finally, it is very difficult for the system developer to guarantee the availability of dependability solutions to cover all the dependability requirements of the targeted application using the application domain concepts.

To provide a concrete example, we introduce a railway case study from the TERESA project[1] called Safe4Rail, which is a simplified version of a real ETCS (European Train Control System) (UNISIG, 2009; Stanley, 2011). The main functionality of this demonstrator is to ensure that the traveled speed and distance do not exceed the authorized maximum values provided by the railway infrastructure. To implement this functionality, the system is composed of multiple subsystems, including the European Vital Computer (EVC), which executes the safety application, and a set of odometry sensors and actuators. The odometry sensors provide the speed and acceleration of the train. With these values, the system must be able to calculate accurate speed and position values (odometry). There is a family of products in the railway sector, including regional trains, tramways, and high-speed trains. These units share common parts, although they differ in distinct ways. For example, consider the calculation of the actual speed and position by Safe4Rail. The implementation varies according to each type of train product, which depends on the safety level to be met and the type and number of sensors and actuators that are involved. These considerations greatly influence how each product is implemented because several issues should be considered: the number of channel redundancies, the diversity of the channels, the monitoring of the channels, and the interaction with assorted data (type, weight, etc.).

A proprietary embedded system has been designed to meet stricter safety regulations. In our case, the SIL4 level is pursued. To achieve this level, several design techniques from related standards, such as IEC-61,508 (IEC, 2010b) and EN-50,126 (CENELEC, 1999), are used, including redundancies, votations, diagnostics, and secure and safe communications. Hence, capturing and providing this expertise by means of a repository of dependability patterns and models can enhance the development of embedded systems. We seek mechanisms that allow a safer, easier and faster safety-critical development process. To illustrate this statement, two different railway industry scenarios are described. The first takes place within a railway manufacturing group, whereas second occurs within an SME.

Scenario 1. *Railway manufacturing group.* The first scenario takes place within a group of railway manufacturers. The group is divided into subsidiary companies that specialize in the development of train and railway infrastructure systems (e.g., traction, central control, infotainment, railway signaling, interlocking, Communication Based Train Control (CBTC) (IEEE, 2004), etc.). The group intends to develop its own safety-related subsystems instead of buying them from providers and competitors. The problem is that the engineering cost associated with these safety-related systems is relatively high; moreover, the number of qualified engineers is limited. Additionally, safety concepts and design patterns are not present in commonly used modeling languages (e.g., UML, SysML, etc.), which leads to design ambiguity. The expected benefits for this scenario are the following: (1) reduce product development cost and time by reusing design patterns for other projects and companies, and (2) reduce the probability of systematic faults by reducing ambiguity.

Scenario 2. *SME company.* In the second scenario, an SME that develops safety-related embedded systems is presented. This company has proven experience in the development of IEC-61,508-based safety-related embedded systems, although the company has little experience in the railway domain. If this company wants to enter the railway market, it must overcome a great barrier and analyze relevant railway standards, adapt the solutions it has been using for years in other sectors, and be ready for success. Some of the problems are the same as those that arise in the first scenario: the development cost for safety-related systems is high, and the number of qualified engineers is limited. Additionally, in this context, systems are developed with different standards derived from IEC-61,508 (the differences are well known). The expected benefits for this scenario are the following: (1) reduce product development cost and time, with the cross-domain reuse of design patterns between projects; (2) reduce the probability of systematic fault by reducing ambiguity; and (3) provide a cross-domain arsenal of design patterns that can be used in new domains.

To address issues related to scenario 1, an infrastructure that allows the reuse of the most common techniques used to achieve the dependability requirements in most subsystems would allow these subsidiaries to reduce efforts, and costs while also reducing the number of required and scarce safety experts. With regard to the issues related to scenario 2, we offer an infrastructure that allows the cross-domain reutilization of the techniques used to achieve the target safety level.

[1] http://www.teresa-project.org/.

1.1. Solution overview

To address the above problem, we propose an approach that combines model-driven technology and pattern-based development to address the design of dependable applications. Our proposed approach makes use of patterns as is primary technique: *Patten Based System Engineering (PBSE)*. PBSE focuses on patterns; from this perspective, it addresses two types of processes: *pattern definition* and *system development with patterns*. Metamodeling techniques are used to represent patterns at a greater level of abstraction. Therefore, patterns can be stored in a repository and can be loaded for desired properties. As a result, patterns can be used as bricks to build applications through a model-driven engineering approach. The associated framework promotes an infrastructure for the modeling of dependability patterns and provides specific transformation engines that can adapt and generate different representations, where patterns are clearly related to domain models. Supported by an MDE tool suite, PBSE assists the dependable system engineering process. The resulting tool-chain supports the two categories of users: "reuse" consumers and "reuse" producers. The former category comprises developers who reuse existing artifacts from the repository, whereas the latter comprises developers of artifacts to be stored in the repository. Such an MDE tool suite utilizes Domain-Specific Modeling Languages (DSMLs) (Gray et al., 2007; Strembeck and Zdun, 2009) built on an integrated repository of modeling artifacts that function as a group, where a pattern is at the heart of development – its role should be specified in all lifecycle stages of development. We use the open-source Eclipse Modeling Framework (EMF) (Steinberg et al., 2009) and its extended version, Eclipse Modeling Framework Technology (EMFT)[2] to build the support tools for our approach. The EMF provides an implementation of the EMOF (Essential MOF), which is a subset of the Meta-Object Facility (MOF) (OMG, 2008) called Ecore[3]. The EMF offers a set of tools to specify metamodels in Ecore and to generate other representations.

1.2. Intended contributions

In order to empirically assess the proposed approach and tooling, we provide evidence of its benefits and applicability through an example of a representative industrial case from the TERESA project, i.e., the Safe4Rail application. The case study indicates that our approach is feasible in a real industrial context and that it can provide useful guidance in reusing existing solutions. This case study is followed by a survey to better understand the perceptions of practitioners regarding our approach. The survey indicates that practitioners agree on the benefits of adopting our approach in a real industrial context. The new approach permits faster production and reduces staff using proven solutions instead of hiring expensive dependability engineers. Because the pattern conceptual model can be applied to multiple problems through a simple extension, these results suggest our work has wider applicability and usefulness. In summary, the work presented in this paper has the following features: (1) a dependability pattern-based approach as a new method for software system engineering based on the reuse of patterns; (2) the design of a set of DSMLs to specify the patterns that is independent of the end-user development applications and execution platforms; (3) the development of a set of tools to support the proposed approach; (4) the application of the approach in the context of railway systems; and (5) a survey to better understand the perceptions of industry practitioners regarding our approach.

[2] https://eclipse.org/modeling/emft/.
[3] Ecore is a meta-meta-model.

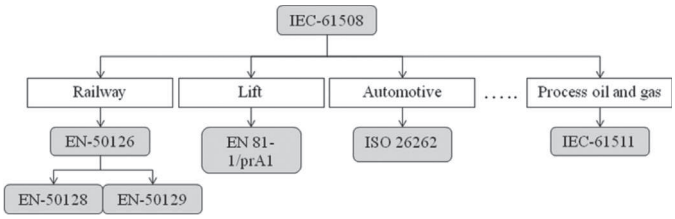

Fig. 1. Domain specific standards derived from IEC 61,508.

The basic formulation of the approach presented in this article has been previously published in a research paper at the 16th International System Design Languages Forum on Model-Driven Dependability Engineering (Hamid et al., 2013). This work extends the ideas described in the earlier paper and presents a holistic approach to the modeling of pattern-based software systems with strong dependability requirements. Specifically, we provide a more comprehensive and complete description of our approach (Section 3) and tool support (Section 4), along with substantial new empirical results to show the feasibility and usefulness of our approach (Section 5).

1.3. Outline

The remainder of the paper is organized as follows. An overview of the modeling framework, including the development context, is presented in Section 2. In section 3, we present our approach to support the pattern-based system and software dependability engineering. Section 4 describes the architecture of the tool suite and presents an example implementation. In Section 5, we present an empirical evaluation of our approach through the TERESA railway case study and a survey of key performance indicators. In Section 6, we discuss the contribution with regard to certification. Related work is discussed in Section 7. Finally, Section 8 concludes and sketches future work directions. We investigate a few open issues, primarily the issues of generalization and implementation, including the usability of the proposed modeling framework.

2. Background

The engineering of embedded systems with safety requirements typically requires the certification of such products according to generic (e.g., IEC-61,508 (IEC, 2010b)) or domain-specific standards (e.g., EN-5012X (CENELEC, 1999)). System engineers must develop a system that complies with required standards, implementing recommended techniques and measures. The cost-effective development and certification of such products is a challenge, and the reusability of proven solutions (design patterns) enables cost and time-to-market reductions. Our work aims at providing a new engineering approach that allows these solutions (techniques and measures) to be reused for development within the same application domain or in a cross-domain scenario.

Many domain-specific standards are derived from IEC-61,508, as shown in Fig. 1. The main characteristic of these standards is that they all share several features (i.e., QM, V&V, life cycle, techniques and measures, etc.) of the main standard, IEC-61,508, whereas other specific parts are solely related to each application domain itself. These standards provide a specific set of techniques and measures that must be implemented to achieve the desired level of safety integrity of an application (either to avoid systematic faults or to control random faults). To foster reuse, developers must adapt the requirements and design solutions of these standards to the concepts of the application domain. As described below, a subset of techniques and measures proposed in the stan-

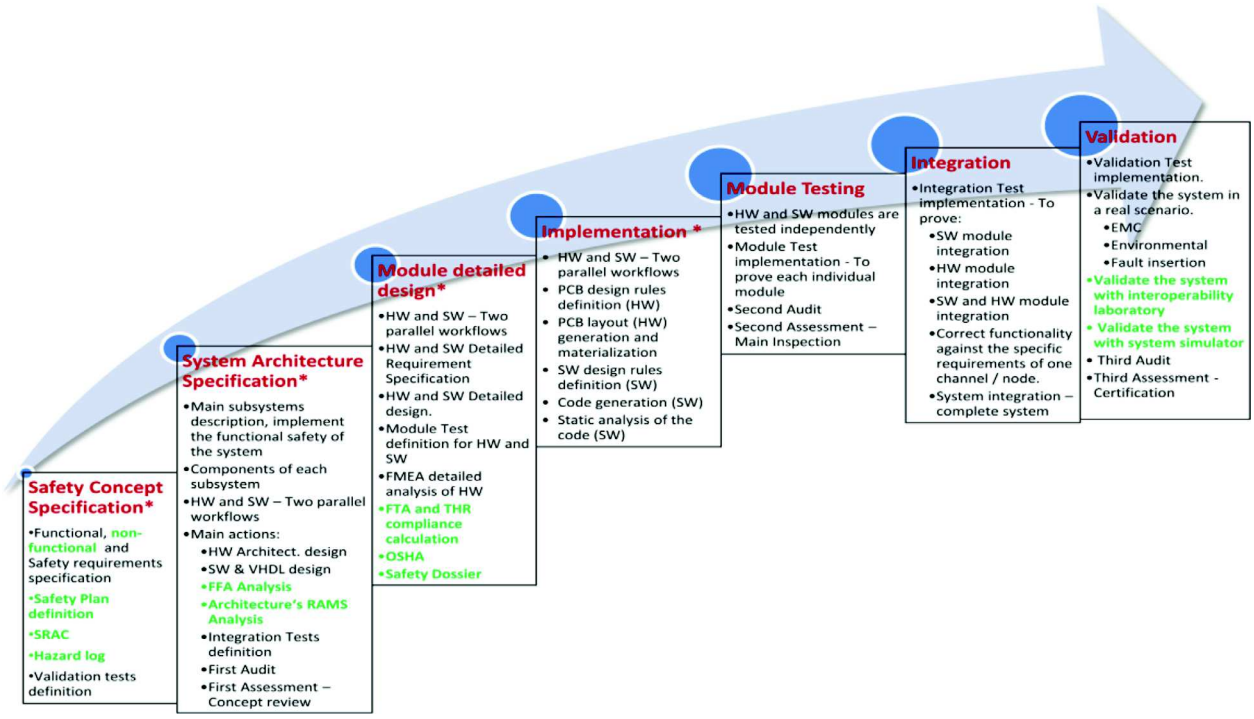

Fig. 2. Reference safety engineering process for railway signaling.

dards can be used to define several dependability patterns used in the demonstrator (Section 5).

Furthermore, most of these techniques and measures are provided in the form of design decisions targeting one stage of the development lifecycle without effective realization. In addition, the format in which the techniques and measures are presented must be improved to ensure that the provided solutions are storable, reusable and appropriate for the automation of software development and analysis. An ontological approach for describing design patterns and their relationships was proposed in (Girardi and Lindoso, 2006) to facilitate understanding and reuse during software development.

Finally, development processes for system and software construction are common knowledge and mainstream practice in most development organizations. Fig. 2 shows major activities/deliverables associated with a reference railway safety engineering process, where items in green are new or different compared with the IEC-61,508 standard. Every phase receives input documents and defines a set of activities to perform and a set of output documents to generate. The activities are assigned with roles and may require specialized tools. The items in green are either not relevant (e.g., validate the system) or not considered within the scope of our case study (e.g., hazard log). In addition, there is a high level of compatibility between the IEC-61,508-3 and EN-50,128 software safety standards, especially regarding their software lifecycles and safety techniques. The engineering of embedded systems with safety has been well established (IEC, 2010b), although methods and tools to support it are lacking.

Therefore, there are two main prerequisites to define the pattern-based dependability engineering methodology. The first is that it must be compatible with current development processes. The objective is not to change the habits of the engineers; instead, easing the acceptance of the approach in industry is the goal. The second prerequisite is that it must be flexible enough to adapt to other specific processes in other domains.

We seek a solution based on the reuse of software subsystems that have been pre-engineered to adapt to a specific domain. The approach described in Section 3 uses MDE techniques to handle the issues described above. Fig. 3 shows the overall life-cycle for a safety related system, from concept to disposal. This figure has been extracted from the standard IEC-61,508. Compared to existing methodologies from relevant standards, our work enhances the provided methodology only during the *realization phase*, although it may have a positive impact in subsequent phases of the overall product lifecycle, i.e., from concept to disposal.

3. Approach

A system architect must work at different levels. Integrating all subsystems while considering the associated dependability requirements in a seamless fashion is challenging given the various critical requirements and uncertainties. We propose a solution for software system development based on the reuse of dedicated subsystems, so-called dependability patterns that have been pre-engineered to adapt to a specific domain. The patterns that are at the heart of our system engineering process reflect design solutions at the domain-independent and domain-specific levels. We use Model-Driven Engineering (MDE) to describe dependability patterns and a methodology for developing dependable software systems using these patterns. The resultant modeling framework reduces the time/cost related to understanding and analyzing system artifact description due to the abstraction mechanisms, and it reduces the cost of the development process due to the generation mechanisms.

The proposed approach (as illustrated in Fig. 4) is composed of six main steps (the numbers in parentheses correspond to those in Fig. 4). The first step (step 1) is responsible for the creation of the conceptual model of dependability patterns. The resulting conceptual model is used to build a DSML to specify dependability patterns (step 2). The dependability expert with the help of the system and software engineering expert uses this DSML to define dependability patterns (step 3). Then, a domain process expert adapts the dependability patterns into a version that is suitable in its system development process (step 4). An example is adaptation

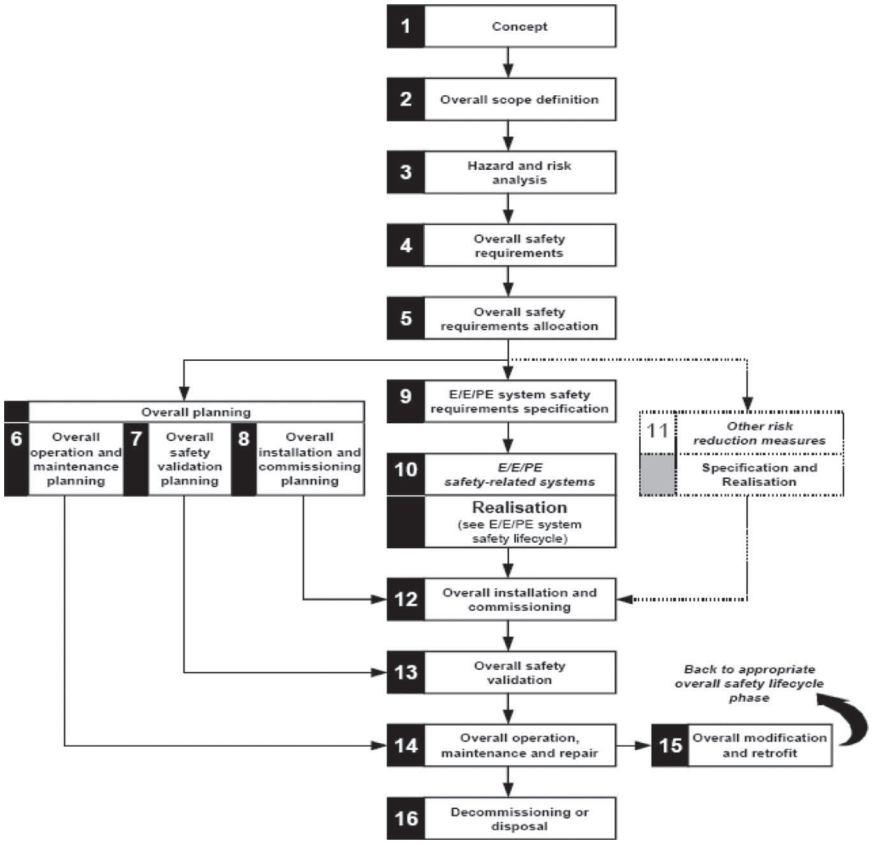

Fig. 3. Overall safety life-cycle (IEC-61,508).

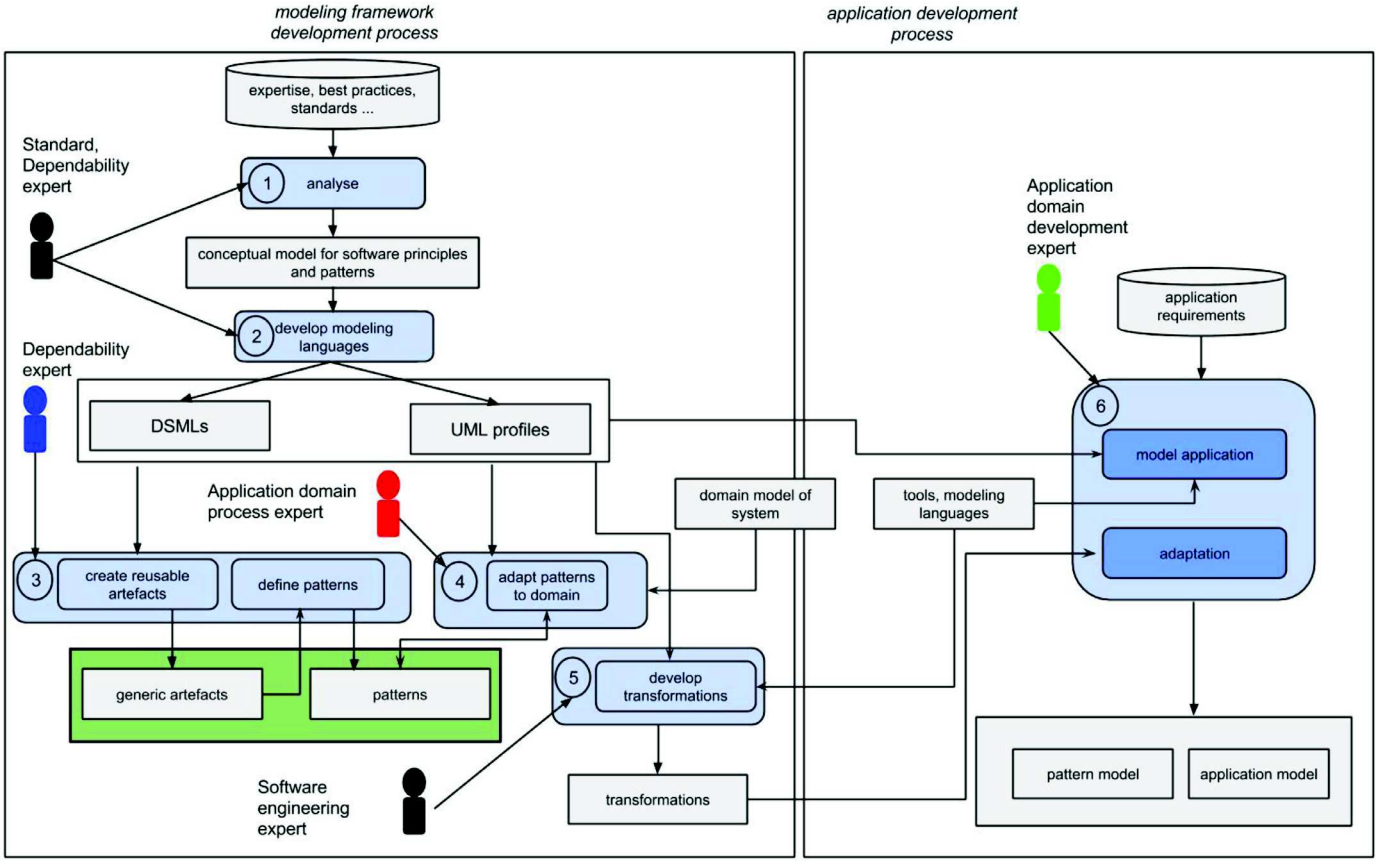

Fig. 4. Methodology for the creation of the PBSE modeling framework.

for compliance with an appropriate standard. We then develop and apply appropriate transformations of a pattern representation in a suitable format for the development environment (step 5). Pattern instantiation as the initial activity to apply a pattern is performed during steps 4 and 5. Finally, the domain engineer reuses the resulting adapted and transformed patterns for the given engineering environment (development platform) to develop a domain application (step 6). The pattern integration-application in the designs activity is performed during this step.

The first two steps (1 and 2) are performed once for a set of domains. The inputs of these steps are expertise, standards and best practices from the dependability expert. Step 3 is performed once for a set of domains. Step 4 is performed once per application domain. Performing step 3 requires knowledge of dependability engineering, whereas step 4 requires knowledge of both dependability engineering and the system development process for a specific application domain. Step 5 is performed once for each development environment. Step 6 is performed once for every system in the application domain. This step requires the availability of knowledge on the specific targeted system and dedicated tools that are customized for a given development platform. In the rest of this section, we present detailed descriptions of the six steps in our approach.

3.1. Step 1: conceptual model of dependability patterns

The idea of design patterns was introduced by an architect (Urbanist), Christopher Alexander (Alexander et al., 1977), not by a software developer. The first objective was to enhance architectural quality, beauty, elegance and harmony to avoid dehumanization of the living environment. Design patterns have a certain number of elements that must be captured by means of a pattern specification language. In *GoF* (Gamma et al., 1995), a design pattern extracts the key artifacts of a common design structure that make it useful for creating a reusable object-oriented design. In our context, we refine the GoF specification to fit with the non-functional needs. Adapting the security pattern definition of (Schumacher, 2003), we define a *dependability pattern* as a description of a particular recurring dependability problem that arises in specific contexts and presents a well-proven generic scheme for its solution. Therefore, a *system of dependability patterns* is a collection of dependability patterns and the relevant guidelines for their implementation, combination and practical use in dependability engineering.

Dependability patterns are defined from a platform-independent perspective (i.e., they are independent of their dedicated implementation mechanisms); they are expressed in a consistent way with domain-specific models. Consequently, they are much easier to understand and validate by application designers in a specific area. To capture this vision, we introduce the concept of the *domain perspective*, where a dependability pattern at the domain-independent level exhibits an abstract solution without specific knowledge of how the solution is implemented with regard to the application domain. The objective is to reuse the domain-independent model dependability patterns for several application domains and allow them to customize those domain-independent patterns with their domain knowledge and/or requirements to produce their own domain-specific artifacts. Thus, the question of how to support these concepts should be captured in the specification languages.

