Process and tool support for design patterns with safety requirements

Ansgar Radermacher, Brahim Hamid, Manel Fredj, Jean-Louis Profizi

Process and tool support for design patterns with safety requirements

ANSGAR RADERMACHER, CEA, LIST, Laboratory of Model driven engineering for embedded systems
BRAHIM HAMID, Brahim Hamid, IRIT, University of Toulouse
MANEL FREDJ, SERMA INGENIERIE, Critical Embedded Software Business Unit, France
JEAN-LOUIS PROFIZI, ALSTOM Transport Information Solutions, France

The requirement for higher Security and Dependability (S&D) of systems is continuously increasing, even in domains tradi-tionally not deeply involved in such issues. Nowadays, many practitioners express their worries about current S&D software engineering practices. New recommendations should be considered to ground this discipline on two pillars: solid theory and proven principles. We took the second pillar towards software engineering for embedded system applications, focusing on the problem of integrating S&D by design to foster reuse. In this paper, we propose to combine design patterns and Model Driven Engineering (MDE) techniques for building component-based applications with safety requirements. The resulting modeling framework serves primarily to capture the basic concepts for specifying safety-oriented design patterns, building an S&D pattern system, and maintain safety properties, with existing modeling artifacts, during the engineering process based on the S&D pattern system. As a proof of concept, we are evaluating the feasibility of the framework through the example of the MooN pattern system for building systems having safety requirements: Communication Based Train Control (CBTC).

Key Words and Phrases: Design patterns, Component, Traceability, Safety, Model Driven Engineering

1. INTRODUCTION

The development of embedded systems [Zurawski 2007] needs to address non-functional requirements such as memory, timeliness, or limited energy. If these systems are safety-critical, the development also need to address additional safety and security as well as dependability (S&D[1]) [Ravi et al. 2004] requirements. Thus, non-functional requirements become more important, since accidents may be caused not only by "wrong" actions but also by "right" actions that are executed too late. Some non-functional requirements have a large impact on the software as well as hardware architecture, for instance an availability requirement can only be guaranteed with a replication of hardware and a suitable allocation of software blocks on this hardware. In addition, safety norms demand that requirements are *traced*, i.e. systematically captured and covered by elements within the system architecture.

[1]In the sequel, we explicitly include functional safety when we talk about S&D.

Thus, development processes need to the demands of S&D systems into account. Component-based software engineering (CBSE) [Crnkovic 2005] helps to capture some facets of the requirements, notably a hierarchical decomposition of the system and explicit allocation of software to hardware components. Still, the integration of S&D features requires the availability of both application domain specific knowledge and S&D expertise at the same time. We propose to capture this expertise through S&D *patterns* as well as integrating these patters into the development process.

The collaborative project SIRSEC[2], within which this work takes place, has the objective to apply safety design patterns to the development of component-based embedded systems supporting railway signaling functions such as CBTC (Communication Based Train Control). The safety functions of the CBTC system are implemented on a set of safety equipments. The hardware and software platform hosts both the safety software application and the middleware with high dependability capabilities. The system has to fulfill safety integrity levle 4 (SIL4), as determined by the safety analysis of the system according to the standards EN 50126, EN50128 and EN 50129 [CENELEC 2000] [CENELEC 2011] [CENELEC 2003].

A design pattern may provide a suitable solution for the constraints or properties of the hardware and software platform, since it captures a proven solution along with its advantages and inconveniences. The pattern thus guides the adaptation of a model based design to this platform. However, the lack of formalization introduces potential risks on the system safety. Hence, the definition and the use of design patterns should be adapted according to safety requirements and assessments, to meet several issues. An important issue is to (partly) *automate the use of design pattern in the software development process.* This requires the formalization of design patterns. Whereas diagrams sketch the structure or behavior of a solution, the underlying notation is often not or not well formalized. Semi-formal languages such as UML are a good starting point, since these are typically also used for application development, in the context of model driven engineering.

Existing approaches using UML also support the application of a pattern. However, the latter results merely in copying a solution into a model. In general, it is then necessary to adapt the copied solution to the system constraints, naming conventions, etc. However, if there is no trace of the original roles within the pattern definition, it is possible to modify the elements of copied solution in an arbitrary way after the pattern has been applied. Thus, the properties of the solution, for instance with respect to safety, might not be valid within the system although a pattern guaranteeing these has been used. In our work, we propose a model and pattern instantiation approach allowing the application designer to *trace pattern application* and thus preserve the safety properties of the pattern along its use.

In our previous work [Hamid et al. 2011], we proposed methods to model security and dependability aspects in patterns and to validate manually whether these still hold in RCES (Resource Constrained Embedded Systems) after pattern application. A meta-model serves as language for describing these patterns. A set of S&D patterns is organized in a pattern system and stored in a repository within the Semco platform[3].

In this paper, we present a new UML profile to model S&D patterns providing new capabilities for pattern application and traceability within the profile. To validate the results, we are developing an MDE tool chain to assist the developers of S&D pattern-based component-based applications. The MDE tool chain was implemented as an extension of the UML model editor Papyrus[4]. We also validate the results by applying our approach to model secure and dependable embedded system in the context of SIRSEC project: the CBTC System. The target audience of this paper are on the one side people working on pattern formalization and on the other those working on process and tools supporting the application of patterns, i.e. integrating the use of patterns into a development cycle.

[2]SIRSEC: Safe Distributed Information System (Système d'Information Réparti Sécuritaire).
[3]http://www.semcomdt.org
[4]http://eclipse.org/papyrus/

The paper is structured as follows. In the next section (Section 2), we examine related work including both academic papers as well as design pattern support in existing modeling tools. Section 3 gives an overview of our approach which is detailed in the following sections. In Section 4, we propose a modeling language-neutral meta-model and a UML profile that enables the specification of design patterns. This presentation is focused on a simplified example of a design pattern (Section 4.1), which serves as a running example throughout the rest of the paper. The use of patterns within the development of a system is addressed in Section 5. This section provides details on how a pattern is applied to an existing UML architecture in a way that it keeps a trace of pattern application. The next section outlines a tool that supports the proposed approach. The paper is concluded in Section 7 by a summary and outlook for future work.

