A Language for Modeling Patterns for Extra-Functional Requirements

Brahim Hamid
IRIT, University of Toulouse, France
118 Route de Narbonne, 31062 Toulouse Cedex 9
France
hamid@irit.fr

***Abstract*—Model-driven engineering is well suited for the development of safe and heterogeneous systems, as it enhances the separation of concerns (i.e security, performance, analysis, simulation, etc.) through a declarative specification of behavior, e.g., by means of models that describe a system's functioning (where there are predefined configurations). In this paper, we deal with the idea of using patterns to describe extra-functional concerns as recurring design problems in specific design contexts, and to present a well-proven generic scheme for their solutions. To achieve this goal, we propose a Pattern Modeling Language to get a common representation to specify patterns for several domains. Our proposition is based on several levels of abstraction, for instance generic design (domain independent) and specific design (domain specific) levels. The aim of the generic design level is to catch, at high level, a set of generic properties by determining in advance if the artifact (e.g., pattern) has or uses a certain kind of generic properties. Then, specific domain design level allows to make more dedicated information. The approach enables us to define an engineering approach based on a repository of models and practices. It ensures separation of engineering concerns and roles between (1) application experts, (2) concerns experts and (3) MDE experts. The advantage of the language is illustrated by the modeling of the authorization pattern.**

***Index Terms*—Multi-Concerns engineering, Extra-Functional Properties, Pattern, Meta-model, Model Driven Engineering.**

I. Introduction

Extra-functional concerns become a strong requirement as well as more difficult to achieve even in safety critical systems. They can be found in many application sectors such as automotive, aerospace, and home control. Such systems come with a large number of common characteristics, including real-time and temperature constraints, computational processing, power constraints and/or limited energy and common extra-functional: such as dependability, security as well as efficiency [12]. Domains dealing with these concerns covers a wide spectrum of applications ranging across embedded real time systems, commercial transaction systems, transportation systems and military space systems, to name a few. The supporting research includes system architecture, design techniques, validation, modeling, software reliability and real-time processing.

The integration of such concerns, for instance security, safety and dependability, requires the availability of both application development and concerns expertises at the same time. Many domains are not traditionally involved in this kind of issue and have to adapt their current processes. Typically, such requirements are developed ad-hoc for each system, preventing further reuse beyond such domain-specific boundaries.

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 automated, even if some level of automatic code generation or even model driven engineering support is applied. Furthermore, many critical systems also have assurance requirements, ranging from very strong levels involving certification (e.g., DO178 and IEC-61508 for safety relevant embedded systems development) to lighter levels based on industry practices.

Over the last two decades, the need for a formally defined safety lifecycle process has emerged. The integration of extra-functional mechanisms is still new in many domains. Hence capturing and providing this expertise by the way of specific patterns can enhance safety critical systems development. Model-Driven Engineering (MDE) provides a very useful contribution for the design of these systems, since it bridges the gap between design issues and implementation concerns. It helps the designer to specify in a separate way extra-functional requirements at an even greater level that are very important to guide the implementation process. Of course, a MDE approach is not sufficient but offers an ideal development context. While using a MDE framework, it is possible to help concerns specialists in their task.

The question remains at which step of the development process to integrate these patterns. As a prerequisite work, we investigate the design process of patterns. The goal of the paper is to propose a new pattern development technique in order to make easy their use in a building process of software applications with multi-concerns support. Reaching this target requires to get (i) a common representation of patterns for several domains and (ii) a flexible structure for a pattern.

According to Buschmann [2], a pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. Unfortunately, most of exiting patterns are expressed in a textual form, as informal indications on how to solve some security problems. Some of them use more precise representations based on UML [1] diagrams, but these patterns do not include sufficient semantic descriptions in order to automate their processing and to extend their use. Furthermore, there is no guarantee of the correctness use of a pattern because the description does not consider the effects of interactions, adaptation and combination. This makes them not appropriate for automated processing within a tool-supported development process. Finally, due to manual pattern implementation use, the problem of incorrect implementation (the most important source of security problems) remains unsolved.

The solution envisaged here is based on meta-modeling techniques to represent patterns at a greater level of abstraction. Therefore, patterns can be stored in a repository and can be loaded in function of desired properties. As a result, patterns will be used as brick to build a applications through a model driven engineering approach.