To foster the reuse of the best practices in software design through patterns, patterns rely on describing the concepts in an abstract way (i.e., domain-independent) and leave it to the software developer to create an implementation. In contrast, our dependability patterns support the software developer in the task of exploiting existing implementations. The knowledge of specific implementations using application domain constructs is captured in domain-specific patterns and thus supports the refinement step. Pattern refinement takes advantage of the fact that applications in the same domain, such as railway systems, have common requirements/standards and perform comparable dependability functions.

The modeling framework presented in this paper provides support for three levels of abstraction: (i) a pattern specification metamodel (SEPM), (ii) a domain-independent pattern model (DIPM), and (iii) a domain-specific pattern model (DSPM). This decomposition allows design applications within the context of dependability by avoiding the extensive complexity that is normally introduced when combining dependability and domain-specific artifacts. Moreover, this approach assists with overcoming the lack of formalization related to the classical textual pattern form.

Definition 1. (Domain). *A domain is a field or a scope of knowledge or activity that is characterized by the concerns, methods, and mechanisms employed in the development of a system. The actual clustering into domains depends on the given group/community implementing the target methodology.*

In our context, a domain may include knowledge of protocols, processes, methods, techniques, practices, OS, HW systems, measurement and certification related to the specific domain. For example, in the group of safety standards, IEC-61,508 is a domain-independent standard, whereas EN-50,126, EN 81 and ISO-26,262 are a railway domain- specific standard, an elevator domain standard and an automotive domain standard, respectively.

To specify dependability patterns, we build on a metamodel for representing these patterns in the form of a sub-system providing appropriate interfaces and targeting dependability properties to enforce the dependability system requirements. The so-called **external interfaces** are used to make the pattern's functionality available to the application, whereas the **technical interfaces** support interactions with dependability primitives and protocols of the application domain, including HW platforms. We capture the dependability capabilities of the pattern through a novel concept called **Dproperty**.

With regard to the artifacts used in the system under development, the first-class citizens of the domain are identified to specialize these artifacts. For example, the specification of a pattern in the domain-independent perspective is based on software design constructs. The specification of such a pattern for a domain uses a domain protocol to implement the pattern solution (see the example of a safety communication pattern given in Sections 3.3 and 3.4). For this purpose, we introduce two concepts: **DIPattern** and **DSPattern**.

These concepts and their related observations are used as a basis for our conceptual pattern modeling language (see Section 3.1.2). The following subsection describes an example to emphasize the issues identified in this paper. Then, the first level of abstraction, namely, the metamodel, is described.

3.1.1. Motivating example: safety communication pattern

As an example of a commonly and widely used pattern, we choose the *safety communication pattern* (IEC, 2010a). In the following, the terms safety communication pattern and black channel pattern are used interchangeably. Distributed safety systems require (safe) data communication between distributed safety functions. For example, the presence of random failures in the transfer of data within different levels of an application, such as transmission errors, repetitions, and deletions, can lead to a loss of information or to improper delivery of information. In a safety-related system or application, data communication is a vital part of their development. This communication could occur at different levels, i.e., on chips, inter-boards, or systems. A safety communication

pattern is required for safety-related systems, where the data communication is directly involved in the implementation of its safety function.

The main purpose of the safety communication pattern is to provide a simple way to guarantee that the communication at different levels in a system is reliable and to provide the capability to guarantee the correct transmission of information to the right destination and at the right time through a standard communication mechanism. The functionality of this pattern is based on the detection of a random software/hardware failure during communication. Safety standards let the designer use a standard communication mechanism to share the information in a safety function, although the reliability of the information is guaranteed through internal safety interfaces, which has the following connotations:

- The elements that share the information must be safety-relevant.
- The platform must support the implementation of an Error Detection Code (EDC).

This strategy leaves the responsibility of this task (at the interfaces of the elements participating in the communication) to detect failures during communication at different levels, such as chips, inter-boards, and systems. The safety communication pattern provides specific interfaces to guarantee the reliability of the information transfer and the authenticity of the participants in the communication. More specifically, the participants in the communication could be independent hardware channels, internal modules, etc.

Let *sender* and *receiver* be two elements acting as communication participants. Let *sender it f* (resp. *receiver it f*) be a *sender* interface (resp. *receiver* interface).

The *sender it f* offers the following services:

- Generate an identifier for establishing the sequence of the information to be sent.
- Generate an identifier for the source and receiver of the information.
- Generate an EDC; to detect failures in the information transfer.
- Pack the information before sending it over the standard communication mechanism. The *receiver it f* offers the following services:
- Unpack the information.
- Check to package before to use the received information to examine the following aspects:
  - the origin of the information (source),
  - the destination of the information (receiver),
  - the order of the information received and
  - that the information has not been corrupted.
- Send the received information to the receiver.

It is important to ensure that the actions of sending and receiving the safety message are cyclical and periodical because elements involved in the communication must always know when the message should be sent and when it should be received. It is important to synchronize both parts of the communication process. How to protect elements that share the information and how to provide the error detection code remain critical problems.

However, these safety communication patterns are slightly different with regard to the application domain. For example, a system domain has its own mechanisms and means to serve the implementation of this pattern, primarily the technique used to enable reliable delivery of data over unreliable communication channels. Depending on the type of failures to be detected and the diagnostic coverage to be achieved, the EDC can be implemented using a set of protocols, such as repetition codes, parity bits, checksums, Cycle Redundancy Checks (CRCs) and hash functions. To summarize, they are similar in their goal and different in their implementation issues, e.g., determining the level of communication in which the pattern is used and the restrictiveness and efficiency of the expected solution. Thus, the motivation is to handle the modeling of patterns by the following abstraction. In what follows, we propose using CRCs (Peterson and Brown, 1961) to specialize the implementation of the safety communication pattern. This solution is already used at the hardware and operating system levels to detect failures.

3.1.2. Pattern specification metamodel (SEPM)

To foster the reuse of patterns in the development of critical systems with dependability requirements, we extend a metamodel from (Hamid et al., 2016) for representing dependability patterns in the form of a subsystem that provides appropriate interfaces and targeting dependability properties to enforce the dependability system requirements. Interfaces are used to exhibit a pattern's functionality and to manage its application. In addition, interfaces support interactions between dependability primitives and protocols within a specific application domain. The principal classes of the system and software engineering pattern metamodel (SEPM) are described with Ecore notations in Fig. 5. Their meanings are explained in more detail in the following paragraphs.

- SepmPattern. This block represents a dependability pattern as a subsystem that describes a solution for a recurring dependability design problem arising in a specific design context. A SepmPattern defines its behavior in terms of provided and required interfaces. Larger pieces of a system's functionality may be assembled by reusing patterns as components of an encompassing pattern or an assembly of patterns; the required and provided interfaces are wired together. A SepmPattern may be manifested by one or more artifacts.
- SepmDIPattern. This is a SepmPattern that denotes an abstract representation of a dependability pattern at the domain- independent level. This is the key entry artifact to model patterns at the domain-independent level (DIPM).
- SepmInterface. A SepmPattern interacts with its environment via SepmInterfaces, which are composed of operations. A SepmPattern represented the provided and required interfaces. A provided interface highlights the services exposed to the environment. A required interface corresponds to services required by the pattern to function properly. We consider two interface types:
  - SepmExternalInterface. This allows the implementation of interactions to integrate a pattern into an application model or to compose patterns. It represents the application element to be used during the pattern integration- application in the designs. In other words, such a pattern element will be replaced with one element from the application design, or created if it exists in the pattern but not in the application (as will be detailed in Section 3.6). Moreover, it will be used to reason about the pattern properties and its provided design solution.
  - SepmTechnicalinterface. This allows the implementation of interactions with dependability primitives and protocols, such as error detection, and specialization for specific underlying software and/or hardware platforms during the deployment activity. It represents the platform element to be used during the pattern integration-application in the designs. In other words, such a pattern element will be replaced with one element from the application domain, or created if it exists in the pattern but not in the application. Please note that an SepmDIPattern does not have SepmTechnicalInterfaces.

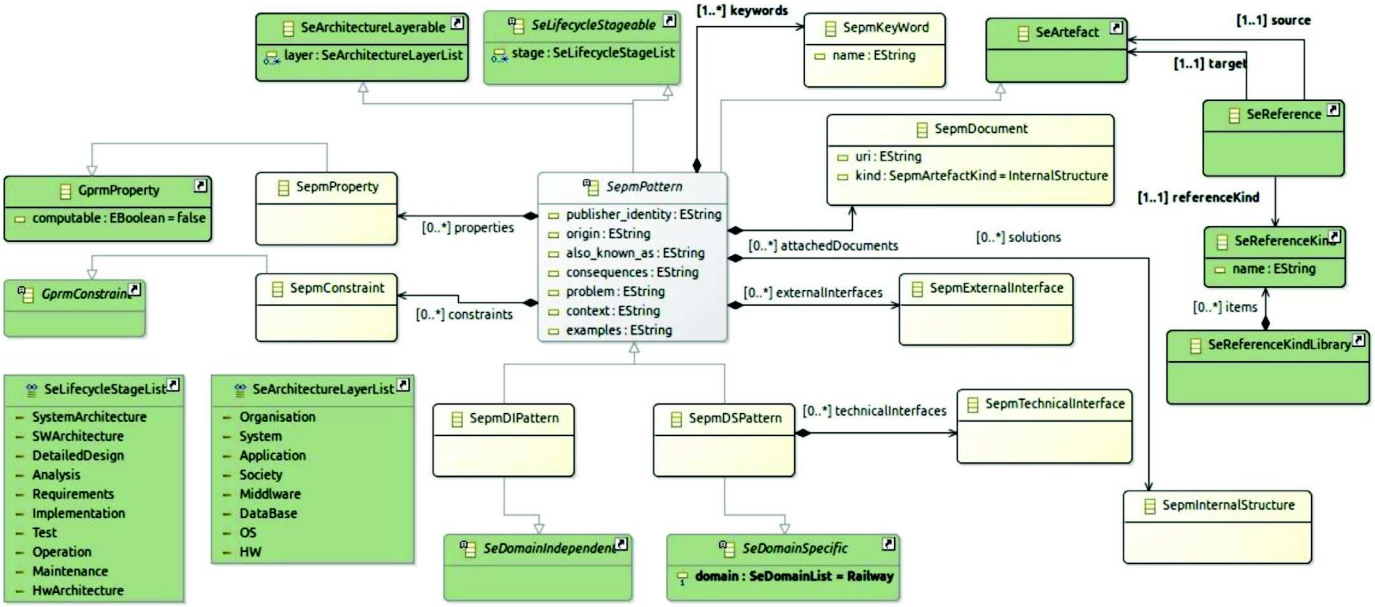

Fig. 5. An overview of the SEPM.

For our example, one may identify the following external interfaces:

– *send(S,d, t).* The application sender *S* sends data *d* at time *t*.
– *receive(R,d,t).* The application receiver *R* receives data *d* at time *t*.
– *send(P,d,crc,n,t).* The pattern sender participant *P* sends data *d* at time *t* with certain CRC *crc* and a sequence number *n*
– *receive(Q,d,crc,n,t).* The pattern receiver participant *Q* receives data *d* at time *t* with a certain CRC *crc* and a sequence number *n*.

- SeReference. This link is used to specify the relationship between patterns with regard to the domain and software lifecycle stage in the form of a pattern language. For example, a pattern at a certain software lifecycle stage *uses* another pattern at the same or at a different software lifecycle stage. SeReferenceKind contains examples of these links.
- seArtifact. We define a modeling artifact as a formalized piece of knowledge for understanding and communicating ideas produced and/or consumed during certain activities of system engineering processes. The modeling artifact may be classified in accordance with engineering process levels.
- SeLifecycleStage. A SeLifecycleStage SeLifecycleStage defines the development lifecycle stage in which the artifact is used. In our study, we focus on dependability pattern models. In this context, we use the pattern classification of Riehle and Buschmann (Riehle and Züllighoven, 1996; Buschmann et al., 1996, 2007).
- SepmProperty. This is a particular characteristic of a pattern related to the concern of interest and is dedicated to capturing its intent. Each property of a pattern is validated at the time of the pattern validating process, and the assumptions are compiled as a set of constraints that must be satisfied by the domain application. Additionally, this concept will serve for pattern classification and identification. The dependability attributes from (Avizienis et al., 2004) are categories of dependability properties. For our example, we define the following dependability properties:
  - *integrity of data.* When data *d* are received by the application receiver *R*, those same data *d* are sent out by the application sender *S* .
  - *data freshness.* It is often desired that the transmitted data *d* are recent. This property states that when an application receiver *R* receives data *d*, the same data are sent by the application sender *S* at most Delta t ago.
  - *non duplication.* Given data that are sent by the application sender *S*, a receiver *R* receives these data at most the same number of times they were sent.
- SepmConstraint. This is a set of requisites of the pattern. If the constraints are not met, the pattern is not able to deliver its properties. For our example, we specify constraints on the CRC computation/checking algorithms, on the correct/incorrect transmission over the network and on the maximal network delay.
- SepmInternalStructure. This constitutes the implementation of the solution proposed by the pattern. Thus, the InternalStructure can be considered as a white box that exposes the details of the SepmDIPattern and the SepmDSPattern. To capture all the key elements of the solution, the SepmInternalStructure manifests two types of structures: static and dynamic. One pattern can have several possible implementations, providing support for pattern variability.
- SepmDSPattern. This is a refinement of SepmDIPattern (as will be detailed in Section 3.4). It is used to build a pattern at the domain-specific level (DSPM). Furthermore, a SepmDSPattern has *Technical Interfaces* to interact with the platform. This is the key entry artifact to model the pattern at the DSPM.

3.2. Step 2: creation of a DSML from a conceptual model of dependability patterns

To create model instances of the proposed metamodels, we must provide concrete syntaxes. The DSML's concrete syntax may be described in any syntax type (textual, tree-structured, tabular, diagrammatic, etc.), depending on the corresponding artifact. For our purpose, we propose using the well-known approach in MDE: a DSML that contains pattern-specific information for several software and system modeling languages and different development environments. This approach is useful because we are storing a library of design patterns in a common repository and are providing one or more adaptations of each pattern to target several applica-

tion domains, e.g., the railway industry, and different development environment domains, e.g., UML. However, our vision is not limited to DSML. For example, in (Radermacher et al., 2013), we defined a UML profile under the UML papyrus tool[4] to specify patterns.

In our context, we use a mixed syntax that combines structured-tree syntax and a UML-based diagrammatic syntax to describe the SEPM's concrete syntax. The basic idea is that the former defines problems, objectives and constraints, whereas the diagrammatic part defines roles and solutions (SepmInternalStructure). The objective behind this separation is that a solution defined in the pattern can be integrated (without losing information) into the application architecture only if both are specified in the same modeling language, e.g., UML. Conversely, the problem statement and objectives are independent of the chosen modeling language. The separation enables solutions defined in different modeling languages to share the same problem definition, which is useful because we are storing design patterns in a common repository where model specifications in the structured-tree syntax are separately managed and stored. A pattern might eventually have multiple solutions defined in different modeling languages. The pattern discovery, i.e., the mechanisms to browse or search patterns within the repository, are based on the non-diagrammatic part.

3.3. Step 3: definition of dependability patterns

Once we have developed the DSML's concrete syntax in Step 2, we can create the set of dependability patterns to share the dependability expertise within the domain of interest. During this step, the patterns are constructed such that they conform to the metamodel description adopted in Step 1. To foster technology reuse across domains, the patterns are stored in a repository, such as the one described in (Hamid, 2014), thus reducing the amount of effort and time needed to design a complex system.

Furthermore, in the context of our work, we use validation reports and documentation generation techniques to validate each pattern. If a pattern is correctly defined, i.e., it conforms to its modeling language, then the artifact is ready for publication in the repository. Otherwise, we can identify any issues from the report and rebuild the pattern by correcting or completing the relevant constructs. Additionally, each pattern is studied to identify its relationships with other patterns belonging to the same application domain based on the engineering process activity in which it is utilized. The purpose of this activity is to organize patterns into a set of pattern systems. Moreover, this step should include all activities that support pattern producers in managing the relationships among these patterns, which can be defined in pattern relationship model libraries. At each stage (phase) n of the system engineering development process, the patterns identified in the previous stage (phase) $n-1$ can assist in the selection process during the current phase. As a prerequisite, we specify model libraries for the classification of patterns. At each stage of the system engineering development process, the appropriate patterns are identified via a classification process. Other work has adopted a similar conceptualization of a pattern-based development approach in the scope of safety engineering (e.g., (Hauge, 2014)), or in the scope of security engineering (e.g., (Uzunov et al., 2013)).

After an initial analysis of the various artifact sources, including standards and existing applications, the designer determines the stage of the engineering process lifecycle (system concept, system architecture, software architecture, and detailed module design) in which each pattern can be defined; moreover, whether the pattern is domain-independent or domain-specific can be determined. For this purpose, we choose to use the pattern classification of Riehle and Buschmann (Riehle and Züllighoven, 1996; Buschmann et al., 1996, 2007), who defined *system patterns, architectural patterns, design patterns* and *implementation patterns* to create the *SeLifecycleStage* model library. In addition, a pattern may be linked with other patterns and associated with property models using a predefined set of reference types, on a very high level (Noble, 1998) or including details on what part of a pattern is used, refined, or combined (Hauge, 2014). Here, we create the *SeReferenceKind* model library to support the specification of relationships across artifacts (e.g., *refines, specializes* and *uses*) as an extension of the relationship classification proposed in (Noble, 1998).

In the context of our work, certain patterns have a meaningful representation at the system level, at which general system blocks are defined and domain concepts are expressed (e.g., system redundancy). However, their representations might not be directly refined in later phases because they represent concepts that are meaningful at only the architectural level. In contrast, other patterns might be meaningful only in later design phases as indirect specializations of an architectural concept, e.g., a data agreement software pattern is a specialization of an architectural system redundancy pattern. In addition, the same pattern may have multiple instantiations and specializations in each phase (e.g., a watchdog driver is linked to a hardware component). Therefore, as shown in Fig. 6, a given design pattern (P2) in the repository might follow a tree-shaped refinement and specialization flow, representing different lifecycle phases, different refinements and specializations, and new pattern representations in later phases. The following is an example of specialization through the process:

1. *P2* (at System Concept Specification phase): Black Channel
2. *P22* (at System Architecture Design phase): Ethernet-Based, Star-Topology Black Channel
3. *P221* (at Software Architecture Definition phase): Ethernet-Based, Star-Topology Black Channel with CRC and sequence number monitoring
4. *P2212* (at Module Detailed Design phase): Ethernet-Based, Star-Topology Black Channel with CRC and sequence number monitoring

The target representation is the DIPM level, while still conforming to the SEPM metamodel. At the DIPM level, this description reveals the following elements: *interfaces* of type SepmExternalInterface, *dependability properties* of type SepmProperty and *solutions* of type SepmInternalStructure. Moreover, for classification (respectively relationship) purposes, additional information may be defined, e.g., *lifecycle stages* of type SeLifecycleStage (respectively *relationships* of type SeReference).

The first task is to create a basic pattern subsystem as an instance of the SepmPattern. The instance is given a name and a set of attributes that correspond to the pattern. The description, with varying levels of abstraction, is managed by inheritance. Once the basic pattern subsystem is specified, interfaces are added to expose some of the patterns functionalities. For each interface, an instance of *SepmExternaInterface* is added to the patterns interface collection. The next step after creating interfaces is the creation of property instances. An instance is created in the patterns property collection to specify every identified dependability property. A property is given a name and an expression based on external interfaces in a property language.

We continue our illustration using the example of the *safety communication pattern*. For the sake of simplicity, we specify only those elements related to both steps 2 and 3that are required to explain our approach. As introduced in Section 3.1.1, data communication must be safe, which leads to two possible approaches (IEC, 2010a):

1. *White channel:* Use of a safety communication channel that is designed, implemented and validated according to the IEC-

[4] http://eclipse.org/papyrus/.

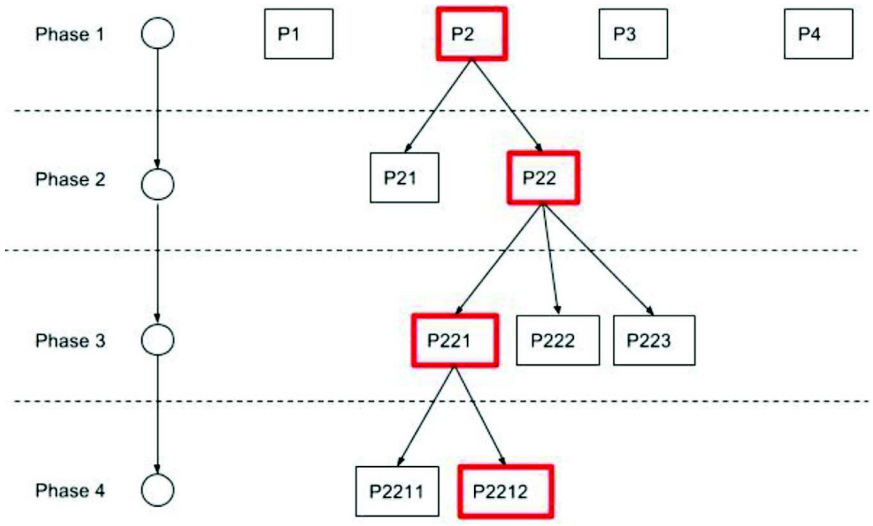

Fig. 6. Tree-shaped pattern refinement and specialization.

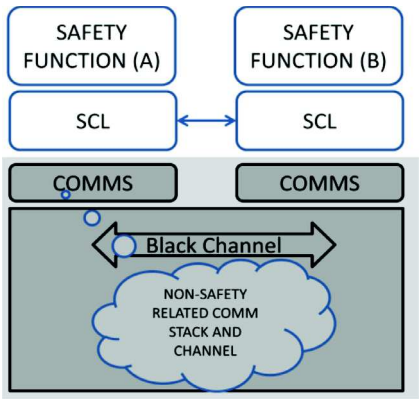

Fig. 7. Safety communication pattern structure.

61,508 standard (IEC, 2010b) and the IEC-61,784-3 (IEC, 2003) or IEC-62,280 (IEC, 2002) standards using a certified safety communication channel, such as TTEthernet (TTE)[5].

2. *Black channel:* Use of a communication channel that is not designed or validated according to the IEC-61,508 standard, where safety measures can be implemented either in the safety functions or in interfaces with the communication layer in accordance with the IEC-61,784-3 or IEC-62,280 standards. The latter is called a *safety communication layer (SCL).*

As shown in Fig. 7, the safety communication pattern is an application level service on top of a non-safety-related communication stack ("comms") that enables "safe" data exchange between safety functions. It must be defined according to a life cycle equivalent to the highest safety level (SIL) in the application, requiring the detection of all possible communication errors, such as corruption, incorrect message order, message outside temporal requirements, message lost, message duplicated, etc.

In our example, an instance of SepmPattern is created and called *SCL*. Fig. 8 shows the SCL pattern for the software architecture. An encapsulation unit is used to prepare the data message to be sent and a de-encapsulation unit to extract the information from the message once it has arrived. The receiver provides an interface where some extra information is added in the sent message, which will be checked by the receiver. This information includes (1) a safety code for the receiver to verify the integrity of the message and (2) a sequence number to ensure the correct arrival order. The sender is also responsible for maintaining a minimum transmission rate that is specified by the requirements of the application. Moreover, the receiver provides an interface to check the received message. This interface includes (1) a safety code to ensure the integrity of the message, (2) a sequence number, which guarantees that the received message is new and not a replica traveling into the communication channel, and (3) a transmission quality checker to react if it falls below a predefined level.

[5] http://www.tttech.com/products/ttethernet/.