2. RELATED WORK

In this section, we examine the state-of-the art of modeling design patterns, chiefly the aspect how to represent a pattern and how to support the use of a design pattern focusing on S&D patterns.

Some existing approaches [Foote et al. 1999] propose specific languages to formalize the pattern description, however, the solutions are still not generic enough or are too complex to be easily applied on every system. For instance, in [Taibi 2007], the author proposes a balanced pattern specification language (BPSL) to describe both structural and behavioral aspects of patterns using formal specifications. This approach enables users to know when and how to apply a pattern, however, it does not provide support for pattern application.

Section 17.5 of UML 2.0 [OMG 2011] proposes to model patterns in form of collaborations in combination with a template signature. The collaboration diagram focuses on the relationship between objects, enabling to visualize the way objects collaborate together. The template signature is a list of formal template parameters, which can be substituted by actual parameters when binding the collaboration to an actual system.

Existing UML tools support the application of design patterns, notably Netbeans and StarUML[5] using the proposal of the UML standard. The main critic of these tools is that *no trace* of the pattern application remains in the application model. Thus, it is possible to make modifications to the pattern without being aware that these may break some of the pattern's properties. This is particularly important in the context of patterns that should guarantee certain safety and security properties.

In the context of verifiability, in [Mapelsden et al. 2002], the authors propose to use DPML (Design Pattern Modeling language) with a tool support DPTool. In DPML, only solutions of design patterns are modeled, i.e., consequences and others pattern specifics are not included in the model. It introduces a notation for visually representing design pattern solution, then, at pattern instantiation. It provides a process that transforms the DPML-specific notation into the UML model of the system and proposes a list of candidates for each role of the solution. This process enables correct and consistent application of the solution on a UML object model. In the case that users incorrectly bind the pattern solution to their UML model, a verification mechanism generates error messages.

In LePUS [Nicholson et al. 2009] (LanguagE for Patterns Uniform Specification, Version 3), the authors propose a visual tool supported modeling language for design pattern that enables formal specification of design pattern solutions using first order predicate logic. LePUS includes visual notation for LePUS formulas, consisting of icons (ellipses, squares and triangles). It also enables verifying whether a Java program satisfies the pattern specification. The main constraint of LePUS is that it is based on formal logic, it accredits the verification with formal basis however it makes difficult for the average software developers to work with as the tool-support is still at the prototyping stage.

[5]netbeans: http://netbeans.org/ StarUML: http://staruml.sourceforge.net/en/

Another alternative for the specification of patterns is the use of pattern primitives [Zdun and Avgeriou 2005]. An interesting approach used to reason about design decision in terms of precise modeling element in a model which represents a pattern. The approach was applied in [Zdun and Avgeriou 2005] for specifying architectural patterns and in [Zdun et al. 2007] for specifying process patterns. For the first concern, a set of pattern primitive (e.g., Callback,Indirection,Grouping,Layering, Virtual Connector, ...) are used as a basic construct for software architectural pattern specification, and then illustrated how software architectural pattern such as broker is built using these primitives. For the second concern, a set of pattern primitives (eg. process flow refinement, process flow steps, Synchronous service invocation,..) for process patterns in the context of process-oriented integration of services are captured and then used for specifying a set of patterns of process-driven SOAs (eg. Microflow integration service).

Several approaches exist in the S&D design pattern literature [Yoder and Barcalow 1998; Giacomo et al. 2008; Yoshioka et al. 2008; Daniels 1997; Tichy et al. 2004]. They allow to solve very general problems that appear frequently as sub-tasks in the design of systems with security and dependability requirements. These elementary tasks include secure communication, fault tolerance, etc. Particularly, [Yoder and Barcalow 1998] presents a collection of patterns to be used when dealing with application security. The proposed catalog includes secure access layer, single access point, check point, etc..

[Daniels 1997] describes a set of patterns for the development of fault-tolerant software applications. These patterns are based on classical fault tolerant strategies such as N-Version programming, recovery block, consensus, voting and acceptance tests. In addition, the paper proposes patterns that combine several strategies. These patterns are therefore called *hybrid* patterns. The expressiveness of the technique is demonstrated through the support of the 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 tolerance techniques in form of fault tolerance patterns. The pattern specification consists of a service-based architectural design and deployment restrictions in form of UML deployment diagrams for the different architectural services. The work is illustrated with an application to guide the self-repair of the system after the detection of a node crash.

In [Harrison et al. 2010], the authors studied the interaction of fault tolerance tactics and architectural patterns. They proposed an empirical study on how to add fault-tolerance tactics, such as fault-detection and replication, to software architectural pattern, such as brokers, pipes and filters, for S&D engineering. The study shows the advantages/disadvantages of modifying these patterns by adding the tactic to the pattern before or after its integration.

To summarize, in software engineering, design patterns are considered as effective tools for the reuse of specific knowledge. However, a gap between the development of the system and the pattern information still exists. This becomes more exciting when dealing with specific concerns namely security and dependability for several application sectors.

3. OVERVIEW OF A PATTERN ENABLED DEVELOPMENT PROCESS

In our work, we promote the development of secure and dependable systems by means of patterns, as shown in Fig. 1. It includes the development of a pattern and its storage as well as its use within a system development process. The pattern development process includes tasks such as the representation, the organization and the definitions of relationships between patterns. The patterns usage focus on the problem of selecting and integrating patterns into an application. A suitable storage, e.g. in form of a central repository facilitates the pattern usage, but this aspect is out of the scope of this paper.

The left side of Fig. 1, i.e. the representation and development of patterns, is addressed in section 4 providing a modeling framework to capture S&D expertise for component-based development in the form of UML profiles and Patterns. The latter are also specified through dedicated UML profiles. The component-based development using the resulted patterns and UML profiles (right part of Fig.1) is addressed in section 5.