The work is conducted in the context of a framework called *SEMCO* for System and software Engineering for embedded systems applications with Multi-COncerns support. We build on a theory and novel methods based on a repository of models which (1) promote engineering separation of concerns, (2) supports multi-concerns, (3) use *patterns* to embed solutions of engineering concerns and (3) supports multi-domain specific process. This project is three-folded: providing repository of modeling artifacts, tools to manage these artifacts, and guidelines to build complete engineering systems.

The rest of this paper is organized as follows. An overview of our approach is presented in Section II. Then, Section III describes in detail the pattern modeling language we propose. Section IV presents in depth the modeling part. In Section V, we examine a test case that has several S&D requirements: Secure Service Discovery. In Section VI, we review most related works that address pattern development. Finally, Section VII concludes this paper with a short discussion about future works.

II. Foundations and Conceptual Framework

The following subsection presents briefly the SEMCO framework, describes an example in order to illustrates the issues identified in the paper. Then, the structure of the pattern modeling is presented.

A. SEMCO Approach

SEMCO is a a federated modeling framework and the goal of Fig. 1 is to highlight the notion of integrated repository of metamodels to deal with system engineering. The proposed approach is to use an integrated repository of models to capture several concerns of safety critical embedded systems namely extra and non functional properties.

These artifacts will be used to capture in order to model all the facets of the system and its parts: logical (software and hardware components) and the infrastructure. They are provided as informal textual document, as semi-formal document using UML, SysML and Eclipse modeling framework, and as formal document using formal frameworks.

Currently, as shown in Fig. 1, SEMCO defines and provides 13 different artifacts types representing different engineering concerns and architectural information.

SEMCO approach promotes the use of patterns as first-class artifacts to embed solutions of extra-functional concerns such as safety, security and performance requirements of systems, specify the set of correct configurations, and capture the execution infrastructure of the systems, supporting the mechanisms to implement these concerns.

In this paper, we focus on the study of *pattern* artifact to deal with extra-functional concerns as recurring design problems in specific design contexts, and to present a well-proven generic scheme for their solutions. For that, we propose a language for modeling design patterns to get a common representation of patterns for several domains in the context of safety critical systems applications. Therefore, such a solution allows to capture appropriate characteristics of design concerns and to utilize several views. We begin describing our motivating example.

B. Motivating Example: Authorization Pattern

The essence of Fig. 4 is to promote the separation of general-purpose services from implementations. In our context, this figure highlights the separation of general-purpose of the pattern from its required mechanisms. This is an important issue to understand the use of patterns to target extra-functional concerns. In which layer related mechanisms are placed depends on the assurance a client has in how the services are in some particular layer. As example of a common and a widely used patterns, we choose the *Authorization Pattern* [13].

For instance, in a distributed environment in which users or processes make requests for data or resources, this pattern describes who is authorized to access specific resources in a system, in an environment in which we have resources whose access needs to be controlled. As depicted in Fig. 2, it indicates how to describe allowable types of accesses (authorizations) by active computational entities (subjects) to passive resources (protection objects). Such a pattern provides support to define possible ways of uses that applies to every level of the system.

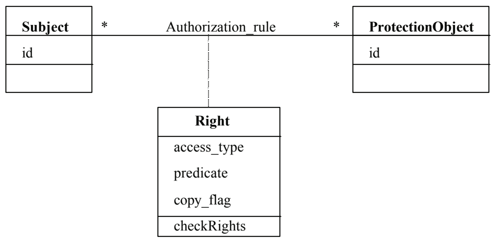

Fig. 2. Authorization Pattern

However, those authorization patterns are slightly different with regard to the application domain. For instance, a system domain has its own mechanisms and means to serve the implementation of this pattern using a set of protocols ranging