3.4. Step 4: adaptation for a specific domain

At the DSPM level, the dependability pattern and some of its related elements are also created by inheritance. Once a SepmDSPattern is created, every pattern external interface is identified and modeled as a refinement of the DIPM's SepmExternalInterface in the pattern's interfaces collection. Then, following the pattern's description of the particular solution that is represented, each of the pattern's technical interfaces is identified and modeled by an instance of SepmTechnicalInterface in the pattern's interfaces collection.

In the context of our experiment, the railway domain-specific pattern must be compliant not only with the generic safety standards but also with railway-specific safety standards. Fig. 9 shows the SCL pattern for the software architecture. The encapsulation and de-encapsulation units described in the domain-independent perspective are refined into sequencer, CRC calculator and software watchdog units. This combination of elements enables the detection of errors. A sequence number is added to the data message to detect an incorrect order of messages and/or messages that are lost. A CRC code is added to the data message to detect message corruption. Each message has an associated period, and a watchdog is used to detect whether the message arrival time is within the specified time range. Thus, the sender provides an interface in which extra information is added to the sent message, which will be checked by the receiver. This information includes (1) a safety code (CRC) to detect message corruption and (2) a sequence number to detect an incorrect order or lost messages. Moreover, the receiver provides an interface to check the received message through a CRC checker. To detect message corruption, the receiver computes a CRC and compares it with the one provided by the sender. However, there are other possible realizations, e.g., using

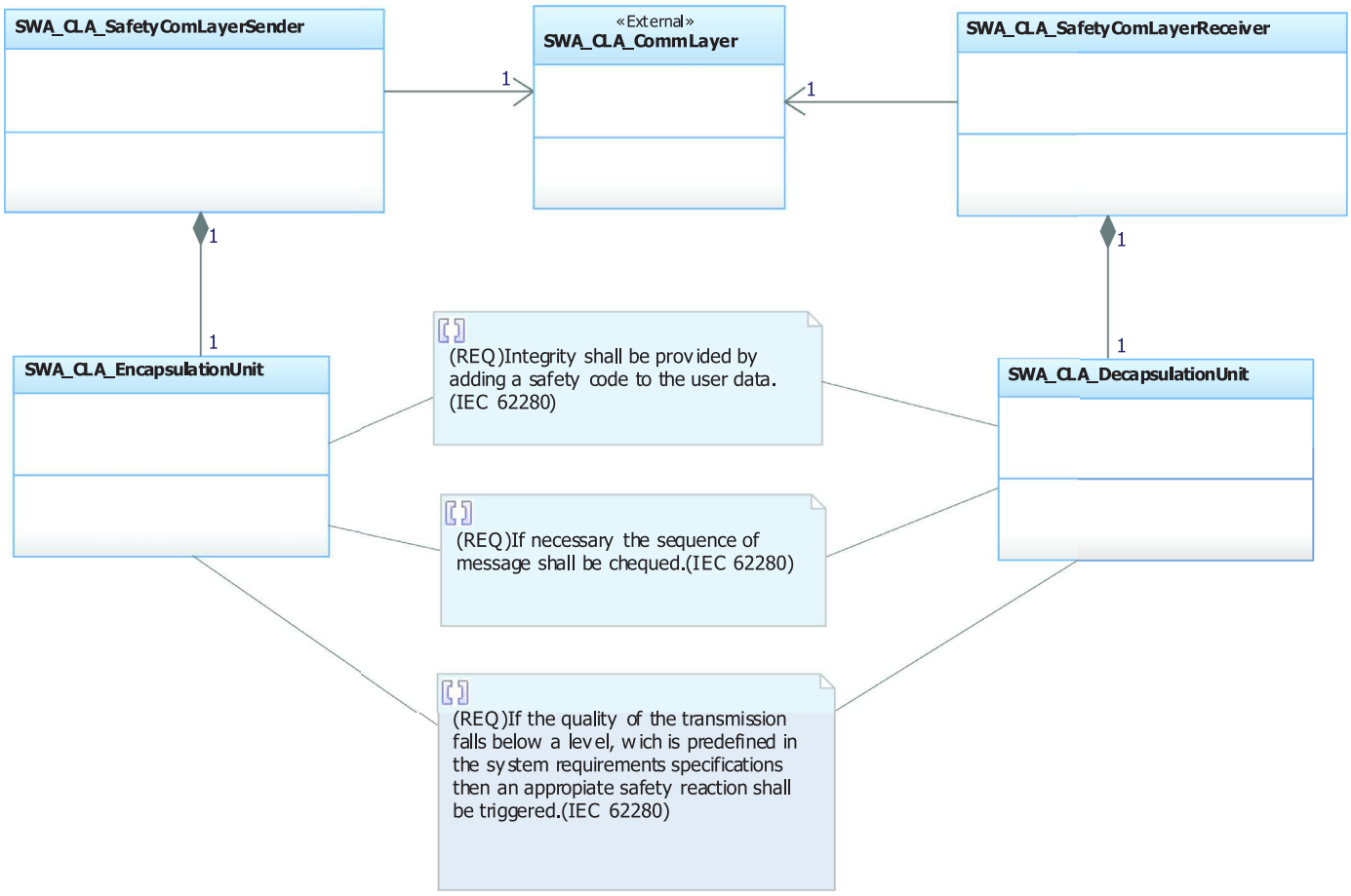

Fig. 8. Black channel for software architecture.

a sequence number checker or using a quality checker through a software watchdog.

3.5. Step 5: adaptation for a specific domain development environment

The final step (step 6) is performed to support the development of a specific system in the application domain. As a prerequisite, step 5 identifies appropriate patterns and creates tailored versions that represent model concepts in the domain of interest and that can be adapted to both the system development process and the development environment. The selection of a pattern is primarily the choice of the developer. There are various considerations that may narrow and simplify this choice. The first is the purpose of the pattern application. Although this purpose cannot be generally formalized, certain patterns address requirements that are defined by domain standards (e.g., safety). If these requirements are stored in a model library and are referenced in the definitions of the patterns, then the selection of patterns could be driven by the selection of (domain) requirements. The second consideration is that patterns can be classified with respect to several properties. One of which is the stage of the engineering process lifecycle discussed in Section 3.3 – a pattern may be relevant to the system, its architecture, or to aspects of its design or implementation. Thus, it must be possible to filter available modeling artifacts based on this classification.

In our context, the mappings from dependability pattern models, which are formalized in a SEPM description language, for a specific domain development environment are supported via model transformations. Once the repository is available[6], patterns can be imported/exported from the repository as XMI standard files that are compatible with processes: *Identification* and *Tailoring*.

Definition 2. (Identification). Identification activities support system engineers in selecting appropriate solutions from the repository. This activity makes it possible to search for and retrieve patterns in accordance with the system requirements. The identification activity consists of the following tasks:

1. Define needs.
2. Search for patterns in the repository.
3. Select the appropriate patterns from those proposed by the repository.

Definition 3. (Tailoring). A tailoring activity involves the retrieval of a pattern and its related model libraries from the repository and their incorporation into the target development environment. This activity enables the reuse of a pattern. The tailoring activity consists of the following tasks:

1. Adapt the selected patterns for the domain-specific process of interest.
2. Import the tailored patterns by transforming them into the domain-specific development environment.

In the context of our work, the target domain development environment is IBM Rational Rhapsody[7], and the descriptions of the

[6] The repository system populated with modeling artifacts.
[7] http://www-03.ibm.com/software/products/en/ratirhapfami.

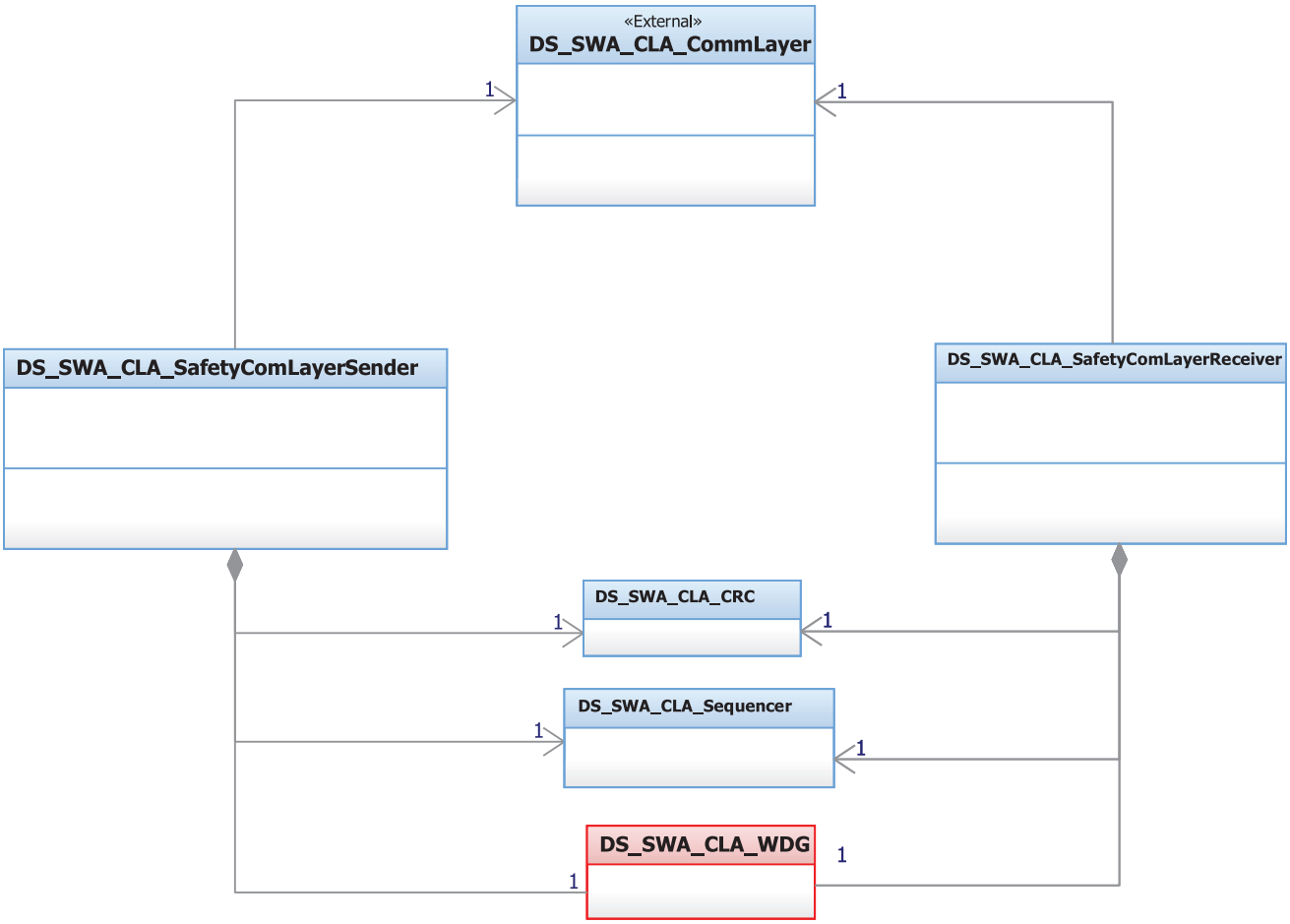

Fig. 9. Railway black channel software architecture.

model transformations are based on the QVT operational language. Therefore, the design of a given pattern can be regarded as a single package that contains one sub-package per lifecycle phase of the engineering process; each of these phases can contain design modules and additional sub-packages associated with particular specializations and refinements. Thus, imported patterns are stored inside a dedicated package that facilitates searching within the package tree of each design. Moreover, to foster reuse, the pattern artifacts related to that phase are instantiated from the repository to the vehicular modeling tool as a reference package. As shown in Fig. 10, each pattern design package generally contains the following items:

- Any information that is required by the end-user pattern integrator, e.g., a UML class or SysML block, with interfaces that enable the interconnection of patterns with a given system design.
- Additional detailed information of interest, e.g., a "structure" package that contains the static internal structure (e.g., class diagram) and the dynamic structure (e.g., sequence diagram).

3.6. Step 6: reuse for a specific system development

This section focuses on the use of patterns in a software development process. The integration of a pattern involves the application of a solution provided by that pattern in an existing application architecture to take advantage of its benefits. We cannot simply copy such a solution into the architecture under development. Instead, we must account for the interplay between elements that already exist in the application and the elements of the pattern. The challenge of this task is that the relationships (e.g., connections, associations or inheritances) defined between elements in the pattern definition must also be established in the application model. Currently, integrating a pattern requires adding new connections, associations, and other factors, and the user must resolve potential conflicts.

To address this issue, a specific activity called *Integration*, which was already studied in (Hamid et al., 2012), is used herein.

Definition 4. (Integration). An integration activity is performed within the development environment when a pattern and its related model libraries are introduced into an application design; it allows the elements of the application to be organized for consistency with the elements of the pattern. The integration activity consists of the following tasks:

1. Preparation. The elements of the pattern are extracted in the form of a role diagram to match/merge them with the elements of the existing application model.
2. Elicitation. Connections between the application model and the pattern based on the role diagram are constructed. This phase is responsible for defining the elements of the application that are used to fulfill the roles identified in the pattern.
3. Consolidation. The pattern is merged with the application. For certain elements of the pattern, they may simply be replaced with elements from the application or newly created elements may be added as they exist in the pattern.
4. Adaptation. An optional phase that offers the opportunity for tailored integration by allowing the user to refine the new application is conducted.

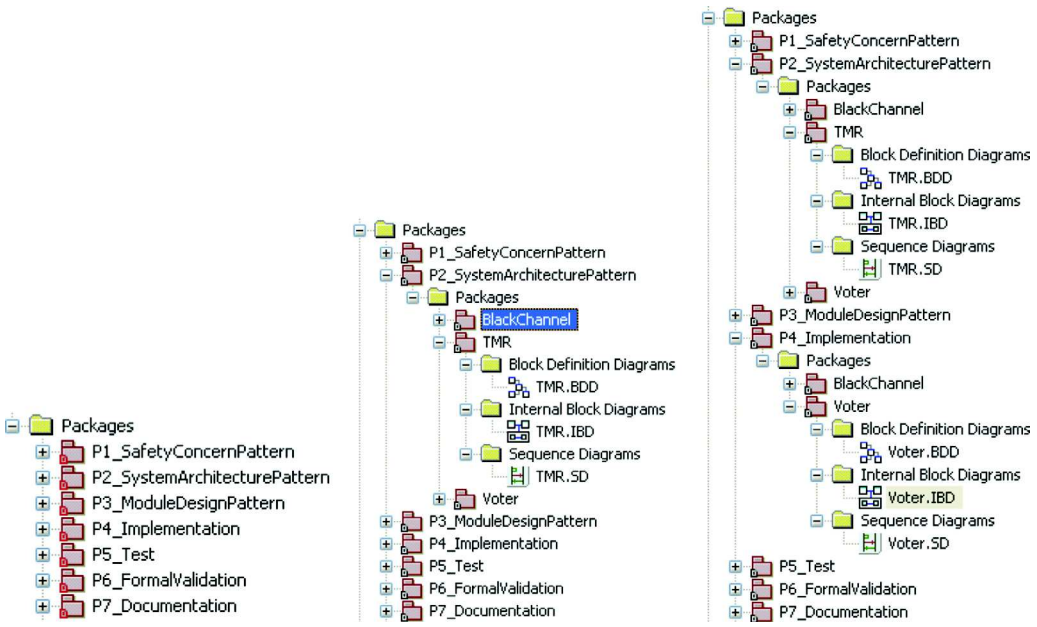

Fig. 10. Pattern design deployed in packages using the IBM Rational Rhapsody tool.

In the context of our example, by executing a tailoring activity, the pattern is exported in an XMI file. Then, it must be imported from Rhapsody. This two-step approach is an intermediate solution for the purpose of demonstration. The envisioned future solution is to install a plug-in in the design tool so that patterns can be imported without external software. As shown in Fig. 10, once the pattern is imported in Rhapsody as a package, a project tree is generated and its artifacts are available in the project. Therefore, in each phase, the system developer executes the search/select task on the repository to tailor appropriate patterns for the modeling environment using the identification and the tailoring processes described in Section 3.5. The developer then integrates them into the application models following an incremental process. For example, the process flow at the software architecture phase can be summarized by the following steps:

1. The software architect searches for different specializations (at the software architecture-definition level) of the patterns to complement the design.
2. The software architect selects the appropriate set of identified patterns.
3. The software architect imports the software architecture design perspective of each pattern into the vehicular modeling environment (Rhapsody) as a reference package. The application developer is responsible for linking the pattern interfaces to integrate the pattern at that specific level.
4. The software architect integrates the pattern into the existing software architecture design diagrams.

4. Tool support

In this section, we propose an MDE tool chain to support the proposed approach and assist the developers of model- and pattern-based dependable software systems. As discussed below, the proposed tool chain is designed to support the proposed metamodels; hence, the tool chain and the remainder of the activities involved in the approach may be developed in parallel. Appropriate tools for supporting our approach must fulfill the following key requirements:

- Enable the creation of the UML class diagrams used to describe pattern metamodel in our approach.
- Enable the creation of a concrete syntax.
- Support the implementation of a repository to store pattern models and the related model libraries for classification and relationships.
- Enable the creation of pattern models and the related model libraries and publication of the results into the repository.
- Support the administration and the internal management of the repository.
- Enable the creation of visualizations of the repository to facilitate its access.
- Enable the creation of application models.
- Enable transformations of the models from the repository format into the target modeling environment.
- Enable the integration of application models and imported patterns.
- Support application-specific code generation.

To satisfy the above requirements, we define four integrated sets of software tools:

- *Toolset A* for populating the repository,
- *Toolset B* for retrieval and adaptation from the repository,
- *Toolset C* to serve as the repository software, including its administration and internal management, and
- *Toolset D* as the augmented target development environment.

There are several environments that can be used to build an MDE tool chain. In this work, the open-source Eclipse Modeling Framework (EMF) and its extended version, Eclipse Modeling Framework Technology (EMFT) are used to build the support tools for our approach. All metamodels are specified using the EMF. The design tools are semi-automatically generated from these metamodels. Several enhancements are added to the generated code, such as creation wizards, to guide the modeling artifact designer in populating the repository. Visual enhancements are added to facilitate the recognition of different concepts as a first step toward a future visual syntax. To describe the model transformations, the QVT operational language (OMG, 2008) is used. The repository is implemented using the Eclipse CDO[8] framework. However, our vi-

[8] http://www.eclipse.org/cdo/.

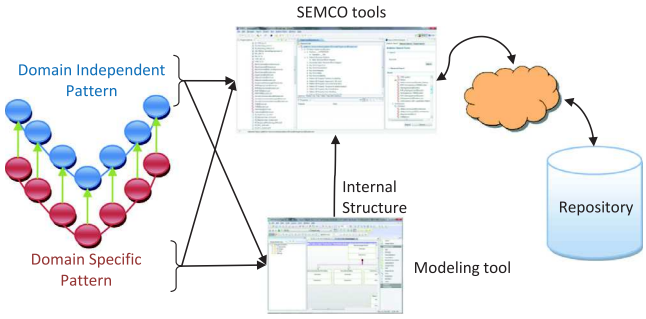

Fig. 11. Pattern designer schematic.

sion is not limited to the EMF platform. Other modeling tools conforming to the requirements of Section 4 can also be used. For example, in (Radermacher et al., 2013), we investigated the use of UML papyrus and its support for the definition of UML profiles to provide tool support for the approach.

Our approach is successfully applied to a case study of PBSE. Specifically, we develop Semcomdt[9] (SEMCO model development tools) as an MDE tool chain to support all the steps in our approach. Semcomdt offers the following features:

- *Gaya* for specifying and implementing a repository to store models,
- *Arabion* for specifying patterns that conform to *SEPM*, and
- *Retrieval* for repository access.

For populating the repository, we construct a pattern design tool (Arabion) to be used by a pattern designer. Arabion interacts with the Gaya repository for publication purposes. As described below, and already described in Section 3.2 from a DSML construction perspective, design patterns are composed of two parts, as visualized in Fig. 11:

- *Structured-tree component.* Pattern definition that defines pattern properties and attributes, such as safety properties, resource constraints, development phases, and relationships. These data are used to ease pattern search and analysis (Hamid, 2015).
- *UML-based diagrammatic component.* Pattern internal structure design files generated via additional tools, e.g., Rhapsody or Papyrus (UML editors), that are stored as XMI files and can be attached to the pattern description file.

The pattern design environment is presented in Fig. 12. There is a design palette on the right, a tree view of the project on the left and the main design environment in the middle. Furthermore, Arabion includes mechanisms for verifying the conformity of the pattern with the SEPM metamodel and for publishing the results to the repository.

For access to the repository by a system engineer, the retrieval tool provides a set of functions to assist in the search, selection and sorting of patterns. For example, as shown in Fig. 13, the tool assists in selecting appropriate patterns through key word searches and lifecycle stage searches. The results are displayed in the search result tree as system, architecture, design and implementation patterns. The tool includes features for export and tailoring using dialogs that are primarily based on model transformation techniques to adapt pattern models to the target development environment.

With regard to Tool set D, IBM's Rational Rhapsody Developer/Software Architecture[10] is used to provide tool support for the other parts of our approach. Other modeling tools can be used in accordance with the target application domain and/or modeling environment. In addition to meeting its expected requirements, Rhapsody is a mature and well-established tool in the industry, making it easier to provide support for our approach and making our approach more likely to be adopted by practitioners, such as those engaged in railway safety-related processes. Rhapsody also allows for application-specific codes to be automatically generated (except for some specific parts that are coded by hand).

Rhapsody is used as the domain-specific design software tool to design (and implement) the system using UML/SysML modeling languages. For example, it can be used to design systems based on packages, where one package might contain design diagrams and/or additional packages. Based on this approach, the design of a given pattern can be considered a single package that contains one sub-package per safety engineering process lifecycle phase; each of these phases might contain design modules and additional subpackages associated with specializations and refinements. Thus, the access tool provides the option to export patterns in a format that can be imported by the Rhapsody tools. Therefore, a customized access tool, such as the one shown in Fig. 14, is developed to construct connections between the railway development environment and the repository of patterns. The access tool offers a GUI that allows the user to search for and select patterns. When a pattern is selected, the access tool instantiates the pattern in the domain-specific tool. Because this task is performed during product development, the selected pattern must be compliant with the current phase of the domain process and with the user tools. By accessing the repository, we introduce features based on model transformation techniques to adapt the pattern model to the target development environment. In our work, the target format is a subset of UML that can be imported using Rhapsody and the model transformations as developed using the Eclipse implementation of QVTO.

The left-hand side of Fig. 14 shows the main window of the railway access user interface. This window has two panels: one for searching and the other for display. The display panel on the bottom shows the name of the selected pattern and its different views, e.g., a graphic view with a class diagram of the pattern implementation. Searching the repository is performed using either the "Name" field to enter part of the name or "Keywords" to enter the desired pattern characteristics. To import the selected pattern into the development environment, as shown on the right-hand side of Fig. 14, the access tool creates a new representation of the selected pattern as a UML package for Rhapsody using model transformation techniques.

To assist in the integration step, we provide support for the various phases. However, certain development environments may already offer native integration support. As shown in Section 3.6, a binding must be established between elements of the application architecture and the roles defined in the pattern. The integration tool supports the developer in two different ways. The first is that it enables a filtered selection of possible elements for binding to be displayed. The second is that it provides the completely automated creation of bindings. If a role remains unbound during integration, the developer can indicate that it has no corresponding element in the application and that a suitable element must be created in the application model. For example, in (Radermacher et al., 2013), we used UML collaborations (OMG, 2011a) for modeling patterns and for role binding to establish links between elements of the application architecture and the roles defined in a pattern. Furthermore, every design pattern has certain constraints that must be satisfied while allowing for a certain degree of modification. Thus, there is a need to ensure that the properties of the pattern remain valid while not preventing acceptable modifications (such as renaming). A verification rule associated with each pattern ensures that the invariants of the pattern can be checked after the pattern has been applied. These validation rules are currently

[9] http://www.semcomdt.org.
[10] http://www.ibm.com/developerworks/rational/products/rhapsody/.

