Trusted Computing Engineering for Resource Constrained Embedded Systems Applications

Deliverable 4.2

Repository Structure Specification

Project:	**TERESA**
Project Number:	**IST-248410**
Deliverable:	**D4.2**
Title:	**Repository Structure Specification**
Version:	**v2.6**
Confidentiality:	**Public**
Editors:	**B. Hamid (IRIT), A. Ziani (IRIT)**
Date:	**14.02.2012**

Part of the Seventh Framework Program
Funded by the EC - DG INFSO

Control Sheet

Approval		
	Name	Date
Prepared	Brahim HAMID and Adel ZIANI	14.02.2012
Reviewed	All Project Partners	14.02.2012
Authorized	Antonio Kung	14.02.2012

Circulation	
Recipient	Date of submission
Project Partners	14.02.2012
European Commission	14.02.2012

Version History			
Version number	Date	Main author	Summary of changes
v2.0	01.10.2011	Brahim HAMID and Nicolas Desnos -IRIT	Initial version
v2.1	01.11.2011	Brahim HAMID -IRIT	Pattern Based Development and Pattern Development
v2.2	01.12.2011	Brahim HAMID -IRIT	Pattern, S&D, Resource and Constraint specifications
v2.3	10.12.2011	Brahim HAMID, Adel ZIANI -IRIT	Repository structure and APIs specification
v2.4	10.01.2012	Brahim HAMID -IRIT	Example of Secure communication and SSL pattern modeling
v2.5	10.02.2012	All	Internal Review V1
v2.6	14.02.2012	All	Internal Review V2

Contents

List of Figures

Executive Summary

The TERESA approach promotes a model-based approach coupled with a repository of modeling artifacts including S&D patterns, S&D models, Resource models and engineering process models and to define an engineering discipline for trust that is adapted to resource constrained embedded systems. The main goal of the WP4 is to define a modeling and development framework to support the specifications, the definitions and the packaging of the TERESA set of modeling artifacts to assist the developers of trusted applications for resource constrained embedded systems. After the presentation of the related background, we discuss the TERESA concepts as the basis for the definition of the common conceptual models. Then, we highlight the associated development framework and how it will be used in an MDE approach for a system development process to build trusted applications for resource constrained embedded systems. The other parts detail the specification models for TERESA modeling artifacts for patterns, properties and constraints, and the specification of the repository respectively.

This deliverable is the revised version of D4.2. The document is restructured as:

- More details about the examples of patterns from domains that will host the TERESA demonstrators. Currently, we use some inputs from the storyboard before TERESA.
- The TERESA conceptual framework is defined.
- The TERESA framework is proposed as an MDE framework with three principle parts: Specification of TERESA modeling artifacts, development of TERESA modeling artifacts and the structure of the TERESA repository and the way the repository stores models and patterns as specifications and instances.
- Each of these modeling artifact is defined. In this deliverable, we focus on the specification of pattern modeling artifacts and on the repository structure. The specification of the property, constraint and the process modeling artifacts are presented in D3.2. The instances of the property, constraint and pattern modeling artifacts based on the TERESA patterns are defined in D4.3. The instances of the process modeling artifacts are presented in D3.4.
- The repository structure and the specification of the API's are improved according to the TERESA requirements.
- We added a set of templates in order to evaluate the completeness of the proposed metamodels.

Some of the works presented in this deliverable have been published separately as referred conference papers:

- Section 5.2 was published as: Enforcing S&D Pattern Design in RCES with Modeling and Formal Approaches, in ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), with B. Hamid, S.Gurgens, C. Jouvray, and N. Desnos,
- A partial result of Chapter 3 was presented in the SD4RCES Workshop(SD4RCES10) as: Model-based security and dependability patterns in RCES: the TERESA approach, ACM DL, with B. Hamid, N. Desnos, C. Grepet, and C. Jouvray.

Some of the results in this deliverable which have not yet been published are:

- An extended version of the result of Section 5.2 is in preparation for submission to the SoSyM journal.

1 Introduction

Embedded systems [1] are not conventional software which can be built using usual paradigms. In particular, the development of resource constrained embedded systems (RCES) addresses constraints regarding memory, computational processing power and/or limited energy.

Non-functional requirements such as security and dependability (S&D) [2] become more important as well as more difficult to achieve. The integration of S&D features requires the availability of both application domain specific knowledge and S&D expertise at the same time. Currently, the integration of S&D mechanisms is still new in many domains (i.e., smart metering or home control), hence embedded systems developers usually have limited S&D expertise. In fact, capturing and providing this expertise by the way of *S&D patterns* can support embedded systems development.

Unfortunately, most of S&D patterns are expressed in textual form, as informal indications on how to solve some security problems. Some of them use more precise representations based on UML 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 of the application of a pattern because the description does not consider the effects of interactions, adaptations and combinations. This makes 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 security issues) remains unsolved.

The concept of model is becoming a major paradigm in software engineering. Its use represents a significant advance in terms of level of abstraction, continuity, generality, scalability, etc... Model Driven Engineering (MDE) is a form of generative engineering [3], in which all or a part of an application is generated from models. It looks promising since it offers tools to deal with the development of complex systems improving their quality and reducing their development cycles. The development is based on model approaches, meta-modeling, model transformation, development process and execution platforms.

MDE may be considered from two points of view: methodologists and developers. According to methodologists, an MDE process should define levels of abstraction, the modeling notations, the abstract syntax, how refinements are performed, how can a model be verified against the upper level model and how it can be validated. Developers consider the application of an MDE process as model driven refinement steps.

OMG MDA (Model-Driven Architecture) [4] is a guide that is currently considered as a reference in the MDE community. It aimed at defining a framework of the MDE. Using UML formalism, the framework offers tools to create generic meta models and meta-models to a specific area or platform (UML profiles).

Model-Driven Engineering (MDE) provides a very useful contribution for the design of trusted systems, since it bridges the gap between design issues and implementation concerns. It helps the designer to specify in a separate way non-functional requirements such as security and/or dependability needs at a higher level of abstraction. This allows implementation independent validation of models, generally considered as an important assurance step.

The question remains at which stage of the development process to integrate S&D patterns. As a prerequisite work, we investigate the design process of S&D patterns using MDE techniques. As we shall see, such a process evolves with other modeling artifacts, mainly artifacts to capture security and dependability properties and constraints. Stored in a repository, S&D patterns and their related modeling artifacts are then made available to be integrated into an MDE process to develop S&D applications based on RCES for various domains.

The motivation driving the modeling and formalization of security and dependability of software has typically been the need to amend the principal characteristics of the system targeting several domains with the same set of user requirements. Achieving this goal requires to get

- (1) a common representation of patterns for several domains,
- (2) a pattern flexible structure,
- (3) a unified formal validation of patterns,
- (4) guidelines for platform specific implementation of the patterns and
- (5) guidelines to guarantee the correctness of the pattern integration step.

The main goal of the $WP4$ is to define a modeling and development framework to support the specifications, the definitions and the packaging of a set of modeling artifacts to assist the developers of trusted applications for resource constrained embedded systems. In this document, we propose an approach for S&D pattern specification and validation that follows the MDE paradigm to produce a repository of integrated models (S&D patterns, Resource models, trust models, engineering process models,...). Security and dependability patterns on domain independent and domain specific level, respectively, that are derived from and associated with domain specific models will help developers to integrate application building blocks with S&D building blocks.

The rest of this document is organized as follows. Chapter 2 presents the background including a review of the most important related work and a synthesis. In Chapter 3, we discuss the TERESA concepts as the basis for the definition of the common conceptual models. In Chapter 4, we introduce our development framework. Then, in Chapter 5 and Chapter 6 we detail the specification of modeling artifacts models for patterns, properties and constraints, and the specification of the repository respectively. Chapter 7 describes a template for feedback on the modeling environment we propose. Finally, Chapter 8 concludes the deliverable with future work, mostly related to the elements of the framework need to be realized. An appendix is added with a list of abbreviations and acronyms.

2 Background

In this chapter, we outline different tools used in the TERESA solution including the *IRIT SEMCO* approach and some related standards. The next section presents a set of common concepts and definitions. The template section gives some concepts about the required information to represent patterns in the domain of software engineering in order to describe S&D patterns for each TERESA application domain. In the next part, we review briefly exiting solutions and related works. We start with security modeling and patterns modeling. Then, we study repository development for software and systems. The chapter is concluded with a synthesis and a discussion.

2.1 SEMCO

The following presents briefly the SEMCO Modeling Platform (System and software Engineering for embedded systems applications with Multi-COncerns) as *IRIT*'s under development system engineering platform.

We build on a theory and novel methods based on a repository of models which (1) promote engineering separation of concerns, (2) support multi-concerns, (3) use *patterns* to embed solutions of engineering concerns and (4) support multi-domain specific processes. This project is three-folded: providing repository of modeling artifacts, tools to manage these artifacts, and guidelines to build complete engineering systems.

SEMCO covers a wide spectrum of applications ranging across DRTES and RCES to name only a few. One of the major considerations in designing multi-concerns systems is to determine at which level of abstraction concerns should be placed. The supporting research includes system architecture, specification, modeling, implementation mechanisms, verification... For example, distributed systems are organized into separate layers following some reference models, e.g. applications, middleware and the operating system services.

SEMCO is a federated modeling framework and the goal of Figure. 2.1 is to highlight the notion of an integrated repository of metamodels to deal with system engineering. The proposed approach is to use an integrated repository of models to capture multiple concerns of safety critical embedded systems, namely extra- and non-functional properties. These artifacts will be used to capture all the facets of the system and its parts in order to model it: logical (software and hardware components) and the infrastructure. They are provided as informal textual document, as semi-formal document using UML, SysML and the Eclipse Modeling Framework, and as formal artifacts using formal frameworks.

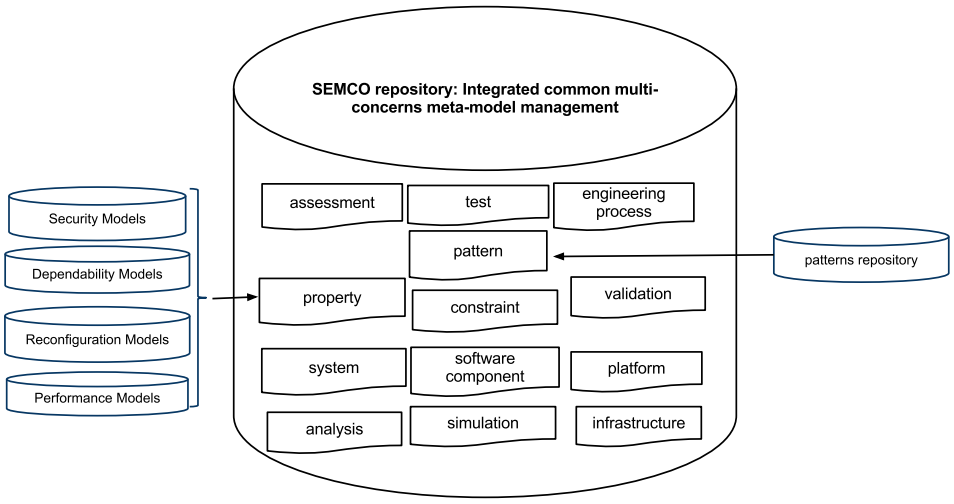


Figure 2.1: SEMCO Generic Structure

SEMCO solution, as depicted in Figure. 2.1, involves widely diverse core expertise ranging from:

- *Software Engineering*
- *Requirement Engineering*
- *Specification*
- *Modeling*
- *Formalization*
- *Implementation mechanisms*
- *Simulation*, ...

Today, design patterns are considered as fundamental technique to build software by capitalizing knowledge to solve occurring problems in many specific domains. *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.

Currently, in addition to the process, pattern and the repository itself artifacts (see Fig. 2.2) SEMCO defines and targets to provide 13 (see Figure 2.1) different artifact types representing different engineering concerns and architectural information.

The framework must cope with multi-concerns and domain specific properties. For this purpose, the proposition presented in this approach is based on a MDE approach and on

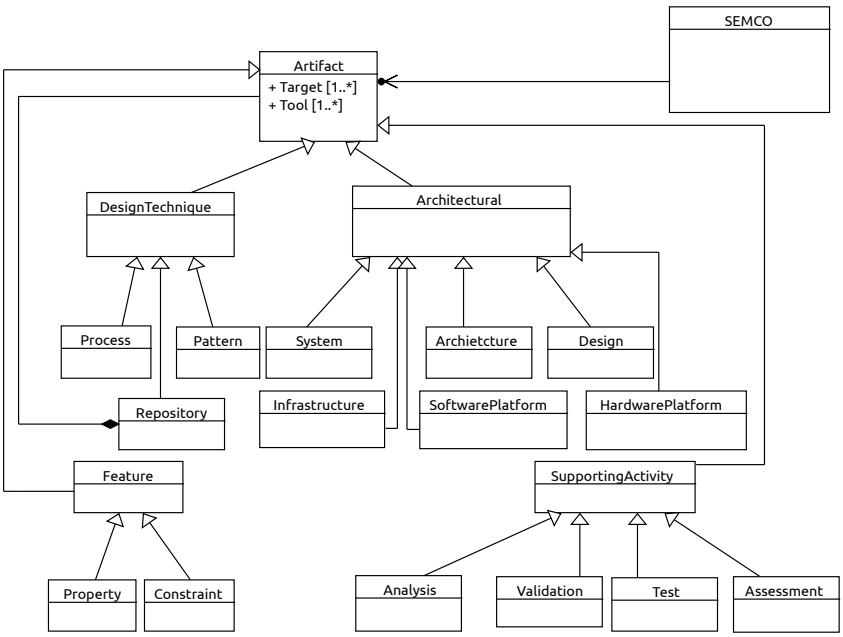


Figure 2.2: SEMCO Foundations and Conceptual Framework

three levels of abstraction: (i) Fundamental Structure Artifact (FSA), (ii) Domain Independent Artifact (DIA) and (iii) Domain Specific Artifact (DSA). The benefit of this structure is to offer a common modeling language for several domains in the context of safety critical systems.

SEMCO targets the definition of DSL-like languages to specify these artifacts at a high level of abstraction and then seeks practical ways to include these artifacts in existing engineering processes using/extending/customizing existing technologies to foster reuse. With regard to the $WP4$, we target a subset of artifacts as shown in Figure. 2.3 with respect the development of trusted RCES applications for four activities sectors: Automotive, Industry Control, Home control and Metering.

2.2 Definitions and Concepts

In this section, we introduce some terms and definitions about patterns and repository. The next section provides example of a template to specify patterns. Reader already familiar with these concepts and definitions may safely proceed to subsequent chapters and consult the definitions and notations herein only as needed.

Definition 2.1 (Pattern)**.** *The* GoF *book [5] defines design patterns as descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.*

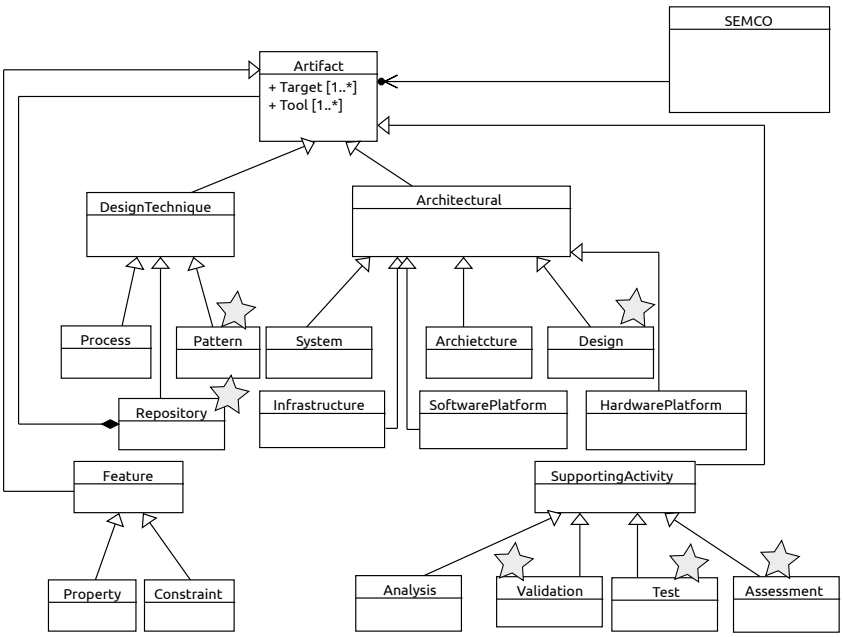


Figure 2.3: TERESA $WP4$

In [5], a design pattern abstracts the key artifacts of a common design structure that make it useful for creating a reusable object-oriented design. They proposed a set of design patterns for several object-oriented design problems. An example of code is depicted as a practical solutions that have been implemented using object-oriented programming languages.