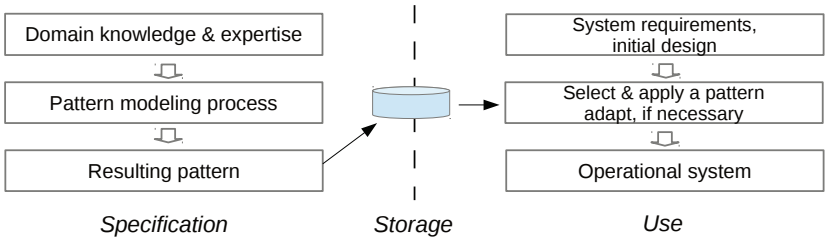

Fig. 1. Overview of a pattern enabled development processes

4. SPECIFYING PATTERNS – METAMODEL AND PROFILE

Design patterns have a certain number of elements that need to be captured by means of a pattern specification language. Our proposition is based on GoF [Gamma et al. 1994] specification, and its adaptation for security engineering [Schumacher 2003]. As we shall see, we refined it in order to fit with the non-functional needs. Adapting the definition of security pattern of [Schumacher 2003], we define a security and dependability pattern as a description of a particular recurring security and dependability problem that arises in specific contexts and presents a well-proven generic scheme for its solution. Therefore, a security and dependability pattern system is a collection of security and dependability patterns, together with guidelines for their implementation, combination and practical use in security and dependability engineering.

For our purpose, we propose to combine two well known approaches in MDE: a domain specific language (DSL) and a UML profile enabling pattern-specific information in UML models. The basic idea is that the former defines problem, objectives and constraints while the latter, the modeling-language specific part defines roles and solutions. The objective behind this separation is that the roles and solution defined in the pattern can only be integrated (without losing information) into the application architecture, if both are specified in the same modeling language, e.g. in UML. Conversely, problem statement and objectives are independent of a modeling language. The separation enables that solutions defined in different modeling languages share the same problem definition. This is useful, since we are storing a library of design patterns in a common repository. 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 modeling language independent part.

In the sequel we shortly present both specification languages: the modeling-language UML-independent part (SEPM) and a UML profile. We begin with a motivating example.

4.1 Motivating Example

Before formalizing design patterns, we illustrate patterns from a case study from the the railway domain: a simplified communication based train control (CBTC) system. It consists of wayside equipments and ground equipments and on-board equipment and communication lines. The application is a set of SW components providing the safety functions of the CBTC that will be replicated N times inside the equipments according to availability requirements and considered safe if M replica are constantly valid. This is namely the M out of N, MooN, architectural concept [Parhami 1994]. Wayside as well as ground as well as on-board equipment consist of replicated cards, typically 3 times. The communication is done via replicated lines (typically) two.

In case of MooN replication, the objective is to enhance the availability of a system while decreasing the probability of errors. MooN is not a single pattern, but merely an S&D pattern system that have different scopes: “node replication”, “Voter” and “Communication protection”. In Figure 2, the responsibility for

safety is shared between the safe application (which is replicated 3 times), the voter and the communication protection. In terms of safety, they shall present, i.e. they are allocated, the same SIL level.

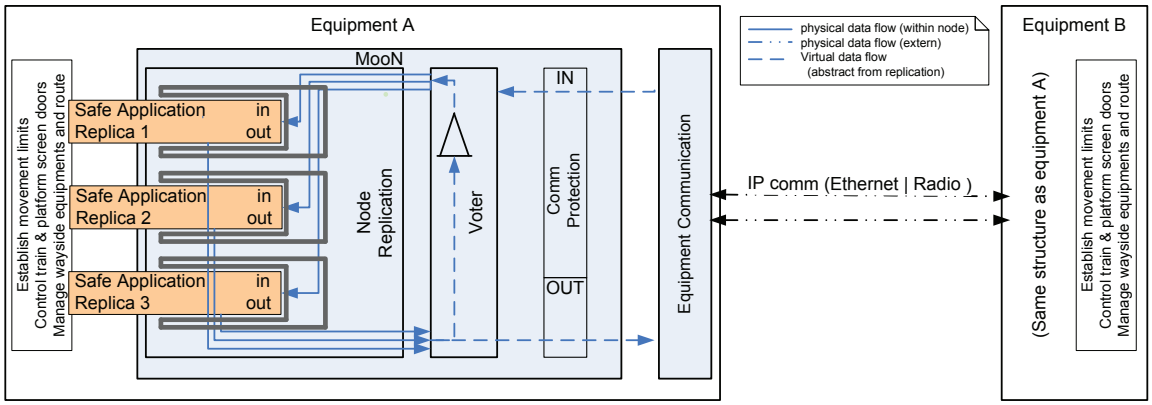

Fig. 2. Functions of a CBTC platform spread over two equipments and using MooN pattern

Now, we describe the pattern system and its constituent parts using a simple template.

MooN pattern system can be described as the following:

—Intent: manage dependability and communication for distributed functions.

—Context: use selected platforms with known availabilities of components and replication capabilities

—Solution: assemble and parametrize pre-designed reference architectures as pre-qualified solutions.

—Pattern language: it is composed of three S&D patterns: node replication, voter and communication protection patterns.

Node replication pattern This pattern allows the used of set of processing resources for the same task. It can be described as the following pattern:

—Intent: assure a high required availability with a set of hardware components. The individual failure rate of each of these component would not guarantee alone the required availability. The balance in the replication decision is in the amount of communication traffic.

—Context: safety applications with high availability

—Solution: replicate hardware, typically by using an equipment with multiple processing cards. Each card may either run the same or a diversified application software. The latter is used to protect against software faults. Results (including intermediate results) need to be synchronized between the cards by means of a *voter*.

Voter pattern In addition, each replica should receive identical inputs and outputs are voted. Intermediate results can be, but do not need to be voted. The voter pattern is a safe arbitrator of replica outputs which uses the safety strategy of the platform and returns to the application its conforming status. So, the replica can decide to isolate itself definitely as invalid data producer. The voter pattern can be described as the following pattern:

—Intent: evaluate a set of data inputs and make a majority vote, either accept only identical values or also those within a predefined error range. The balance in the decision must respect safety integrity level and performance constraints

—Context: either a hardware device or a pure software function.

—Solution: interface the hardware voter or spread the voter function between the hardware cards. Manage the quality of service with the voter parameters.