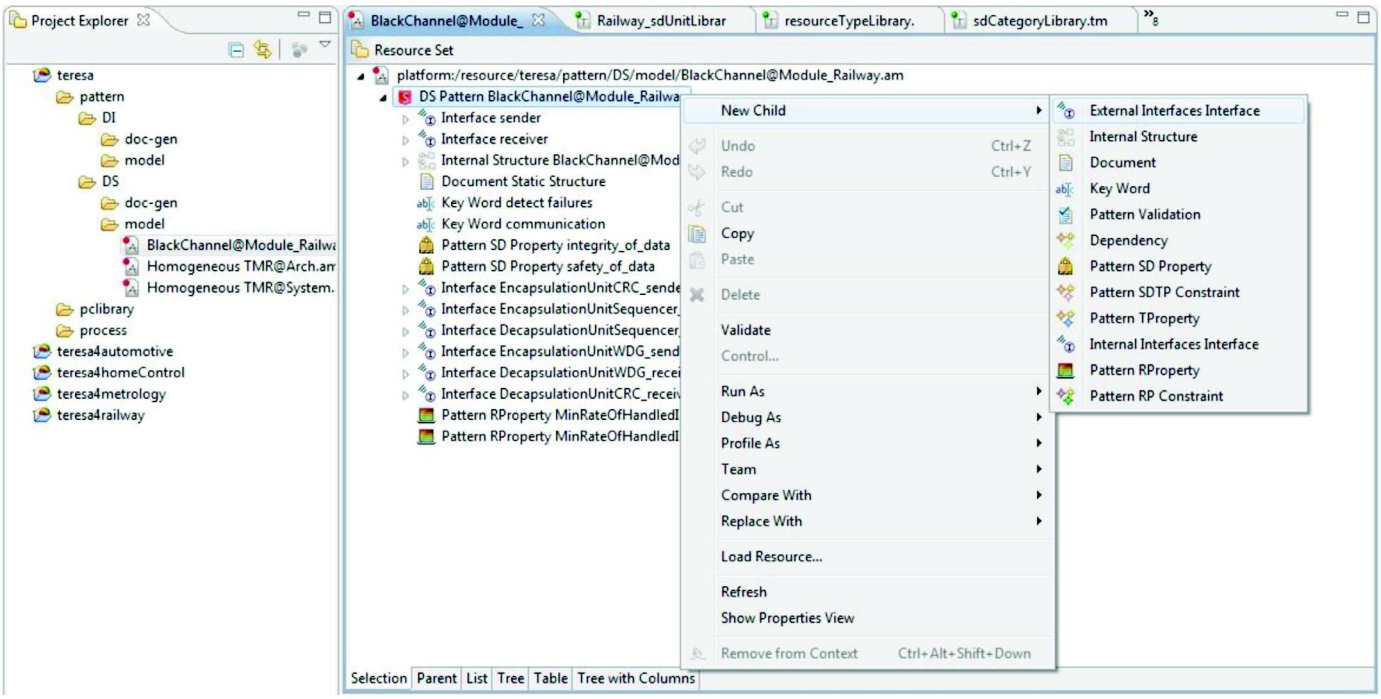

Fig. 12. Designing a pattern.

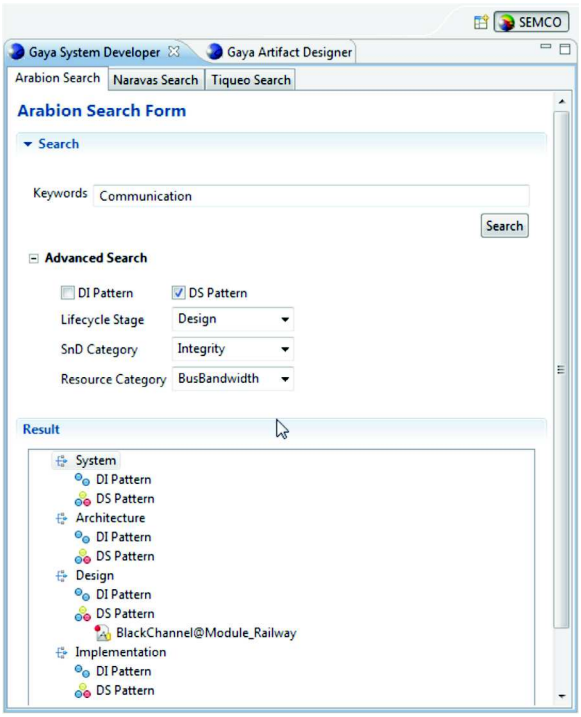

Fig. 13. Access tool.

implemented in the programming language of the target development environment (e.g., C++, Java, etc.); alternatively, a constraint language, such as Object Constraint Language (OCL) (OMG, 2010), could be used. However, the realization of such rules is beyond the scope of this paper.

5. Evaluation

In this section, we first report on an industrial case study performed in the railway domain (Section 5.1), followed by a description of a survey performed among railway domain experts to better understand their perceptions of our approach (Section 5.2). The case study enables us to determine that the pattern-based approach leads to a reduced number or to a simplification of the engineering process steps, whereas the survey assists in assessing whether domain experts agree on the benefit of adopting the pattern-based approach in a real industrial context.

5.1. Case study

In the context of the TERESA project, we evaluate our approach in the construction of an engineering discipline that is adapted to resource constrained systems by combining the MDE process and a model-based repository of dependability patterns and their related property models.

In this subsection, the adaptation of railway processes to incorporate the pattern-based approach is described. We test which of the provided tools are able to support the pattern integration or assist the engineering process. In this context, the extendibility of the pattern repository with new patterns and the extendibility of existing patterns are observed. Furthermore, we evaluate the usefulness of the patterns with respect to increasing engineering productivity. In the presentation of the case study and its results, we describe only a small portion of this system. We do not show the complete resulting model because it contains proprietary information from our industrial partner.

5.1.1. Nature of the case study

One of the case studies that serves as a TERESA demonstrator is set in the railway domain. For confidentiality reasons, we do not reveal the name of the collaborating partner. This is a very conservative domain in which dependability is a key requirement for most subsystems. Thus, the railway domain is a highly appropriate sector in which to apply our approach. It is not uncommon to find

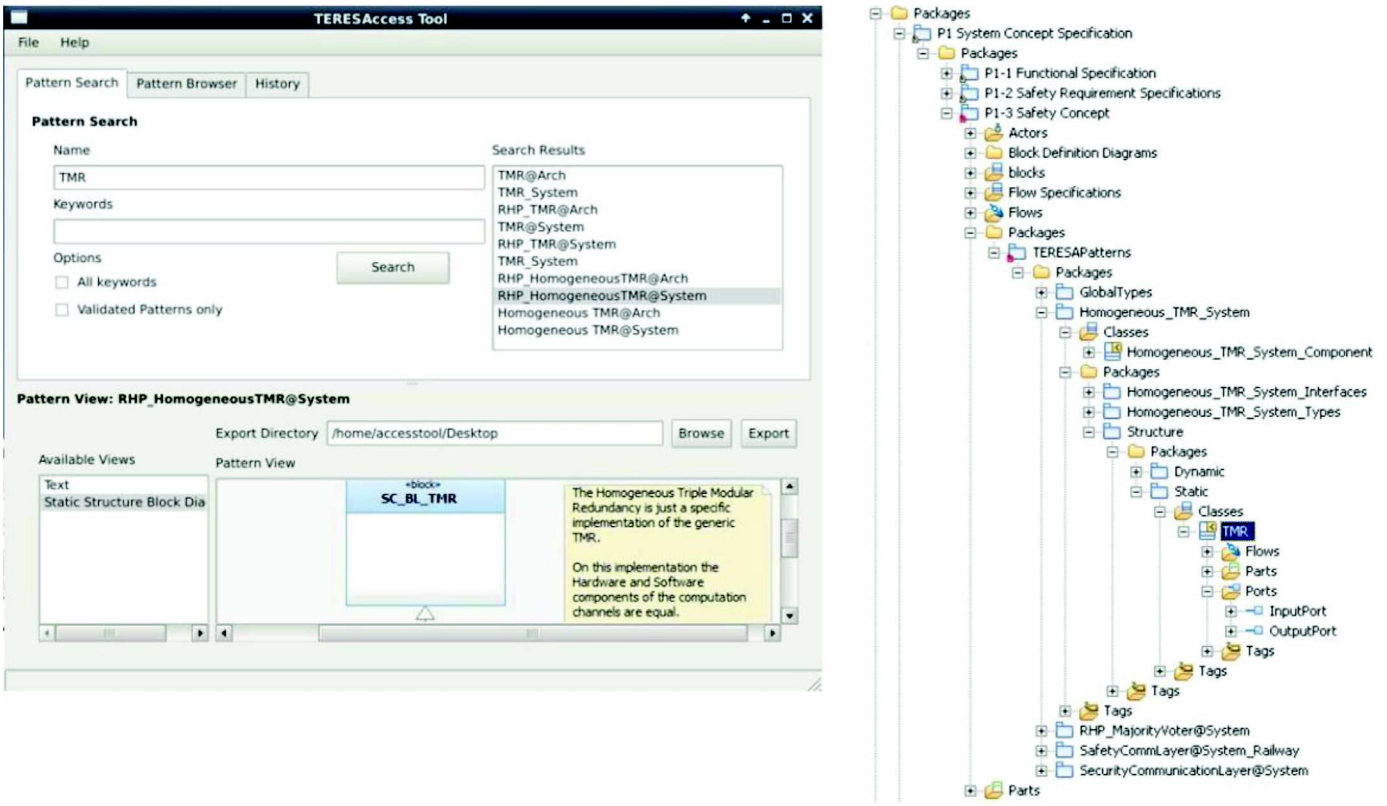

Fig. 14. Railway access tool.

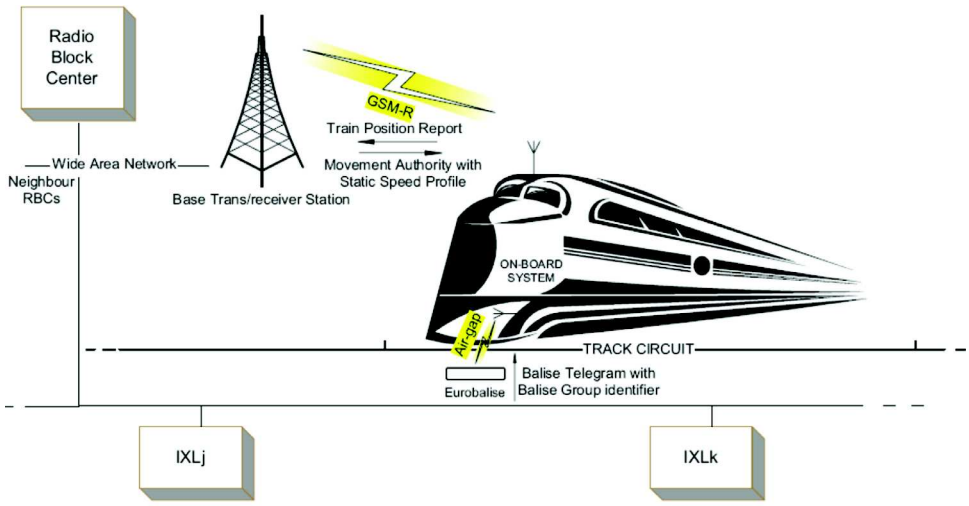

Fig. 15. ERTMS/ETCS diagram.

situations in this industrial domain in which the reuse of system and software modeling artifacts by means of a repository could accelerate and support the development of safety-related subsystems. We demonstrate the applicability of our proposed framework using the *Safe4Rail* demonstrator, which is a simplified version of a real ETCS (European Train Control System) for signaling, control and train protection (see Fig. 15). Additionally, for confidentiality reasons, we use a small but realistic setting to illustrate the dependability pattern-based approach proposed herein. The main functionality of this demonstrator is to ensure that the speed and distance traveled do not exceed the authorized maximum values specified by the railway infrastructure. Safe4Rail is responsible for emergency braking in a railway system. Its task is to detect whether the brake should be activated. Most importantly, the emergency brake must be activated when something goes wrong.

At every position, the braking curve provides three speed limits, which are used to make decisions about when to activate the brakes (see Fig. 16):

1. When the current speed exceeds the warning speed limit, the system must activate a warning signal to advise the driver that the train is approaching a dangerous speed.
2. If the driver does not take any action and the service speed limit is exceeded, the system must activate the service brake.

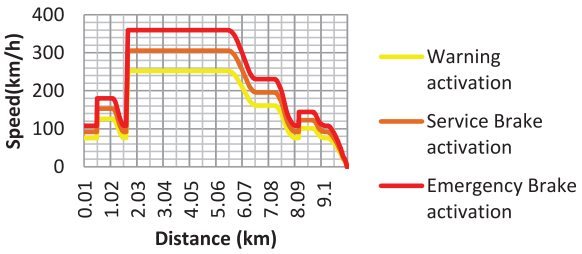

Fig. 16. ERTMS/ETCS supervision limits and braking curves.

3. If the train continues accelerating and exceeds the final limit, the system will deactivate the acceleration and activate the emergency brake to stop the train completely.

The fundamental functionality of the system is based on a set of inputs from the rail track, assorted sensors, balises, etc. Beginning with these inputs, it performs various calculations to determine whether the emergency brake should be activated. An output signal is sent if necessary. Fig. 17 provides an overview of the entire system architecture; a description for each subsystem that contributes to the system is also provided. Furthermore, the following list provides the requirements to fulfill by these components:

1. *Clock*. Generates a periodic event that triggers the system to estimate the current position and speed and to supervise the train to ensure it complies with the current track restrictions.
2. *Environmental conditions*. Represent the physical interaction between the environment (train, track, etc.) and the sensors of the system.
3. *Balise*. Represents a balise installed on the track that supplies the train supervision system with new information regarding the current position and the track conditions.
4. *Safe Train Interface*. Represents the actuators for the application.
5. *Supervision system*.
   (a) *Balise reader*. Detect and read the information provided by the balise on the rail.
   (b) *Supervision*. The main component of the system responsible for carrying out the functionality of the system.
   (c) *Sensors*. Provide the system with the actual position and speed of the train and the track conditions.
   (d) *User interface*. The driver interacts with the system through this interface.

A more complete description of these components can be found in TERESA, (2013).

5.1.2. Description of the application

The safety requirements of the proposed case-study are extracted from a simplified analysis of the ERTMS/ETCS standard and considering that the objective of the case study is not the development of the complete certifiable and interoperable ERTMS/ETCS system but the provision of a case study that is as realistic as possible and can be developed within the scope our study. In the Safe4Rail system, there are three main cases, which are described below. Fig. 18 provides a diagram of the selected cases, illustrating their relationships and their classification as safety-relevant or non-safety-relevant cases. These use cases can be described as follows:

1. Activate emergency brake and perform diagnostics (when the system is in standby mode).
2. Supervise train speed and position (when the system is in supervision mode).

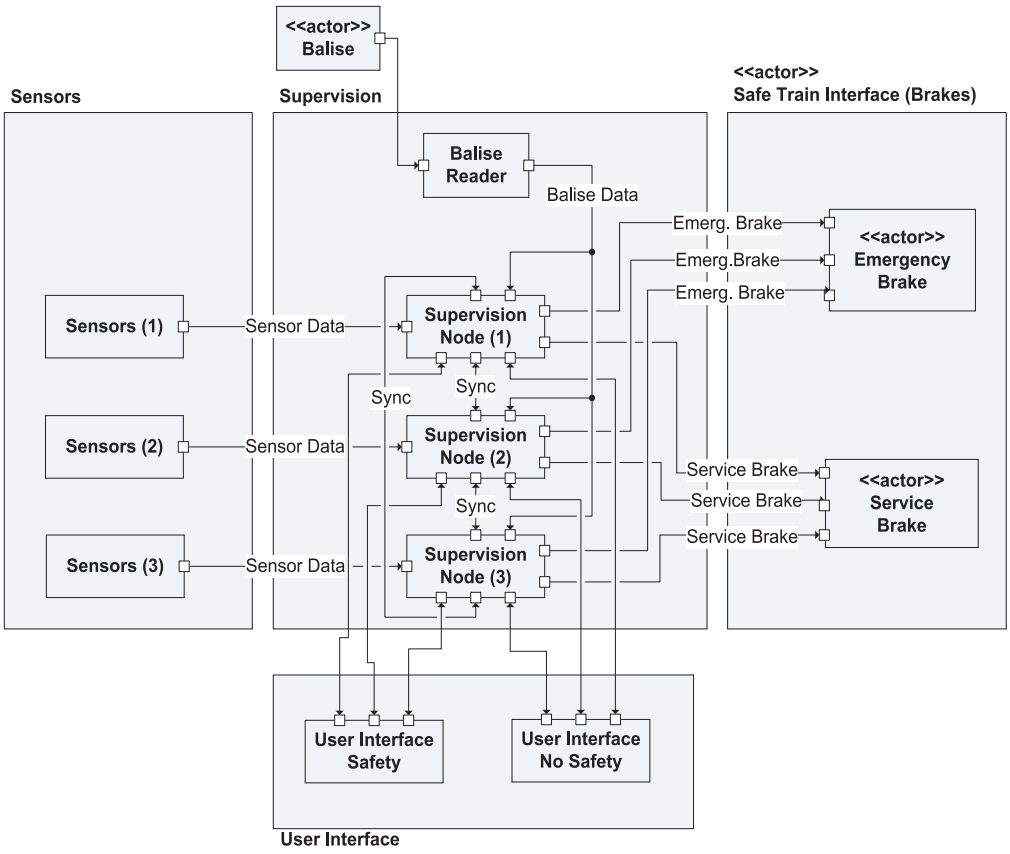

Fig. 17. Safe4Rail system components.

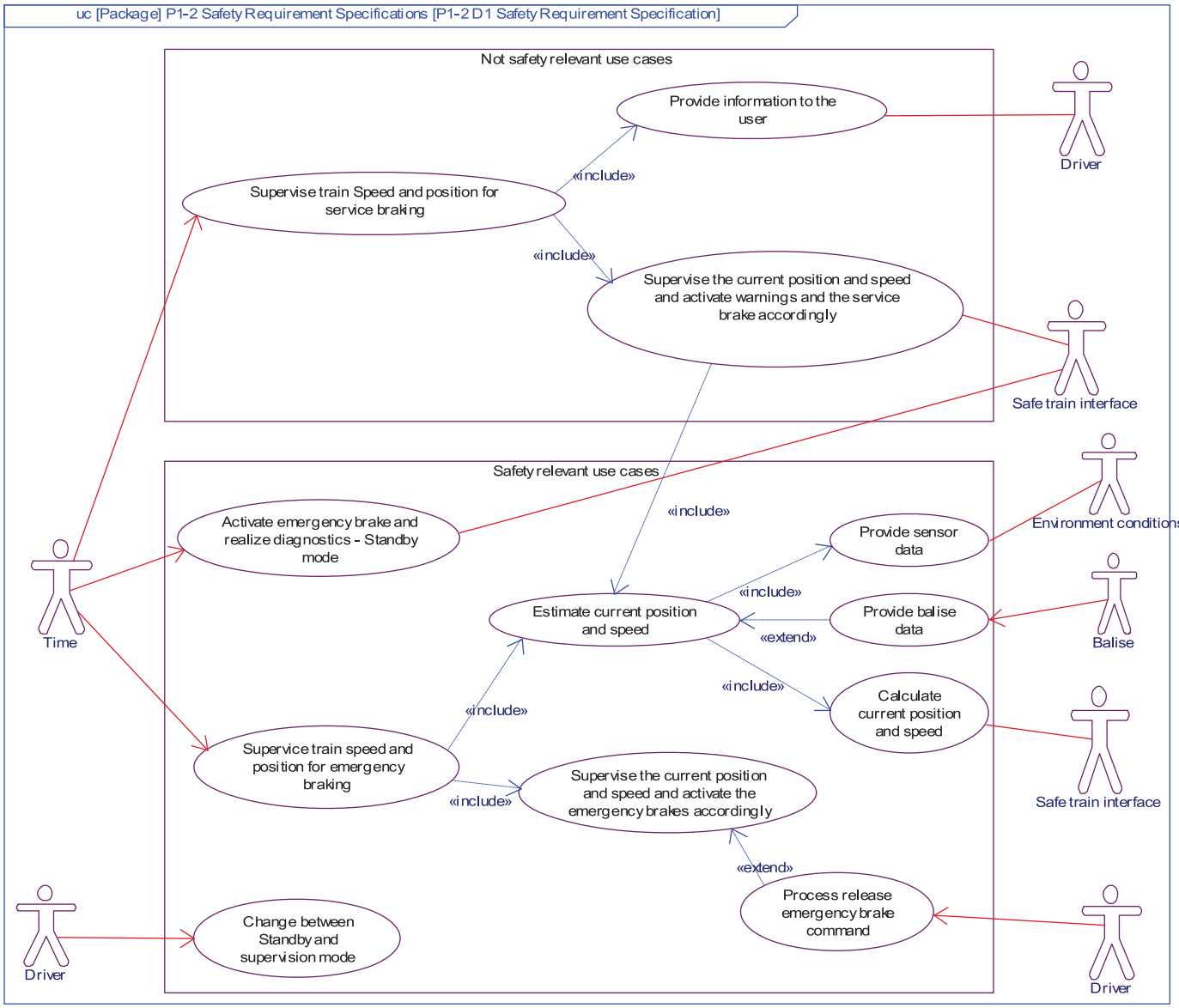

Fig. 18. Part of a Safe4Rail use-case diagram.

(a) Estimate current position and speed.
   i. Obtain sensor data.
   ii. Obtain balise data.
   iii. Calculate current position and speed.

(b) Supervise the current position and speed and activate warnings and brakes accordingly.
   i. Process release emergency brake command.

(c) Provide information to the user.

3. Switch between the standby and supervision modes.

Based on the previous analysis, the following safety requirements are specified and shown in Fig. 19. The requirements described below are a summary (with some revisited information) of the safety requirements:

1. (SIL4) Supervise train traveling speed and distance. The system shall ensure that the train's traveling distance and traveling speed do not exceed the maximum authorized safety values, which are the movement authority (MA) and speed profiles, respectively.

(a) (SIL4) Odometry. The system estimates the traveling speed and distance with bounded absolute errors (ABS DIST ERR MAX and ABS SPEED ERR MAX for a maximum distance between DIST MAX BALISE and a maximum speed of 500 km/h).

(b) (SIL4) Mode. The system safely manages modes and their transitions:
   i. No power. The system remains in a safe state.
   ii. Standby. The system remains in a safe state.
   iii. Supervision: The system ensures that the train traveling distance and traveling speed do not exceed the maximum authorized values for safety, namely, the movement authority (MA) and speed profiles, respectively. (This implies the execution of multiple sub-safety functions, such as 'communicate with control centers', 'limit supervision', etc.)

(c) (SIL4) Limit supervision: Limit supervision. The system updates the maximum distance and maximum speed profiles with received commands and compares the estimated traveling distance and speed (odometry) with these limits. If any 'safe authorized limit' is exceeded (distance and/or speed), the safe state is activated.