The idea of design patterns was introduced not by a software developer, but by an architect (Urbanist), Christopher Alexander [6]. The first objective was to enhance the architectural quality, beauty, elegance and harmony in order to avoid the dehumanization of the living environment. It should be noted that the [5] work focuses on the object-oriented design, but introduces some ambiguities leading to use the term *design pattern* to refer to any pattern which addresses issues of software architecture, design or programming implementation. Several generalizations of this basis to describe software design patterns in general are proposed in literature. The most popular is the work presented in the book of Patterns of Software Architecture [7]. As depicted in the following definition, they made an important distinction between these three conceptual levels by categorizing them into architectural patterns, design patterns, and idioms (or coding patterns).

Definition 2.2 (Pattern)**.** *We adopt the following definition from [7]: 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.*

In this book, three types of patterns are defined:

- *Architectural Patterns.* An architectural pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- *Design Patterns.* A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.
- *Idioms.* An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Design patterns are medium-scale patterns comparing to architectural patterns but they are still at a higher level than the bare programming language.

In [8], authors made similar distinctions with the used language and its relation with the problem space or the solution space viewpoint. They introduced the following classification:

- *Conceptual Patterns.* A conceptual pattern is a pattern whose form is described by means of terms and concepts from an application domain.
- *Design Patterns.* A design pattern is a pattern whose form is described by means of software design constructs, for example objects, classes, inheritance, aggregation and use-relationship.
- *Programming Patterns.* A programming pattern is a pattern whose form is described by means of programming language constructs.

Definition 2.3 (Repository)**.** *According to Bernstein and Dayal [9], a repository is a shared database of information on engineered artifacts.*

In their paper, Bernstein and Dayal give a fundamental overview of repository technology as well as functional requirements of a repository. They introduce the fact that a repository has:

- a *Manager* for modeling, retrieving, and managing the objects in a repository,
- a *Database* to store the data,
- and *Functionalities* (to interact with the repository).

These definitions and their related observations were used as a basis for our conceptual pattern modeling language and repository structure (see Chapter 3).

2.3 Documentation Approach: 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. From the literature and Deliverable D2.1, we propose the following common pattern documenting artifacts in order (1) to build a first common understanding and to (2) represent examples of S&D patterns from our industrial partners in a uniform way. As we shall see, this template inspired our proposal: the S&D pattern modeling language.

- Name / Generalities.
  - Identification: 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.
  - Version.
  - Description / Intent /Properties to ensure.
  - Classification (creational, structural, beahvioral, ...).
  - Level: To be used at architecture, design or implementation level of the software development process.
  - Motivation.
  - Applicability.
  - ...
- *Problem*. Informal description of a problem that needs an appropriate solution
  - To be applied on and when.
  - Explain the problem and its context: Describes the conditions where the problem occur, when the pattern applies and whether or not in can be applied in view of other design constraints.
  - Conditions to be met to apply the pattern.
  - ...
- *Notation*.
  - Modeling Langauge (UML, SySML, ...).
  - Programming Language.
  - Code.
  - Pseudo code.
  - Text.
  - Example.
  - ...
- *Solution (independently of a particular application domain)*. The solution describes how to solve the problem. Indicates with class, sequence, and other UML diagrams, the form of the solution. As proposed in the [5], the design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities.
  - Description of the element of the design.
  - Description of the relationship between design elements.
  - Description of the responsibilities of the elements.
  - Description of the collaboration between elements.
  - Interfaces: To encapsulate patterns interaction functions. In other words, the way the pattern interact with its environment.
  - ...
- *Technical Solutions (for a particular application domain)*.
  - SW/HW service.
  - Low level middeleware service.
  - Properties to ensure: The kind of properties the pattern provides to resolve the requirements.
  - Safety constraint. Describes constraints for a reasonable and correct use of the pattern.
  - Resource constraint.
  - ....

- *Consequences*. Consequences and trade-offs of its use.
  - Results in applying the pattern.
  - Tradeoff in applying the pattern.
  - Design decision associated to the pattern.
  - Benefit of applying the pattern.
  - Impact on system extensibility.
  - Impact on system flexibility.
  - Impact on system portability.
  - ....
- Related Patterns. Mentions related patterns.
  - Links between patterns used at the same level (architecture, design or implementation).
  - Links between patterns used at different levels.

2.4 Examples of the TERESA Patterns

Deliverable D4.2 describes a subset of modeled patterns following the abstract informal pattern descriptions from Deliverable D2.1. This description is made in an informal way. The objective is to extract the common information we need to propose a common representation of the TERESA S&D patterns. This subset contains the most important patterns for our Use cases. The modeling of this subset of patterns is presented in Deliverable D4.3. Currently, we are working on the modeling of the other patterns of D2.1 and the results will presented in Deliverable D4.4.

TMR Pattern

Name / Generalities:

Identification: TMR
Version: 0.0
Description / Intent: Triple Modular Redundancy (TMR) is a fault-tolerant architectural design pattern, in which three computational channels perform a process and the result is voted producing a single output (e.g. 2oo3, 1oo3).
Classification: Structural.
Level: Safety level, Design level, Implementation level.
Motivation: To enhance reliability and safety in situations where there is no fail- safe state.
Applicability: This pattern is required for safety critical application in which a high safety integrity level is desired. Besides, it is a right solution for applications in which additional time to provide a correct output response in the presence of faults is forbidden.

Problem:

Context: A safety critical application that has high availability requirements, and that is exposed to different types of random faults.
Description: Provide protection against random and systematic (heterogeneous implementation) faults with the additional constraint that when a fault is detected, the input data should not be lost, nor should additional time be required to provide a correct output response.
Applied on: Safety critical architectures.
Applied when: The system must continue working even in the presence of a fault. Availability requirements are high.
In the presence of a fault, the result should be available without additional time.
Preconditions: SIL level to meet by the application.
The safety function has been identified.

Notation:

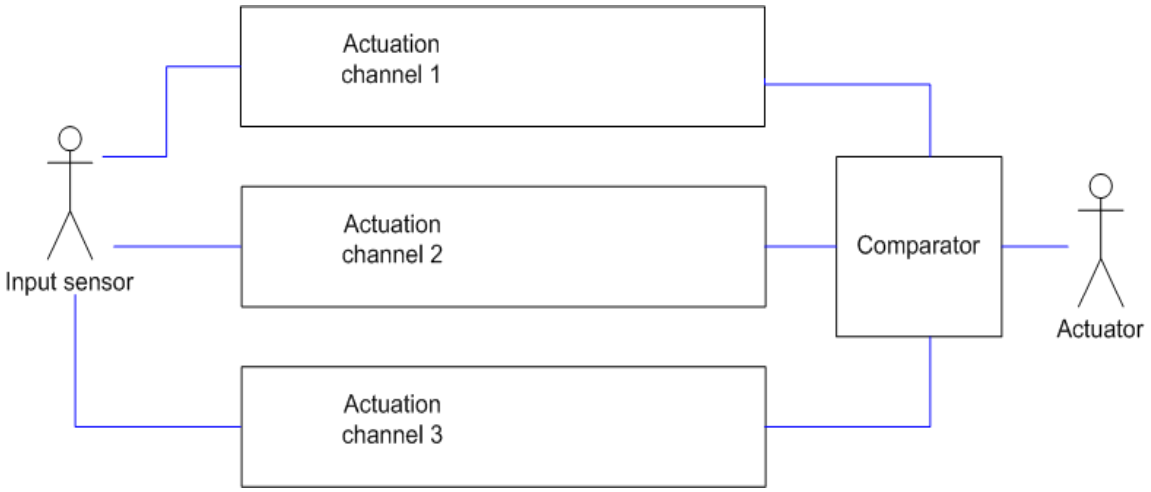
Modeling Language: SySML
Programming Language: None
Code: None
Pseudo code: None
Text: Description for the safety concept

Solution:

Description of the element of the design: This pattern is a structural pattern. It describes some strategies to follow at the definition of the system architecture. These strategies are as follow:

- Define a replicate structure (three channels)
- Independence between channels execution
- Define the degree of difference between each design of each channel (degree of heterogeneity).

Description of the relationship between design elements:



Description of the responsibilities of the elements:

- *Input sensor :* This is the source of information used to control the actuator. This can also be redundant.
- *Actuation channel:* The three channels could be all replicated homogeneous structures or some part like

HW or SW could be heterogeneous. All process the same input data, and perform identical operation, all channels executing in parallel

-*Comparator:* Takes the three outputs and implements a majority wins policy, discarding data from one channel if it deviates significantly from the other two. The Comparator just takes into account computational and time-lag jitter when it compares the values and decides which differences are significant and which are not.

-Actuator: It is the actual device performing the actuation

Description of the collaboration between elements: This pattern is a structural pattern, all the relation has been described in previous sections. The main points to remark are that the independence between channels must be guaranteed, and a comparator must be provided to select the correct output response.

Technical Solution:

Design pattern Voter: Depending on the kind of application, the design cost, and the kind of faults to detect. Some possible approaches are as follow:

- TMR Homogeneous
- TMR SW Heterogeneous
- TMR HW Heterogeneous

Consequences:

Results in applying the pattern:

- The system can provide the safety-related functionality regardless the presence of random/systematic faults.
- Not extra time is required to provide a correct response output in the presence of a fault.

Tradeoff in applying the pattern:

- Cost will increase because the number of replicate channels. In addition, this cost will be even higher if the channels are different, because that implies independent design flow for each channel. Also the cost will increase, because it is necessary to implement a comparator to get the correct response output

Design decision associated to the pattern:

- Use a comparator

Benefit of applying the pattern:

- Detection of random / systematic faults
- Increase the availability of the application

Impact on system extensibility:

- The system extensibility is penalized because whenever additional functionality is added to the system, this must be replicated in each redundant channel.

Impact on system flexibility:

- All the resources related to the redundant part of the system are replicated. Replicated CPU, memories, others.

Impact on system portability:

- If the system uses the heterogeneous version of this pattern, the system portability is much more complicated, because it has the same work of porting three independent designs.

Related Patterns:

Links between patterns used at the same level: None

Links between patterns used at different levels:

Voter Pattern: This pattern will be used to implement the comparator required for getting the correct response output.

Voter Pattern

Name / Generalities:

Identification: Voter

Version: 0.0

Description / Intent: This pattern requires the replication of the safety relevant part (independent channel). The results of each channel must be compared in order to know if there has been a failure in one of the channels.

Classification: Structural, Behavioral.

Level: Safety level, Design level, Implementation level.

Motivation: To detect and in some cases mask transient failures in the application.

Applicability: This pattern is required for application where solution as redundancy architecture is implemented. The degree of applicability is directly related to the SIL level to be achieved.

Problem:

Context: A critical application that must perform a safety-related functionality regardless the presence of a transient fault.

Description: A transient fault that could lead to a failure is not detected.

Applied on: Redundant architectures.

Applied when: The system must continue working even in the presence of a fault. Availability requirements are high. In the presence of a fault, the result should be available without additional time.

Preconditions: The system architecture has been defined (TMR, Redundant Architecture). The safety function has been identified. Data type to vote has been defined, and the type of voting algorithm has been selected.

Notation:

Modeling Language: UML, SySML

Programming Language: C++

Code: None

Pseudo code:

```
//Carry out the exact voting
int VoteServiceOutputsSB::vote(bool arg1, bool arg2, bool arg3, bool errnoNode2, bool errnoNode3,
bool& resul) {
  //Check the arguments
  //No degraded
  if (!errnoNode2 && !errnoNode3){
        }
  /No inputs from node 2
  }else if(errnoNode2 && !errnoNode3){/
        }
  //No inputs from node 3
  }else if(!errnoNode2 && errnoNode3){
        }
//FAILURE!!!
  }else{
  }
}
  //Check the results of the voting
void VoteServiceOutputsSB::checkVotes() {
  //Get the status of the error of each node
  bool errnoNodeN = getFailureMiddleware().localFailure;
  // Checking of votes
  //Voting no degraded
  if (!errnoNode2 && !errnoNode3){
     //If the local inputs differs from the voted inputs; throws F_OutputsExactLocalFailure
     exception
     if ( checkRangesExactVoting(sMaxAux, sMaxVoted, sMaxAux) == 1 ||
     checkRangesExactVoting(vWarningAux, vWarningVoted,vWarningAux) == 1 ||
     checkRangesExactVoting(serviceBrakeAux, serviceBrakeVoted) ||
     checkRangesExactVoting(warningAux, warningVoted)){
        }

     //If the three inputs are different from each other; throws F_OutputsExactAgreeFailure
     exception
        if(checkRangesExactVoting(sMaxAux,
     serviceOutputsNode2.sMax,serviceOutputsNode3.sMax) == 2 ||
     checkRangesExactVoting(vWarningAux,
     serviceOutputsNode2.vWarning,serviceOutputsNode3.vWarning) == 2 ){
        }

  //Voting degraded
  }else{
```

```
// To detect Error in node2 and 3
    if (errnoNode2){
        result =
checkRangesExactVoting(serviceBrakeAux,serviceOutputsNode3.serviceBrake);
        result2 =
checkRangesExactVotingDegraded(sMaxAux,sMaxVoted,serviceOutputsNode3.sMax);
        result3 =
checkRangesExactVotingDegraded(vWarningAux,vWarningVoted,serviceOutputsNode3.vWarning
);
        result4 = checkRangesExactVoting(warningAux,serviceOutputsNode3.warning);
}

//If the local inputs or the remote inputs differ from the voted inputs; throws
F_OutputsExactLocalFailure exception
if (result > 1 || result2 > 1 || result3 > 1 || result4 > 1){
    FailureSB_t failure = itsServiceApplicationSB->getFailure();
        failure.F_OutputsExactLocalFailure = true;
        itsServiceApplicationSB->setFailure(failure);
}else{
    FailureSB_t failure = itsServiceApplicationSB->getFailure();
        failure.F_OutputsExactLocalFailure = false;
        itsServiceApplicationSB->setFailure(failure);
}
}
}
```

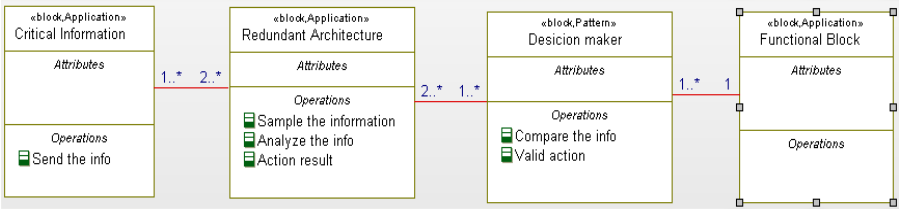
Text: Description for the safety concept.

Proof of concept provide by the formal validation – Detailed design phase

Solution:

Description of the element of the design: The functionality of this pattern is based on the detection of differences between its inputs (internal / external outputs of the application).

Description of the relationship between design elements:



Description of the responsibilities of the elements:

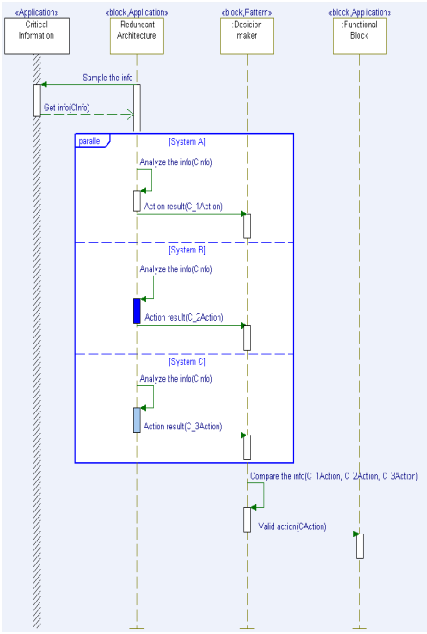
Redundant Architecture (Application): Provide the inputs required by the Decision maker for carrying out the comparison/voting.

Decision maker:

Compare the data coming from the independent channels.

Send the decision taken from the comparison/voting to the actuator.

Description of the collaboration between elements:



In the Sequence Diagram above is important to remark the concurrency of the operation within the voter. To express this concurrency the ***Interaction Operation*** artefact has been used along with the ***Parallel*** stereotype and a different color has been selected for each input data to emphasize this issue.