Communication protection pattern This pattern provides a service which ensures that the data from the sender arrives to the receiver in an ordered way and been prepared to correct a fault in the message, avoiding any kind of resending. This is achieved by build a safety interface in both sides of the channel and will comply EN 50159 [CENELEC 2010]. This pattern has multiple objectives:

(1) managing throughput: if different parts of a message are sent over the communication lines,

(2) enhanced availability and response time: when multiple identical messages are sent and depending on the validity check strategy or

(3) enhanced safety: the message can only be decoded, if different parts of it are received via both lines.

The communication protection pattern can be described as the following pattern:

—Intent: assure integrity of data passed over a communication link. The balance is between the portability (without re-qualification on each platform), conformity with standards, and the continuous possibility to upgrade while in operation

—Context: distributed communication between equipments

—Solution: use a message transformation architecture – assemble a set of fixed transformations in a specific order. A transformation could piggyback additional data (notably CRC data) to messages or perform more complex transformations such as encryption and splitting. Each transformation has a proven, unchangeable solution complying with EN50159 [CENELEC 2010].

The equipment communication is providing the external exchange facilities, usually a redounded IP communication through Ethernet and Radio; it is not modeled as a pattern.

The outlined patterns do not only address software, but also hardware and allocation. Software and hardware components are enriched with non-functional properties that describe the requirements on the software side and capabilities on the hardware side (e.g. reliability of processing modules or data link). Thus, we can only reason about the replication architectures (and patterns), if the modeling language is sufficiently rich to represent the properties of roles that are identified within a pattern. In section 4.3 we tackle the use of a suitable system modeling language that supports component-based architectures with non-functional properties.

Another aspect is the importance of validation: a major added value of applying the example patterns is to assure that a desired property, for instance a required availability or the protection of the communication channel is actually achieved. Thus, we need to trace pattern application and execute validation rules defined within the patterns, as addressed in section 5.

4.2 System and Software Engineering Pattern Metamodel – SEPM

Our pattern specification language is described using a meta-model which we call System and Software Engineering Pattern Metamodel (SEPM). It constitutes the base of our pattern modeling language, describing the concepts (and their relations) required to capture all the facets of patterns.

The principal classes of the metamodel are described with ecore notation – a core element of the Eclipse modeling framework[6] notation in Fig. 3 and the link with the property models. Security and Dependability (S&D) and resource models are used as model libraries to define the S&D and resource properties and constraints of the pattern. The reader is referred to [Ziani et al. 2011] for a full description of the properties metamodel. Note, however, that in concept such a modeling part is similar to the one proposed in [OMG 2008].

In addition to defining pattern concepts, the pattern metamodel provides an instantiation mechanism that enables the separation between the domain independent and domain specific on the one hand, and between the level of design on the other hand. The elements of the meta model are described in the sequel through the example of communication protection pattern presented previously. Note, however, that for the sake of clarity we only detail those elements related to S&D concerns.

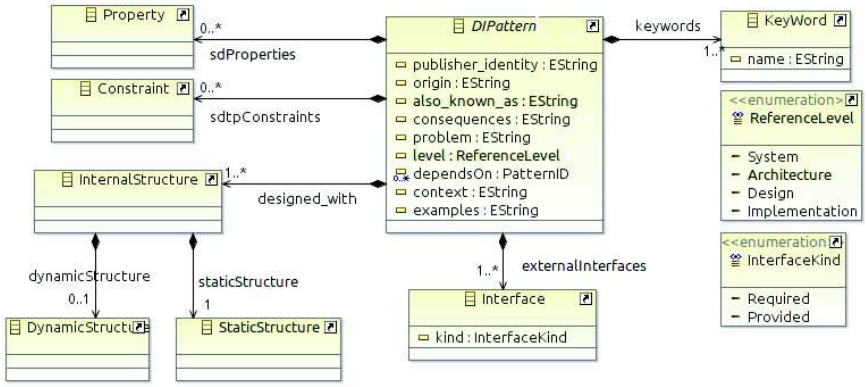

Fig. 3. The (simplified) SEPM metamodel

—*DIPattern.* This block represents a modular part of a system that encapsulates a solution of a recurrent problem. A *DIPattern* defines its behavior in terms of provided and required interfaces. As such, a *DIPattern* serves as a type whose conformance is defined by these provided and required *interfaces*. A *DIPattern* may be manifest by one or more artifacts, and in turn, that artifact may be deployed to its execution environment. The *DIPattern* has some fields to describe the related recurring design problem that arises in specific design contexts. These fields are based on the GoF [Gamma et al. 1994]. This is the key entry artifact to model pattern at domain *independent* level.

—*Interface.* DIPattern interacts with its environment via interfaces that have a modeling-specific representation.

[6]http://www.eclipse.org/modeling/emf/

—*Property.* Is a particular characteristic of a pattern. Each property of a pattern will be validated and the used assumptions will be compiled as a set of constraints. For our example, we define the following properties:
  —*integrity of data.* whenever some data d is received by the application receiver R, that same data d is send out by the application sender S.
  —*data freshness.* it is often desired that transmitted data d is fresh in terms of recentness. This property states that whenever an application receiver R receives some data d, the same data was sent by the application sender S at most Delta_t ago.
  —*non duplication.* given some data, that is sent by the application sender S, a receiver R will get this data at most the same number of times it was sent.
—*Constraint.* A set of requisites of the pattern. If the constraints are not met, the pattern will not be able to deliver its properties. For our example, we may specify constraints on the CRC computation/checking algorithms, on the correct/incorrect transmission over the network and on the maximal network delay.
—*Internal Structure.* Constitutes the implementation of the solution proposed by the pattern. Thus the *InternalStructure* can be considered as a white box which exposes the details of the *DIPatterns*. In our case, the internal structure is given by a means of a UML model, UML diagrams presenting the solution are presented in the next section.