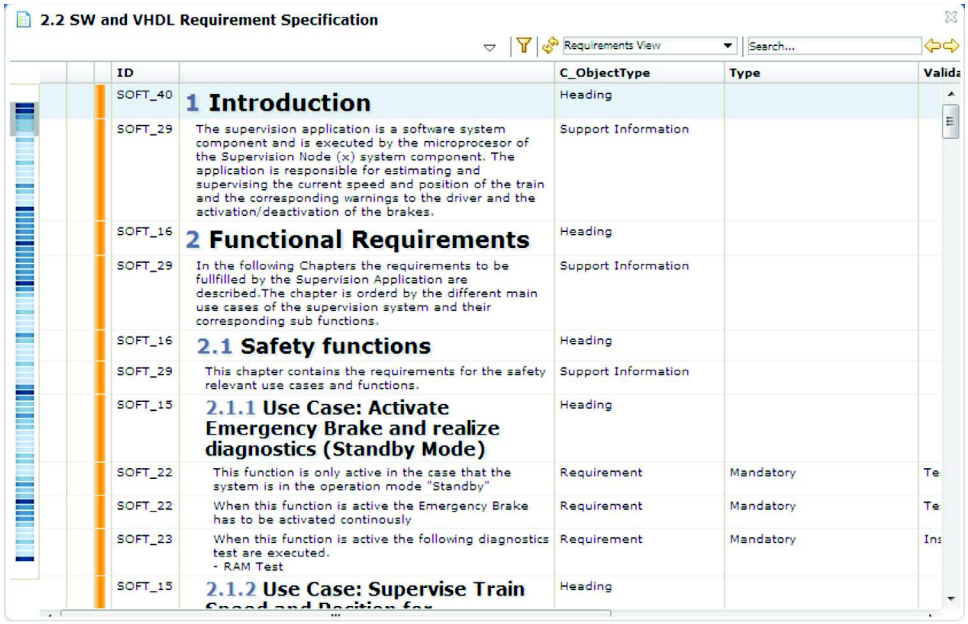

Fig. 19. A portion of the Safe4Rail requirements (software).

Table 1
List of patterns.

Pattern	Origin
Hypervisor	EN-50,129 (Table E.4; Separation of safety-related sys-tems from non safety-related systems)
Watchdog	EN-50,126, IEC-61,508–3 (Table A.11)
Triple Modular Redundancy (TMR)	IEC-61,508-2 (Tables 2 and 3), EN-50,129 (Table E.4)
Majority Voter	IEC-61,508-2 (Table A.2,A.3A.4)
Reciprocal Monitoring	IEC-61,508–2 (Table A.4),EN-51,028 (Fault Detection & Diagnosis)
Data Agreement	Book "Real-Time Systems:Design Principles for Dis-tributed Embedded Applications"
Black Channel - Safety Com-munication Layer	IEC-61,508–2 / IEC-61,784

(d) (SIL4) Rearm: Once the safe state is activated by the 'limit supervision' safety function, the system can be rearmed (release emergency brake) only when the 'train is stopped' and the 'driver commands rearm'.

2. The safe state is when the emergency brake is activated (which means that the system is in the safe state, i.e., the train is stopped).

5.1.3. Results

Here, we present the results of our case study. Because we elements of this study were discussed in Sections 3 and 4 to explain our approach, we provide only an overview of the outcomes of the case study (steps 3–6).

Definition of dependability patterns (step 3). In this step, the system architects analyze system safety requirements and identify possible safety patterns to be used. Table 1 presents the list of patterns to be used in the railway demonstrator. This list populates the repository of dependability patterns for the railway domain through the MDE tool set presented in Section 4.

The Arabion editor is used to create the corresponding set of seven patterns. Arabion uses a set of property libraries to determine the type of the pattern properties. Then, pattern publication into the repository is triggered by running the publication feature of Arabion. However, the publication feature requires that the validation tool be executed to guarantee pattern design validity. In the context of this study, we examine only conformance validity with regard to the pattern metamodel. However, formal verification may be applied using an additional verification framework (Rodano and Giammarc, 2013; Hamid et al., 2016). Finally, the repository management tool is used to define the relationships between the patterns.

Adaptation for a specific domain (step 4). The domain-specific perspective of the pattern is dependent on the solution/product; each (commercial) implementation provides different characteristics and features. In the context of our work, a railway-specific model is constructed based on previous patterns using the tool suite. Here, we present a subset while focusing on the specific railway realizations.

- *Hypervisor.* The hypervisor virtualizes the real platform hardware to allow the application and the operating system (partition) to "feel like" they are executed in a standalone manner on real hardware. The commercial hypervisors are XtratuM,[11] RTS hypervisor,[12] and Lynx hypervisor.[13]
- *Watchdog.* The watchdog checks whether the monitored application or a particular (safety) application process is running into its time base or executed in the correct order (Powel, 2003). Two possible watchdog architectures are possible: (1) with a time window (or time range watchdog), which defines two different time periods for the monitoring process (i.e., Tlow and Thigh), and (2) without a time window, where a unique time-out value is considered (Tmax).
- *Triple Modular Redundancy (TMR).* This is a fault-tolerant redundant architecture in which three computational channels

[11] www.virtical.eu.
[12] www.caf.net.
[13] www.alstom.com/power/renewables/wind/.

perform a safety computation and the result is used to produce a single safe output. The domain-independent version is defined according to the IEC-61,508-based fail-safe embedded system that provides a single random hardware fault-tolerance (HFT = 1; IEC-61,508-2, Table 3). The railway-specific version is defined according to the EN-50,126-based fail-safe embedded system with a "composite fail-safety" technique (EN-50,129 B.3.1 Effects of single faults, Table E.4).

- *Majority Voter.* According to IEC-61,508, the majority voter is a safety technique that provides a safe output based on the majority principle (*M* out of *N*, e.g., 2*oo*3), masking failures in one of at least three hardware channels. In the railway-specific application, two new methods are added to represent two major majority vote types, i.e., the bit-exact voter and the approximate voter. The bit-exact voter compares all input data bits, considering only inputs that are bit-identical as being equal. The approximate voter establishes a criterion to decide when inputs are considered equivalent to attain majority voting results.
- *Reciprocal Monitoring.* Also known as "monitored redundancy" (IEC-61,508), is a monitoring and checking pattern between *N* data providers. If one of the providers is sending an erroneous data stream the other entities will detect and accuse it of having a fault. In our context, we consider the *MooN* reciprocal monitoring which is based a *MooN* Majority Voter to have the result of the voted data by the other computation units of the system. Once the vote has concluded it compares its own data and the data of the rest participants, if *M* or more computation units had a coherent answer to the vote the system is running in an acceptable way. If less than *M* computation unit have a coherent answer the system is running in a not acceptable way, or if the own computation unit has no coherent answer, the computation unit is turned to a fault-safe state.

We use the same process flows as applied for the domain-independent representation, although the appropriate features of the toolset are used to create and deposit the corresponding domain-specific representations of these patterns into the repository.

System developer perspective: reuse of existing dependability patterns (steps 5 and 6). This process is relevant to both step 5 (patterns identification and tailoring) and step 6 (integration). The first activity in this process is to construct an access tool for the railway domain, such as the one presented in Fig. 14. In our case, the target format is a subset of UML that can be imported using Rhapsody. Thus, the set of railway patterns is imported into the Rhapsody environment as a set of packages using the railway access tool (see Fig. 10).

In the safety railway process model, which is depicted in Fig. 20, the developer begins with engineering requirements and subsequent system specifications. In each phase, the system developer executes the search/select task on the repository to tailor appropriate patterns for the modeling environment using the access tool and integrates these patterns into the application models following an incremental process. Fig. 21 shows the simplified flow for an iteration within a software architecture definition phase. Moreover, the system developer can use the pattern designer tool (Arabion) to develop custom solutions when the repository fails to yield appropriate patterns during this phase.

The safety concept diagram (see Fig. 22) provides a high-level architectural perspective in which major safety requirements, techniques and concepts used to augment the safety of the system are represented. The practical application of our approach begins at this point, where the system architect and RAMS (Reliability, Availability, Maintainability, and Safety) architect open the access tool and log in.

1. Both the system architect and the RAMS architect analyze the system safety requirements and identify the possible architectures and safety techniques to be used. They begin defining the European vital computer (EVC). They identify triple modular redundancy (TMR) as a design pattern of interest to reach a SIL4 via a "composite fail-safety" technique (EN-50,129 B.3.1 Effects of single faults, Table E.4). As shown in Fig. 14, they type and search the TMR to find the most suitable design pattern instantiation for this phase. They analyze all possible options, select "TMR@System" and click "Export". By clicking the "Export" button, the pattern is exported as a Rhapsody-referenced package to the selected directory.
2. As shown in Fig. 14, once the pattern is imported in Rhapsody as a package, a project tree is generated and its artifacts are available for use in the project
3. Both the system architect and the RAMS architect continue refining the safety concept (see Fig. 22), searching for design patterns in the access tools and importing them whenever a suitable design pattern is found:
   (a) The TMR requires an external safety hardware majority voter (its MajorityVoter) implemented with two independent majority voters. Majority voters require six digital outputs of the TMR (three per majority voter, with each digital output controlled by a different computation channel) to generate two independent emergency brake majority commands to the train interface (see Fig. 22).
   (b) The TMR is connected to the following safety and non-safety related subsystems (see Fig. 22). It follows the "separation of safety-related systems from non-safety-related systems" technique described in EN-50,129 (Table E.4).
      i. Odometry input sensors (its OdometrySensors SC): A simplified odometry requires three (non-safety related) encoders (itsEncoderSensor SC), each of them connected end-to-end to a single computational channel
      ii. The (safety) balise reader (itsBaliseReader SC)
      iii. The (safety) DMI (itsDMI SC)
   (c) Initial decisions regarding the internal structure of the TMR (see Fig. 22):
      i. A black channel is selected to enable communication between the computation channels. Therefore, a safety communication layer pattern is already integrated.
      ii. A data agreement protocol can be used to reach an agreement regarding the input values to be used by the computation channels (input sensors are connected end-to-end to a single computation channel). This enables bit-exact execution of software to simplify diagnosis.
   (d) As shown in Fig. 22, the replicated software executes the supervision safety function (Supervision-SystemPIM) and safety techniques ("DataAgreement" and "SafetyCommLayer"). The system supervision function is based on three main functionalities (sensor reading, supervision, interface with balise reader and interface with the user (driver)). The main safety techniques are "TMR software redundancy", "data agreement" and "safety communication layer".

The system architecture shown in Fig. 23 specifies the Safe4Rail system decomposition and the relationship between the different blocks that compose the system. At this stage, the system architect makes several architectural decisions (based on the safety concept and requirements) and accesses the repository to search and import suitable refined and specialized design patterns of interest:

1. The TMR is implemented with the following features:
   (a) Three homogeneous nodes ("SupervisionNode") connected to two Ethernet switches in star topology and composed of the following:

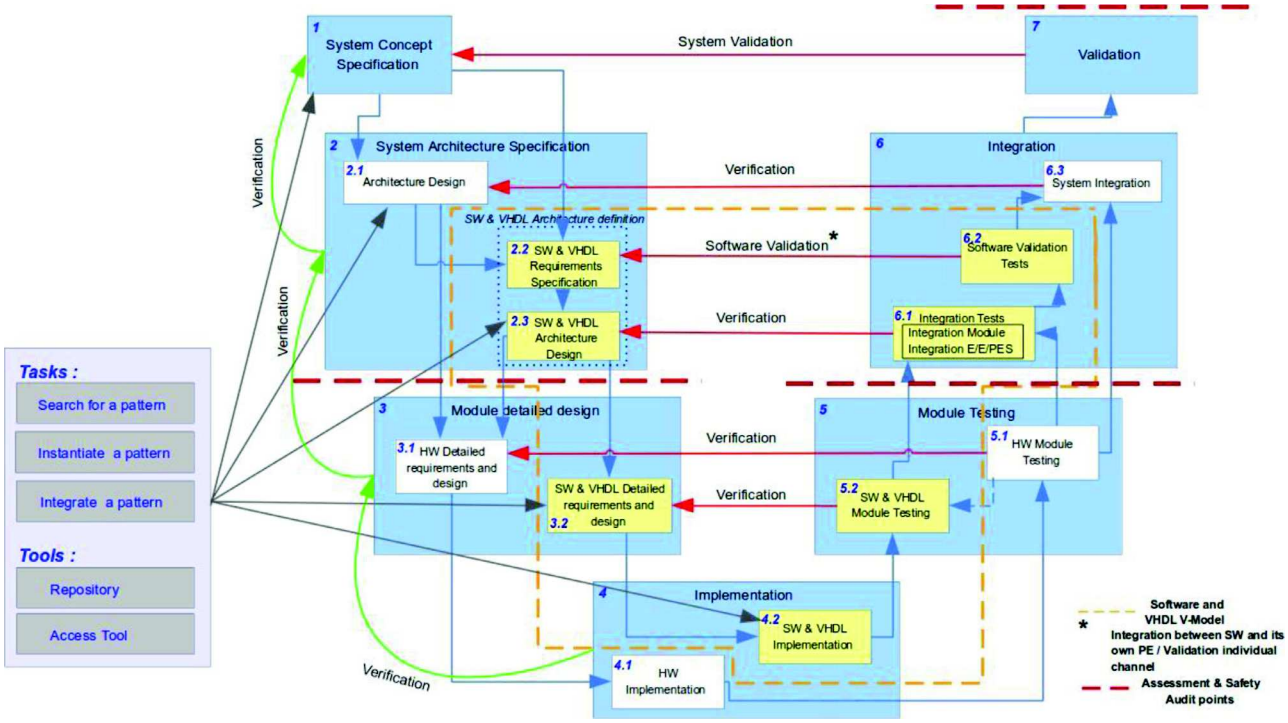

Fig. 20. An overview of the railway reference process model.

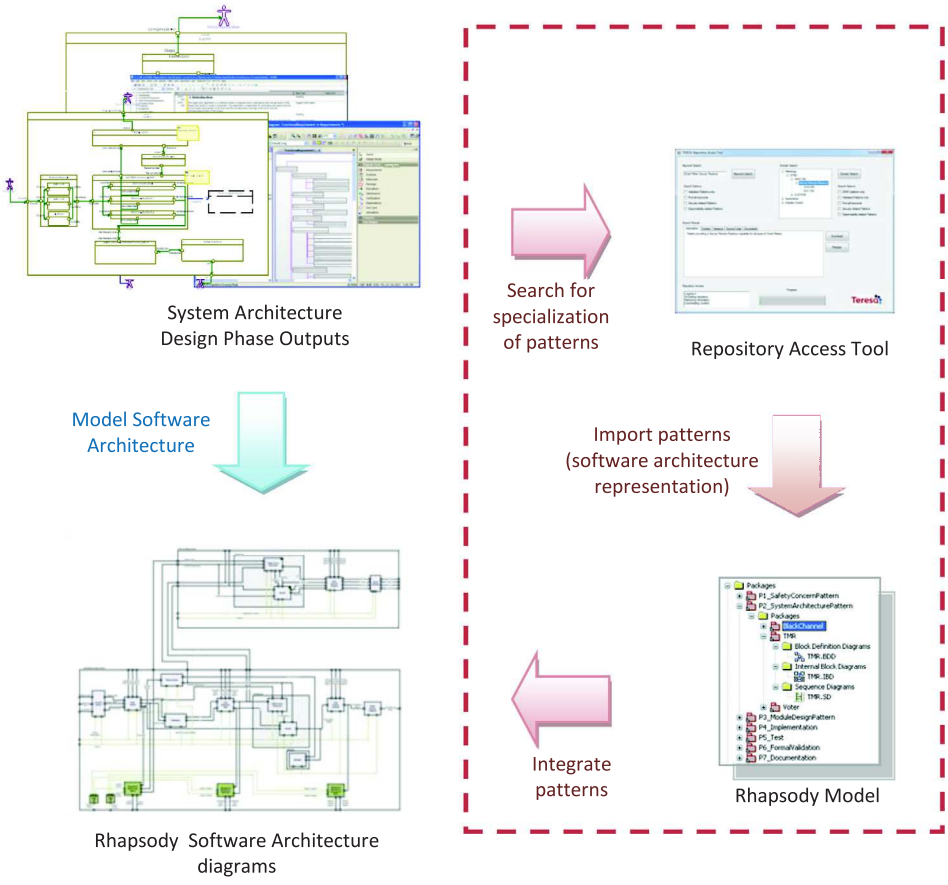

Fig. 21. Simplified process flow using our approach at the software architecture definition phase.

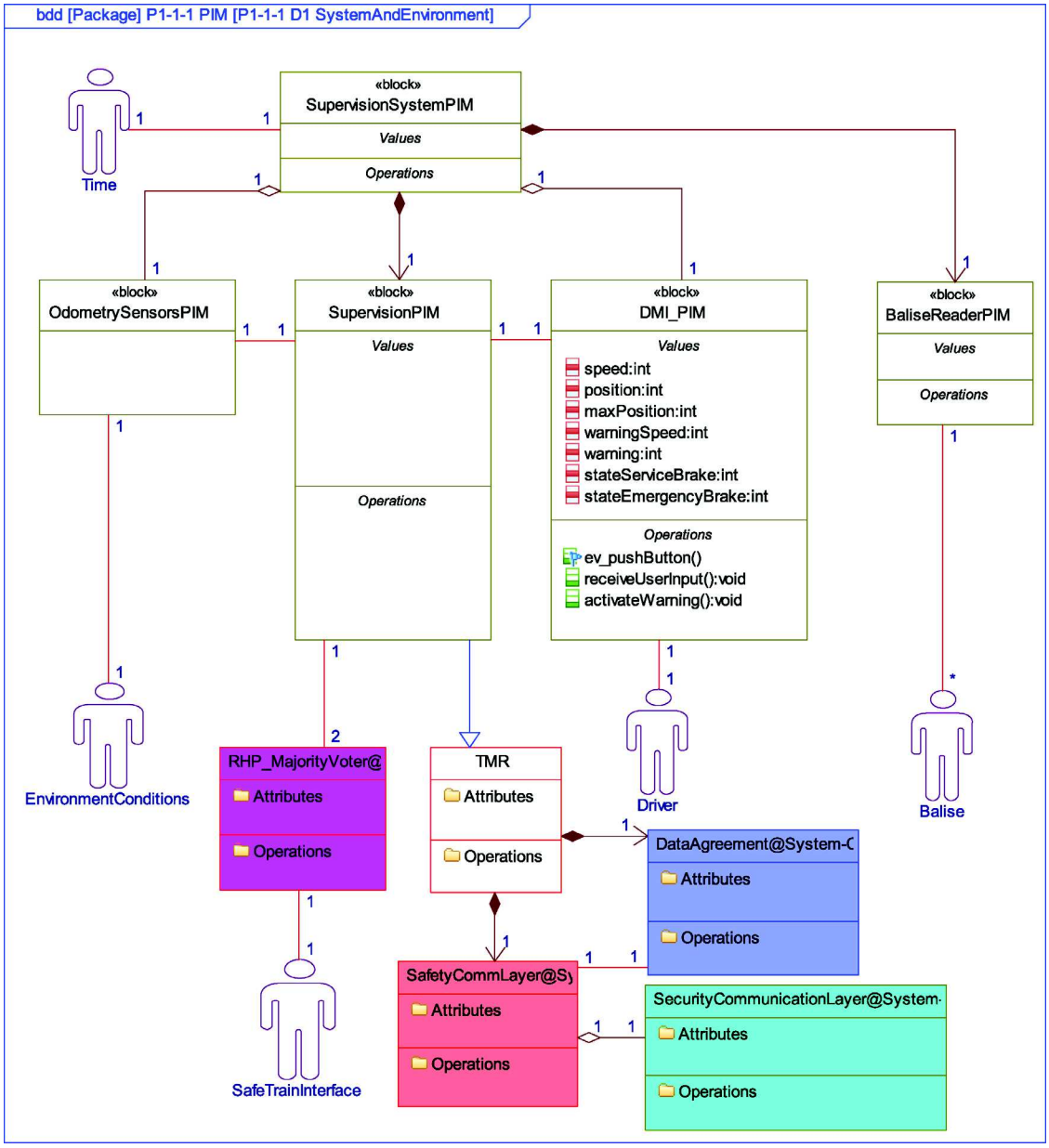

Fig. 22. Overview of the Safe4Rail safety concept.

i. a computing unit microcontroller (“ComputingUnit”, “Microcontroller”),
ii. an “FPGA” that provides safety and non-safety related inputs/outputs (“IO Safety” and “IO No Safety”),
iii. a watchdog as a pattern,
iv. a software application (“SupervisionApplication”) executed at the computing unit that integrates the safety communication layer as a pattern to support safe communication between the replicated communication channels,
v. the use of a hypervisor (“DI SA BL Hypervisor”) as a pattern to enable integration in a single software application (“SupervisionApplication”) that contains the following:
   - safety software, such as software functionalities and safety techniques previously described in the safety concept, and
   - non safety-related application software such as the communication stack of the black channel.

(b) Two Ethernet switches (“EthernetSwitch”) associated with a black channel.
2. A single (safety) balise reader (“BaliseReader”).
3. A black channel communication protocol (Ethernet/EtherCAT).

During these phases, new design patterns are imported from the repository based on the system architect decisions. For example, the selection of a supervision node with a single microcontroller results in the use of a hypervisor to integrate safety and non-safety-related software in a single microcontroller.

The software architect continues refining the software architecture. The emergency supervision safety function shown in Fig. 23 is refined, leading to the software architecture shown in Fig. 24. Then, additional software architectural decisions and analyses are made; additional safety techniques are identified. The software architect accesses the repository to search and import suitable refined and specialized design patterns according to the identified techniques and integrates them. Therefore, new patterns that are not represented in previous phases are initially introduced at this phase.

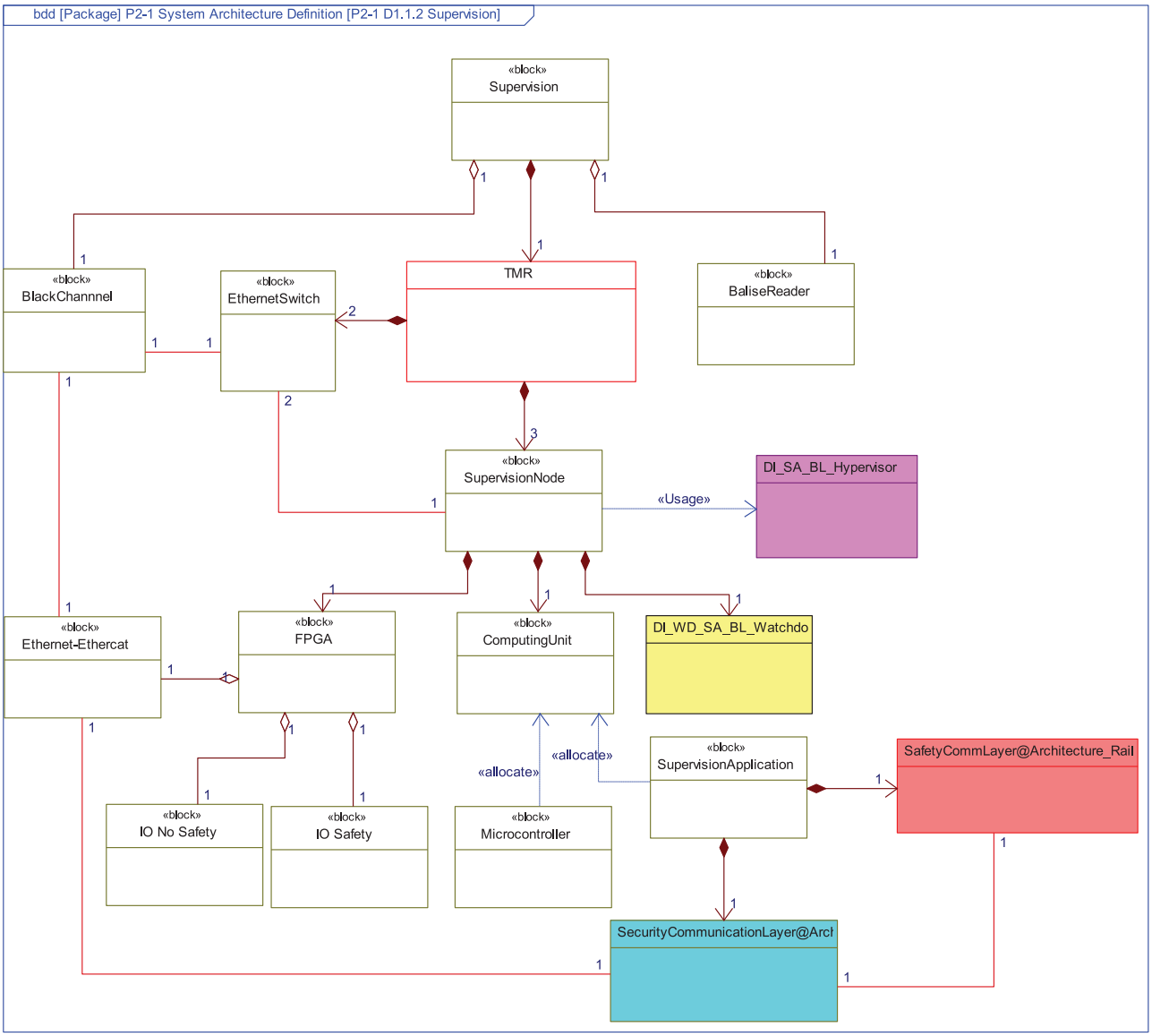

Fig. 23. Overview of the Safe4Rail system architecture.