Besides, it is important to remark that the redundant architecture provides independent inputs to the decision maker. This independence is represented with different names of the actions (C_NAction) provide to the *"Decision Maker"*. The *"Critical Information"* could also be independent. This may come from different sources. For simplicity of the previous diagram this case has not been represented

Technical Solution:

Depending on the type and the number of input data, the algorithm used to vote could change. Some possible approaches are as follow:

- Inexact Voting
- Majority Voter
- Exact Voting
- Approval Voting
- Comparison
- ….

Consequences:

Results in applying the pattern:

- The system can provide the safety-related functionality regardless the presence of a random failure.
- The system can react in the presence of a fault.

Tradeoff in applying the pattern:

- Overhead and cost could increase owing to the "Redundant Architecture" of the system and the actions taken to monitor the status of the "Decision maker".

Design decision associated to the pattern:

- Redundant architecture

Benefit of applying the pattern:

- Detection of random failures
- Increase the availability of the application

Impact on system extensibility:

- The system extensibility does not appear to be affected, extra functionality could be added to the system without any impact, because all the functionality of the pattern are implemented at the interface of the modules, and is independent to the rest of the functionality.

Impact on system flexibility:

- The system is more flexible, because the pattern could adapt to different kinds of data input, and can change the internal algorithm used to carry out the voting.

Impact on system portability:

- The Voter could be implemented in HW or SW. This pattern does not impose any kind of restriction to the system portability.

Related Patterns:

Links between patterns used at the same level: None

Links between patterns used at different levels:

TMR Pattern: This pattern needs at least one Voter Pattern to vote its outputs. In some application, the TMR pattern used internal Voter Pattern to implement the synchronization of the independent channels.

Communication Ring

Name / Generalities:

Identification: Communication Ring

Version: 0.0

Description / Intent: The purpose of this pattern is to guarantee that the information share between the different components "Master / Slaves" of a distributed system is received by the Master regardless the presence of a random failures.

*Classification: S*tructural, Behavioral.

Level: Safety level, Architectural level, Design level, Implementation level.

Motivation: To detect failures during bus communication.

Applicability: This pattern is required for application where ring topologies are used in the communication of the different participant of the safety relevant application. Even in application not safety related, this kind of pattern could be used to increment the availability of the result at the master.

Problem:

Context: A communication critical application that must guarantee that the information share between the different components "Master / Slaves" of the system is received by the "Master" of the system regardless the presence of a random failure.

Description: The presence of random hardware failure in the system is not detected and this can lead to a loss of the information needed to perform the safety-related functionality.

Applied on: Communication Ring topologies.

Applied when: The system must continue working even in the presence of a fault. In the presence of a fault, the result should be available without additional time.

Preconditions: The Ring topology must be defined (Master and Slaves identified).

The selected bus for the application must support a bidirectional communication, and a way to send the frame in opposite directions of the line at the same time (e.g. two independent lines, Ethernet line, etc).

Notation:

Modeling Language: UML, SySML

Programming Language: C++

Code: None

Pseudo code: None

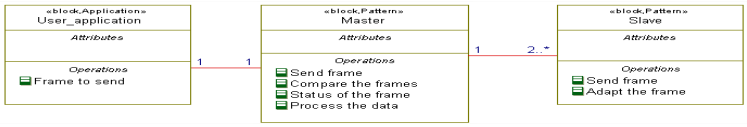
Text: Description for the safety concept

Proof of concept provide by the formal validation – Detailed design phase

Solution:

Description of the element of the design: The functionality of this pattern is based on the detection of a random hardware failure during the communication. Fault tolerance systems use information redundancy strategies to increase the system availability in the presence of a HW or SW fault. The Communication Ring pattern "Master /Slaves" supports sending information in both directions and at the same time decides if all the required information has been received.

Description of the relationship between design elements:



Description of the responsibilities of the elements:

User application: Send the frame to process (new information).

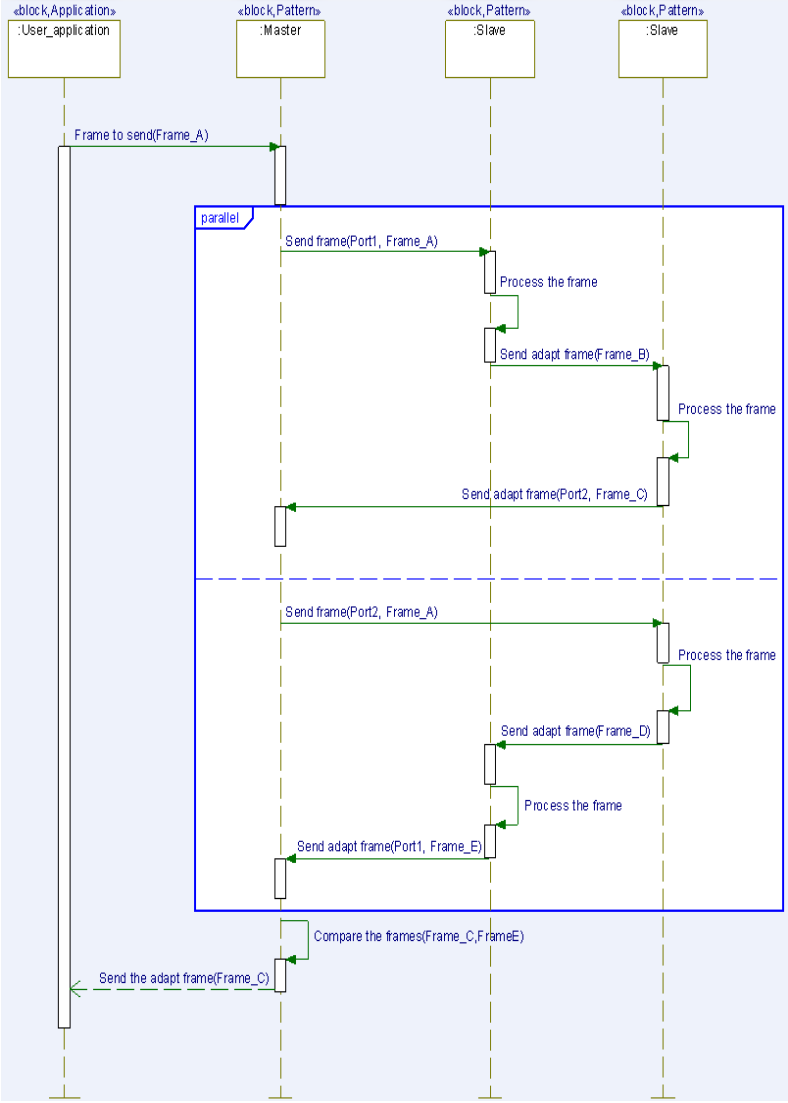
Master:

- Send the frame in both directions of the ring.
- Compare the frames coming from both direction of the ring.
- Evaluate the status of the frame received.
- Process de data received from the slaves.

Slave:

- Receive the frame from the Master or from other Slave.
- Adapt the frame with the new information and with its own identifier.
- Send the frame to other Slave or to the Master.

Description of the collaboration between elements:



Technical Solution:

The implementation of this pattern is directly linked to a specific communication mechanism that supports full duplex communication. Some strategies could be:

- Ethercat – Ethernet based fieldbus system. This protocol uses Ehternet full-duplex physical layers to support different kind of topologies as for example ring topology.
- Complete hardware redundancy. Two independent communication mechanisms with extra lines in order to detect failures in the communication.

Consequences:

Results in applying the pattern:

- The "*User application*" request can be carried out in a distributed system regardless the presence of a transient fault.

- The information resulting from the actions performed in each component "*Master/Slaves*" of the distributed system can be collected and analyzed in the "*Master*" regardless the presence of a transient fault.

Tradeoff in applying the pattern:

- The overhead of the application is increased because the information has to be sent twice in both direction of the Ring topology.

Design decision associated to the pattern:

- Redundant information.
- Redundant hardware.
- The application of this pattern is restricted to a type of bus that supports a bi-directional communication. Besides, the bus must support a way to send the information in opposite direction of the communication line at the same time.

Benefit of applying the pattern:

- Detect failures caused by a defect in the information transfer.

Impact on system extensibility:

- None

Impact on system flexibility:

- The flexibility of the system might look decremented because the constraints imposed by the type of communication mechanism (ring topology) required for its implementation.
- From a point of view of distributed systems, this solution easy the way to detect failures in communication. The flexibility of this kind of systems might be increased because the diagnostic coverage of the system will increase depending on the quality of the strategy used to guarantee the authenticity and the reliability of the data shared between slaves.

Impact on system portability:

  - The impact of this pattern in the system portability is negative, because its implementation is directly related to a specific type of communication mechanism that allows a full duplex communication.

Related Patterns:

Links between patterns used at the same level: None

Links between patterns used at different levels: None

Black Channel

Name / Generalities:

Identification: Black Channel

Version: 0.0

Description / Intent: The main purpose of this pattern is to simplify the way to guarantee that a communication at different levels in a system is reliable. This strategy leaves the responsibility of this task, at the interfaces of the elements participant in the communication.

Classification: Structural, Behavioral.

Level: Safety level, Architectural level, Design level, Implementation level.

Motivation: To detect failures during communication at different levels as for example on chip, inter boards, systems, others.

Applicability: This pattern is required for safety related systems where the data communication is directly involved in the implementation of its safety function.

Problem:

Context: In a safety related system or in application, the data communication is a vital part in its development. This communication could be at different levels: On chip, inter boards, systems, others.

Description: The presence of random failures in the transfer of data within different levels of an application, such as, transmission errors, repetitions, deletion, or others, can lead to a loss of information, or to an improper delivery of information. This pattern provides the capability to guarantee the correct transmission of the information to the right destination and at the right time through a standard communication mechanism.

Applied on: Data communication at different level of an application, such as, inter board communication, on chip communication, or others.

Applied when: Data communication is implied in the implementation of the safety function. The communication of safety related information is carried out through a standard communication mechanism.

Preconditions: The elements that share the information must be safety-relevant

The platform must support the implementation of an error detection code.

Notation:

Modeling Language: UML, SySML

Programming Language: C++

Pseudo code:

```
//Prepare and send outputs message
void Task_ServiceApplication::sendDataToMiddleware(EmergencyAppEmergency2Service & e2s) {

//The previous function uses the prepareMessage function to prepare the message and the sendData to sent information. E.g.:
        crc_.prepareMessage(tmp_data.service, local_node_id_, step_);
        middleware_.sendData(tmp_data);
        }

//Prepare the message
void prepareMessage(T& msg, S source, uint32 seq) {
        msg.CYCLE = seq;
        msg.SOURCE = source;
        msg.CRC = 0;
        msg.CRC = calcCRC(msg);
        }

//Calculate the CRC
uint32 calcCRC(T& in) {
        unsigned char buf[sizeof T];
        uint32 size = ((T&)in).pack(buf);
        uint32 crc = crcFast(buf, size);
        return crc;
        }

crc crcFast(unsigned char const message[], int nBytes) {
        }

//Send the message
bool DDSMiddleware::sendData(data & p) {
        return writer_data_->write(*((data_Type*)&p));
        }
```

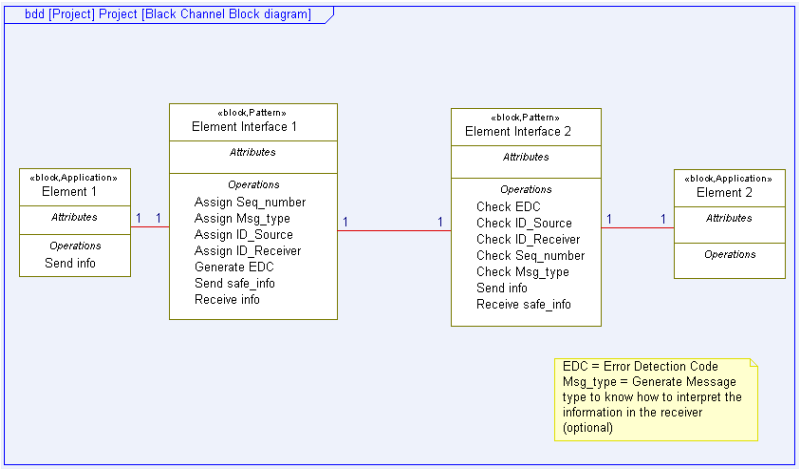
Text: Description for the safety concept. Proof of concept provide by the formal validation – Detailed design phase

Solution:

Description of the element of the design: The functionality of this pattern is based on the detection of a random hardware failure during the communication. Safety standards let the designer to use standard communication mechanism to share the information in a safety function, but with the condition to guarantee the reliability of the information through internal safety interfaces.

The Black Channel provides specific interfaces to guarantee the reliability of the information transfer, and the authenticity of the participants of the communication.

Description of the relationship between design elements:



Description of the responsibilities of the elements:

Element N: Participants of the communication. It could be independent hardware channels, internal modules, or others.

Interface 1:

- 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 Error Detection Code; to detect failures in the information transfer.

- Pack the information before send it over the standard communication mechanism.

Interface 2:

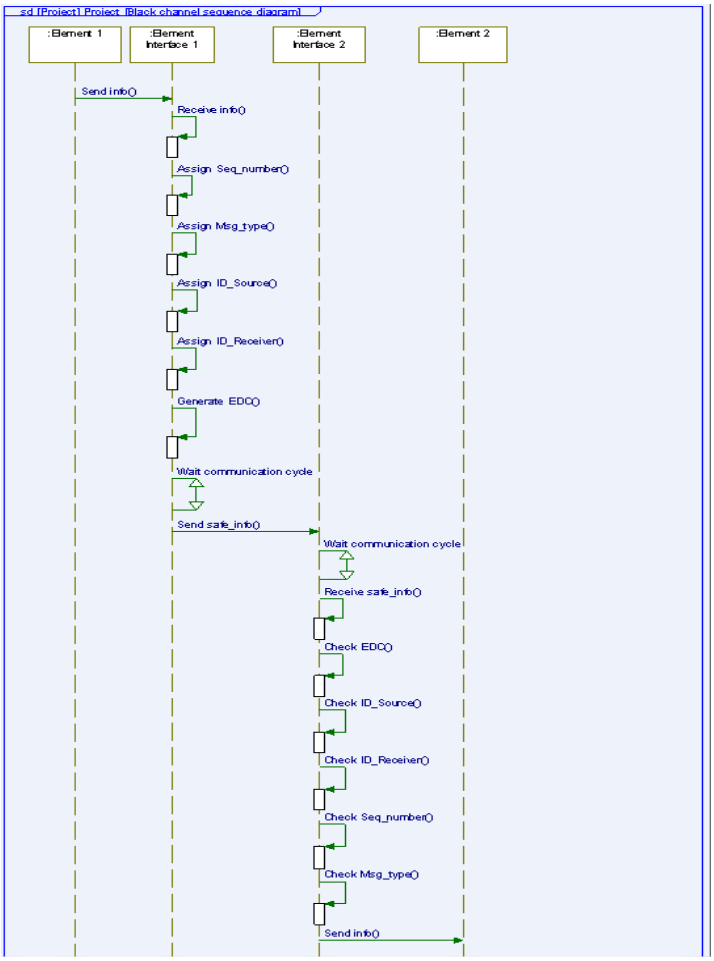
-Unpack the information.

-Before use the information received, check the package to probe:

- Origin of the information (source)
- Destination of the information (receiver)
- Order of the information received
- Information has not been corrupted
- Others….

-Send the information received to the receiver.

Description of the collaboration between elements: In the sequence diagram below is important to remark that the actions of sending and receiving the safety message must be cyclical and periodical because to the fact that elements involved in the communication must always know when the message should be sent and when it should be received.



Technical Solution:

The implementation of this pattern is carry out in the elements participant in the communication. The pattern is comprised by two kind of interfaces, one in charge of generating the package (data + safety techniques), and send the packages, and the other one in charge of receiving the package, unpack the data, and check the data. The package is composed of the data, plus some safety techniques as follow:

- Identifier of the source
- Identifier of the receiver
- Sequence number - To identify the sequence of the package to be sent.
- Message type – To know how to handle the received data.
- Error Detection Codes – This technique enable reliable delivery of data over unreliable communication channels. Depending on the type of failures to detect, and the diagnostic coverage to be achieved, the type of the EDC can change. Some EDCs are:
    - Repetition codes
    - Parity bits
    - Checksum
    - Cycle redundancy checks (CRC)
    - Hash functions

It is important to remark, that both the source and the receiver always must know when the data is sent and received. It is important to synchronize both parts of the communication.

Consequences:

Results in applying the pattern:

- The information shared in an application at different levels can be transmitted in a secure manner over a standard communication mechanism in order to avoid corrupt the information by different kind of intruders.