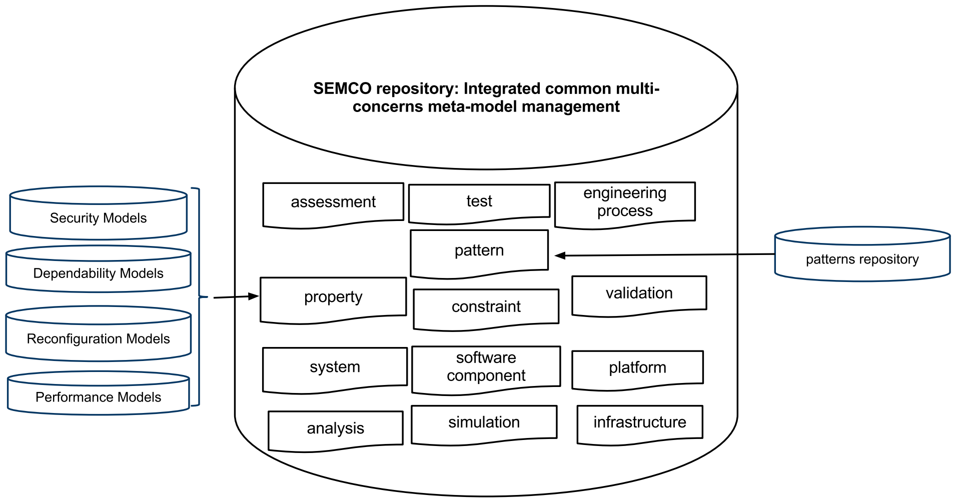

Fig. 1. SEMCO Generic Structure

from RBAC (Role Based Acess Control), FireWall, ACL'S (Acess Control List), Capabilities, and so on. For more breadth and depth, the reader is referred to [15]. In summary, they are similar in the goal, but different in the implementation issues for instance to determine for each active entity that can access resources, which resources it can access, and how it can access them. So, the motivation is to handle the modeling of patterns by following abstraction. In the followings, we propose to use *Capabilities* [15] to specialize the implementation of the authorization pattern. This solution is already used at the hardware and operating system level to control resources.

More specifically, the access rights of subjects with respect to objects are stored in an *access control matrix* M. each subject is represented by a row and each object is represented by a column. An entry in such a matrix $M[s, o]$ contains precisely the list of operations subject s are allowed to request on object o. More efficient way to store the matrix is to distribute the matrix row-wise by giving each subject a list of capabilities it has for each object. Without such a capability for a specific object means that the subject has no access rights for that object. Then, requests for resources are intercepted and validated with the information in the capabilities. The interception and the validation are achieved by a special program usually referred to as *reference monitor*. For instance, whenever a subject s requests for the resource r of object o, it sends such a request passing its capability. The reference monitor will check whether it knows the subject s and if that subject is allowed to have the requested operation r, as depicted in Fig. 3. Otherwise the request fails. It remains the problem of how to protect a capability against modification by its holder. One way is to protect such a capability (or a list of them) with a signature handed out by special certification authorities named *attribute certification authorities*.

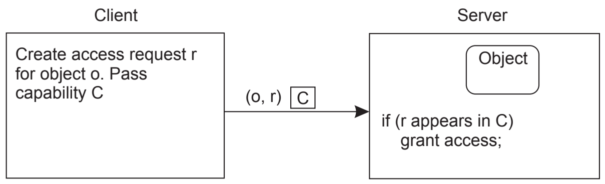
Client
Server
Create access request r for object o. Pass capability C
(o, r) C
Object
if (r appears in C) grant access;

Fig. 3. Protecting Resources using Capabilities

C. Pattern Metamodel Structure

One of the major considerations in designing multi-concerns safety critical systems is to determine at which level of abstraction concerns should be placed. The supporting research includes specification, modeling, implementation mechanisms, verification, etc. to name a few. For example, distributed systems are organized into separate layers following some reference models, e.g., applications, middleware and the operating system services. Combining the layered organization of target applications, domain specific systems and patterns modeling leads roughly to what is shown in Fig. 4.

The framework must cope with multi-concerns and domain specific properties. For this purpose, the proposition presented in this paper is based on a MDE approach and on three levels of abstraction: (i) Pattern Fundamental Structure (PFS), (ii) Domain Independent Pattern Model (DIPM) and (iii) Domain Specific Pattern Model (DSPM). Firstly this decomposition aims at allowing the design of multi-concerns applications in the context of safety (since combining extra-functional concerns and domain specific artifacts introduces a great complexity), and secondly to overcome the lack of formalism of the classical pattern form (e.g., textual).