This is the case of the reciprocal monitoring pattern, which is only over interest at this stage. When the software architect must define how to monitor the nodes, he/she can use the access tool to determine whether there is a pattern to implement this functionality. The reciprocal monitoring pattern is selected, imported and integrated in the software architecture diagrams.

5.1.4. Discussion

This subsection has described the adaptation of railway processes to incorporate the proposed approach centered around a model-based repository of dependability patterns. The procedure used for developing the case study closely followed the approach described in Section 3. Given the dependability pattern requirements, a conceptual model is built that fulfills these design requirements. The next step is the creation of a Domain-Specific Modeling Language (DSML) for the specification of pattern models. This work was done by the author. Then, the dependability expert, with the help of the system and software engineering expert, defines the dependability patterns and begins populating the repository. Then, a domain process expert adapts the patterns into a format that is suitable for the system development process. Finally, a domain engineer reuses the resulting pattern models that have been adapted and transformed for the given engineering environment to develop a domain application. For the purpose of our study, we have developed Semcomdt (SEMCO Model Development Tools) as an MDE tool chain to support all steps of our approach using EMFT. The repository is implemented using Java and the Eclipse CDO framework.

The creation of the conceptual model of the dependability pattern required approximately 4 person months. The creation of the DSML for the dependability pattern took approximately 6 person months. The implementation of the MDE tool chain took 12 person months. The construction of the domain model (i.e., Safe4Rail system model) took another 3 person months. The process of populating the repository took one month. The proposed tool chain is designed to support the proposed metamodels, and hence, the tool chain and the remainder of the activities involved in the approach may be developed in parallel. This activity needs to be performed only once for a given set of domains. We expect the effort for the creation of a DSML and the development of tools to be less for future applications, as we had to address several technical details in relation to using EMFT and CDO in our first application. We tested

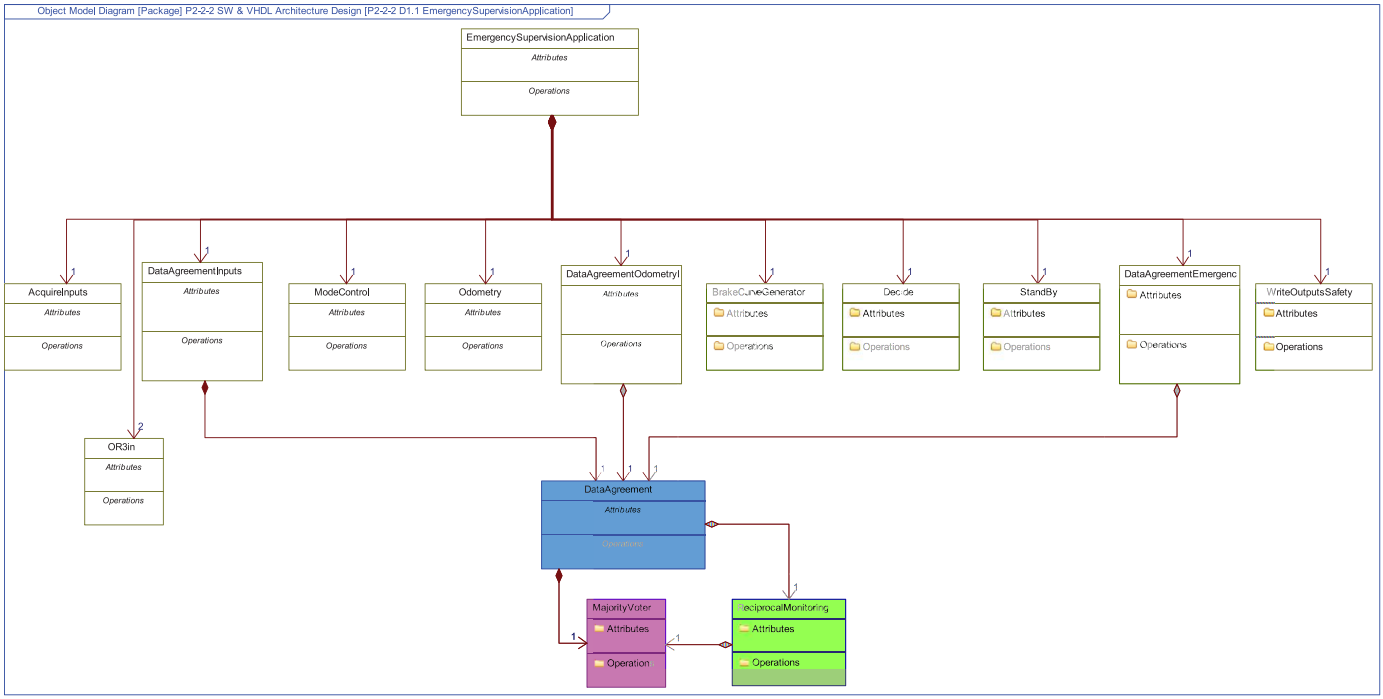

Fig. 24. Overview of the Safe4Rail software architecture.

which of the provided tools were best able to support pattern integration, thus assisting in the engineering process. In this context, the extendibility of the pattern repository with new patterns and the extendibility of existing patterns were observed. Furthermore, we evaluated to what extent such patterns are useful in increasing engineering productivity, as described in the following section.

5.2. Survey

After the completion of our case study, our approach and the solution of our case study were presented to industry practitioners through a survey to collect feedback. This survey presents a preliminary evaluation of the proposed approach using Key Performance Indicators (KPIs). A performance indicator or key performance indicator (KPI) is an industry term referring to a type of performance measurement. KPIs are commonly used to evaluate the success of a particular activity, project or company. The objective of the survey is to determine whether domain experts agree on the benefits of adopting the pattern-based approach in a real industrial context. In the following, we present and discuss the design and results of this survey.

5.2.1. Context and description of the methodology for experiment

The approach and the solution of our case study presented in Section 5.1 were presented to the industry practitioners for evaluation. The purpose of this survey is to provide an overview of the software architecture for a small but sufficiently complex system that can be used to illustrate our proposed approach. Therefore, the estimations given for the railway domain provide two values with associated argumentation:

1. *Large and complex safety systems*, such as on-board ERTMS/ETCS, and
2. *Intermediate safety systems*, such as traction control safety supervision.

Among the fifteen participants in the experiment, nine are TERESA members from the railway domain. Moreover, five dependability experts (DEPs) participated in the survey: three from the railway domain and two from other domains. Six software engineering (SENs) experts participated in the survey: four from the railway domain and two from other domains. In addition, four project managers (MGs) participated in the survey: two from the railway domain and two from other domains.

The participants also knew that the experimenter did not have any influence on their job evaluation scores or on their career evolution and progress, as the experimenter was external to their organizations. Statistical analyses, mainly for the averages of the values provided by the participants, are performed by peers on anonymized data (both subjects and scales) independently from the experimenter. Moreover, the researcher agreed to use the survey only for research purpose as anonymized results.

We have created a questionnaire shown below, about the background of the subjects related to dependability in general. We explained in the questionnaire that “dependability-related experience” covers the following: (1) participating in the development of projects related to dependability; (2) constructing and reviewing dependability project reports and deliverables; (3) attending tutorials and workshops on dependability engineering processes and standards; and (4) self-reading of dependability engineering standards.

Q1. Is dependability engineering an important aspect of your job?

- Yes
- NO

Q2. How much experience do you have with dependability engineering related activities?

- Less than 6 Months
- More than 6 months but less than 12 months.
- More than 1 year but less than 2 years
- More than 2 year

The purpose of our study is to address the following two research questions:

- **RQ1:** Does the proposed approach reduce the effort involved in developing a new application (design and implementation)?
- **RQ2:** Is the effort involved to engineer a new version of an existing application to add a dependability property acceptable?

To address these two questions, we consider KPIs as a collection of metrics for quantifying the objectives of the approach, monitoring the related activities and assessing the expected results. The participants were asked to scale their estimation on a scale from 0% to 100%, 0% being the lowest and 100% the highest value of estimation regarding the different KPIs. The averages of the values provided by the participants are calculated for the analysis. To answer **RQ1**, we evaluate whether the approach leads to a reduced number of steps in the engineering process or to their simplification, and we assess whether the domain experts agree on the benefits of adopting the approach in a real industrial context for the development of new applications. For **RQ2**, we measure the ability of the approach to integrate dependability solutions into existing products.

In the following, KPIs are presented with an estimated target value submitted at the beginning of the case studies, and real values are estimated upon completion of the case studies. Both values are averages because the individual values highly depend on certain factors, such as the size of the project, the product requirements, the expertise of the engineering team, the use of the approach facilities and the availability of the appropriate patterns. In both cases, KPIs are estimated based on previous knowledge of implementing such systems or equivalents. A description of the considered constraints and assumptions is discussed below.

1. *Development costs.* Cost reduction is at the forefront of the challenges associated with system engineering. Nonetheless, methods to reduce costs differ greatly. Reducing the cost of consumer electronics is typically bound to the hardware, whereas cost reduction for industrial embedded systems is primarily related to the engineering times. In the context of our experiment, the railway domain follows the second criterion because only a few hundreds/thousands of devices are manufactured. Two examples of KPIs that can be used to measure the reduction in cost are described as follows:
   - *(K1) Overall engineering cost.* Cost to develop a new application (design and implementation) with dependability requirements using a target reduction of 10% to 20% (on average).
   - *(K2) Percentage of reused code.* Amount of code reused from existing patterns in a new development with dependability requirements using a target reduction of 20% to 60% (on average).
2. *Time to market.* Market pressures are forcing a continuous evolution of the systems present in the market. Consumer electronics is a good example, where every other day a new product appears. In general, there is a dual timing pressure, i.e., reduce the time to market of systems and reduce the time to market of new models of systems. Here, two example metrics to measure this reduction are described:
   - *(K3) Engineering time.* Time to develop a new application (design and implementation) with dependability requirements using a target reduction of 10% to 25% (on average).
   - *(K4) Re-engineering time.* Time to engineer a new version of an existing application to add a dependability property with an estimated target reduction of 40% (on average).
3. *Product quality.* Product quality is the degree to which customer needs and expectations are satisfied. Here, product quality is related to the effectiveness and efficiency of the reused code from prior patterns. We provide two examples of metrics to measure this quality:
   - *(K5) Errors in reused code.* Number of errors appearing in code reused from patterns using a target value of 10 as a factor of the probability of errors in reused code with respect to new code.
   - *(K6) Code quality.* Readability and compliance of the reused code with the required standard with an estimated target value of compliance of 100%.
4. *Post-deployment support.* The maintenance of the system while it is in operation typically spans several years. Railway systems are designed to be in operation for more than 20 years; thus, there is a need to provide support and reduce maintenance both effort and costs. There is a wide degree of diversity among the different domains of interest. The maintenance expectations are lower in consumer electronics, whereas they are critical in the railway domain. The following describes two example metrics for post-deployment support:
   - *(K7) Maintenance cost.* Total cost and effort associated with bugs being detected after deployment using a target reduction of 10% to 20% (on average).
   - *(K8) Incident response.* Time and effort for identifying affected products from reused code/concepts with an estimated target reduction of 30% to 50% (on average).

As discussed below, some of these categories are closely related to each other. For example, every improvement in time to market, product quality or post-deployment support has a positive impact on cost.

5.2.2. Experimental validity

Threats to internal validity. To obtain scientifically sound results that enable comparison between metrics associated with a standard development approach and those associated with our approach, it is necessary to perform a study in which comparable systems are designed in parallel by equivalent engineering teams. However, this method is not feasible within the scope of our work or within the scope of the TERESA project because of a lack of resources required to develop the system addressed in the case study twice. Moreover, it is even less feasible to obtain such metrics from the development of real products, such as ERTMS/ETCS, for which the development and certification costs are several tens of millions of Euros and confidentiality is an issue. Therefore, metrics for comparison are estimated based on previous knowledge of the implementation of such systems or their equivalents. This estimation is thoroughly elaborated and discussed, including descriptions of the considered constraints and assumptions. Therefore, a notable threat to the internal validity of this study is the possible lack of a common understanding of concepts between the researchers involved in the development of the approach and the participants. To minimize this threat, the quality of certain elements of the training and preparation procedures could be improved. One would expect that in a real environment setting, an engineer using any engineering method would have some experience with that method. Another potential threat to the study's internal validity is the influence of relationships between the researchers and the participants because both are involved in the TERESA project. This issue is addressed by including six participants out of the TERESA consortium. Finally, the definition of the target values of the KPIs may influence the results because the participants may not respect the evaluation context. This issue could be completely resolved only if several evaluation methods are configured to define the target values.

Threats to external validity. As noted by the authors of (Wohlin et al., 2000), external validity may be used to measure whether the results of a certain experiment can be generalized be-

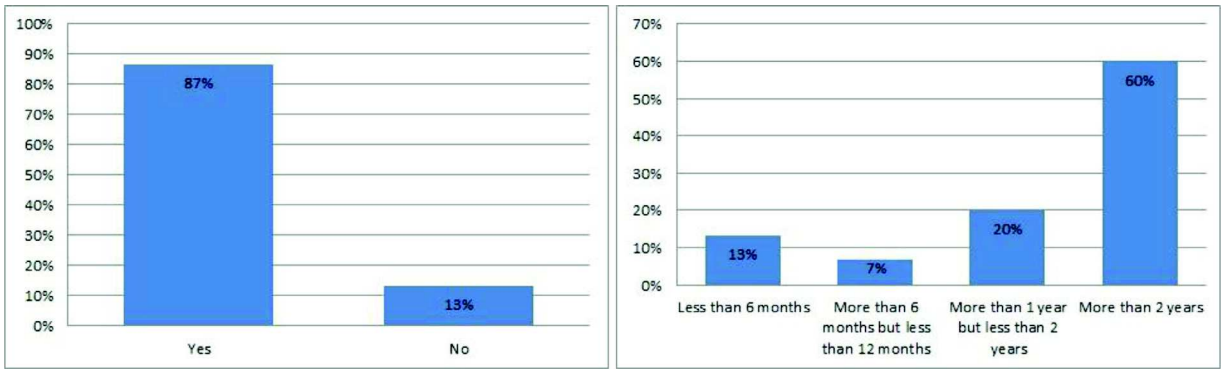

Fig. 25. Subject dependability engineering background (Q1 and Q2).

yond the experimental context. Because the proposed approach is targeted toward software engineers, one threat to external validity lies in the skills of the participants. Most participants possess dependability expertise. Therefore, it would be constructive to evaluate the approach using participants who are not experts in dependability engineering. Nevertheless, six participants in the conducted experiment possess high software engineering skills, and the remaining participants previously participated in the development of several software projects with a focus on dependability. Therefore, all the participants may be considered to be at an advanced stage in their practice as software engineers; therefore, they can be regarded as comparable to the targeted software engineering population. Other obvious external validity threats are the size of the project, the product requirements, the use of the tool facilities, and the availability of appropriate artifacts during development. However, these issues are not relevant because the size and context of the tasks addressed in the experiment are similar to those of typical cases in the industry. In addition, the participants were required to complete their assigned tasks during the time allotted for the experiment. However, in an industry context, software engineers are typically provided sufficient time to address the development of a software system. Finally, the significance of each KPI and its related class of responses may differ between different domains. To minimize the influence of this concern, appropriate metrics must be defined and ranked for each application domain. Thus, the collected metrics should be objective, measurable, relevant to the targeted project and comparable to the situation before using the proposed approach. Success is occasionally defined in terms of making progress toward goals; however, more often, success is simply the repeated achievement of some level of an operational goal (e.g., zero defects, 10/10 customer satisfaction, etc.). Accordingly, choosing the right KPIs depends on having a good understanding of what is important to the domain of interest or the organization.

5.2.3. Result and discussion

Fifteen people completed the questionnaire. A broad range of industry sectors was represented, with respondents from railway, automotive, metrology and software development sector. Based on the responses obtained, depend- ability engineering was an important aspect of the job for all but two participants (left part of Fig. 25). Among these two participants, one is working as project manager and the other as software engineer. Both of them would be engaged in dependability engineering activities in the future and hence had started to develop their skills in this field. Overall, 60% of the participants had over two years of dependability engineering experience and a further 20% had at least one year of experience with dependability engineering (right part of Fig. 25). All participants are familiar with modeling tools.

With regard to the overall engineering cost, we estimate that the development of large and complex systems (ERTMS/ETCS), respectively intermediate systems, is reduced by an average of 12.5%, respectively 30% (left part of Fig. 26). The overall engineering cost is reduced as follows. During the safety concept, system architecture, software architecture and module detailed design phases, a reduction in the time to formalize and document the design is observed using already-developed design patterns that include all necessary safety information and reducing the effort required to document detailed descriptions by hand. Moreover, a reduction in the time and effort associated with verification is also observed because the design patterns are already verified and provide a common understanding for both designers and verifiers. Finally, a reduction in the time and effort associated with RAMS analysis is observed because the design patterns are already verified and provide a common understanding for both designers and RAMS engineers.

The reduction in development cost at the implementation and test phases may be justified by the fact that design patterns are already implemented as software (meeting all required coding standards), and unit tests are already implemented by means of generation mechanisms provided by the modeling tools. Therefore, each design pattern is already tested (even unit test results are stored), and guidelines for integration tests might be provided.

Finally, verification and validation activities must be extensively tested in different implementations, assuming that design patterns are already used in several applications. Therefore, the real test unit coverage is higher, and the probability of finding an integration test or validation bugs associated with the design pattern is very low. This reduces the verification or validation effort where the correction of bugs is time- and effort-consuming. Note that verification and validation consumes approximately 50% of the overall project cost for high-integrity-level systems (e.g., SIL4). Safety design patterns, such as those selected for the case study (data agreement, safety communication layer, etc.), provide a safety foundation that is difficult to verify and expensive to validate in real system developments. They require the participation of multiple replicated communication channels, temporal constraints, etc., as opposed to system-specific safety functions that commonly perform computations. Therefore, the availability of these key foundational pre-verified safety design patterns can significantly reduce the verification and validation effort for a given system.

With regard to the percentage of reused code, we estimate that during the development of large and complex systems (ERTMS/ETCS), resp. intermediate systems, code is reused at an average of 12.5%, respectively 43% (right part of Fig. 26). The selected case study, namely, ERTMS/ETCS on-board railway signaling acting as a large and complex safety system, is a representative SIL4 safety embedded system in which multiple design patterns can be

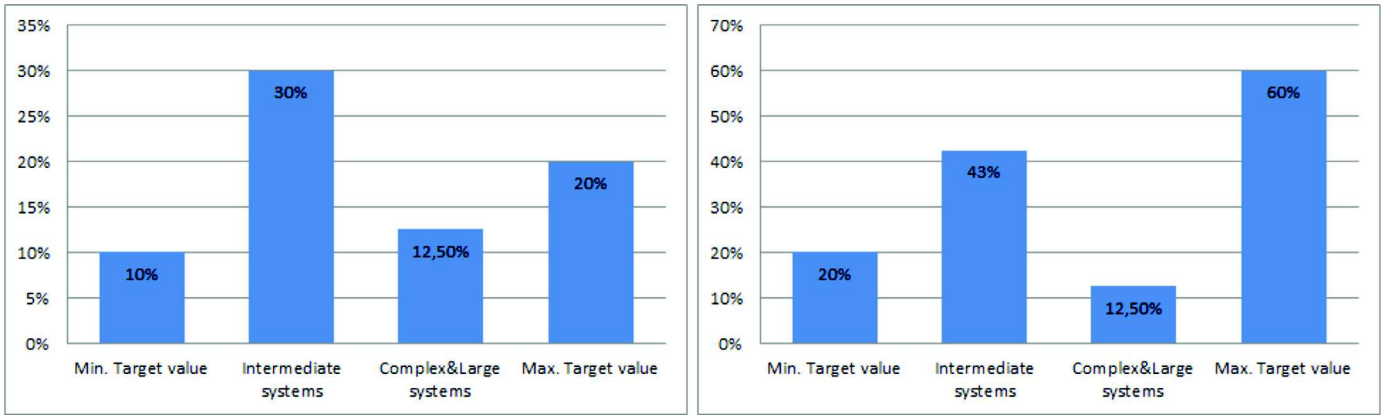

Fig. 26. Development costs (K1 and K2).

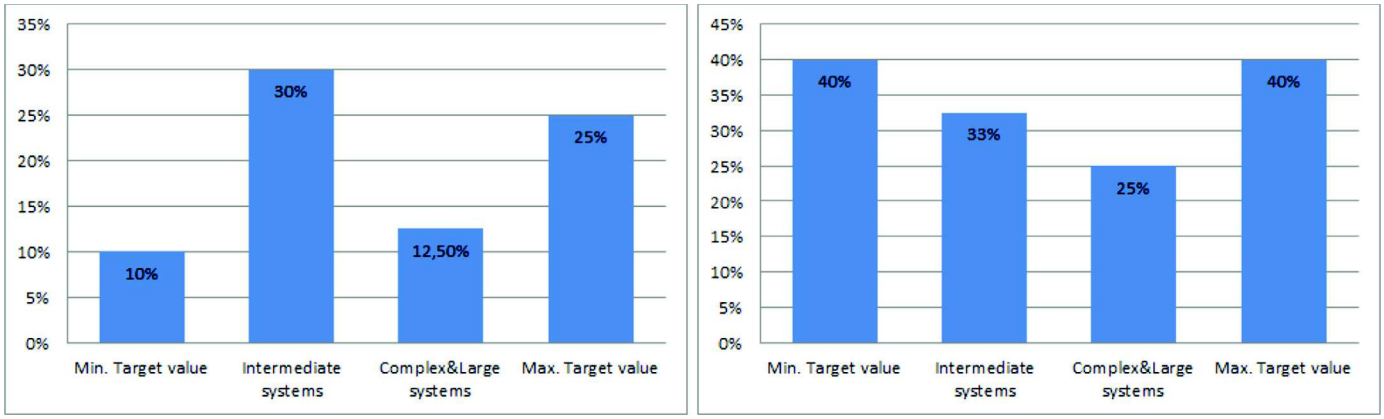

Fig. 27. Product quality (K3 and K4).

instantiated. However, ERTMS/ETCS is a highly complex system in which most of the software application implements the safety and functional requirements established by the standard for interoperability. Selected and integrated design patterns provide the safety skeleton and safety architectural foundation of the system (key foundation), where the system-specific application is deployed and executed. However, the ratio between system-specific software and software that can be provided by design patterns is less than 0.1.

This is a paradox because other safety subsystems of the train that perform intermediate complexity safety functions might be developed with a much smaller set of design patterns, although the ratio of design-pattern-based safety software compared to system-specific safety software might be at least one to one. For example, the safety function of a railway traction system (SIL2) acting as an intermediate safety system has an intermediate complexity. It must compare already acquired current, voltage and temperature measurements with given minimum and maximum thresholds and perform a small set of coherency checks on the measurements.

With regard to engineering, we estimate that the development of large and complex systems (ERTMS/ETCS), resp. intermediate systems, is reduced by an average of 12.5%, respectively 30% (left part of Fig. 27).