Tradeoff in applying the pattern:

- The implementation of this pattern increases the computational cost of the application. This increase is caused by the calculation of the EDC.
- The amount of information through the communication mechanism is increased because the transmitted data size is larger. This increase is caused by the extra information needed for identification and error detection of the data.

Design decision associated to the pattern:

- Implementation of specific safety interfaces in the elements involved in the communication.
- Extent the information, to support error detection in the communication, authenticity of the participants, others.

Benefit of applying the pattern:

- Detect failures caused in the transmission of the data over a standard communication mechanism (unreliable channel).

Impact on system extensibility:

- In non-safety related application, this pattern is also useful because might contribute to transform a standard channel in a reliable one.

Impact on system flexibility:

- The flexibility of the system is increased because a same pattern can be used in different data communication at different levels, and because the pattern can be easily adapted to a solution more or less restrictive, just changing the type of EDC used for error detection.

Impact on system portability:

- System portability is not affected. All the functionality of this pattern is implemented at the interfaces of the modules to communicate (independent of the communication mechanism), and is really flexible for platforms with less resources, because the EDC algorithm to use could be less efficient, but depending of the application, it could still provide the required functionality.

Related Patterns:

Links between patterns used at the same level: None

Links between patterns used at different levels: None

HMAC Pattern

Name / Generalities:

Identification: HMAC

Version: 0.0

Description / Intent: The purpose of this pattern is to guarantee that communication in an application is secure.

Classification: Structural, Behavioral.

Level: Safety level, Architectural level, Design level, Implementation level.

Motivation: To guarantee the data integrity and the authenticity of the message.

Applicability: This pattern may be used in different kind of application where the data share in the application might be subject to an attack that can corrupt it.

Problem:

Context: A critical application that must guarantee the integrity of the information shared between different components "Nodes, channels, etc" through an unreliable medium.

Description: This pattern provides the capability to ensure the integrity of the data and the authenticity of the message transmitted over a standard communication mechanism.

Applied on: Mainly to external interfaces, where the access to the data is easier, but it can be implemented at any level of communication of an application.

Applied when: The risks of external intruders that may corrupt the information in order to generate a failure in the application are high. The communication of the information is carried out through a standard communication mechanism.

Preconditions: The elements that share the information are part of the community of users, and each one knows the secretKey. The platform must support the implementation of a hash function.

Notation:

Modeling Language: UML, SySML

Programming Language: C++

Code: None

Pseudo code: None

Text: Description for the safety concept

Proof of concept provide by the formal validation – Detailed design phase

Solution:

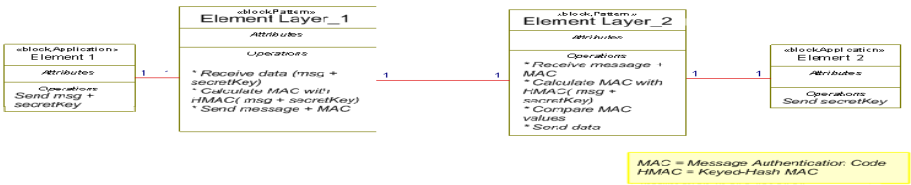
Description of the element of the design:

This pattern is characterized to detect any change in the information shared in the application. Moreover, this pattern guarantees that only the authorized participants of the communication can participate in it.

HMACs have two functionally distinct parameters, a message input and a secret key known only to the message originator and intended receiver(s). Additional applications of keyed-hash functions include their use in challenge-response identification protocols for computing responses, which are a function of both a secret key and a challenge message.

An HMAC function is used by the message sender to produce a value (the MAC) that is formed by condensing the secret key and the message input. The MAC is typically sent to the message receiver along with the message. The receiver computes the MAC on the received message using the same key and HMAC function as was used by the sender, and compares the result computed with the received MAC. If the two values match, the message has been correctly received and the receiver is assured that the sender is a member of the community of users that share the key.

Description of the relationship between design elements:



Description of the responsibilities of the elements:

Element N:

They are the participants of the communication. It could be independent hardware channels, nodes, internal modules, or others. They are in charge of sending and receiving the information to share; besides they are in charge of providing the authentication key.

Element Layer 1:

This element acts as an interface to the elements that take part in communication. It is responsible for implementing security techniques to ensure the integrity of the message at the source side.

- Receive the information and the authentication key coming from the source.

- Calculate the MAC using the Hash function. The Hash function used the information and the authentication key to calculate the MAC.

- Send the information, plus the MAC over the standard communication mechanism.

Element Layer 2:

This element also acts as an interface to the elements that take part in the communication. In this case, this element is in charge of check the integrity of the information received at the receiver side.

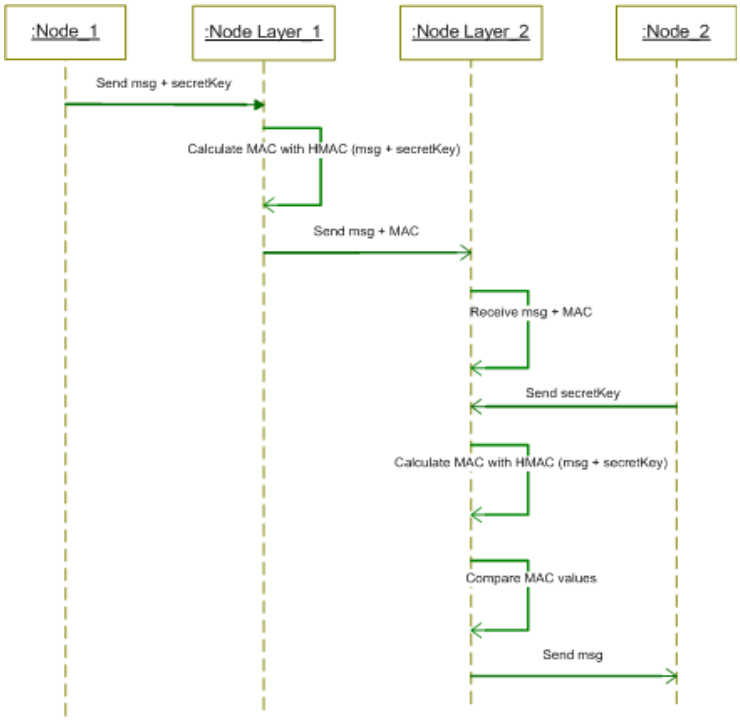
- Receive the information and MAC coming from the security interface (Element Layer 1).

- Receive the authentication key coming from the receiver.

- Calculate its own MAC with the received information and the authentication key.

- Compare the MAC received, with the previous MAC calculated, to check if the information received is valid.

Description of the collaboration between elements:



Technical Solution:

The implementation of this pattern is carry out in the elements participant in the communication. The pattern is comprised by two kind of interfaces, one in charge of generating the MAC through the Hash function, and send the package (info + MAC), and the other one in charge of receiving the package, unpack the data, generate its own MAC, and check the info. The package is composed of the information to share, plus the MAC code. The MAC code is calculated with a Hash function. HMACs have two functionally distinct parameters:

- Information input
- Authentication key -Secret key-

It is important to remark that the strength of this strategy depends on the properties on the underlying cryptographic algorithm (Hash function) used to calculate the MAC. Some example of hash functions are:

- MD5
- SHA1
- SHA256

Consequences:

Results in applying the pattern:

- The shared information in an application at different levels can be transmitted securely via a standard communication mechanism, preventing different types of attacks such as brute force attacks.

Tradeoff in applying the pattern:

- The implementation of this pattern increases the computational cost of the application. This increase is caused by the calculation of the MAC with the hash function.
- The amount of information through the communication mechanism is increased because the transmitted data size is larger. This increase is caused by the extra information needed by the message authentication code (MAC).

Design decision associated to the pattern:

- Implementation of specific interfaces in the elements involved in the communication.
- Extent the information with a Message Authentication Code, to guarantee the integrity of the information to share.
- Cryptographic key –Secret key- used in conjunction with the information to produce the MAC through a hash function, to guarantee the authenticity of the parties involved in the communication.

Benefit of applying the pattern:

- Providing a way to check the integrity of information transmitted over or stored in an unreliable medium.

Impact on system extensibility:

- This pattern has a direct impact on the extensibility of the system, because it can transform an application where the security of the information is not guarantee, to an application where the authenticity of the parties involved in the communication and the integrity of the message are guarantee without the use of any additional mechanism.

Impact on system flexibility:

- The system can be more flexible because, normal applications can be easily adapted to a secure application, only with the implementation of this pattern at different communication levels.
- The pattern may offer different solutions to the system for the same security problem in communications. These solutions can take into account for example the computational cost to calculate the MAC, and it can use less strength cryptographic algorithms, etc.

Impact on system portability:

- The use of this pattern has a minor impact in the portability of the system, because it can be seen as an independent interface related to only the parties involved in the communication. Moreover, these interfaces could be implemented even in software, firmware or hardware.

Related Patterns:

Links between patterns used at the same level: None, *Links between patterns used at different levels:* None

Reciprocal Monitoring Pattern

Name / Generalities:

Identification: Reciprocal Monitoring

Version: 0.0

Description / Intent: The purpose of this pattern is to detect as early as possible failures in the application by dynamic comparison.

*Classification: S*tructural, Behavioral.

Level: Safety level, Architectural level, Design level, Implementation level.

Motivation: To detect as early as possible failures in the application, in order to prevent the propagation of these failures to the rest of the application, and the likely generation of new failures that could lead to system fault.

Applicability: This pattern can only be used in application where architectural strategies as redundancy are selected. At least two independent channels are required for the implementation of this pattern.

Problem:

Context: A critical application must guarantee the detection of failures as early as possible.

Description: This pattern provides the capability to share information between the channels as for examples final results, internal results, test data in order to detect as early as possible a failure in one of the channels**.**

Applied on: Replicated systems that require an early detection of failures that may lead to an incorrect result.

Applied when: In applications where the desired diagnostic coverage is high, it is recommended by the safety standard to use diagnostic techniques such as monitoring –High Diagnostic Coverage-Redundant Architectures

Preconditions: System architecture has been defined.

The data to be monitored has been selected.

Notation:

Modeling Language: UML, SySML

Programming Language: C++

Code: None

Pseudo code: None

Text: Description for the safety concept

Proof of concept provide by the formal validation – Detailed design phase

Solution:

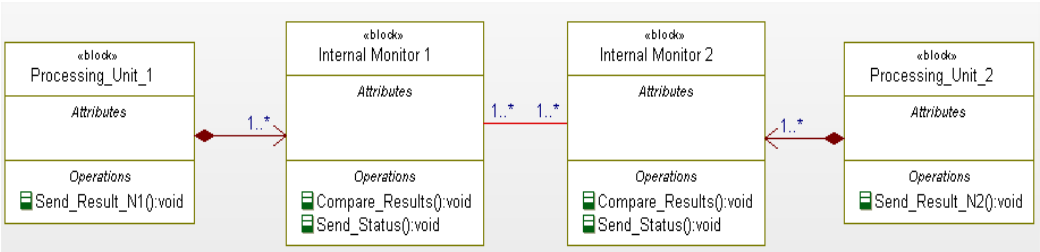
Description of the element of the design:

This pattern is characterized to detect any difference between the compared units / channels. This pattern performs a comparison of final results, internal results or test data in order to detect any failure in a unit/channel. Each unit/channel has a monitor implemented internally.

In this pattern, the internal monitor of each unit/channel, received the data coming from the other units/channels and its own unit/channel, and compare these data in order to detect any difference. In case any difference is detected, the internal monitor should inform the other channels that a channel has failed.

It is important to remark that the independence between the units/channels must be guarantee.

Description of the relationship between design elements:



Description of the responsibilities of the elements:

Processing Unit N: This element is the application by itself. In this case, the application must be replicate in independent units/channels in order to carry out the reciprocal monitoring. This element is responsible for sending the information to:

- Its internal monitor.
- To the internal monitors of the other channels.

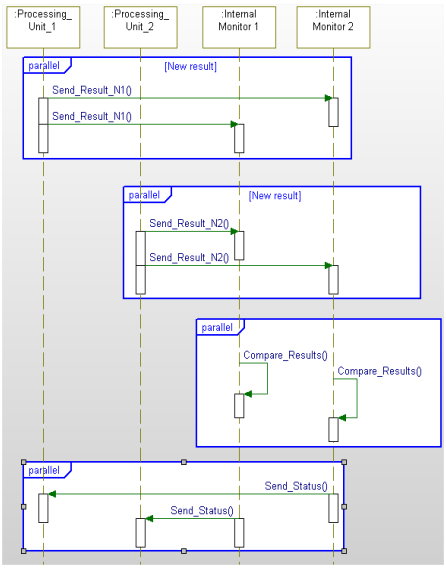
Internal Monitor N: This element must be implemented in each one of the unit/channel to be monitored. The independence between the monitor and the application must be guarantee.

- This element receives the information coming from its unit/channel and from the other units/channels.

- Compare the information received

- Send the information about the status of the other channels to the corresponding channel.

Description of the collaboration between elements:



Technical Solution:

The implementation of this pattern is carry out inside each unit/channel to be monitored. The word reciprocal coming from the idea, that each unit/channel is responsible for monitoring the results/data of the other units/channels in order to detect any difference, without use an independent monitoring element.

The reciprocal monitoring is always performed between two independent units/channels, but it might be use also in triple redundant architectures. In this case for example, the first unit/channel will have a reciprocal monitor that compares the results of the first unit channel with the results of the other two channels independently, and will generate and separate output for each comparison.

It is important to remark that the independence of each unit/channel must be guarantee. Moreover, the independence between the unit/channel and its internal monitor must be also guarantee.

Consequences:

Results in applying the pattern:

- A failure can be detected before it can be propagate to the output.

Tradeoff in applying the pattern:

- The implementation of this pattern increases the computational cost of the application by the introduction of the internal monitors, and the information shared between the units/channels.

Design decision associated to the pattern:

- Implementation of specific internal monitors to carry out the comparison between the units/channels.
- Redundant Architectures

Benefit of applying the pattern:

- Increase the diagnostic coverage of the application, because high number failures might be detected at different stages of the application.
- The propagation of the failures through the application will be reduced.

Impact on system extensibility:

- This pattern might help in the extensibility of systems where it is required to increase the diagnostic coverage of the application. This pattern can help to detect failures at early stages of the application, and it does not require generation of extra modules. The functionality of the pattern can be implemented inside the units/channels.

Impact on system flexibility:

- This pattern may be adapted to different data types, and can be used at different stages of the application to compare different types of results. It can be used in different type of redundant architectures and compare the results of two, three or n channels.

Impact on system portability:

- None

Related Patterns:

Links between patterns used at the same level: None

Links between patterns used at different levels: None

Secure Remote Readout (SSR) Pattern

Name / Generalities:

Identification: Secure Remote Readout (SRR)

Version : 1.0

Description / Intent /Properties to ensure: Pattern allows remote readout of measurement data generated by the Smart Meter ensuring integrity and authenticity of origin of the measurements.

Classification: Behavioural, pattern provides communication between components

Level: Architecture level, Detailed Design level, Implementation level

Motivation: The measurement tuples of a Smart Meter are read out remotely. They are transmitted over possibly insecure communication networks. Integrity and authenticity of origin of the measurement tuples have to be ensured. These properties are provided by the pattern. The lawful requirements for metering devices comprise the demand of measuring data being non manipulable from recording until billing (Transparency from measurement recording until billing for all market participants). The Secure Remote Readout is used to ensure integrity and authenticity of origin of the transferred data and to prevent manipulation.

Applicability: Pattern may be applied to metering devices equipped with a remote readout functionality.

Problem:

Ordinary measurement tuples are easily manipulable. Authenticity of their origin and their integrity have to be assured when transferring measurement data from the Smart Meter to the Remote Meter Readout Center. Also the compulsory relevant log file needs to be securely read out (guaranteeing authenticity and integrity of every single entry) by the Metrology Institute/ Notified body.