III. Pattern Modeling Language

This section is dedicated to present our pattern modeling framework. As we shall see, the originality of this approach

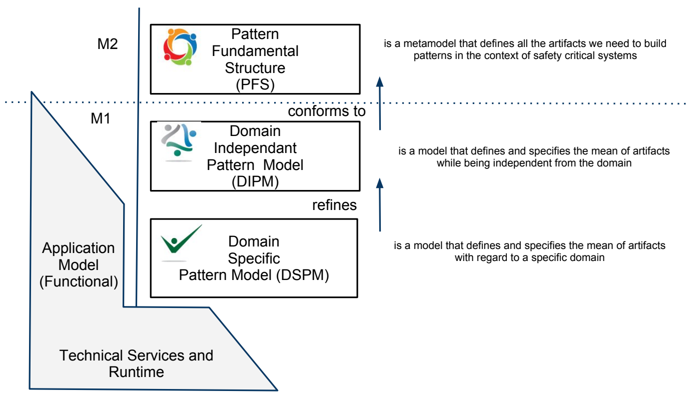
M2
Pattern Fundamental Structure (PFS)
is a metamodel that defines all the artifacts we need to build patterns in the context of safety critical systems
M1
conforms to
Domain Independant Pattern Model (DIPM)
is a model that defines and specifies the mean of artifacts while being independent from the domain
refines
Domain Specific Pattern Model (DSPM)
is a model that defines and specifies the mean of artifacts with regard to a specific domain
Application Model (Functional)
Technical Services and Runtime

Fig. 4. Pattern Modeling Framework Structure

is to consider patterns as building blocks that expose services (via interfaces) to deal with concerns (properties). This pattern definition provides a clear and flexible structure. Moreover, the modularity it enables allows to tame the complexity of large systems. As introduced into the last section, our pattern modeling language is based on three levels of abstraction. We start the description with a template inspiring our proposal (the patterns modeling language). Then, the first level of abstraction, namely PFS, will be described.

A. Patterns Documenting Model: Template

For our best knowledge, there is no consensus about the required information to represent patterns in the domain of software engineering, particularly when dealing with extra-functional properties. For this reason, we propose the following template. Note, however, that our proposition is based on GoF, and we deeply refined it in order to fit with the non-functional needs.

B. Pattern Fundamental Structure

The Pattern Fundamental Structure (PFS), as depicted in Fig. 6, is a meta-model which defines a new formalism for describing patterns and which constitutes the base of our pattern modeling language. Such a formalism describes all the artifacts (and their relations) required to capture all the facets of patterns. Here we consider patterns as building blocks that expose services (via interfaces) and manage properties (via features) yielding a way to capture meta-information related to patterns and their context of use. These pattern are specified by means of a domain-independent generic representation and a domain-specific representation. The next section details the principal classes of our meta-model, as described with UML notations in Fig. 6. Fig. 7 depicts in more details the meaning of principal concepts used to edit a pattern.

IV. Modeling Patterns

As mentioned earlier, our modeling framework promotes to use three levels of abstraction: (i) Pattern Fundamental Structure (PFS), (ii) Domain Independent Pattern Model (DIPM) and (iii) Domain Specific Pattern Model (DSPM). In

- *Name*. Meaningful word or phrase for the pattern to facilitate the documentation. This unique name gives a first idea about the pattern purpose,
- *Also known as*. Describes known occurrences of the pattern,
- *Related Patterns*. Here related patterns are mentioned
- *Example*. To show the use of the pattern,
- *Context*. To describe the conditions where the problem occur,
- *Problem*. Informal description of a problem that needs an appropriate solution,
- *Solution*. The solution describes how to solve the problem,
- *Precondition*. The set of conditions have to be met in order to be able to use the pattern,
- *Postcondition*. The impact of the pattern integration,
- *Attributes*. The set of information to configure and customize the pattern,
- *Properties*. The kind of properties the pattern provides to resolve the requirements.
- *Constraints*. This part describes constraints for a reasonable and correct use of the pattern.
- *Structure*. Indicates with class, sequence, and other UML diagrams, the form of the solution.
- *Interfaces*. To encapsulate patterns interaction functions. The way the pattern interact with its environment.

Fig. 5. Extra-Functional Pattern Template

this section, the required artifacts will be pointed out while following the two abstraction levels (i.e., DIPM and DSPM). These two levels with a authorization pattern presented in Section II-B are illustrated. Note, however, that for lack of space we only specify those principle elements.

A. Domain Independent Pattern Model (DIPM)

This level is intended to generically represent patterns independently from the application domain. This is an instance of the PFS. As we shall see, we introduce new concepts through instantiation of existing concepts of the PFS meta-model in order to cover most existing patterns in safety critical applications. In our case study, the DIPM of the authorization pattern consists of two communicating entities. The authorization pattern is defined as followed:

- Properties. At this level, we identify one property: *confidentiality*.
- External Interfaces. The authorization pattern exposes its functionalities through function calls:
  - $request(S, AT, PR)$: the subject S sends request about access type AT concerning the protected resource PR.
- Internal Structure. The behavioral of authorization pattern can be modeled by a UML Sequence Diagram following Fig. 2.