With regard to the re-engineering, we estimate that the development of large and complex systems (ERTMS/ETCS), respectively intermediate systems, is reduced by an average of 25%, respectively 33% (right part of Fig. 27).

In this study, product quality is measured by two metrics, i.e., errors in the reused code and code quality. For the first metric, it is estimated that a reduction by a factor of 10 in the probability of errors in re-used code with respect to new code can be achieved and even surpassed if the design pattern is already extensively used in multiple applications. Pre-engineered safety design patterns are already verified and probably being used extensively in multiple applications in which identified bugs have been previously corrected and the design pattern has been updated. With regard to the code quality, safety design patterns developed by/for an end-user should be 100% compliant with the required standards.

After developing the demonstrator and considering the previous estimation of K2 (percentage of reused code), we estimate that the maintenance cost is reduced in large and complex (ERTMS/ETCS), resp. intermediate systems, by an average of 12.5%, respectively, 25% (left part of Fig. 28). An example is the reduction of complex incident responses associated with the operation of safety replicas (e.g., data agreement, safety communication layer, etc.) that require a considerable amount of time to be analyzed and solved.

After developing the demonstrator and considering the previous estimation of K2 (percentage of reused code) and that safety design patterns provide key foundational patterns for the development of safety systems, we estimate that the time and effort for incident response is reduced in large and complex (ERTMS/ETCS) systems, resp. intermediate systems, by an average of 20%, respectively, 40% (right part of Fig. 28). An example is the reduction of complex incident responses associated with the operation of safety replicas (e.g., data agreement, safety communication layer, etc.) that require a considerable amount of time to be analyzed and solved.

A comparison of the resulting KPIs in the two cases is shown in Fig. 29, where the estimated KPI values for the newly proposed approach at the end of the case study are presented. From this com-

Fig. 28. Post-deployment support (K7 and K8).

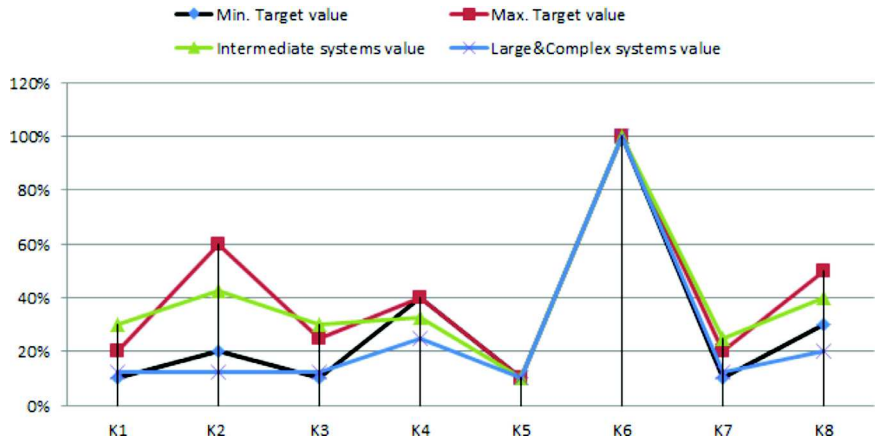

Fig. 29. Intermediate, large&complex safety systems.

parison, it appears that using the dependability pattern-based approach brings significant advantages in dependability engineering, especially for intermediate safety systems.

Embedded system development projects in the railway domain are commonly developed by multidisciplinary teams that can be geographically dispersed. Therefore, the existence of a repository with pre-engineered patterns (designed and verified) can reduce the time to market (and overall project cost), ease the complexity management, reduce the probability of systematic faults and increase the robustness of the developed products. Such effects are primarily related to each design pattern providing a concise representation of a concept (e.g., redundancy, as specified by IEC-61,508), and/or well-known and pre-engineered solutions that fit naturally within the company's engineering lifecycle.

The selected design patterns defined in the context of this case study target the development of (railway) safety embedded systems and provide either common IEC-61,508-based safety techniques (e.g., majority voter) or representative safety solutions (e.g., hypervisor). Instantiating and thereby reusing existing patterns reduces the cost to develop an application. Therefore, it becomes possible to produce results faster and to reduce staff efforts using proven solutions and reusing test cases instead of hiring expensive dependability engineers. Furthermore, by producing results faster, the time to market is reduced. In addition, to increase the chances of reducing development costs and the time to market in the future, additional patterns should be defined to extend existing patterns. It becomes possible to respond to the needs of customers and provide the desired patterns. Furthermore, the potential to exploit the approach over a long period of time becomes feasible.

6. Certification

Depending on the application requirements, different levels of assurance can be involved, including the most stringent certification (e.g., EN-5012 for railway systems and DO-178B for airborne systems). The certification process is required in all application sectors in which a safety function is included. The safety certification of a given safety system requires a demonstration that the system is safe for its purpose according to a given safety certification standard, e.g., EN-50,129 for the railway domain. This process requires at least the demonstration that the system has been developed according to a safety engineering process lifecycle that is compliant with the relevant standard. This lifecycle reduces the probability of systematic faults and ensures that the developed system is safe for its purpose using state-of-the-art techniques and measures that mitigate, detect and tolerate random and systematic faults.

6.1. Supported safety techniques

The safety techniques recommended by the IEC-61,508 (IEC, 2010b) safety standard, which can also generally be used in other domains, such as EN-5012X railway safety standards. The following provides a brief explanation of the given support for each technique:

1. *Fault detection and diagnosis.* As described by the relevant standard (IEC, 2010b), the goal is to detect faults in a system that might lead to a failure, thus providing the basis for countermeasures that can minimize the consequences of failures. This

approach is supported by safety techniques described by the standard (e.g., redundancy TMR) (IEC, 2010b) and state-of-the-art research (Kopetz, 2011), which provide well-known solutions to well-known problems that are suitable to be developed as safety patterns. Within our approach, the following cross-domain fault detection and diagnosis safety patterns are developed: TMR, majority voter, reciprocal monitoring, data agreement and watchdog.
2. *Modular approach* (see Table B.9 in (IEC, 2010b)). It is assumed that safety patterns are developed using a safety engineering process lifecycle, thus meeting the given recommendations for a modular approach (software module size limit, software complexity control, information encapsulation, etc.).
3. *Use of trusted/verified software modules and components (if available).* As described by the relevant standard (IEC, 2010b), the goal is to avoid the need for software designs and elements to be extensively revalidated or redesigned for each new application and to take advantage of a pre-existing software elements that are verified for a different application and for which a body of verification evidence exists. Safety patterns directly support this safety technique, providing trusted/verified software modules. Safety patterns should be developed according to an appropriate safety engineering process and safety integrity level, delivered with a body of verification and, whenever possible, a use/verification history (based on systems that have previously demonstrated successful usage).
4. *Semi-formal methods* (see Table B.7 in (IEC, 2010b)). It is assumed that safety patterns are developed using at least semi-formal methods, such as UML modeling, finite-state machines and sequence diagrams.
5. *Computer-aided design tools* (see B.3.5 in (IEC, 2010b). As described by the relevant standard, the goal is to perform the design procedure more systematically and to include appropriate automatic construction elements that are already available and tested. It is assumed that safety patterns are developed using a safety engineering process lifecycle and associated computer-aided design tools.

Dependable embedded systems with safety requirements that are designed with a semi-formal method, such as UML (that might also use trusted/verified software), typically require a complementary text description to explain the relevant safety concepts and techniques used in the design to be verified by an independent team. The use of safety patterns should reduce the probability of systematic faults by enabling the representation of safety concepts and techniques in such a way that the probability of misunderstanding between the design and verification team is reduced. For this purpose, safety patterns should clearly represent a safety concept/technique, be self-contained and be verified by dependability experts.

6.2. Certification scenarios

It is assumed that the end user has a safety engineering process lifecycle that is compliant with the relevant standard and a set of qualified safety engineers, which are the minimum requirements to proceed with the development and certification of a safety product. As a result, two basic scenarios are defined for the certification of safety embedded systems that use dependability design patterns:

1. In-house-constructed dependability design patterns and repository. The end user develops and reuses safety patterns developed with the same safety engineering process and tools used in the development of safety embedded systems. Already developed and verified safety patterns can be reused as trusted/verified software modules and components to develop certifiable products, reusing the design, development and unit test. Safety patterns can also be certified with a modular certification approach (Rushby, 2007; Althammer et al., 2008) if feasible (e.g., black channel safety communication layer (SCL)).
2. Third-party dependability design patterns and repository. The end user integrates and reuses safety patterns developed by a trusted third party (commercial, open-source, etc.) using a safety engineering process compliant with the safety standard. The integration can include defining a modeling perspective that is compatible with the end-user safety engineering process (encapsulating the details) or integrating the safety pattern as is in the design if both processes are sufficiently compatible. For example, the first approach (compatible model perspective) can be used for generic patterns, such as real-time operating systems and hypervisors. The decision is dependent on the specific project and should consider meeting constraints associated with the safety technique that addresses the reuse of trusted/verified software modules and components and efficiency/compatibility criteria from the engineering process perspective. The end user assumes the responsibility of demonstrating that integrated design patterns are safe for their purpose, which can require a large effort if integrated design patterns are not certifiable.

The availability of safety patterns that have been previously certified (certifiable or, if possible, modular certification) can further improve the development and certification effort by reducing the associated cost and effort to a minimum. If the safety pattern has already been certified (modular certification), the end user requires a modeling representation to be used in the design (compatibility of engineering process), verifies the appropriate use of the safety pattern (meeting constraints and hypotheses stated in the modular certification) and defines/executes the necessary integration tests. However, the internal design, development and unit test that are part of the modular certification can potentially be taken 'as is', and the provider should provide the required documentation to support the certification and additional test frameworks to be executed and passed in each certification project (e.g., to check software correctness for a given processor). For example, this is applicable to certifiable real-time operating systems and safety patterns developed within the scope of our experiment (e.g., safety communication layer (SCL)).

7. Related work

The ideas of system architecture and dependability modeling and analysis are not new. However, to the best of our knowledge, the combination of model-based and pattern-based dependability engineering is new. In this section, we describe our vision in these areas and discuss their relationship to software system engineering and pattern-based engineering. The pattern concept was first introduced by Alexander et al. (1977). A pattern addresses a specific, recurring problem in the design or implementation of a software system. It captures expertise in the form of reusable architecture design themes and styles that can be reused even when algorithms, component implementations, or frameworks cannot.

7.1. Incorporating dependability in system and software engineering

Over the past two decades, the need for formally defined safety lifecycle processes has emerged because the inevitable requirement for better processes eventually pushed control systems to a level of complexity in which sophisticated electronics and programmable systems became the optimal solution for control and safety protection (Smith and Simpson, 2004). With these emergent requirements, many safety lifecycles have been proposed by dif-

ferent associations, such as IEC and ANSI/ISA. These safety lifecycles have been adopted by different domains or companies with some modifications to adapt different requirements (for example, domain-specific requirements). However, because the fundamental differences between a traditional development process and a safety lifecycle are immense, e.g., different types of safety checks and the safety relationships between these checks and phases, modeling these different safety lifecycles with traditionally used process metamodels is not simple and direct.

In system engineering, dependability may be compromised in several system layers. Dependability is typically considered when design decisions are made, leading to potential conflicting situations. The integration of dependability features requires the availability of simultaneous system architecture expertise, domain-specific application knowledge and dependability expertise to manage the potential consequences of design decisions on the dependability of a system and on the remainder of the architecture. For example, at the architectural level, incorporating dependability means having a mechanism (which may be a component or integrated into a component). Development processes for system and software construction are common knowledge and mainstream practice in most development organizations. Unfortunately, these processes offer little support for meeting dependability requirements. Over the years, research efforts have been invested in methodologies and techniques for dependable software engineering, although dedicated processes have been proposed only recently (Bernardi et al., 2012; Panesar-Walawege et al., 2013; Ni et al., 2015).

7.2. Pattern-based development

The supporting research activities in PBSE examine three distinct challenges: (a) mining (discovering patterns from existing systems), (b) hatching (selection of the appropriate pattern), and (c) applying (effective use during the system development process). These three challenges often involve broad core expertise, including formal logic, mathematics, stochastic modeling, graph theory, hardware design and software engineering. In our work, we study only the last two challenges, targeting the (i) development of an extendible design language for modeling patterns in dependable distributed embedded systems (Hamid et al., 2016) and (ii) a methodology to improve existing development processes using patterns (Hamid et al., 2013). The language must capture the core elements of the pattern to support its (a) precise specification, (b) appropriate selection and (c) seamless integration and use. The first aspect is related to pattern definition, whereas the second and third aspects are more related to problem definition. From the pattern-based system and software engineering methodological perspective, only a few works (Abowd et al., 1995; Soundarajan and Hallstrom, 2004; Zdun and Avgeriou, 2008) have addressed this concern. They are harmonized with the use of patterns in each system and software development lifecycle stage. However, existing approaches using patterns often target one stage of development (architecture, design or implementation) due to the lack of formalisms ensuring (1) the specification of a pattern at different levels of abstraction, (2) relationships that govern their interactions and complementarity and (3) the relationship between patterns and other artifacts manipulated during the development lifecycle and those related to the assessment of critical systems.

Several approaches exist in the dependability design pattern literature (Giacomo et al., 2008; Daniels and Vouks, 1997; Tichy et al., 2004). They allow solutions to very general problems that appear frequently as sub-tasks in the design of systems with security and dependability requirements. These elementary tasks include safety communication and fault tolerance. In developing fault-tolerant software applications, the use of patterns leads to well-structured applications; (Daniels and Vouks, 1997) described a hybrid set of patterns to be used in the development of fault-tolerant software applications. These patterns are based on classical fault-tolerant strategies, such as N-version programming, recovery blocks, consensus, and voting. In addition, the hybrid pattern structure can be constructed through a recursive combination of *N*-version programming and others. They also addressed the power of the technique in support of advanced software voting techniques. Extending this framework, (Tichy et al., 2004) proposed a framework for the development of dependable software systems based on a pattern approach. They reused proven fault-tolerant techniques via fault-tolerant patterns. They demonstrated their framework using an application to guide the self-repair of a system after the detection of a node crash.

In Kodituwakku et al. (2003), a mathematical structure was proposed for the organization of patterns depending on several categories. An ontological approach for selecting design patterns was proposed in Girardi and Lindoso (2006) to facilitate understanding and reuse during software development. In their paper, the authors presented an ontology that describes the design pattern format and their relationships. They used a pattern system/language to facilitate the design, integration, selection and reuse of these patterns. A multidimensional classification based on architectural levels, concerns, stages, and other aspects was described in VanHilst et al. (2009). Another aspect that has been considered is system perspectives. Based on the idea of the Zachman framework (Zachman, 1987) (classification based on system perspectives and interrogatives), the Microsoft patterns and practices group classification (Trowbridge et al., 2004) distinguishes the following elements: (a) merits (clearly identifies the context of each pattern and helps identify missing patterns), (b) flaws (more dedicated to functional patterns; non-functional patterns tend to cover many levels of system development and many interrogatives), and (c) improvement (add icons in each pattern to provide classifications).

In the context of patterns in dependable software system development, (Serrano et al., 2008) explained how pattern integration can be achieved using a library of precisely described and formally verified solutions. Conceptually, our modeling framework is similar to that proposed in Serrano et al. (2008). Nevertheless, they used a rigid structure (a pattern was defined as a quadruplet), and consequently, their approach is not usable for capturing specific characteristics of patterns for several domains. Another attempt was made by Boussaidi and Mili (2005), who created a metamodel for both the problem and the design pattern. Then, using a mapping between the two models, they were able to create an integrated model using model transformations. Although we found similarities between this approach and ours, we wanted to go further than the transformation by defining a full process for a proven integration and be able (within this defined process) to allow the user to freely alter the automatic result while always checking the final correctness.

Usually, these design artifacts are provided as a library of models (sub-systems) and as a system of patterns (framework) in the more elaborate approaches. However, there remains a lack of modeling languages and/or formalisms dedicated to specifying these design artifacts and understanding their reuse in software development automation. More precisely, a gap between the development of systems using patterns and the pattern information remains. Most patterns are expressed in a textual form, as informal indications on how to solve individual design problems. Some of them use more precise representations based on UML diagrams, although these patterns do not include sufficient semantic descriptions to automate their processing and to extend their use. Furthermore, the correct application of a pattern is not guaranteed because the description does not consider the effects of interactions, adaptation and combination, making them inappropriate for

automated processing within a tool-supported development process. Finally, due to manual pattern implementation, the problem of incorrect implementation (the most important source of safety issues) remains unresolved.

Recently, Hauge, (2014) presented a pattern-based approach called Safe Control Systems (SaCS) for the development of conceptual safety designs. The SaCS provides three artifacts: (1) a process for the systematic application of patterns as development support; (2) a set of patterns in the form of a library; and (3) a pattern language to define patterns and to specify the application of patterns for safety design conceptualization. This work is similar to our work in its goal, e.g., determining the level of abstraction and life-cycle stage in which the pattern is used and how to define relationships between patterns in order to efficiently combine them. These two works are complementary and their integration should improve the existing pattern-based development approaches. A successful application of our framework attempts to demonstrate the resulting opportunity for applying pattern-based development combined with the benefits of model-based engineering.

7.3. Model-driven engineering and domain-specific language

The modeling concept is becoming a major paradigm in system engineering, particularly in system software engineering (Selic, 2003; Schmidt, 2006; France and Rumpe, 2007). Its use represents a significant advance in terms of the level of abstraction, continuity, and generality. It offers tools to address the development of complex systems, improving their quality and reducing their development cycles (Liebel et al., 2014). Modeling languages based on precise metamodels and transformations are key elements of MDE (Atkinson and Ku¨hne, 2003). With the use of modeling languages, software engineering models a particular system with the goal to be complete and accurate in the context of the system requirements. If done properly, model-driven engineering allows this model to be verified using formal analysis or execution and, later, to generate the source code required to implement the system via transformations (Selic, 2003; France and Rumpe, 2007). Domain-specific languages (DSLs) (France and Rumpe, 2005; Gray et al., 2007) are languages that are specifically tailored to the needs of a particular problem or application domain. Domain experts can understand, validate, modify, test, and sometimes even develop such languages. DSLs are frequently used in MDE (Selic, 2003).

The importance of models and MDE in dependability engineering was highlighted by Gran et al. (2007); Hamid et al. (2008) and Biehl et al. (2010), and confirmed in a recent empirical study on the state of modeling in the embedded domain (Bernardi et al., 2012; Panesar-Walawege et al., 2013; Liebel et al., 2014; Ni et al., 2015) because code generation and simulation are heavily used; the use of modeling in that field has been reported as highly positive. In this context, (Bernardi et al., 2012) proposed a UML profile compliant with MARTE (OMG, 2011b) to address dependability analysis and modeling. Such a profile allows one to conduct a quantitative dependability analysis of software systems modeled with UML. They focus on the following facets of dependability: reliability, availability and safety. In Hamid et al., (2008), we proposed a methodology that associates a model-driven approach with component- based development to design distributed applications with fault-tolerant requirements. UML-based modeling is used to capture application structure and related non-functional requirements thanks to the complementary profile called the FT profile, which is an extension of a subset of QoS&FT and uses the NFP (non-functional properties) sub-profile of MARTE (profile for modeling and analysis of real-time embedded systems). Stereotypes dedicated to fault tolerance specify the fault-detection policy, replication management style and replica group management. From this model descriptor, files are generated (according to the deployment and configuration standard (D&C)) to build boot code (static deployment) that instantiates, configures and connects components and to load configured components. Within this process, component replication and FT properties are declaratively specified at the model level and are transparent for the component implementation.

7.4. Pattern modeling languages

The first attempt to model patterns is the GoF (Gamma et al., 1995), where each pattern is described by UML diagrams. However, there are only natural texts and a few examples to link the diagrams together and explain the integration. This is not sufficient for our objectives. Therefore, UMLAUT was proposed by Guennec et al., (2000) as an approach to formally model design patterns by proposing extensions to the UML metamodel 1.3. They used OCL language to describe structural and behavioral constraints. These constraints are defined using metamodels of specified UML elements via meta collaboration diagrams. The mechanisms of association of these meta-level diagrams to their instance levels (instances of design patterns) are then defined, allowing one to model design patterns accurately via the UML language. This work is illustrated through two examples of design patterns: visitor and observer.

By specifying design patterns as metamodels and defining a set of features to handle the models, the RBML (role-based metamodeling language) proposed by Kim et al., (2003) attempts to bridge the gap between the pattern and its use. The RBML formalism, which is based on UML, is able to precisely capture various design perspectives of patterns, such as static structure, interactions, and state-based behavior. Each one is characterized by a specific RBML metamodel type: (1) SPS (static pattern specifications) is a structural design pattern specification that allows one to express the static view, (2) IPS (interaction pattern specification) represents the design pattern in terms of possible interactions between different roles, and (3) SMPS (state machine pattern specifications) can add a behavioral perspective to describe the various states in which a pattern may lie in its execution. However, the integration by itself remains not clearly defined.

Another issue raised in DPML (design pattern modeling language) (Mapelsden et al., 2002) and in LePUS (Gasparis et al., 2008) is visualization. These languages both use a combination of modeling and metamodeling. In Gasparis et al., (2008), a formal and visual language called LePUS was presented for specifying design patterns. It defines a pattern in an accurate and complete based on a formula in Z with a graphical representation. A diagram in LePUS is a graph whose nodes correspond to variables and whose arcs are labeled with binary relations. The framework promoted by LePUS is interesting, although the degree of expressiveness proposed to capture the intent and abstract the solution of a pattern is too restrictive. In addition, there is a lack of relationship between the pattern and its instantiation. With regard to the integration of patterns in software systems, the DPML (design pattern modeling language) (Mapelsden et al., 2002) allows the incorporation of patterns in UML models.

The recently completed FP6 SERENITY project has introduced a new notion of security and dependability (S&D) patterns. SERENITY's S&D patterns are precise specifications of validated S&D mechanisms including a precise behavioral description, references to the S&D properties, constraints on the context required for deployment, information describing how to adapt and monitor the mechanism, and trust mechanisms. The S&D SERENITY pattern is specified following several levels of abstraction to bridge the gap between abstract solution and implementation. These abstraction levels are S&D classes, S&D patterns and S&D implementation.

Such validated S&D patterns and the formal characterization of their behavior and semantics can also be the basic building blocks for S&D engineering in embedded systems. Serrano et al. (2008) explained how this can be achieved using a library of precisely described and formally verified security and dependability (S&D) solutions, i.e., S&D classes, S&D patterns, S&D implementation and S&D integration schemes. Moreover, Giacomo et al. (2008) reported an empirical experience regarding the adoption and elicitation of S&D patterns in the air traffic management (ATM) domain, demonstrating the power of using patterns as guidance to structure the analysis of operational aspects when used at the design stage.

Existing formalization attempts for patterns (Mikkonen, 1998; Soundarajan and Hallstrom, 2004) fall short in handling the inherent variability in pattern descriptions (Zdun and Avgeriou, 2008), and they focus primarily on a very limited design and architecture pattern scope. They do not yet address specific domains, such as security and safety. For the first type of approach (Gamma et al., 1995), design patterns are usually represented by diagrams with specific notations, such as UML object diagrams, that are accompanied by textual descriptions and examples of code to complete the description. Furthermore, their structure is rigid (context, structure, solution, etc.). Unfortunately, the use and/or application of a pattern can be difficult or inaccurate. In fact, the existing descriptions are not formal definitions and sometimes leave ambiguities regarding the exact meaning of the patterns. There are some promising and well-proven approaches (Douglass, 1998) based on Gamma et al., (1995). However, this type of technique does not afford the high degree of flexibility in the pattern structure that is required to reach our objectives. Thus far, patterns have been used in systematic engineering approaches for various tasks, such as classification and organization, pattern selection based on security requirements (Weiss and Mouratidis, 2008), analyzing and modeling security requirements (Cheng et al., 2003), and measuring the introduced security level (Fernandez et al., 2010). A similar situation is prevalent for safety patterns, which are surveyed in Preschern et al. (2013, 2014) and formalized in Armoush, (2010). In Daniels and Vouks (1997) and Tichy et al. (2004), the pattern specification consists of a service-based architectural design and deployment restrictions in the form of UML deployment diagrams for different architectural services. Conceptually, our modeling framework is similar to that proposed in the SERENITY project. Nevertheless, the pattern structure is rigid (a pattern is defined as quadruplet) and is thus unusable for capturing specific characteristics of S&D patterns. However, the SERENITY project proposes several levels of abstraction to bridge the gap between abstract solution and implementation, which intends to not capture a common representation of patterns for several domains.