Notation:

Modeling Langauge (UML, SySML, ...): UML
Programming Language: C/C++
Code: N/A
Pseudo code:

```
secure_remote_readout( data, private_key )
{
// Hash data and store result in digest
digest = hash( data )

// Get random number
random_number = get_random_number()

// Sign digest using the private_key and the random number
signature = generate_signature( digest, private_key, random_number)

//Return the signature
return signature
}
```

Text:
The input parameters for the interface function “generate_secure_remote_readout” are the data (e.g. measurement tuples) to be secured by a digital signature and the private key.
Using internal interfaces, data is hashed, result is stored in digest and a random number is generated.
The digest and the random number are signed and the resulting signature is returned

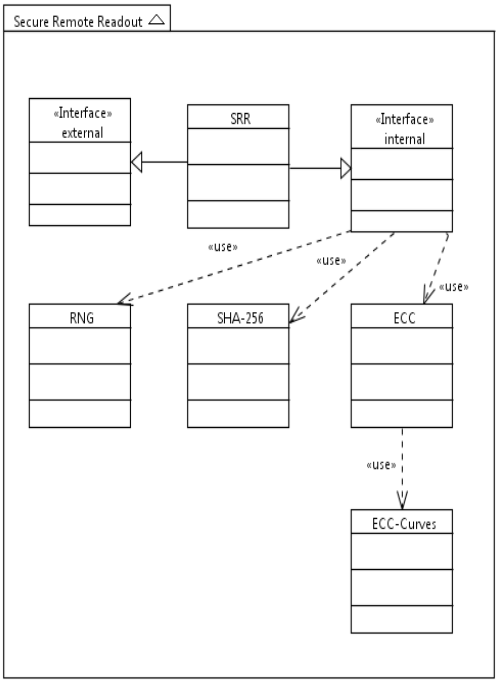
Solution:

The data to be remotely read out (Measurement data and compulsory relevant log for meters) is protected against manipulation with the help of digital signatures. In the moment the measurement data becomes available, a timestamp is appended and a digital signature is created over the data and the timestamp. These three parts (data, timestamp, signature) are concatenated to a tuple. Any of the later processing steps (e.g. saving to persistent memory, transferring to the remote readout center) works on the tuple as a whole.

Description of the element of the design:

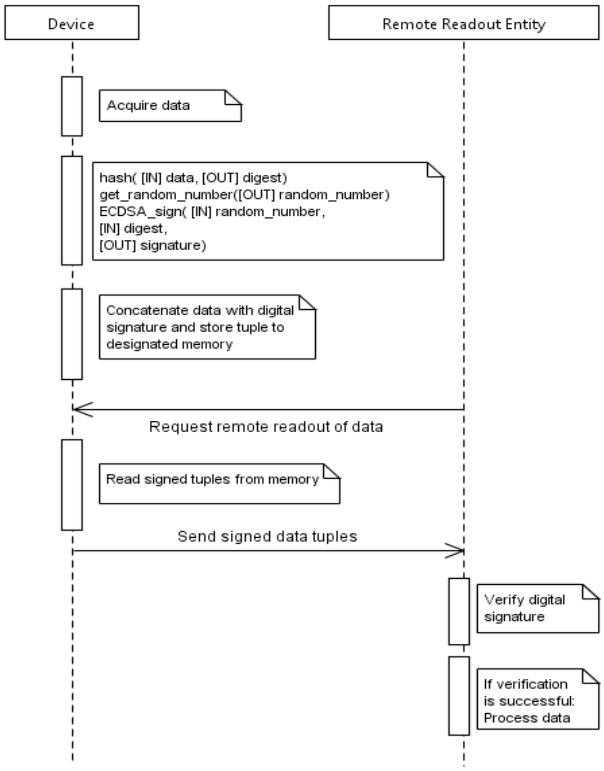
- SRR: Main class/module, provides the external interface for access
- RNG: Random number generator providing random numbers needed for the digital signature algorithm
- SHA-256: Hash algorithm (SHA-256) needed for creating the message digest
- ECC: Provides digital signature functionalities, based on ECC.
- ECC-Curves: Provides necessary curve parameters for the ECC algorithm

Description of the relationship between design elements:



Description of the responsibilities of the elements: SRR provides the main functionalities. Internal interfaces of the pattern are: RNG, SHA-256, ECC, Using ECC-Curves

Description of the collaboration between elements:



Technical Solutions:

Pattern provides a software solution: Hash- or signature algorithm may be also provided as a hardware service.

Properties to ensure:
Security properties: Authenticity of origin of the data transmitted, Integrity of the data transmitted

Resource constraint: Possible resource constraints are:

- Computational (CPU)
- Memory (RAM, (Flash)-ROM)
- Energy

Consequences:

Results in applying the pattern: Authenticity of origin and integrity of the remote readout data tuples is guaranteed

Tradeoff in applying the pattern: Additional memory is required, especially a secure memory for the private key

Design decision associated to the pattern: None

Benefit of applying the pattern: Authenticity of origin and integrity of the remote readout data tuples is guaranteed

Impact on system extensibility: None

Impact on system flexibility: When applying the pattern to a resource constrained system, it requires additional CPU- time, memory and energy. Depending on the system parameters and the cryptographic algorithms, only a limited number of secure remote readout data tuples may be generated in a certain period of time.

Impact on system portability: None

Related Patterns

Links between patterns used at the same level: The Secure Remote Readout (SRR) pattern provides internal interfaces for the following patterns:

1. ECC En/Decryption
2. Elliptic Curve Digital Signature Algorithm ECDSA
3. Random Number Generator

Links between patterns used at different levels: N/A

Secure Software Download (SSD) Pattern

Name / Generalities:

Identification: Secure Software Download (SSD)

Version: 1.0

Description / Intent /Properties to ensure: Secure software download for embedded systems like meters, electronic control units et cetera, wherever a software update is necessary without exchanging the physical system.

Classification: Behavioural, pattern provides communication between components

Level: Architecture level, Detailed Design level, Implementation level

Motivation: A target device stays in use at the customer site for a longer period of time, e.g. 20 years without being directly accessed by service personnel. It is connected to a possibly unsafe communication network. For remotely updating the software running on the system, a secure software download mechanism is needed.

Applicability: Pattern may be applied to resource constraint embedded systems devices in the field requiring the functionality to remotely update the device's firmware in a secure way. Possible devices are smart electricity meters or electronic control units in cars.

Problem:

A device in the field needs a software update. The update is provided over possibly unsafe communication networks. It has to be ensured the meter only downloads and installs certified software images (in regard to compulsory relevance). Especially authenticity of origin and integrity of the firmware image must be guaranteed.

Notation:

Modeling Langauge (UML, SySML, ...): UML

Programming Language: C/C++

Code: N/A

Pseudo code:

```
//Check if download command is valid
if (download_command == valid)
{
execute_secure_software_download();
}
execute_secure_software_download()
{
// Receive firmware image and store it in designated memory
Receive_image(designated_memory);
// Check authentiticty and other relevant information before flashing
```

```
if (verify_image(public_key) == valid)
{
// Mark new image as active
set_image_active(image);

// Reboot system with new image
System_reboot();
}
```

Text: See comments above

Solution:

The technique of digital signatures is used to ensure authenticity of origin and integrity of the firmware image. The update takes place in three steps. At first the client that provides the new software has to authenticate at the target system, so the target system can enable the download functionality. Second, the software is transferred into a designated secure memory region at the target system and the integrity and authenticity of the software image is checked on the target system. Third, the target system is restarted and the bootloader loads the new software image from the designated memory and starts it.

Description of the element of the design:

• SSD : Main class/module, provides main functionality

• <<interface>> : Secure Software Download. It provides interface for the Secure Software Download functionality to the system

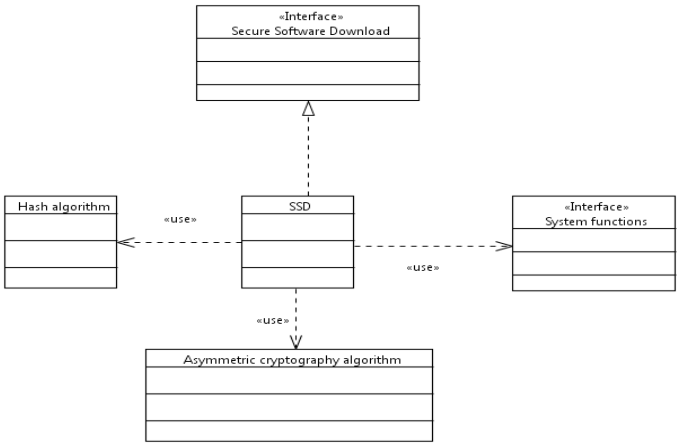
• Hash algorithm: is needed for creating a message digest

• Asymmetric cryptography algorithm

Provides an asymmetric cryptography algorithm for verifying digital signatures

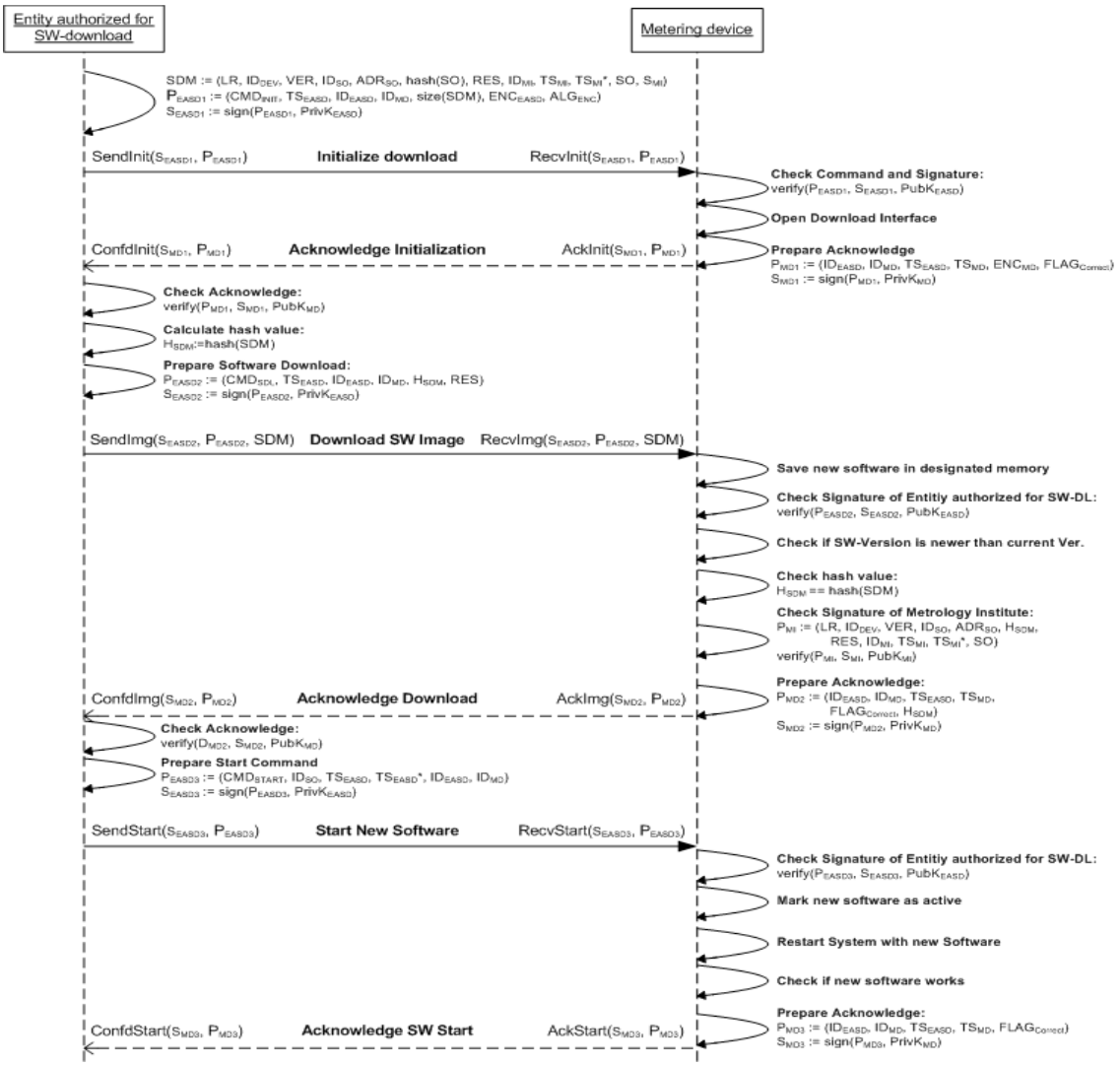
• <<interface>>: Internal interface, needed to provide access to system functions, like reboot.

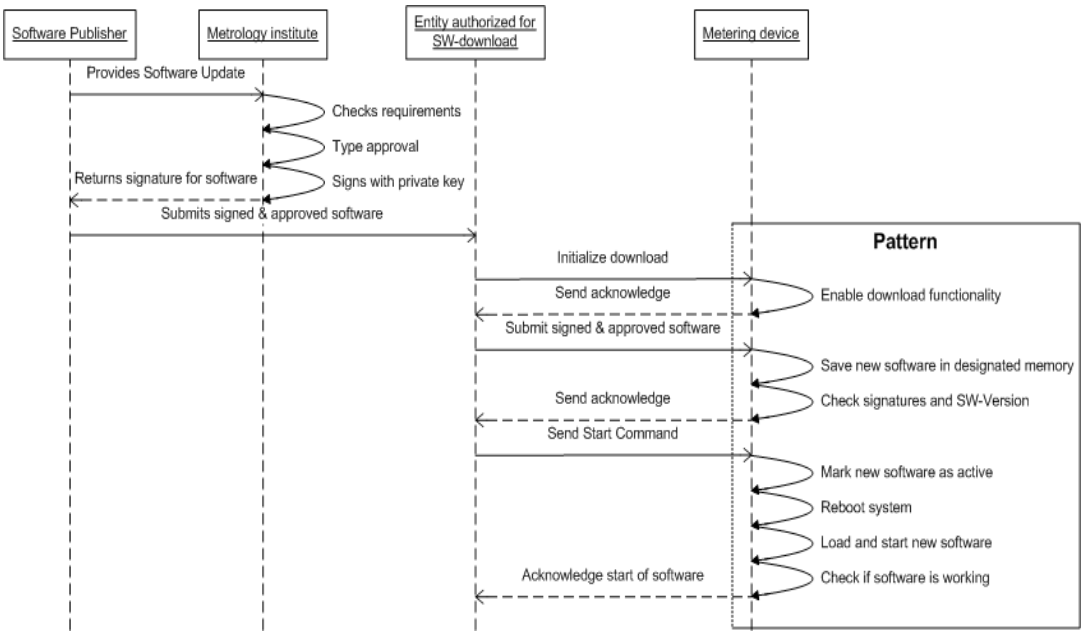
Description of the relationship between design elements:



Description of the collaboration between elements:

The dynamics of the software update process is shown in the two sequence diagrams below. The first one gives an overview about the whole process starting with the provision of the software update and ending with the successful start of the new system software. The second diagram shows the interaction between the Entity Authorized for Software Download and the Metering Device in more detail and highlights the data transmitted during the process.





Technical Solutions:

Pattern provides a software solution. Hash- or signature algorithm may be also provided as a hardware service.

Security properties: Access Control for enabling the Secure Software Download of the system, Authenticity of origin of the software image transmitted, Integrity of the software image transmitted

Resource constraint: Possible resource constraints are:

- Computational (CPU)
- Memory (RAM, (Flash)-ROM)
- Energy

Consequences:

Results in applying the pattern: System is remote updatable.

Tradeoff in applying the pattern: Additional memory is required for the software image and the cryptographic keys

Design decision associated to the pattern: None

Benefit of applying the pattern: Authenticity of origin and integrity of the software image is guaranteed.

Impact on system extensibility: None

Impact on system flexibility: When applying the pattern to a resource constrained system, it requires additional memory.

Impact on system portability: None

Related Patterns:

Links between patterns used at the same level : The Secure Remote Readout (SRR) pattern provides internal interfaces for the following patterns:

. ECC En/Decryption

. Elliptic Curve Digital Signature Algorithm ECDSA

. Random Number Generator

Links between patterns used at different levels: None

NIST SP 800-90 Hash-DRBG Pattern

Name / Generalities:

Identification:
NIST SP 800-90 Hash_DRBG – DRBG (Deterministic Random Bit Generator) mechanism based on a hash function

Also Known As: Pseudo Random Number Generator (PRNG)

Version: 1.1

Description / Intent:

In short: Generate a sequence of pseudorandom bits using Hash_DRBG based on a hash function including a truly random seed.

In detail:

1. The Hash_DRBG is a specific DRBG specified in NIST SP 800-90. For a detailed algorithm description including the instantiate, reseed and generate functions we refer to Section 10.1.1 of [1]. The instantiate function is used prior to the generation of pseudorandom bits. This initialization is necessary to optain the entropy and nonce inputs needed to support the security strength. After the initalization the generate function can be used to generate pseudo random bits until a reseed is required to insert additional entropy.

2. The Hash_DRBG is FIPS140-2 approved [3].

3. It is based on a hash function that is non-invertible or one-way. The Hash_DRBG algorithm specified in [1] is designed to use any approved hash function. The maximum security strength that can be supported by Hash_DRBG is the security strength of the hash function used. The corresponding security strength for hash functions are provided in [2] , Chapter 5.6.1. In [1] only the four security strengths are recommended: 112, 128, 192, and 256. Table 1 specified in [1] provides the values that shall be used for each approved hash function.