B. Domain Specific Pattern Model (DSPM)

The objective of the specific design level is to specify the patterns for a specific application domain. This level offers artifacts at down level of abstraction with more precise *information* and *constraints* about the target domain. This modeling level is a refinement of the DIPM, where the specific

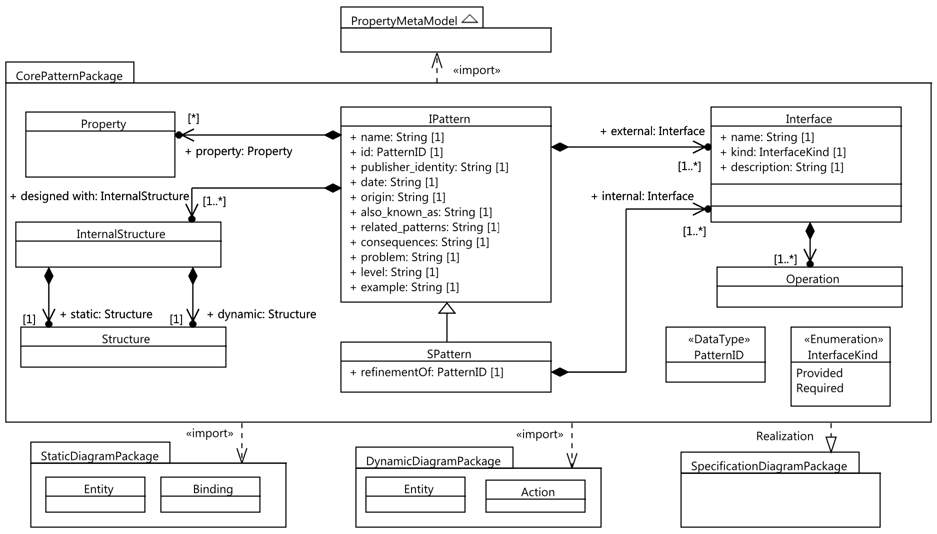

Fig. 6. Pattern Fundamental Structure

characteristics and dependencies of the application domain are considered. Different DSPM would refine a same DIPM for all needed domain. For instance, when using Capabilities as a mechanism related to the application domain to refine the authorization pattern at DSPM, we introduce the following artifacts:

- *Properties.* In addition to the refinement of the property identified in the DIPM, at this level, we consider: *deny unauthorized access*, *permit authorized access*, and *efficiency* properties.
- *External Interfaces.* This is a refinement of the DIPM external interface:
  $request(S, AT, PR, C)$: the subject S sends request about access type AT concerning the protected resource PR passing its capability C.
- *Internal Interfaces.* Let the subset of functions related to the use of capabilities to refine the authorization pattern:
  - $sign(C)$: the certification authority signs the capability C,
  - $verifyCert()$: the attribute capability certificate is verified,
  - $extractCap()$: the capability is extracted from the certificate,
  - $checkRight(S, AT, PR, C)$: the reference monitor verifies, using the capability, whether PR appears in the C.
- *Internal Structure.* The behavioral of authorization based on capability and reference monitor can be modeled by a UML Sequence Diagram following the description in Section II-B.

V. Secure Service Discovery for Home Control

In the following, an example will illustrate the approach point defined in the previous sections. The current trend aims at integrating more intelligence into the homes to increase services to the person. For this purpose, electronics equipments are widely used while providing easy and powerful services. However, to integrate all the services automatically requires a plug and play like system. For this issue, this example aims at providing a pattern for home control domains which provides a secure service discovery. Compared to a usual service discovery, this pattern will use a secure channel in order to protect all data. Fig. 8 shows two use cases: (i) adding a new equipment (ii) updating the current configuration. The Fig. 8 is described in form of UML notations and highlights the interfaces of the pattern in order to support the two main use cases. In the next subsections, the interface and the static internal structure will be pointed out while following the two abstraction levels (i.e., DIPM and DSPM) proposed by the paper.