4.3 Profile

As explained at the end of the example section, a modeling language for dependable systems with replication needs to be sufficiently rich to capture properties of the domain, such as hardware capabilities or interaction properties. Since UML is extensible via profile and widely used, we choose it as modeling language for application development. It is enriched via the profile MARTE for non-functional properties, and the profile FCM providing a flexible component model that enables us to refine ports, connectors and containers. For more information on FCM, consult [Jan et al. 2011], in the scope of this paper it is sufficient that FCM enables us to precise how interactions between components are realized. A further profile allows us to state whether a component is replicated and which replication schema (passive, active with vote, ...) shall be used.

Since UML is the base modeling language, we define a specification language for S&D pattern by means of a UML profile. The resulting profile allows pattern authors to use UML to specify S&D patterns. In addition to the concepts of the meta-model, we provide new capabilities for pattern application and traceability within the pattern specification language. In the next section, we show (1) how these techniques are developed and integrated into the UML editor Papyrus[7], and (2) how a pattern is applied during an engineering process for modeling applications with S&D requirements. In the sequel, we describe the most important elements of the profile.

—Pattern
  —*description.* the design pattern is principally a (named) holder for the description of problem, the solutions and the properties of these solutions.
  —*modeling solution.* a suitable and intuitive option is to map it to a UML package, i.e. to *extend* the UML meta-model element “package” with a stereotype “Pattern”. All elements that further characterize the pattern are elements *within* this package.

  Fig. 4 shows the definition of the MooN pattern system by means of UML and the design pattern profile. The MooN pattern itself is a UML package, problem and intention are stereotyped comments. Global properties, such as the reference level are specified by means of the profile. The sub patterns appear as nested packages.

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

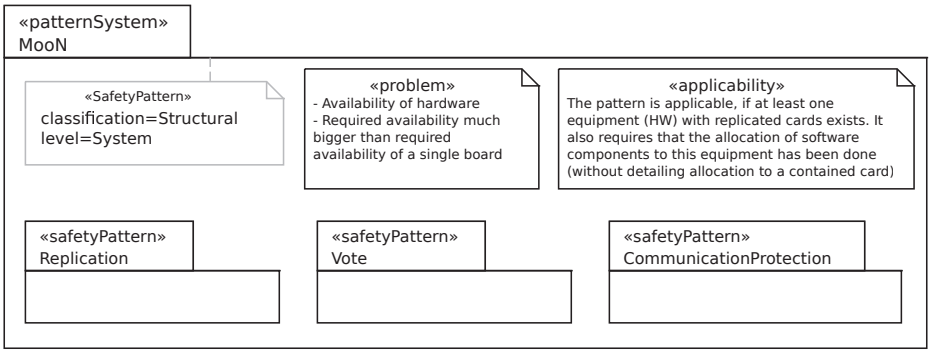

Fig. 4. The MooN design pattern modeled with UML and the design pattern profile

—Property
  —*description.* An intrinsic property of a pattern along with a value. It might reference a safety property defined by a standard, for instance the safety integrity level. Further examples are properties of the referenced model elements, such as a hardware capability.
  —*modeling solution.* We model properties via profiles dedicated to safety-critical systems. The mentioned hardware capability is a stereotype with two attributes: availability and error rate. Values for those are specified using MARTE's value specification language. We do not enter into more details, as this is not the focus of this paper.

—Constraint
  —*description.* A constraint that is defined over the roles within a pattern.
  —*modeling solution.* A UML constraint specified either via OCL or Java. This constraints references the associated roles via its constrained elements link. An example of a constraints is the calculation of the availability in function of the number of replicas.

—Role
  —*description.* Identify a role within a collaboration
  —*modeling solution.* We use the standard UML mechanism of a role within a UML collaboration, as shown in the state of the art in section 2. Roles are attributes within the collaboration, i.e. named and typed parts within the collaboration (similar to a composite class). The type may be used to characterize the role, e.g. by defining obligatory ports or associations.

Fig. 5 shows the formalization of two patterns, on the left-hand side the voter pattern, on the right-hand side the protection of communication between equipments. The former identifies two communicating safety-applications (application1 and application2) as roles, the latter in addition the hardware equipments (equipment1 and equipment2, each containing a fixed number of replicated cards) on which the applications are allocated. Please note that we use a generalization relationship between the solution class and the collaboration defining roles. Whereas a solution is not a specific variant of a collaboration capturing roles, the main objective is to provide a pragmatic way to enable referencing (inherited) roles within a solution. Please note that we show a variant of the patterns dedicated to two collaborating applications, in general, an arbitrary number of applications can be modeled via the cardinality of the roles. Please also note, that this formalization only holds in the context of component-based development (and not e.g. object-oriented development), in which hardware elements are also represented by parts in a certain way in the system model.

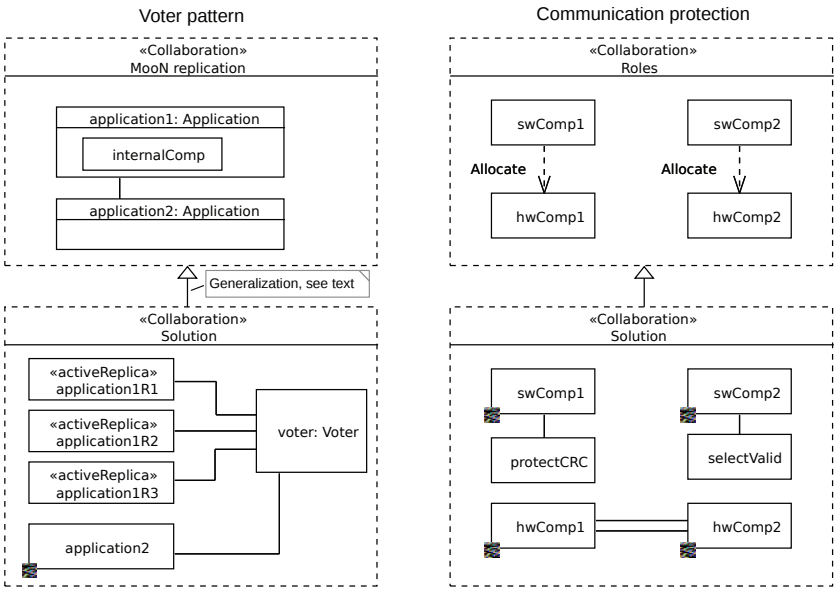

Fig. 5. Two patterns of the MooN pattern system: vote and communication protection

—Solution

—*description.* This element is a description of structure (or behavior) of a solution to the problem stated in the design pattern. For our context, the focus is to show the structure of a solution.