From a different point of view, we agree with the argumentations given in Zdun and Avgeriou, (2005) to justify why the precise specification and formalization of a pattern by definition restricts its “degree of freedom for the design”, and hence there are no success stories of works dealing with pattern development. This is not only related to dependability patterns. Note however, that these works do not address the validation activity which is an important issue in any design activity and more particularly in dependability engineering. We claim that dependability is subject to rigorous and precise specification and the proposed literature (to the best of our knowledge) fails to meet these two objectives. To remedy these contradictory needs, we support the specifications of dependability patterns at two levels of abstractions, domain-independent and domain-specific, in a semi-formal representation through metamodeling techniques. This allows to support some variability of the pattern, and hence to foster reuse.

8. Conclusion

In this paper, we propose a pattern-based development approach to address dependability through a model-driven engineering approach. The approach is composed of several steps and is based on metamodeling techniques that enable the specification of dependability patterns. It is also based on model transformation techniques for the purposes of generation. The defined metamodel points to a common representation for several contexts of use. First, this approach aims to allow design automation through the reuse of a dependable application. Second, it aims to overcome the lack of formalism in a conventional text-based approach. The approach empowers system and software engineers to reuse solutions for dependability without specific knowledge of how the solution is designed and implemented. This feature enables one to work at a higher abstraction level, which may significantly reduce the cost of engineering a particular system.

We begin by specifying a conceptual model of the desired patterns and proceed by designing modeling languages that are appropriate for the content. The results of these efforts are then used to specify and define the dependability as a pattern (e.g., in the form of properties, design diagrams, etc.). Developing an application using pattern-based development processes and thus reusing existing patterns requires finding and tailoring suitable patterns into a form that is appropriate for the targeted development environment. The integration phase of our approach allows a domain engineer to reuse the resultant patterns that have been previously adapted and transformed for a given engineering environment (development platform) to develop a domain-specific application. Thus, we provide an overall pattern- based system engineering (PBSE) framework and an operational architecture for a tool suite to support the proposed approach. An example of such a tool suite, called Semcomdt, is constructed using EMFT and a CDO-based repository and is currently provided in the form of Eclipse plugins. In addition, the tool suite promotes the separation of concerns during the development process by distinguishing the stakeholder roles. Access to the repository is customized with regard to the development phases, the stakeholders domain and system knowledge

Furthermore, we evaluate the usefulness of the patterns for increasing engineering productivity. The dependability captured in a pattern (e.g., in the form of properties, design solutions, etc.) is based on its generality, i.e., we determine whether the same design solution can be successfully used to instantiate the railway safety sector engineering processes and whether they can be used to instantiate other processes. We intend to demonstrate that the dependability pattern-based approach leads to a reduced number or to a simplification of the engineering process steps. The design solutions that are provided should support the developer regarding dependability issues and reduce the error frequency. We demonstrate that the application of the proposed approach brings important benefits to development engineers. This statement is demonstrated via the implementation of the demonstrator. First evidence from the case study and the key performance indicator survey indicates that users are satisfied with the pattern-based approach. The approach paves the way to allow users to define their own road-maps based on the PBSE methodology. The first evaluations are encouraging. However, they also highlight one of the main challenges, i.e., the automatic search for the user to derive those “dependability patterns” from the requirement analysis. However, being aware of all functionality benefits requires years of experience in the practical industry. Because model-driven engineering is not yet common in all embedded domains, it becomes more difficult to find acceptance among companies accustomed to manually implementing software solutions. However, this methodology will become increasingly common, and our approach contributes

to the exploitation of this engineering methodology in future. The practical case study in this paper shows that the developed tools can be successfully used to support the entire engineering process. The SEMCO tools can be adapted to a domain's engineering processes, and the provided transformation techniques make it possible to support several target platforms. The extension to industry companies in different domains is given by the extendibility and flexibility of the tools offered by the SEMCO tool suite. The following steps aim to garner acceptance in the industry and to extend our approach to industry companies.

Finally, we discuss the pattern-based approach for certification support, arguing that our approach may provide artifacts to ease the certification and sketching future directions in this regard. Patterns have supported application engineers in generating the documents required by the certification authority. Even the certification phase may profit from our approach. For the evaluator, the use of well-structured and formally validated patterns and their direct contribution to development process documentation can hasten the evaluation work.

In our future work, we plan to study the automation of the model search and tailoring tasks, and a framework allowing a simpler specification of constraints would be beneficial. Our vision is for patterns to be inferred from the browsing history of users and constructed from a set of previously developed applications. As we look to the future, we can use existing work on reuse scenarios and design space exploration (Tomer et al., 2004; Hegedu¨s et al., 2015; Hamid, 2015). We would also like to study the integration of our tools with other MDE tools. The objective is to show the process flow and the integration of the tools in the domain tool chains, whereas the intention is not to resolve the low-level details of the approach integration. For that purpose, we must implement other types of software and means of generating validated artifacts, such as programming language code and certification artifacts, that are capable of producing a restrictive set of artifacts that comply with domain standards. The required pattern representation at the same level may differ from one domain to another; thus the access tool is responsible for providing the information in the required format. The layout of the access tool depends on the sector particularities; thus, a new "skin" must be defined every time a new sector is considered. Moreover, the access tool must be extended with a transformation capability for different toolsets. We would also like to study the preservation of design decisions through modeling artifacts (That et al., 2014). Concurrently, more sophisticated techniques to derive artifact relationships can be implemented, possibly using different domains, to reduce the complexity of designing systems of modeling artifacts. Additionally, we will seek new opportunities to apply the proposed approach to other domains. This requires an instantiation of the full software engineering tool and method and an evaluation across the experiences of many users across many domains. Finally, we would like to enhance the proposed integration process by automating the detection of conflicts between the modeling artifact structure and the existing application architecture and propose solutions in a manner similar to that in which the merging tools operate.

Acknowledgments

This work was initiated within the context of the SEMCO project. It was supported by the European FP7 TERESA project and by the French FUI 7 SIRSEC project. Particular thanks go to Adel Ziani and Jacob Geisel for their valuable assistance in the implementation and development of the SEMCO tools. In addition, we would like to thank the TERESA consortium members for their participation in the survey and the implementation of the case study.

References

Abowd, G., Allen, R., Garlan, D., 1995. Formalizing style to understand descriptions of software architecture. ACM Trans. Softw. Eng. Methodol. 4 (4), 319–364.

Agresti, W., 2011. Software reuse: developers' experiences and perceptions. J. Softw. Eng. Appl. 4 (1), 48–58.

Alexander, C., Ishikawa, S., Silverstein, M., 1977. A pattern language. Center for Environmental Structure Series, 2. Oxford University Press ISBN 9780195019193.

Alexander, R., Kelly, T., Kurd, Z., McDermid, J., 2007. Safety Cases for Advanced Control Software: Safety Case Patterns. Final Report, NASA Contract FA8655-07-1-3025. Tech. rep., University of York.

Althammer, E., Schoitsch, E., Sonneck, G., Eriksson, H., Vinter, J., 2008. Modular certification support - the decos concept of generic safety cases. In: Proceedings of 6th IEEE International Conference on Industrial Informatics (INDIN). IEEE, pp. 258–263.

Armoush, A., 2010. Design Patterns for Safety-Critical Embedded Systems. Dissertation, Embedded Software Laboratory - RWTH Aachen University, [Accessed: December-2013]. URL http://aib.informatik.rwth-aachen.de/2010/2010-13.pdf

Atkinson, C., Ku¨hne, T., 2003. Model-driven development: a metamodeling foundation. IEEE Softw 20 (5), 36–41.

Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Depend. Secure Comput. 1, 11–33.

Bernardi, S., Merseguer, J., Petriu, D., 2012. Dependability modeling and analysis of software systems specified with UML. ACM Comput. Surv 45 (1), 1–2 248.

Biehl, M., DeJiu, C., To¨rngren, M., 2010. Integrating safety analysis into the Model-based development toolchain of automotive embedded systems. SIGPLAN Not 45 (4), 125–132.

Boussaidi, G.E., Mili, H., 2005. Representing and applying design patterns: what is the problem? In: Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MODELS). Springer, pp. 186–200.

Buschmann, F., Henney, K., Schmidt, D., 2007. Pattern-Oriented software architecture. A Pattern Language for Distributed Computing, 4. Wiley ISBN 978-0470059029.

Buschmann, G., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. Pattern-Oriented Software Architecture: a system of patterns, 1. John Wiley and Sons ISBN 978-0471958697.

CENELEC, 1999. EN 50129: railway applications. Communication, signalling and processing systems. Safety related electronic systems for signalling.

Cheng, B.H.C., Konrad, S., Campbell, L.A., Wassermann, R., 2003. Using security patterns to model and analyze security. In: Proceedings of IEEE Workshop on Requirements for High Assurance Systems, pp. 13–22.

Daniels, K.K.F., Vouks, M.A., 1997. The reliable hybrid Pattern: a generalized software fault tolerant design pattern. In: Proceedings of the Pattern Language of Programs (PLoP), pp. 1–9.

Douglass, B.P., 1998. Real-time UML: Developing Efficient Objects for Embedded Systems. Addison-Wesley ISBN 0-201-32579-9.

Fernandez, E.B., 2013. Security patterns in practice: Building secure architectures using software patterns. Software Design Patterns. Wiley ISBN 978-1-119-99894-5.

Fernandez, E., Yoshioka, N., Washizaki, H., VanHilst, M., 2010. Measuring the level of security introduced by security patterns. In: Proceedings of International Conference on Availability, Reliability, and Security (ARES). IEEE Computer Society, pp. 565–568.

France, R., Rumpe, B., 2007. Model-driven development of complex Software: a research roadmap. In: Future of Software Engineering (FOSE). IEEE Computer Society, pp. 37–54.

France, R.B., Rumpe, B., 2005. Domain specific modeling. Softw. Syst. Model. 4 (1), 1–3.

Gamma, E., Helm, R., Johnson, R.E., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley ISBN 978-0-470-05902-9.

Gasparis, E., Nicholson, J., Eden, A.H., 2008. LePUS3: an object-oriented design description language. In: Diagrammatic Representation and Inference. Springer, Berlin Heidelberg, pp. 364–367.

Giacomo, V.D., Felici, M., Meduri, V., Presenza, D., Riccucci, C., Tedeschi, A., 2008. Using security and dependability patterns for reaction processes. In: Proceedings of the 2008 19th International Conference on Database and Expert Systems Application. IEEE Computer Society, pp. 315–319.

Girardi, R., Lindoso, A.N., 2006. An ontology-based knowledge base for the representation and reuse of software patterns. ACM SIGSOFT Softw. Eng. Notes 31 (1), 1–6.

Gran, B.A., Fredriksen, R., Thunem, A.P.-J., 2007. Addressing dependability by applying an approach for model-based risk assessment. Reliab. Eng. Syst. Safety 92 (11), 1492–1502.

Gray, J., Tolvanen, J.-P., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J., 2007. Domain-Specific modeling. In: Fishwick, P. (Ed.), Handbook of Dynamic System Modeling. Chapman & Hall/CRC, pp. 1–20. Ch. 7.

Guennec, A.L., Sunye, G., Jeźeźquel, J.-M., 2000. Precise modeling of design patterns. In: Proceedings of the Unified Modeling Language: Advancing the Standard Third International Conference. Springer-Verlag, pp. 482–496.

Hamid, B., 2014. A model-driven methodology approach for developing a repository of models. In: Proceedings of the 4th International Conference Model and Data Engineering - (MEDI). Springer, pp. 29–44. Vol. 8748 of LNCS.

Hamid, B., 2015. Interplay of security&dependability and resource using Model–driven and Pattern-based development. In: Proceedings of IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE Computer Society, pp. 254–262.

Hamid, B., Geisel, J., Ziani, A., Bruel, J., Perez, J., 2013. Model-Driven engineering for trusted embedded systems based on security and dependability patterns. In: Proceedings of the 16th International SDL Forum. Springer, pp. 72–90. Vol. 7916 of LNCS.

Hamid, B., Gu¨rgens, S., Fuchs, A., 2016. Security patterns modeling and formalization for pattern-based development of secure software systems. In: Innovations in Systems and Software Engineering, 12. Springer, pp. 109–140.

Hamid, B., Percebois, C., Gouteux, D., 2012. A methodology for integration of patterns with validation purpose. In: Proceedings of European Conference on Pattern Language of Programs (EuroPlop). ACM DL, pp. 1–14.

Hamid, B., Radermacher, A., Lanusse, A., Jouvray, C., Ge´rard, S., Terrier, F., 2008. Designing fault-tolerant component based applications with a model driven approach. In: Proceedings of IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS). Springer, pp. 9–20. Vol. 5287 of LNCS.

Hauge, A., 2014. SaCS: A Method and a Pattern Language for the Development of Conceptual Safety Design. University of Oslo Doctoral thesis[Accessed: June -2016]. URL https://www.duo.uio.no/bitstream/handle/10852/41717/1/dravhandling-Hauge.pdf.

Hegedu¨s, A., Horva´th, A., Varro´, D., 2015. A model-driven framework for guided design space exploration. Autom. Softw. Eng. 22 (3), 399–436.

Henninger, S., Corre^a, V., 2007. Software pattern Communities: current practices and challenges. In: Proceedings of the 14th Conference on Pattern Languages of Programs. PLOP '07, 14. ACM, pp. 1–14. 19.

IEC, 2002. IEC 62280: railway applications - Communication, signalling and processing systems. Safety related communication in transmission systems.

IEC, 2003. IEC 61784: industrial communication networks.

IEC, 2010a. IEC 61508-2: Functional safety of electrical/electronic/programmable electronic safety-related systems Part 2: Requirements for electrical / electronic / programmable electronic safety-related systems.

IEC, 2010b. IEC 61508: functional safety of electrical/electronic/programmable electronic Safety-related systems.

IEEE, 2004. IEEE 1474.1-2004: standard for communications-based train control (CBTC) performance and functional requirements.

Kim, D., France, R.B., Ghosh, S., Song, E., 2003. A role-based metamodeling approach to specifying design patterns. In: Proceedings of the 27th International Com- puter Software and Applications Conference (COMPSAC): Design and Assessment of Trustworthy Software-Based Systems. IEEE Computer Society, pp. 452–457.

Kodituwakku, S.R., Saluka, R., Bertok, P., 2003. Pattern Categories: a mathematical approach for organizing design patterns. In: Revised papers from the Third Asia-Pacific Conference on Pattern Languages of Programs (KoalaPLoP). ACS, pp. 63–73. Vol. 13 of CRPIT.

Kopetz, H., 2011. Real-Time Systems - Design Principles for Distributed Embedded Applications. Springer ISBN 978-1441982360.

Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J., 2014. Assessing the State-of-Practice of model-based engineering in the embedded systems domain. In: Model-Driven Engineering Languages and Systems. Springer International Publishing, pp. 166–182. Vol. 8767 of LNCS.

Mapelsden, D., Hosking, J., Grundy, J., 2002. Design pattern modelling and instantiation using DPML. In: Proceedings of the Fortieth International Conference on Tools Pacific. Australian Computer Society, Inc, pp. 3–11.

McClure, C., 1997. Software Reuse Techniques: Adding Reuse to the System Development Process. Prentice-Hall, Inc.

Mikkonen, T., 1998. Formalizing design patterns. In: Proceedings of the 20th International Conference on Software Engineering (ICSE). IEEE Computer Society, pp. 115–124.

Ni, S., Zhuang, Y., Cao, Z., Kong, X., 2015. Modeling dependability features for real–time embedded systems. IEEE Trans. Depend. Secure Comput. 12 (2), 190–203.

Noble, J., 1998. Classifying relationships between object-oriented design patterns. In: Proceedings of the Australian Software Engineering Conference (ASWEC). IEEE Computer Society, pp. 98–107.

OMG, 2008. Meta Object Facility (MOF) 2.0 query/view/transformation (QVT), Version 1.0. http://www.omg.org/spec/QVT/1.0/. [Accessed: January-2013].

OMG, 2010. Object Constraint Language (OCL), Version 2.2. http://www.omg.org/spec/OCL/2.2. [Accessed: January-2013].

OMG, 2011a. UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE), Version 1.1. http://www.omg.org/spec/MARTE/1.1/ , [Accessed: January-2013].

OMG, 2011b. Unified modeling language (UML), version 2.4.1. http://www.omg.org/spec/UML/2.4.1. [Accessed: January-2013].

Panesar-Walawege, R.K., Sabetzadeh, M., Briand, L., 2013. Supporting the verification of compliance to safety standards via model-driven engineering: Approach, tool-support and empirical validation. Inf. Softw. Technol. 55 (5), 836–864.

Peterson, W., Brown, D., 1961. Cyclic codes for error detection. Proc. IRE 49 (1), 228–235.

Powel, D.B., 2003. Real-time design patterns : robust scalable architecture for real–time systems. Object Technology Series. Addison-Wesley ISBN 978-0201699562.

Preschern, C., Kajtazovic, N., Ho¨ller, A., Kreiner, C., 2014. Pattern-based safety development Methods: overview and comparison. In: Proceedings of the 19th European Conference on Pattern Languages of Programs (EuroPLoP), 28. ACM DL, pp. 1–28. 20.

Preschern, C., Kajtazovic, N., Kreiner, C., 2013. Building a safety architecture pattern system. In: Proceedings of the 18th European Conference on Pattern Languages of Program (EuroPLoP), 17. ACM DL, pp. 1–17. 55.

Radermacher, A., Hamid, B., Fredj, M., Profizi, J.-L., 2013. Process and tool support for design patterns with safety requirements. In: Proceedings of European Conference on Pattern Language of Programs (EuroPlop), 8. ACM DL, pp. 1–8. 16.

Ravi, S., Raghunathan, A., Kocher, P., Hattangady, S., 2004. Security in embedded systems: design challenges. ACM Trans. Embed. Comput. Syst. 3 (3), 461–491.

Riehle, D., Züllighoven, H., 1996. Understanding and using patterns in software development. Theor. Pract. Object Syst. 2 (1), 3–13.

Rodano, M., Giammarc, K., 2013. A formal method for evaluation of a modeled system architecture. Procedia Comput. Sci. 20, 210–215.

RTCA, 1992. DO-178B: Software Considerations in airborne Systems and Equipment Certification.

Rushby, J., 2007. Just-in-time certification. In: Proceedings of 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007). IEEE, pp. 15–24.

Schmidt, D., 2006. Model-Driven engineering. IEEE Comput. 39 (2), 41–47.

Schmidt, D., Buschmann, F., 2003. Patterns, frameworks, and middleware: their synergistic relationships. In: Proceedings of 25th International Conference on Software Engineering. IEEE, pp. 694–704.

Schumacher, M., 2003. Security engineering with patterns - Origins, theoretical Models, and new applications. Lecture Notes in Computer Science, 2754. Springer ISBN 978-3-540-45180-8.

Selic, B., 2003. The pragmatics of model-driven development. IEEE Softw. 20 (5), 19–25.

Serrano, D., Mana, A., Sotirious, A.-D., 2008. Towards precise and certified security patterns. In: Proceedings of 2nd International Workshop on Secure Systems Methodologies Using Patterns (Spattern). IEEE Computer Society, pp. 287–291.

Smith, D., Simpson, K., 2004. Functional Safety. Routledge ISBN 978-0-08-047792-3.

Soundarajan, N., Hallstrom, J., 2004. Responsibilities and Rewards: specifying design patterns. In: Proceedings of the 26th International Confer- ence on Software Engineering. IEEE Computer Society, pp. 666–675.

Stanley, P., 2011. ETCS For Engineers. EurailPress ISBN 978-3777104164.

Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2009. EMF: Eclipse Modeling Framework 2.0, 2nd Edition Addison-Wesley Professional ISBN 0321331885.

Strembeck, M., Zdun, U., 2009. An approach for the systematic development of domain-specific languages. Softw.: Pract. Exp. 39 (15), 1253–1292.

TERESA, 2013. Specification of Platform. Deliverable D6.1 – TERESA/WP6/D6.1, IST Project IST-248410. [Accessed: January-2013].

That, M.T., Sadou, S., Oquendo, F., Fleurquin, R., 2014. Preserving architectural decisions through architectural patterns. Automated Softw. Eng. 23 (3), 427–467.

Tichy, M., Schilling, D., Giese, H., 2004. Design of self-managing dependable systems with UML and fault tolerance patterns. In: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems. ACM, pp. 105–109.

Tomer, A., Goldin, L., Kuflik, T., Kimchi, E., Schach, S., 2004. Evaluating software reuse Alternatives: a model and its application to an industrial case study. IEEE Trans. Softw. Eng. 30 (9), 601–612.

Trowbridge, D., Cunningham, W., Evans, M., Brader, L. [Accessed: December-2015].

UNISIG, 2009. Safety requirements for the technical interoperability of ETCS in levels 1 & 2, issue 2.5.0.

Uzunov, A.V., Fernandez, E.B., Falkner, K., 2013. Engineering security into distributed Systems: a survey of methodologies. J. Univ. Comput. Sci. 18 (20), 2920–3006.

VanHilst, M., Fernandez, E.B., Braz, F., 2009. A multidimensional classification for users of security patterns. J. Res. Pract. Inf. Technol. 41 (2), 87–97.

Weiss, M., Mouratidis, H., 2008. Selecting security patterns that fulfill security requirements. In: Proceedings of the 16th IEEE International Requirements Engineering Conference. IEEE Computer Society, pp. 169–172.

Wohlin, C., Runeson, P., Ho¨st, M., Ohlsson, M., Regnell, B., Wessleń, A., 2000. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell, MA, USA.

Zachman, A., 1987. A framework for information systems architecture. IBM Syst. J. 26, 276–292.

Zdun, U., Avgeriou, P., 2005. Modeling architectural patterns using architectural primitives. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. ACM, pp. 133–146.

Zdun, U., Avgeriou, P., 2008. A catalog of architectural primitives for modeling architectural patterns. J. Inf. Softw. Technol. 50 (9-10), 1003–1034.

Ziani, A., Hamid, B., Bruel, J., 2012. A model-driven engineering framework for fault tolerance in dependable embedded systems design. In: EUROMICRO-SEAA. IEEE, pp. 166–169.

Dr Brahim HAMID is an associate professor at the University of Toulouse Jean-Jaurès (France) and he is a member of the IRIT-MACAO team. He got his Ph.D. degree in 2007 in the area of dependability from the University of Bordeaux. He has been an assistant professor at ENSEIRB. Then he worked as a post-doc in the modeling group at the CEA. His main research topics are software languages engineering, at both the foundations and application level, particularly for resource constrained systems. He works on security, dependability and software architecture. Furthermore, he is an expert in model-driven development approaches both in research and teaching.

Dr. Jon Pérez is a researcher at IKERLAN research centre since 2002 who works in the design and development of SIL4 safety-critical embedded systems for railway signaling (ERTMS/ETCS). He has received a M.Sc. in Electronics & Electrical Engineering at the University of Glasgow and completed his PhD in Computer Science at TU WIEN in the field of safety-critical embedded systems. He has previously worked for Motorola Semiconductor in the field of multicore DSPs. Research interests focus on distributed real-time and safety-critical embedded systems. He is leading the Electronics Department and the Embedded Systems Research Group in Ikerlan from June 2011.