A. Representation at DIPM: Person using a remote Internet-Box.

Regarding to the interface, it is necessary to declare some operations which allow the user to check if new implementations exist (i.e., update) and to detect the context (i.e., new equipment). Then, it is necessary to define all properties addressed by the pattern.

Fig. 9 illustrates the representation of secure connection at DIPM. At this level, we deal with a person using a remote InternetBox. As mentioned in the previous section, we choose *Authorization pattern* (or Access control) for security property. Regarding the internal structure of the pattern, we consider the following: a person uses a multi-media device which communicates with an internetBox via a Wifi connection.

B. Specialize a pattern through the DSPM: Subscriber using a remote OperatorBox

The interfaces must be adapted in order to match with the specific communication used in the domain. Regarding to the properties, at the DIPM, we only specify a very generic security property. At this level, it is possible to refine this property by defining the mechanisms, the length of the

- *IPattern*. this block represents a modular part of a system that encapsulates a solution of a recurrent problem. An *IPattern* defines its behavior in terms of provided and required interfaces. As such, an *IPattern* serves as a type whose conformance is defined by these provided and required *interfaces*. Larger pieces of a system's functionality may be assembled by reusing patterns as parts in an encompassing pattern or assembly of patterns, and wiring together their required and provided interfaces. An *IPattern* may be manifest by one or more artifacts, and in turn, that artifact may be deployed to its execution environment. The *IPattern* has some fields to describe the related particular recurring design problem that arises in specific design contexts. These fields are based on the GoF [5] information as described in Fig 5. This is the key entry artifact to model pattern at domain *independent* level.
- *Interface*. *IPattern* interacts with its environment with *Interfaces* which are composed of *Operations*. An *IPattern* owns provided and required interfaces. A provided interface is implemented by the *IPattern* and highlights the services exposed to the environment. A required interface corresponds to services needed by the pattern to work properly. So, larger pieces of a system's functionality may be assembled by reusing patterns as parts in an encompassing pattern or assembly of patterns, and wiring together required and provided interfaces. Finally, we consider two kinds of interface:
  - *External interfaces* allow implementing interaction with regard to the integration of a pattern into an application model or to compose patterns.
  - *Internal interfaces* allow implementing interaction with the platform. For instance, at a low level, it is possible to define links with software or hardware module for the cryptographic key management. These interfaces are realized by the *SPattern*. Please,note an *IPattern* does not have *InternalInterface*.
- *Property*. is a particular characteristic of a pattern related to the concern dealing with.
- *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 *IPatterns*. In order to capture all the key elements of the solution, the *Internal Structure* is composed of two kinds of *Structure*: *static* and *dynamic*. Please, note that a same pattern would be have several possible implementations[a].
- *SPattern*. inherits from *IPattern*. It is used to build a pattern at DSPM. Furthermore a *SPattern* has *Internal Interfaces* in order to interact with the platform. This is the key entry artifact to model pattern at domain *specific* level.

[a]Usually referred to as variants of design patterns.

Fig. 7. Pattern MetaModel Dependencies

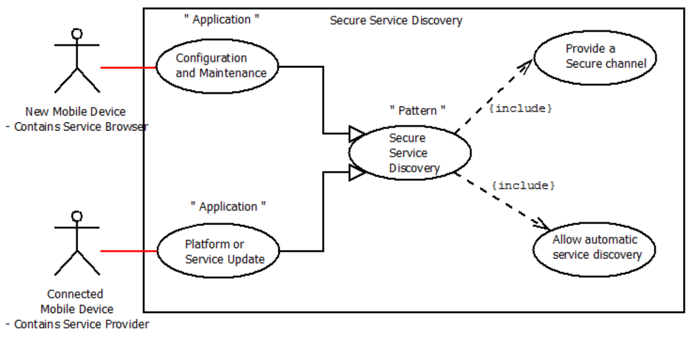
" Application "
Secure Service Discovery
Configuration and Maintenance
Provide a Secure channel
New Mobile Device - Contains Service Browser
" Pattern "
{include}
Secure Service Discovery
{include}
" Application "
Platform or Service Update
Allow automatic service discovery
Connected Mobile Device - Contains Service Provider