4. The seed used to instantiate the DRBG must contain sufficient entropy to provide an assurance of randomness. The quality of the entropy is very important for the use of this PRNG! Appendix C of [1] provides guidance on entropy and entropy sources. In order to obtain an unpredictable output of the DRNG, the seed needs to be kept secret.

5. Besides the entropy a noce has to be provided. As stated in [1], Section 8.6.7, the nonce shall be either:

    1. An unpredictable value with at least (1/2 security_strength) bits of entropy,

    2. A value that is expected to repeat no more often than a (1/2 security_strength)-bit random string would be expected to repeat.

Table 1: Definitions for Hash-Based DRBG Mechanisms [1]

	SHA-1	*SHA-224*	*SHA-256*	*SHA-384*	*SHA-512*
Supported security strengths	*See [2]*				
highest_supported_security_strength	*See [2]*				
Output Block Length (outlen)	*160*	*224*	*256*	*384*	*512*
Required minimum entropy for instantiate and reseed	*security_strength*				
Minimum entropy input length (min_length)	*security_strength*				
Maximum entropy input length (max_length)	$\leq 2^{35}$ *bits*				
Seed length (seedlen) for Hash_DRBG	*440*	*440*	*440*	*888*	*888*
Maximum personalization string length (max_personalization_string_length)	$\leq 2^{35}$ *bits*				
Maximum additional_input length (max_additional_input_length)	$\leq 2^{35}$ *bits*				
max_number_of_bits_per_request	$\leq 2^{19}$ *bits*				
Number of requests between reseeds (reseed_interval)	$\leq 2^{48}$				

Classification (creational, structural, behavioral ...): Creational
Level: to be used at implementation level of the software development process
Applicability: Sufficient entropy and a nonce must be available to instantiate the DRBG. According to the desired security strength the corresponding SHA hash algorithm must be available.

Problem:

To be applied on and when pseudorandom bits have to be generated. The selection of a DRBG mechanism depends on several factors. More information concerning the factors of the selection are provided in the *General DRBG Pattern* [4].

Conditions to be met to apply the pattern: See Applicability.

Notation:

- Programming Language: Example is available in ANSI-C.
- Pseudo code: See [1], Appendix F.1

Solutions:

- Hash_DRBG sequence diagrams

  The following diagrams show sequences for initialization, reseeding and generating.

Figure 1: Hash-DRBG Initialization

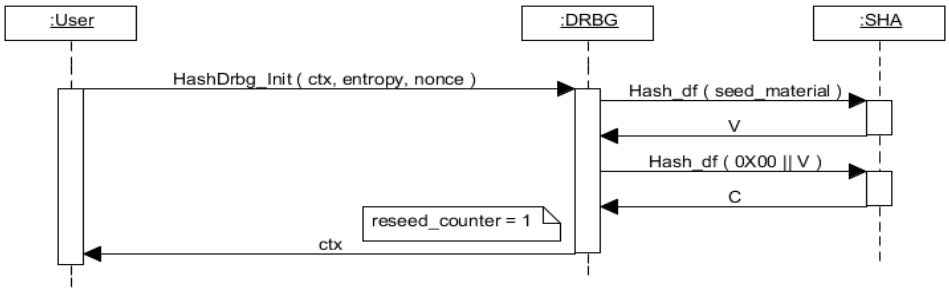


Figure 2: Hash-DRBG Reseed

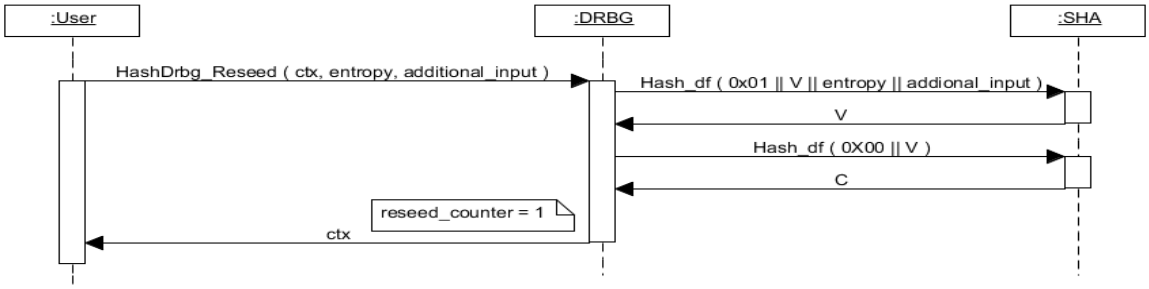
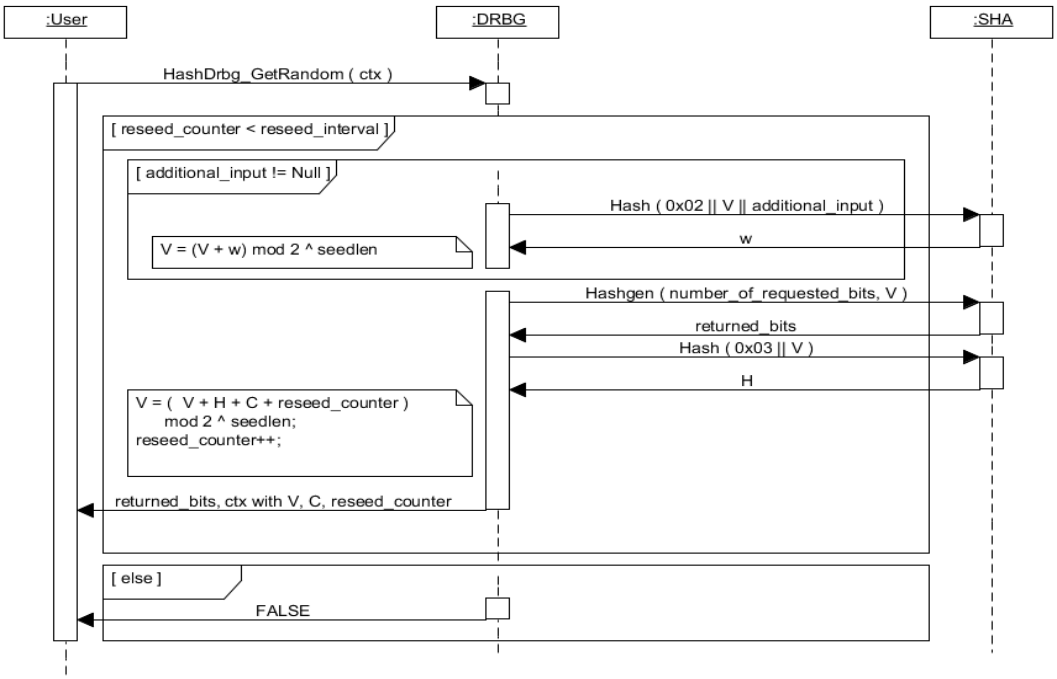


Figure 3: Hash-DRBG Generate



- For the Init and the Reseed function the Hash_DRBG requires an entropy input. The entropy length to be provided has to be at least the security strength, e.g. for a 128 bit security strength, entropy must to be at least 16 bytes.

- In addition, for the Init function a nonce input with at least (1/2 security strength) bit extra entropy or a value that is expected to repeat no more often than a (1/2 security strength)-bit random string would be expected to repeat is required. The remaining bits shall be filled with a personalization string, which makes every instance of this PRNG unique, for example the serial no. of the host device is a good value.

Information to be included in a general DRBG pattern:

- Several diverse DRBG mechanisms provide acceptable security, specified in NIST SP 800-90. (not DUAL_EC_DRBG)

- The DRBG mechanisms can be based on hash functions, block ciphers and number theoretic problems.

- The selection of a DRBG mechanism depends on several factors, including the security strength to be supported and what cryptographic primitives are available. An analysis of the consuming application's requirements for random numbers should be conducted in order to select an appropriate DRBG mechanism. A detailed discussion on DRBG mechanism selection is provided in Appendix G of [1]

Technical Solution (Example Implementation):

- Implemented conform to MISRA-C 2004 and ANSI-C coding style.

- Applicable for any platform providing an ANSI-C conform compiler - from 8 bit to 64 bit.

- Optimized for code-size while satisfying stringent performance-constraints.

- The implementation enables to select the desired security strength in Bits (128, 192, 256).

- It is based on a SHA hash algorithm, i.e. SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512 that must be available.

- As external interface the Hash_DRBG implementation provides the three functions Init, Reseed and Generate.

- A PRNG-instance is represented by a context containing the following data:

```
typedef struct {
    /** seed vector   */
    UINT8 seed[HashDrbg_SEED_LEN];
    /** state vector   */
    UINT8 state[HashDrbg_SEED_LEN];
    /** reseed counter   */
    UINT32 counter;
} HashDrbg_ContextT;
```

– The entropy length HashDrbg_ENTROPY_LEN must be the array size of values for INIT and RESEED calls! For 128 bit security strength, it must to be at least 16 bytes

- The nonce length HashDrbg_NONCE_LEN must be the array size of values for INIT and RESEED calls! For 128 bit security strength, it must to be at least 8 bytes.

- The seed length HashDrbg_SEED_LEN is set according to the standard to 440 bit for SHA-1,SHA-224, and SHA-256 and to 888 bit for SHA-384 and SHA-512.

- Init function: HashDrbg_Init

This function initializes the PRNG. It has to be called once before using the PRNG!

```
extern BOOL

HashDrbg_Init(
    HashDrbg_ContextT* ctx,
  const UINT8 entropy[],
  const UINT8 nonce[]

);
```

Parameters:

[inout]	*ctx*	The PRNG context HashDrbg_ContextT.
[in]	*entropy*	Pointer to the array with entropy input.
[in]	*nonce*	Pointer to the array with nonce.

Returns:
The following presents the instantiate process, see [1] Chapter 10.1.1.2. Let Hash_df be the hash derivation function specified in [1] Section 10.4.1.

Values:

V	*ctx->seed*
C	*ctx->state*
Reseed_counter	*ctx->counter*

Process:
1. seed_material = entropy || nonce (|| personalization_string).
2. seed = Hash_df (seed_material, seedlen).
3. V = seed.
4. C = Hash_df ((0x00 || V), seedlen). Comment: Preceed V with a byte of zeros.
5. reseed_counter = 1.
6. Return context with V, C, and reseed_counter as the initial_working_state.

- Reseed function: HashDrbg_Reseed

Compute a new seed and a new state with newly generated entropy. This function must be called if the reseed counter is higher than the specified *reseed_interval.* In this implementation the default value for the *reseed_interval* has been adopted from the NIST example pseudo code in [1], Appendix F to 100,000 requests.

```
extern BOOL
HashDrbg_Reseed(
    HashDrbg_ContextT* ctx,
  const UINT8 entropy[]

);
```

Parameters:

[inout]	*ctx*	The PRNG context.
[in]	*entropy*	Random Number Array to add entropy to seed and state.

Returns:
TRUE on error.

The following shows the reseeding process, refer to [1] Chapter 10.1.1.3. Let Hash_df be the hash derivation function specified in [1] Section 10.4.1.

Values:

V	*ctx->seed*
C	*ctx->state*
Reseed_counter	*ctx->counter*

Process:
1. seed_material = 0x01 || V || entropy (|| additional_input).
2. seed = Hash_df (seed_material, seedlen).
3. V = seed.
4. C = Hash_df ((0x00 || V), seedlen). Comment: Preceed with a byte of all zeros.
5. reseed_counter = 1.
6. Return context with V, C and reseed_counter for the new_working_state.

- Generate: HashDrbg_GetRandom

Returns a random number derived from the actual state. The internal state is updated.

```
extern BOOL
HashDrbg_GetRandom(
   HashDrbg_ContextT* ctx,
   UINT8 rnd[]

);
```

Parameters:

[inout]	*ctx*	The PRNG context.
[out]	*rnd*	Pointer to the array with random bytes.

Returns:
TRUE on error.

The following shows the reseeding process, refer to [1], Chapter 10.1.1.4. Let Hash be the selected Hash-Function.

Values:

V	*ctx->seed*
C	*ctx->state*
Reseed_counter	*ctx->counter*

Process:
1. If reseed_counter > reseed_interval, then return an indication that a reseed is required.

2. If (additional_input =! Null), then do
2.1 w = Hash (0x02 || V (|| additional_input)).
2.2 $V = (V + w) \bmod 2^{seedlen}$.

3. (returned_bits) = Hashgen (requested_number_of_bits, V).
4. H = Hash (0x03 || V).
5. $V = (V + H + C + reseed\_counter) \bmod 2^{seedlen}$.
6. reseed_counter = reseed_counter + 1.
7. Return SUCCESS and returned_bits and the context with new values of V, C and reseed_counter for the new_working_state.

The following sequence shows the Hashgen process used in HashDrbg_GetRandom, refer to [1] Section 10.1.1.4:

Hashgen (*requested_number_of_bits*, *V*):
Input:
1. *requested_no_of_bits*: The number of bits to be returned.
2. *V*: The current value of *V*.

Output:
1. *returned_bits*: The generated bits to be returned to the generate function.

Hashgen Process:

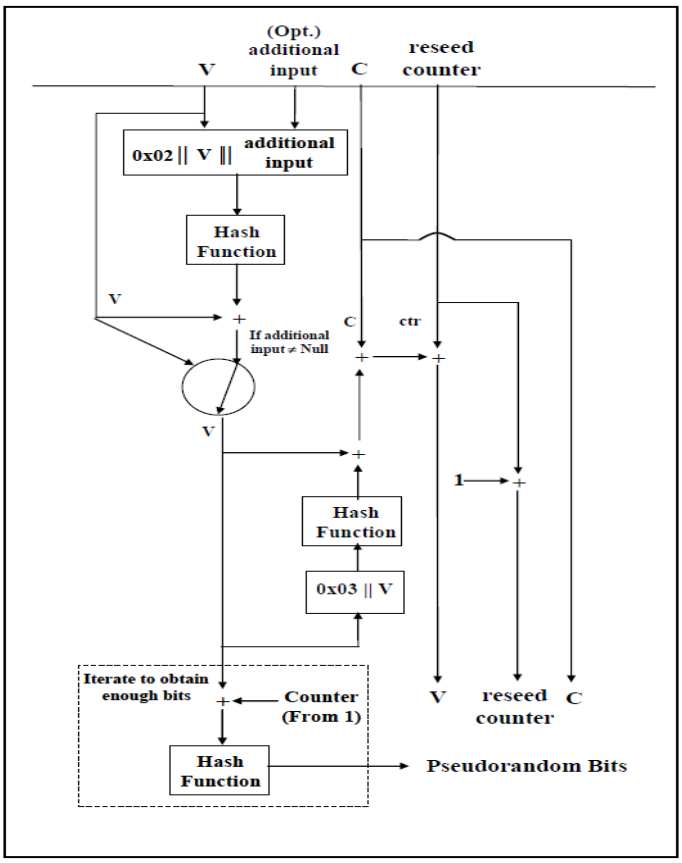
1.

$$m = \frac{\text{requested\_number\_of\_bits}}{\text{outlen}}$$

2. *data* = *V*.
3. *W* = the *Null* string.
4. For *i* = 1 to *m*
4.1 *wi* = Hash (*data*).
4.2 *W* = *W* || *wi*.
4.3 *data* = (*data* + 1) mod $2^{seedlen}$.
5. *returned_bits* = Leftmost (*requested_no_of_bits*) bits of *W*.

6. Return *returned_bits*.

Figure 4: Hash-DRBG Generate Algorithm [1], Section 10.1.1.1



- Internal interfaces:

  Besides a main configuration file that specifies the data types, the Hash_DRBG implementation is based on the required SHA implementation (SHA1, SHA224, SHA256, SHA384, or SHA512).

Consequences:

Design decision associated to the pattern: The selection of a DRBG mechanism depends on several factors. More information concerning the factors of the selection are provided in the *General DRBG Pattern* [4].
Benefit of applying the pattern: For generating pseudorandom bits based on a hash function the Hash_DRBG pattern provides helpful information for the system developer. The example implementation is suitable for a variety of platforms and allows configuring the security strength.

Related Patterns:

Links between patterns used at the same level (architecture, design or implementation): SHA hash algorithm pattern:

- The Hash_DRBG is based on a SHA hash algorithm.
- Entropy source pattern: Sufficient entropy must be provided to instantiate the DRBG.

Links between patterns used at different levels: General DRBG pattern: The Hash_DRBG is a specific DRBG implementation.