—*modeling solution.*The solution is a stereotyped collaboration. An example is shown in Fig. 5.

Whereas it is possible to specify roles and solution in the same collaboration, we often separate these two, letting the solution inherit from the collaboration. This enables the specification of multiple solutions for the same set of roles. If separated, the solution refines the role collaboration via inheritance, enabling the reuse of the roles that are already defined. In many cases, roles re-appear directly in a solution with new connections or associations, as in case of the inter equipment communication: the equipment roles have a dual connection and the applications pass data to/from a protection and selection component, respectively. Note that the connections between equipment1 and equipment2 represent physical links.

—Trace

—*description.* We need to identify two different kinds of traces. One links an element of the application model to a role defined within a pattern. The constraints defined within a pattern use the binding information to validate the bound application model elements. The second kind of trace is a link between the pattern and requirements. Their modeling is not in the scope of this paper.

—*modeling solution.* Use a stereotyped UML collaboration-use. The stereotype attributes store information when and by whom the pattern has been applied. The collaboration use has several role binding relationships to roles defined in the pattern. This is shown in Fig. 6 for the example of a CBTC system: the collaboration-use element in the middle contains the binding relationships, e.g. from cbtcApp to swComp1. In the used UML modeler, the targets of the role binding are only shown by their name. Therefore, we added stereotyped dependencies to the elements of the pattern solution (in the right hand side of the figure 6) for illustration purposes – we did that for the two software components only, the other elements are bound as well in the model.

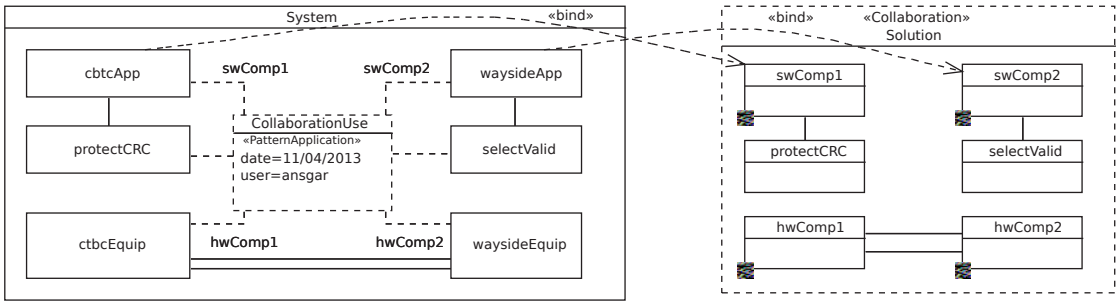

Fig. 6. Achieve traceability via role binding for the communication protection example

We do not need extra specification of traceability such as proposed by [Mirakhorli and Cleland-Huang 2011], since this binding to a role within a pattern serves as traceability link. An interesting aspect is that one of the patterns proposed by Mirakhorli and Cleland-Huang enriches the semantics of a traceability link by mapping it to a common proxy instead of distinguishing several kinds of trace relationships. The same is true for establishing a binding with a role within the pattern, since this role provides an implicit semantics for the traceability link behind the binding.

5. INTEGRATING DESIGN PATTERNS INTO THE DEVELOPMENT PROCESS

This section focuses on the use of safety patterns. We sketch the user guide in order to identify the requirements to adapt tools such as the Eclipse platform. The user shall follow the steps selection, instantiation, integration and validation. The usage context is the design of the software application, in our case a component model for a CBTC equipment. The MooN pattern is the upper part of the middleware that is hosting the application component; it is mainly a container providing availability. The rest or lower part of the middleware has to provide the vote between safe replicas and safety protected communication between replica and the voter and from/to the outside of the equipment.

Fig. 7 shows a development process with design patterns, starting with an initial architecture (which may be quite incomplete). The use of a pattern is an iterative process in which the user has to perform a set of actions: the first is to *select* a pattern from a set of available patterns. This choice may be filtered in a suitable way by the tool, as shown in section 5.1. It is followed by the integration of the selected pattern into the application model in section 5.3. Integration consists in copying elements of the pattern into the application model or binding existing elements to roles within the pattern. It is important to keep the role binding information to enable a trace that the pattern has been applied. Since a pattern is not a fixed solution, the application is typically adapted after pattern integration. Therefore, we need to ensure that these modifications do not violate assumptions that are made by the pattern. This is done by validation mechanisms, outlined in section 5.4.

5.1 Pattern selection

The selection of a design pattern is a primarily a choice of the developer. There are different aspects that may narrow and thus simplify the choice. The first is the objective of the pattern application. While it can not be formalized in general, some patterns address requirements that are defined by safety standards. If these requirements are stored in a model library and the pattern definition references the requirements, the pattern selection could be driven by the selection of a (safety) requirement.

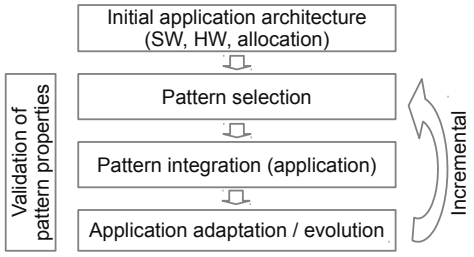

Fig. 7. The development process when using patterns

The second aspect is that patterns can be classified with respect to several properties. One is the reference level identified in section 4: a pattern may be relevant for the system, its architecture, its design or implementation aspects. Thus, it must be possible to filter available patterns with respect to this classification.

5.2 Role binding

As shown in the preceding section (paragraph trace), a *binding* between elements of the application architecture and a role defined in the pattern needs to be established. The developer is supported by the integration tool in two different ways. The first is to show only a filtered selection of possible elements for binding, in case of the equipment only those with replicated cards are candidates for binding. The second is to create bindings completely automated, for instance if application components affected by replication can be identified via the allocation to an equipment that has replicated cards. If a role remains unbound during integration, the developer indicates that no element of the application corresponds to it and the element must thus be created in the application model.