Fig. 8. Secure Service Discovery Use Cases

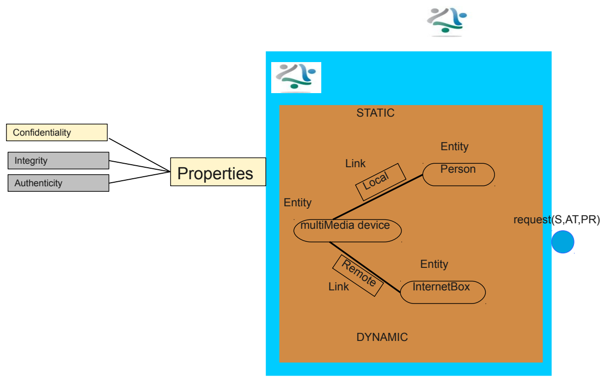

Fig. 9. Service Discovery example: Person using a remote InternetBox (DIPM)

keys, etc. Moreover, it is possible to add new properties. For instance, a RCES property can be added like the cryptographic time.

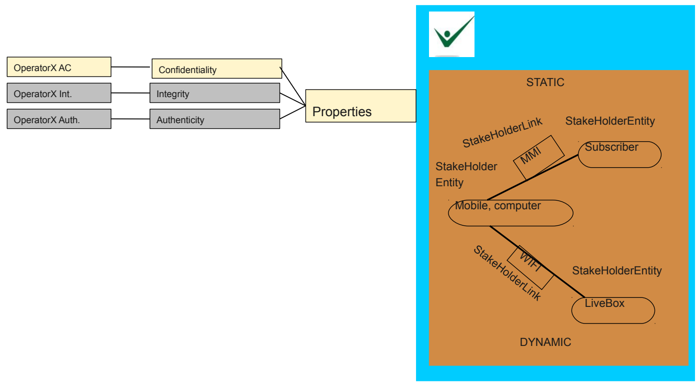

Fig. 10. Person using a remote InternetBox (DSPM)

Fig. 10 illustrates the representation of secure connection at DSPM. At this level, the pattern deals with an operator subscriber using a 'Operato'Box. For instance, we choose OperatorX AC for *Access Control*. The information expressed by the internal structure of the pattern is the following: an operator subscriber person uses a computer which is connected to a 'Operator'Box.

VI. Related Works

Design patterns are a solution model to generic design problems, applicable in specific contexts. Since their appearance, and mainly through the work of Gamma et al [5], they have attracted much interest. The supporting research includes domain patterns, pattern languages and their application in practice. Several tentatives exist in the literature to deal with patterns for specific concern [17], [7], [18], [3], [16]. 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. The pattern specification consists of a service-based architectural design and deployment restrictions in form of UML deployment diagrams for the different architectural services.

To give a flavor of the improvement achievable by using specific languages, we look at the pattern formalization problem. *UMLAUT* [8] is an approach that aims to formally model design patterns by proposing extensions to the UML meta model 1.3. They used OCL language to describe constraints (structural and behavioral) in the form of meta collaboration diagrams. In the same way, *RBML(Role-Based Meta modeling Language)* [10] is able to capture various design perspectives of patterns such as static structure, interactions, and state-based behavior. The framework *LePUS* [6] offers a formal and visual language for specifying design patterns. It defines a pattern in an accurate and complete form of formula with a graphical representation. A diagram in LePUS is a graph whose nodes correspond to variables and whose arcs are labeled with binary relations.

With regard to the integration of patterns in software systems, the *DPML (Design Pattern Modeling Language)* [11] allows the incorporation of patterns in UML class models. Recently, [14] explains how pattern integration can be achieved by using a library of precisely described and formally verified security and dependability solutions. Other domain specific solutions as [7], [18] exist.

While many patterns for specific concern have been designed, still few works propose general techniques for patterns. For the first kind of approaches [5], design patterns are usually represented by diagrams with notations such as UML object, annotated with textual descriptions and examples of code. There are some well-proven approaches [4] based on Gamma et al. However, this kind of techniques does not allow to reach the high degree of pattern structure flexibility which is required to reach our target. The framework promoted by LePUS [6] is interesting but the degree of expressiveness proposed to design a pattern is too restrictive.

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.

VII. Conclusion and Future Work

Extra-functional and non-functional concerns are not building blocks added to an application at the end of the life cycle. It is necessary to take into account this concern from the requirement to the integration phase. In this paper, we promote the use of patterns to provide practical solutions to meet these requirements and follow a MDE-based approach to specify such patterns. Indeed, MDE solutions allows to meet several concerns around one model while ensuring coherence between all businesses.