[1] NIST SP 800-90
[2] NIST SP 800-57
[3] FIPS 140-2 Annex C
[4] General DRBG pattern

ECC Key Pair Generation/Validation Pattern

Name / Generalities:

Identification: Elliptic Curve Cryptography Key Pair Generation and Validation Pattern

Version: 1.0

Description / Intent:

In short: Asymmetric key pair generation for Elliptic Curve Cryptography (ECC).
In detail: According to [1], Section 4.3 the elliptic curve key pair generation is described as follows: An elliptic curve key pair is associated with a particular set of domain parameters $D = (q, \mathrm{FR}, S, a, b, P, n, h)$. The public key is a randomly selected point Q in the group $\langle P \rangle$ generated by P. The corresponding private key is $d = \log_P Q$.
Algorithm: Key pair generation
Input: Valid elliptic curve domain parameters $D = (q, \mathrm{FR}, S, a, b, P, n, h)$.
Output: Public key Q, private key d.

1. Randomly or pseudorandomly select an integer d in the interval $[1, n - 1]$.
2. Compute $Q = dG$.
3. Return (Q, d).

Conditions: The entity A generating the key pair must have the assurance that the domain parameters are valid (see [1], Section 4.2). The association between domain parameters and a public key must be verifiable by all entities who may subsequently use A's public key. In practice, this association can be achieved by cryptographic means (e.g., a certification authority generates a certificate attesting to this association) or by context (e.g., all entities use the same domain parameters).
Observe that the problem of computing a private key d from the public key Q is precisely the elliptic curve discrete logarithm problem. Hence it is crucial that the domain parameters D be selected so that the *elliptic curve discrete logarithm problem (ECDLP)* (see [1], Section 4.1) is intractable. Furthermore, it is important that the numbers d generated be "random" in the sense that the probability of any particular value being selected must be sufficiently small to preclude an adversary from gaining advantage through optimizing a search strategy based on such probability.

Classification (creational, structural, beahvioral ...): Behavioural, pattern provides key pair generation process

Level: Implementation level

Applicability: This security pattern may be applied for ECC key pair generation in general. The domain parameters *D* must be valid and a source for sufficient random numbers must be available to establish the random private key *d*.

Problem:

To be applied on and when an elliptic key pair has to be generated. The pattern enables to generate ECC key pairs needed for digital signature generation and verification based on elliptic curve cryptography.

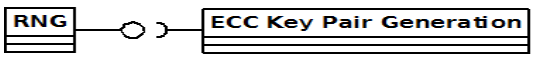
Notation:

Programming Language: Example is available in ANSI-C.
Pseudo code: See Algorithm "Key pair generation"

Solution:

Description of the element of the design

- User: Entity requesting the ECC key pair
- Random Number Source: Entity providing random numbers needed for the key pair generation. For this issue for instance a DRBG can be employed.
- ECC Key Pair Generation Entity
- Entity generating the corresponding public key for a given private key *d*.



Description of the collaboration between elements:

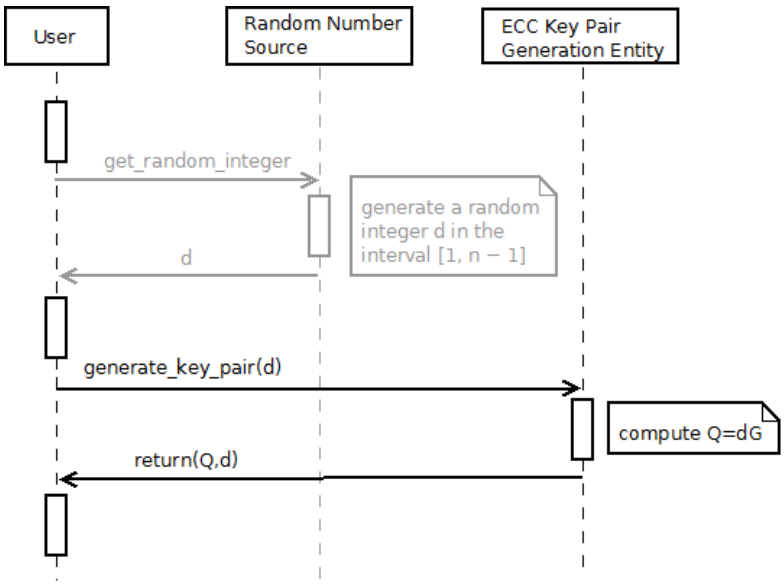


Figure 5: ECC Key Pair Generation Sequence Diagram

Technical Solution (Example Implementation):

- Implemented conform to MISRA-C 2004 and ANSI-C coding style.
- Applicable for any platform providing an ANSI-C conform compiler - from 8 bit to 64 bit.
- Optimized for code-size while satisfying stringent performance-constraints.
- The ECC Key Pair Generation and Validation implementation provides the function Key Generation.

The ECC public key structure EscECC_PublicKeyT contains the following data fields:

UINT8	*x*	Pointer to x (ECC_KEY_BYTES)
UINT8	*y*	Pointer to y (ECC_KEY_BYTES)
UINT32	*counter*	Reseed counter

- Key Generation function: EscEcc_PublicKeyGeneration

Providing the private key field element *d* as input this function generates the corresponding public key based on the domain parameters. The private key field element d must have a length of the ECC key bytes.

```
extern BOOL
        EscEcc_PublicKeyGeneration(
         EscEcc_PublicKeyT* pQ,
        const UINT8 d[]
);
```

Parameters:

[out] *pQ* Public key point (affine).
[in] *d* Private key field element d of length ECC_KEY_BYTES.

Returns:

Non-zero on error.

- Embedded Public key validation: EscEcc_EmbeddedPublicKeyValidation

This function performs an embedded public key validation, which checks that the input parameter Q is not the infinity-point, is a properly represented element in GF(p) and that is lies on the elliptic curve

```
extern BOOL
        EscEcc_EmbeddedPublicKeyValidation (
         const EscEcc_PublicKeyT* pQ
);
```

Parameters:

[in] pQ Public key point (affine)

Returns:

Non-zero on parameter or validation error

- Public key validation: EscEcc_PublicKeyValidation

This function performs a public key validation, which checks the same points as the embedded validation function. Additionally it validates that Q*n is the infinity point.

```
Extern BOOL
        EscEcc_PublicKeyValidation(
        const EscEcc_PublicKeyT* pQ
 );
```

Parameters:

[in] pQ Public key point (affine)

Returns:

Non-zero on parameter or validation error

- In order to call the EscEcc_PublicKeyGeneration function first the domain parameters have to be specified and the private key *d* has to be generated randomly using a sufficient random source.
- Description of the relationship between design elements

Consequences:

Results in applying the pattern: The pattern is used to generate ECC key pairs needed for the ECC digital signature generation and verification.

Benefit of applying the pattern: The ECC key pair generation pattern provides helpful information for the system developer. The example implementation is suitable for a variety of platforms and allows configuring the security strength.

Related Patterns:

Links between patterns used at the same level (architecture, design or implementation): Random number generator pattern: The ECC Key Pair Generation Pattern is based on a random number pattern.
Links between patterns used at different levels: ECC pattern: The ECC Key Pair Generation Pattern is used for generating the key pairs needed for the elliptic curve cryptography described in the ECC pattern.

[1] D. Hankerson, A. Menezes, S. Vanstone: Guide to Elliptic Curve Cryptography, Springer 2004

Elliptic Curve Cryptography Pattern

Name / Generalities:

Identification: Elliptic Curve Cryptography based on the algebraic structure of elliptic curves over finite fields
Also Known As: ECC Pattern
Version: 1.0

Description / Intent:
In short:
Cryptographic computations on Elliptic Curves for encryption purposes or in order to realise several cryptographic schemes like Elliptic Curve Diffie-Hellman (EC-DH) or Elliptic Curve Digital Signature Algorithm (EC-DSA).
In detail:

6. ECC is an approach to public-key cryptography. For a detailed algorithm description including the elliptic curve groups, key generation and encryption functions we refer to Section 3 of [1].
7. The operation for ECC signature generation is the same as the one used for ECC decryption. The operation for ECC signature verification is the same as the one used for ECC encryption.
8. In order to use ECC all parties must agree on all the elements defining the elliptic curve (domain parameters). The parameters should be chosen so that the EC-Discrete-Logarithm-Problem (EC-DLP) is resistant to all known attacks. For that purpose several domain parameters of elliptic curves for several common field sizes were published by NIST in [2] and [3] which were chosen for optimal security and implementation efficiency. The domain parameters are:
   1. Prime p (in prime case) or m and f (in binary case)
   2. Constants $a,b \in F_p$, which define the elliptic curve in $y^2 = x^3 + ax + b$ and satisfy $4a^3 + 27b^2 \neq 0 \bmod p$
   3. Base point (generator) $P = (x_p, y_p)$
   4. n (order of P) with $n*P = \infty$
   5. *cofactor* $h = \#E(F_q)/n$
9. Besides the domain parameters each party needs a key pair with $k_{priv} \in [1, n-1]$ and $k_{pub} = k_{priv} * P$
10. Elliptic Curve based schemes are based on the assumption that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is infeasible. The size of the elliptic curve determines the difficulty of the problem.
    The problem of determining k_{priv}, given the domain parameters and k_{pub}, is the EC-discrete logarithm problem. To avoid successfull attacks on EC-DLP it is necessary that $\#E(F_q)$ be divisible by a sufficient large prime n. At a minimum $n > 2^{160}$ is recommended.
    Because the EC-DLP is much more difficult to solve than the DLP in finite fields, ECC needs much smaller keys than other public-key methods at an identical security level. See table 1 for a comparison.

Table 1: Comparison of security level (symmetric, asymmetric, ECC) [4]

Security level/symmetric key size	RSA and DH key Size	Elliptic Curve Key Size	Ratio of DH-Cost: EC Cost
80	1024	160	3:1
112	2048	224	6:1
128	3072	256	10:1
192	7680	384	32:1
256	15360	521	64:1

Classification (creational, structural, behavioral ...): Creational
Level: Implementation level
Applicability: A sufficient key generation function and appropriate domain parameters must be available. If the signature scheme should be used a HASH function and a RNG is required.

Problem:

To be applied on and when public-key cryptography based on elliptic curves is needed for different cryptographic schemes:

1. Provide confidentiality for small data items (public-key encryption)
2. Digital signature scheme (EC-DSA)
3. Key establishment in order to use the shared secret key in a symmetric-key protocol

Notation

Programming Language: Example is available in ANSI-C.
Pseudo code:
According to [1], Section 4.4.1 the elliptic curve digital signature algorithm (ECDSA), which apperas in the ANSI X9.62, FIPS 186-2. IEEE 1363-2000 and ISO/IEC 15946-2 standards, is described as follows:

Algorithm: ECDSA signature generation
Input: Domain parameters D = (q, FR, S, a, b, P, b, h), private key d, message m
Output: Signature (r,s)

1. Select $k \in_R [1, n-1]$.
2. Compute $kP = (x_1, y_1)$ and convert x_1 to an integer x_1
3. Compute $r = x_1 \bmod n$. If $r = 0$ then go to step 1.
4. Compute $e = H(m)$.
5. Computer $s = k^{-1}(e + dr) \bmod n$. If $s = 0$ then go to step 1.
6. Return (r,s)

Algorithm: ECDSA signature verification
Input: Domain parameters D = (q, FR, S, a, b, P, b, h), public key Q, message m, signature (r,s)
Output: Acceptance or rejection of the signature.

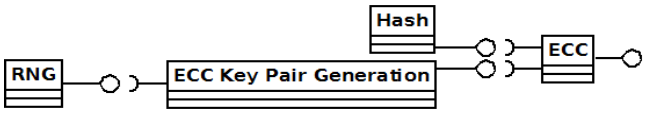
1. Verify that r and s are integers in the interval [1, n-1]. If any verification fails then return ("Reject the signature").
2. Compute $e = H(m)$.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5. Compute $X = u_1P + u_2Q$.
6. If $X = \infty$ then return ("Reject the signature").
7. Convert the x-coordinate x_1 of X to an integer x_1. Compute $v = x_1 \bmod n$.
8. If $v = r$ then return ("Accept the signature"). Else return ("Reject the signature").

Solution:

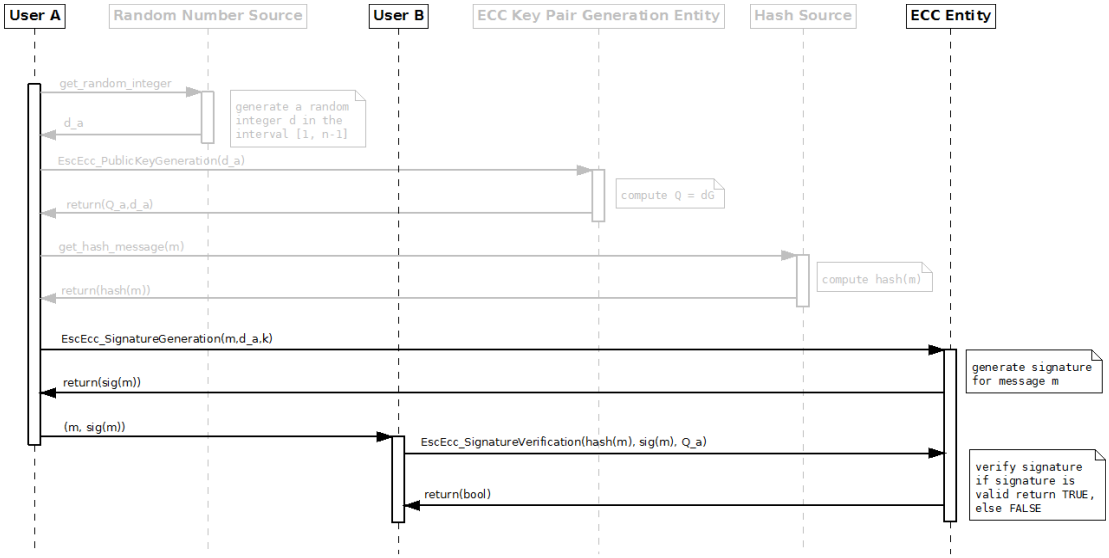
Description of the element of the design:

- User A
  Entity requesting an ECC key pair and sign a message
- User B
  Entity which receives the signed message and verifies it
- Random Number Source
  Entity providing random numbers needed for the key pair generation. For this issue for instance a deterministic random bit generator (DRBG) can be employed.
- ECC Key Pair Generation Entity
  Entity generating the corresponding public key for a given private key d.
- Hash Source
  Entity which generates the corresponding hash value for a given message m.
- ECC Entity
  Entity which signs messages and verifies signatures.

Description of the relationship between design elements:



Description of the collaboration between elements:



Technical Solution (Example Implementation):

- Provides EC-DH and EC-DSA
- Implemented conform to MISRA-C 2004 and ANSI-C coding style.
- Applicable for any platform providing an ANSI-C conform compiler - from 8 bit to 64 bit.
- Optimized for code-size, but offers an option to increase speed while also increasing code-size and stack used.
- The implementation enables to select the desired key strength in Bits (160, 192, 224, 256).
- The ECC implementation provides the following external interface:
- Signature generation: EscEcc_SignatureGeneration

This function generates the corresponding signature for a hashed message.

```
extern BOOLEscEcc_SignatureGeneration(
        EscEcc_SignatureT* ecc_sig,
          const UINT8 msg_hash[[],
          const UINT8 d[],
          const UINT8 k[]
  );
```

Parameters:

[out]	ecc_sig	ECC signature
[in]	*msg_hash*	Hash of signing message of length ECC_KEY_BYTES.
[in]	*d*	Private key field element d of length ECC_KEY_BYTES.
[in]	*k*	Field element unpredictable integer k of length ECC_KEY_BYTES.

Returns:

Non-zero on error.

- Signature verification: EscEcc_SignatureVerification

This function verifies the corresponding signature for a hashed message.

```
extern BOOL
        EscEcc_SignatureVerification(
        const UINT8 msg_hash[[],
        const EscEcc_SignatureT* ecc_sig,
        const EscEcc_PublicKeyT* pQ
  );
```

Parameters:

[in]	*msg_hash*	Hashed signing message of length ECC_KEY_BYTES.
[in]	*ecc_sig*	ECC signature
[in]	*pQ*	Public key point Q

Returns:
Non-zero on error.