5.3 Pattern integration

Pattern *integration* means to apply a solution that is proposed by a design pattern in an existing application architecture. We may not simply copy such a solution into the architecture, but need to take the binding (previous section 5.2) of elements *already existing* in the application to roles identified in pattern into account. The challenge of this task is that relationships (e.g. connections, associations or inheritance) defined between roles in the pattern definition must be established in the application model as well. For example, the solution of the MooN pattern defines that the communication between a producer and a consumer is done by means of a dedicated interaction component that takes care of passing exchanged data to the voter, as shown in Fig. 5. However, application1 and application2 might already be connected via a different interaction component that is conflicting with the proposition in the pattern. Currently, applying a pattern adds new connections, associations, etc. and the user has to resolve potential conflicts. We aim to automate the detection of conflicts and propose solutions in a similar way as merge tools work.

5.4 Pattern validation

Design patterns have certain constraints that must hold while allowing a certain degree of modifications. For instance, the MooN pattern requires that software parts with a required availability need to be allocated to an equipment with a suitable number of replicated cards. This number needs to be calculated the moment the pattern is integrated, but the developer may exchange the hardware after the pattern integration has been done. The new hardware might have different capabilities that change the number of required replicas. Thus, there is a need to assure that properties of the pattern remain valid while not preventing modifications (such as for instance a renaming). The basis for *traceability* are the binding relationships between elements of the application architecture and roles identified in the pattern.

A verification rule associated with the pattern assures that invariants of the pattern can be checked after the pattern has been applied. In case of the MooN pattern, we implemented several constraints compatible with the Eclipse EMF validation framework. An example of such a rule is a check that the model element that is bound to the "equipment role" identified in the pattern is a hardware node containing the required number of replicated cards. The validation rules are currently implemented in Java, alternatively, a constraint language such as OCL could be used. However, the realization of such rules is not in the focus of this paper.

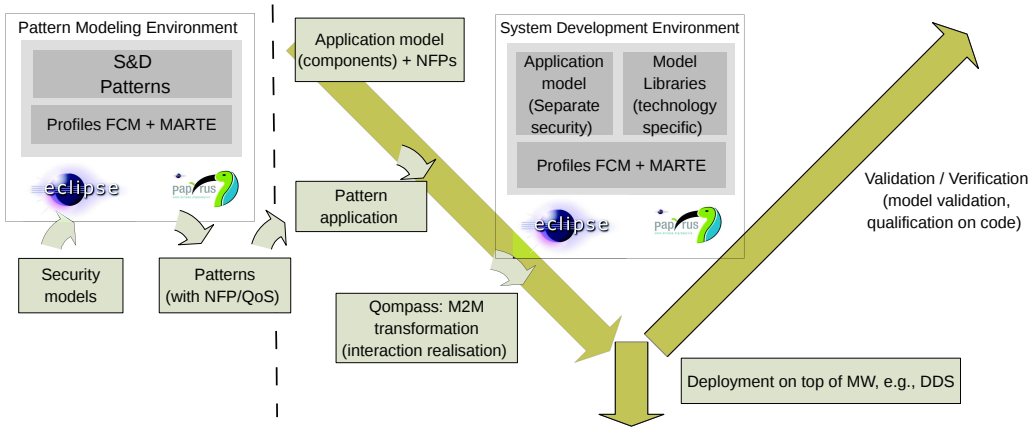

Fig. 8. Developed tool-chain

6. MODEL-DRIVEN TOOL CHAIN

To validate the modeling concepts (UML Profile) introduced in this paper, we have developed a model-driven tool chain to support the S&D engineering process based on our our approach. The tool-suite is visualized in Fig. 8. It consists of two tools: (1) the pattern development environment and (2) the system development environment. Both are based on the Eclipse based modeler Papyrus. The UML models are extended with a set of UML profiles including the S&D pattern profile developed in the context of this work.

The system development tool supports the pattern application. Fig. 9 shows a screen-shot of the pattern selection (on the left) and role-binding dialog (on the right) supported by our tool. The example is based on the communication protection pattern. Once a pattern is selected, roles need to be bound. The tool shows the available roles and displays candidates for binding. Roles can be bound automatically, if the pattern definition provides a Java function with this support. If a pattern is applied, an associated collaboration role containing role bindings is added to the system model.

After pattern allocation and deployment, a further model-to-model transformations support the expansions of connectors to interaction components and the code is finally deployed on a middleware such as DDS or MyCCM. Both aspects are out of scope for this paper.

7. CONCLUSIONS

Traditionally, design patterns are advantageous for solution reuse, however, they are hardly used when it comes to safety. This paper shows that safety design patterns have specific challenges related to the integration of safety requirements within the design pattern and their respect all along the process of system

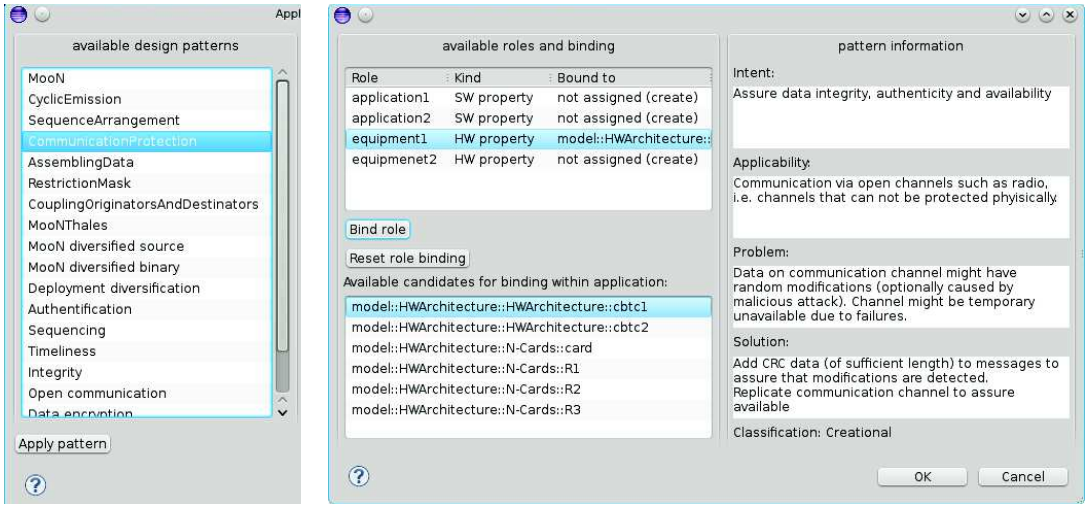

Fig. 9. Example of a pattern selection and application dialog including role binding