Here, we propose a common pattern modeling language to design multi-concerns safety critical system applications. This kind of application requires an adapted language to design it. Indeed, a classical form of pattern is not sufficient to tame the complexity of such application – complexity occurs because of both the concerns and the domain management. To reach this objective and to tame this complexity, our language is based on an advanced form of pattern using a MDE approach. The proposed approach is structured in 3-layer architecture. The first one corresponds to a metamodel which defines a generic structure of patterns. Then, two other layers are an instance of the metamodel. The two last levels allows us to integrate domain specific features at the end of the process.

The benefit of this structure is to offer a common language for different domain application. So far, this common language encompasses four industrial sectors, namely, home control, industry control, automotive, and metering [9].

As a side remark, note that our goal is to obtain an even high level abstraction to represent patterns to capture several facets of extra-functional and non-functional concerns in the different domain of safety critical systems applications, not an implementation of a specific solution. The key is then to show that the major sectors of such systems applications dealing with such concerns become covered by our approach. This result raises new and previously unanswered questions about general techniques to model these kind of patterns. We believe that this result is of particular interest to build a multi-concerns systems discipline that is suited to a number of sectors in safety critical systems.

The next step of this work consists in implementing other patterns including those for security, safety, reconfiguration and dependability to build a repository of multi-concerns patterns. Another objective for the near future is to provide guidelines concerning both the integration of all the presented results in a more global process with the pattern life cycle (i.e., create, update, store patterns) and the integration of pattern in an application. All patterns are stored in a repository. Thanks to it, it is possible to find a pattern regarding to concern criteria. At last, guidelines will be provided during the pattern development and the application development (i.e., help to choose the good pattern).

Acknowledgments. This work is initiated in the context of SEMCO framework. It is supported by the European FP7 TERESA project and by the French FUI 7 SIRSEC project.

REFERENCES

[1] OMG Unified Modeling LanguageTM (OMG UML), superstructure version2.2, 2009. Version 2.2 is a minor revision to the UML 2.1.2 specification. It supersedes formal/2007-11-02.

[2] G. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: a system of patterns*, volume 1. John Wiley and Sons, 1996.

[3] F. Daniels. The reliable hybrid pattern: A generalized software fault tolerant design pattern. In *PLOP 97*, 1997.

[4] B. P. Douglass. *Real-time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.

[5] E. Gamma, R. Helm, R. E. Johnson, and J.Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] E. Gasparis, J. Nicholson, and A. H. Eden. Lepus3: An object-oriented design description language. In *In: Gem Stapleton et al. (eds.) DIAGRAMS, LNAI 5223*, pages 364–367, 2008.

[7] V. Di Giacomo and al. Using security and dependability patterns for reaction processes. pages 315–319. IEEE Computer Society, 2008.

[8] A. L. Guennec, G. Sunyé, and J-M. Jézéquel. Precise modeling of design patterns. In *In Proceedings of the third International Conference on the Unified Modeling Language (UML'2000)*, pages 482–496. Springer-Verlag, 2000.

[9] B. Hamid, N. Desnos, C. Grepet, and C. Jouvray. Model-based security and dependability patterns in rces: the teresa approach. In *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems,SD4RCES '10*, pages 1–4.

[10] D k Kim, R. France, S. Ghosh, and E. Song. A uml-based metamodeling language to specify design patterns. In *Patterns, Proc. Workshop Software Model Eng. (WiSME) with Unified Modeling Language Conf. 2004*, pages 1–9, 2004.

[11] D. Mapelsden, J. Hosking, and J. Grundy. Design pattern modelling and instantiation using dpml. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 3–11. Australian Computer Society, Inc., 2002.

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

[13] M. Schumacher, E. Fernandez, D. Hybertson, and F. Buschmann. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2005.

[14] D. Serrano, A. Mana, and A-D Sotirious. Towards precise and certified security patterns. In *Proceedings of 2nd International Workshop on Secure systems methodologies using patterns (Spattern 2008)*, pages 287–291. IEEE Computer Society, September 2008.

[15] A. S. Tanenbaum and M. Steen. *Distributed systems, principles and paradigms, 2/E.* Prentice-hall, Inc, 2007.

[16] M. Tichy and al. Design of self-managing dependable systems with uml and fault tolerance patterns. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 105–109. ACM, 2004.

[17] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. In *Conference on Pattern Languages of Programs (PLoP 1997)*, pages 1–31, 1998.

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