design. Since a solution proposed by a pattern is typically manually adapted to the concrete application context, we need to assure that this adaptation does not break the pattern's leitmotiv. Therefore, invariants of the pattern are checked by associated validation rules. To this aim, we propose the meta-model SEPM that enables to organize design patterns into a library easing their classification and their search. We further propose a UML profile that focuses on the description of pattern solutions with UML.

Some stereotypes in the profile contain information that is already present in the SEPM modeling-language, for instance the problem statement. We allow this duplication in order to enable a pattern developer to specify a complete pattern in a single environment. A transformation (in our case implemented with the OMG standard QVT – Query/View/Transformation) synchronizes the information, e.g. the problem statement between the UML model and the above-introduced SEPM modeling language, neutral pattern repository.

As a proof of concept, we evaluate our framework through the example of the MooN pattern for building systems having safety requirements as CBTC. As continuity to our present work, we envision to extend our tools in order to automate the whole process of design pattern definition, storage, search and application and offer a complete framework that supports pre-certified, safety-oriented design patterns. The application of such design patterns lightens certification steps that safety systems have to undergo.

Acknowledgment. This work was partially supported by the FUI (Fond Unique Interministeriel) project SIRSEC, the Systematic and i-Trans clusters and FEDER.
We like to thank our shephard Uwe Zdun and the participants of the Europlop writers workshop for their constructive critic and valuable hints for improvement of this paper.

REFERENCES

CENELEC. 2000. EN 50126-1: Railway Applications: the Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS).

CENELEC. 2003. EN 50129: Railway Applications: Communication, signalling and processing systems - Safety-related electronic systems for signalling.

CENELEC. 2010. EN 50159: Railway Applications: Communication, signalling and processing systems - Safety-related communication systems.

CENELEC. 2011. EN 50128: Railway Applications: Communication, signalling and processing systems - Software for railway control and protection systems.

Crnkovic, I. 2005. Component-based software engineering for embedded systems. In *Proceedings of the 27th international conference on Software engineering*. ACM, New York, NY, USA, 712–713.

Daniels, F. 1997. The reliable hybrid pattern: A generalized software fault tolerant design pattern. In Proc. of the Pattern Language of Programs (PLoP'97). *Pattern Language of Programs Proceedings*.

Foote, B., Rohnert, H., and Harrison, N. 1999. *Pattern Languages of Program Design 4*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Giacomo, V. D., Felici, M., Meduri, V., Presenza, D., Riccucci, C., and 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, Washington, DC, USA, 315–319.

Hamid, B., Gürgens, S., Jouvray, C., and Desnos, N. 2011. Enforcing S&D pattern design in RCES with modeling and formal approaches. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*. MODELS'11. Springer-Verlag, Berlin, Heidelberg, 319–333.

Harrison, N., Avgeriou, P., and Zdun, U. 2010. On the impact of fault tolerance tactics on architecture patterns. In *2nd International Workshop on Software Engineering for Resilient Systems (SERENE 2010)*. ACM, New York, USA.

Jan, M., Jouvray, C., Kordon, F., Kung, A., Lalande, J., Loiret, F., Navas, J., Pautet, L., Pulou, J., Radermacher, A., and Seinturier, L. 2011. Flex-eWare: a flexible model driven solution for designing and implementing embedded distributed systems. *Software: Practice and Experience 42,* 6.

Mapelsden, D., Hosking, J., and Grundy, J. 2002. Design pattern modelling and instantiation using DPML. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. CRPIT '02. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 3–11.

Mirakhorli, M. and Cleland-Huang, J. 2011. A pattern system for tracing architectural concerns. In *18th conference on Pattern Languages of Programs (PLoP)*. hillside.

Nicholson, J., Gasparis, E., Eden, A. H., and Kazman, R. 2009. Automated verification of design patterns with lepus3. In *Proceedings of the 1st NASA Formal Methods Symposium*.

OMG. 2008. OMG. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems,beta 2.

OMG. 2011. *Unified Modeling Language: Superstructure, Version 2.4.1*. OMG. OMG Document formal/2011-08-06.

Parhami, B. 1994. Voting Algorithms. *IEEE Transactions on Reliability 43,* 4, 617–629.

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

Schumacher, M. 2003. *Security Engineering with Patterns - Origins, Theoretical Models, and New Applications*. Lecture Notes in Computer Science Series, vol. 2754. Springer.

Taibi, T. 2007. *An Integrated Approach to Design Patterns Formalization*. IGI Publishing, 1–19.

Tichy, M., D.Schilling, and H.Giese. 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*. WOSS '04. ACM, New York, NY, USA, 105–109.

Yoder, J. and Barcalow, J. 1998. Architectural patterns for enabling application security. In *Conference on Pattern Languages of Programs (PLoP 1997)*.

Yoshioka, N., Washizaki, H., and Maruyama, K. 2008. A survey of security patterns. *Progress in Informatics* 5, 35–47.

Zdun, U. and 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*. OOPSLA '05. ACM, New York, NY, USA, 133–146.

Zdun, U., Hentrich, C., and Dustdar, S. 2007. Modeling process-driven and service-oriented architectures using patterns and pattern primitives. *ACM Trans. Web 1,* 3.

Ziani, A., Hamid, B., and Trujillo, S. 2011. Towards a unified meta-model for resources-constrained embedded systems. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 485–492.

Zurawski, R. 2007. Embedded systems in industrial applications - challenges and trends. In *SIES*.