- Shared Secret computation (DH): EscEcc_ComputeSharedSecret

This function computes the shared secret z based on ECDH

```
extern BOOL
        EscEcc_ComputeSharedSecret(
        UINT8 z[],
        const UINT8 dA[],
        const EscEcc_PublicKeyT* pQB
);
```

Parameters:

[out]	*z*	Shared secret field element z of length ECC_KEY_BYTES.
[in]	dA	Private key field element d of A of length ECC_KEY_BYTES.
[in]	pQB	Public key point of B (affine)

Returns:
Non-zero on error.

- Internal interfaces: The ECC pattern has no internal interfaces
- Besides several specified data types the ECC implementation provides functionality for all necessary base operations and comparisons on elliptic curves:
  - Increment
  - Addition
  - Subtraction
  - Multiplication
  - Modular reduction
- In addition it is possible to convert the curve data to different data types.

Consequences:

Results in applying the pattern: The pattern is used to provide ECC functionality for different cryptographic schemes like EC-DSA or EC-DH.
Benefit of applying the pattern: The ECC provides many advantages for system developer. By using die EC-DH it is possible to share a common secret key. ECC needs smaller keys as other public-key schemes at an identical security level because the EC-DLP is much more difficult than the DLP in finite fields and is there for a good choice if memory or computation resources are limited, e.g. on smartcards.

Related Patterns:

Links between patterns used at different levels:

- ECC-Key-Pair-Generation pattern: The ECC Key Pair Generation Pattern is used for generating the key pairs needed for the elliptic curve cryptography described in the ECC pattern.
- Hash Pattern: For the signature of a message, the hash of the message is signed. Hence, first a Hash pattern has to be used to generate the hash of the message.

[1] D. Hankerson, A. Menezes, and S.A. Vanstone, *Guide to elliptic curve cryptography*, Springer-Verlag, 2004
[2] NIST, *Recommended Elliptic Curves for Federal Government Use*, Juli 1999, see http://csrc.nist.gov/encryption
[3] FIPS 186-3, Federal Information Processing Standards Publication, *Digital Signature Standard (DSS), 2009,* see http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf
[4] http://www.nsa.gov/business/programs/elliptic_curve.shtml

2.5 State of the Art on S&D Modeling Languages

Many studies have already been done on modeling security in UML. [10] presents an extension of UML, UMLsec, that enables to express security relevant information within the diagrams in a system specification. UMLsec is defined in form of a UML profile using the UML standard extension mechanisms. [11] presents a modeling language for the model-driven development of secure, distributed systems based on UML. Their approach is based on role-based access control with additional support for specifying authorization constraints. SecureUML is a modeling language that defines a vocabulary for annotating UML-based models with information relevant to access control.

In the context of MDE, [12] proposed a UML profile compliant with MARTE [13] to deal with dependability analysis and modeling. Such a profile allows to capture quantitative dependability analysis of software systems modeled with UML. Particularly they focus on the following facets of dependability: reliability, availability and safety.

In [14], we proposed a methodology which associates a model driven approach with component based development to design distributed applications that has fault-tolerance requirements. UML based modeling is used to capture application structure and related non-functional requirements thanks to the complementary profile named FT profile which is composed of 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 Deployment and Configuration standard (D&C)) to build bootcode (static deployment) which instantiates, configures and connects components and to load configured components. Within this process, component replication and FT properties are declaratively specified at model level and are transparent for the component implementation.

2.6 State of the Art on Pattern Modeling and S&D Concern

The concept of pattern was first introduced by Alexander [6]. A pattern deals with 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, which can be reused even when algorithms, component implementations, or frameworks cannot. With regard to S&D aspects, Yoder and Barcalow [15] were the first to work on security pattern documentation. Many contributions on S&D patterns can be found in literature [15, 16, 17, 18, 19].

The 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.

2.6.1 Pattern Languages

To give a flavor of the improvement achievable by using specific languages, we look at the pattern formalization problem. *UMLAUT* was proposed by Guennec et al. [20] as an approach that aims to formally model design patterns by proposing extensions to the UML meta model 1.3. They used an OCL language to describe constraints (structural and behavioral). These constraints are defined on meta-models of specified UML elements in the form of meta collaboration diagrams. Mechanisms of association of these meta level diagrams to their instance level (instances of design patterns) are then defined. This allows to model design patterns accurately in UML language. This work is illustrated through two examples of design patterns: visitor and observer.

In the same way, Kim et al. [21] presented *RBML (Role-Based Meta modeling Language)*. RBML is able to capture various design perspectives of patterns such as static structure, interactions, and state-based behavior. This language is based on the meta-modeling of design patterns and offer three specifications: Structural, Behavioral and Interactive. Each one is characterized by a kind of RBML meta-model: (1) SPS (Static Pattern Specifications) is a specification of structural design pattern which allows to express the static view, (2) IPS (Interaction Pattern Specification) represents the design pattern in terms of possible interactions between different roles, (3) SIMP (StateMachine Pattern Specifications) can add a behavioral view point to describe the various states in which it may lie in its execution.

Another issue raised in [22] and [23] is visualization. Eden et al. [22] presented a formal and visual language for specifying design patterns called *LePUS*. 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)* [23] allows the incorporation of patterns in UML class models.

2.6.2 S&D Patterns

Several approaches exist in the S&D design pattern literature [15, 16, 17, 18, 19]. 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, [15] presents a collection of patterns to be used when dealing with application security. The proposed catalogue includes secure access layer, single access point, check point, etc..

In developing fault-tolerant software applications, the use of patterns would lead to well structured applications. That is, [18] 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 and recovery block, consensus, voting. In addition, the hybrid pattern structure can be constructed through recursive combination of N-Version programming and the others. The work addressed also the power of

the technique through the support of the advanced software voting techniques. Extending this framework, [19] 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 addition to the above, the recently completed *FP6 SERENITY* project has introduced a new notion of 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 provided 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 – S&D classes, S&D patterns and S&D implementation. Such validated S&D patterns, along with the formal characterization of their behavior and semantics, can also be the basic building blocks for S&D engineering for embedded systems. [24] explains how this can be achieved by using a library of precisely described and formally verified security and dependability (S&D) solutions, i.e., S&D classes, S&D patterns, and S&D integration schemes. The work of [16] reports an empirical experience, about the adopting and eliciting S&D patterns in the Air Traffic Management (ATM) domain, and show the power of using patterns as a guidance to structure the analysis of operational aspects when they are used at the design stage. A survey of approaches to security patterns is proposed in [17].

2.7 State of the Art on Repository

This section aims to present related work on the repository concept in order to determine the most appropriate structure for the TERESA repository. The repository concept is used in different research fields. Among them we can distinguish: model repository, pattern repository, software repository, repository of software components, ontology repository for the semantic web, web services repository, etc. In our context, we focus on the review of the most relevant works: model repository and pattern repository.

2.7.1 Model Repository

In the Model Driven Development (MDD), model repositories [25, 26, 27, 28, 9] are used to facilitate the interoperability of models and tools by managing modeling artifacts. In Biology, the CellML Model Repository [29] provides free access to over 330 biological models. The CellML Model Repository deals with the versioning and uses the repository to store version information at the model level. When a model is modified then a new version is created and added to the repository. Model repositories are often built as a layer on top of existing basic technologies (for instance, databases).

In order to ease the query on the repository, metadata can be added to select the appropriate artifacts. Therefore some repositories exist which are composed solely of metadata. For instance, as presented in the standard ebXML [30] and an ebXML Repository Reference Implementation [31], a service repository can be seen as a metadata repository that contains metadata about location information to find a service. In [27], the authors proposed a reusable architecture decision model for setting-up model and metadata repositories. They aimed to design data for model and metadata repositories. In addition, some helpers are included in the product for selecting a basic repository technology, choosing appropriate repository metadata, and selecting suitable modeling levels of the model information stored in the repository. In [32], they proposed a repository implementation with storing and managing of artifacts support. The supported artifacts are: metamodels, models, constraints, meta-data, specifications, transformation rules, code, templates, configuration or documentation, and their metadata.

The running ReMoDD (Repository for Model Driven Development) project [28] focuses on the MDD for reducing the effort of developing complex software by raising the level of abstraction at which software systems are developed. This approach is based on a repository that contains artifacts that support research and education in MDD. The ReMoDD platform has been developed providing a set of tools to interact with the repository. Concretely, ReMoDD artifacts consist of: documented MDD case studies, examples of models reflecting good and bad modeling practices, modeling exercises and problems that can be used to develop classroom assignments and projects. Another issue is graphical modeling tool generation as studied in the GraMMi project [33]. In this project the repository is based on three levels of abstraction (metametamodel, metamodel and model). The repository stores both metamodels (notation definitions) and models (instantiation definitions). The repository access is made thanks to an interface provided by itself. GraMMi's Kernel allows to manage persistent objects. So this kernel aims at converting the objects (models) in an understandable form for the user via the graphical interface.

Recently, the MORSE project [34] proposes a Model-Aware Service Environment repository, for facilitating services to dynamically reflection models. MORSE addresses two common problems in MDD systems: traceability and collaboration. The model repository is the main component of MORSE and has been designed with the goal to abstract from specific technologies. MORSE focuses on runtime services and processes and their integration and interaction with the repository.

Closely related to implementation issues, we can cite Netbeans MDR [35] and Eclipse CDO [36]. The first one is the well known model repository implementation that stores models and model instances. The second one, allows to store models, but no XMI model instances. Furthermore, both repositories support versioning, security aspects, etc.

2.7.2 Pattern Repository

Patterns are stored in repositories to comprehensibly explain their classification. The organization of patterns in a repository allows to discover the relationships among them and

to facilitate the selection of the most appropriate ones. The repository may have a structure in order to optimize the accesses (selecting patterns with criterion's and publishing new patterns into). Finding the appropriate pattern to solve a particular security or/and dependability problem is difficult because of the lack of a scientific classification scheme for S&D patterns. In the following we discuss some work related to such a problem: classification schemes to help in finding the appropriate pattern.

Some classifications are based on security concepts. For example, ISO/IEC 13335 [37] provides a definition of the five key concepts: security, confidentiality, integrity, availability and accountability. A pattern classification scheme based on these domain level concepts, will facilitate pattern mining and pattern navigation. An implicit culture approach for supporting developers in choosing patterns suitable for a given problem is described in [38]. In this vision, the repository contains patterns that are selected depending on the history of their use regarding decisions made by other developers to deal with related problems. In [39], a mathematical structure is proposed for organizing patterns depending on several categories. An ontological approach for selecting design patterns is proposed in [40] to facilitate the understanding and reuse during software development. In their paper, the authors present an ontology which describes the design pattern format and their relationships. They use a pattern system/language in order to facilitate the design, integration, selection and reuse of design patterns.

Most existing classifications in the literature [41] are based on:

- Applicability is used to protect resources against unauthorized use, disclosure or modification. In addition, applicability is used to make predictable and uninterrupted access to resources or services.
- Product and process (structural and procedural)
- Logical tiers (a) web: this tier takes into account the external requests, authentication and authorization (b) business: this tier takes into account the security services in the business like RBAC (c) integration: this tier facilitates secure integration with external data sources
- Security concept (confidentiality, integrity, availability and accountability) – see ISOIEC 13335-1 and ISO 7498-2 [42, 43].

Another aspect that has been considered is system viewpoints. Based on the idea of the Zachman Framework [44] (classification based on system viewpoints and interrogatives) the Microsoft Patterns and Practices group Classification [45] distinguishes the following elements: (a) Merits (clearly identifies the context of each pattern, help to identify missing patterns), (b) Flaws (more dedicated to functional patterns – non-functional patterns tend to cover many levels of system development and also many interrogatives), (c) Improvement (add icons in each pattern to provides classifications).

2.8 Synthesis and discussion

This section summarizes on the described background, TERESA requirements and the state of the art.

2.8.1 S&D Pattern Modeling

2.8.1.1 Synthesis

While many S&D patterns have been designed, still few works propose general techniques for S&D patterns. As most of them use the same generic concepts to characterize patterns, there is no real consensus about what a pattern is. Furthermore, the term pattern is often ambiguous because it is used to encode the solution of a recurrent problem and to deal with a model in the better case instead of the pattern implementation. In software engineering, design patterns are considered as effective tools for the reuse of specific knowledge. However, there is still a gap between the development of the system and the pattern information. Table 2.1 summarizes the different approaches.

For the first kind of approaches [5], design patterns are usually represented by diagrams with notations such as UML object diagrams, 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. Actually, the existing descriptions are not formal definitions and sometimes leave some ambiguities about the exact meaning of patterns. There are some promising and well-proven approaches [46] based on Gamma et al. However this kind of techniques do not allow to reach the high degree of flexibility in the pattern structure which is required to reach the TERESA target.

The visualization technique promoted by LePUS [22] is interesting but the degree of expressiveness proposed to design a pattern is too restrictive.

UMLsec [10] (approach based on modeling security in UML) and the TERESA proposal are not in competition but they complement each other by providing different view points to the secure information system. In concept, the TERESA modeling framework is similar to the one proposed in the SERENITY project. Nevertheless the pattern structure is rigid (a pattern is defined as quadruplet) and consequently is not usable to capture specific characteristics of S&D patterns. Please note, however, that SERENITY proposes several levels of abstraction to bridge the gap between abstract solution and implementation but not to get a common representation of patterns for several domains.

2.8.1.2 Discussion

In the TERESA approach, different levels of abstraction are used to get a common representation of patterns for several domains. In the TERESA concern, systems include

Approach	Representation Language	Structure Kind	Abstraction Levels
Gamma et al. [5]	*textual* - use: documentation or just for modeling	*rigid:* not usable to capture specific characteristics of S&D patterns)	*no:* difficult to use in a building process
Approaches based on Gamma et al. [46]	*textual and graphical* - use: documentation and diagrams	*flexible:* usable to capture specific characteristics of S&D patterns (but handling is not easy)	*no:* difficult to use in a common building process for several application domains
UMLsec [10]	*graphical* - use: UML diagrams	UML profiles	usable at design level
LePUS [22]	*visual language* - framework for specifying design patterns	*rigid:* not usable to capture specific characteristics of S&D patterns	usable at design level
SERENITY [47]	*pattern modeling language* - use: model transformation	*rigid:* not usable to capture specific characteristics of S&D patterns	*yes:* the goal is to bridge the gap between the abstract solution and the implementation
Our approach	*pattern modeling language* - use: model transformation	*flexible:* thanks to CBSE structure, adequate to deal with RCES	*yes:* the goal is to propose a common representation of patterns for several domains (PFS, DIPM, DDPM)

Table 2.1: Pattern development life cycle comparisons

a combination of hardware and software components. This may add some difficulties to build a simple modeling framework. An even high level of abstraction is proposed to represent S&D patterns to capture several aspects of security and dependability in the different domains of RCES, not an implementation of a specific solution. Other issues are:

- In order to integrate a pattern in a system (application), some significant additional information about the pattern is required. For instance, the interfaces and their requirements. The goal is to capture how the system interacts with the patterns, and how the internal structure of the pattern interacts with other patterns in the case of composite patterns.
- What kind of interfaces should be used? Especially when dealing with software and hardware components?

- What form of graphical representation should be used? UML, SysML or other diagrams?
- How to deal with the previous points with regards to several levels of abstraction?

2.8.2 Pattern Repository

2.8.2.1 Synthesis

The main goal of a repository is to store data and to offer a set of actions to interact with it. Most of the time a repository has to provide the following actions: store files, authentication and access control, check-out/check-in files, file versioning, file metadata storage and data search. Some of the classical formats used are XML, XMI and XSD. Then the graphical view of artifacts can be made by using some format transformations. This one can be realized thanks to XSLT (eXtensible Stylesheet Language Transformation) transformations to ease the reuse of XML solutions. On another hand, XBRL (eXtensible Business Reporting Language), an XML format, also allows the interoperability of information. Thus, an XBRL file can be converted to a standard format such as HTML ASCII and also PDF, which will provide an ergonomic aspect.

2.8.2.2 Discussion

In TERESA approach, the following questions arise about the specification of the repository:

- What kind of visualization interfaces to use?
- What kind of interaction interfaces to use? (some information about how artifacts are published and how artifacts are queried must be provided by the repository)
- How to organize the data ?
  - what kind of metadata to make the selection of data?
  - what kind of data structure to use?
  - what kind of use is supported? (static and/or dynamic).
- how to store the data ?
  - what kind of format to use in order to store the data?

3 Conceptual Framework

As stated before, a repository is a data structure that stores artifacts and that allows the user to publish and to select them. In the context of the TERESA project, the main objective of the repository is to propose a set of modeling artifacts for several domains. As a result of analysis of the requirements (D4.1 and D2.1) we have identified a set of modeling artifacts mainly, S&D patterns, Resource models, S&D and Trust models, constraint models and process models. Combining these modeling artifacts seems to be a very promising approach for building TERESA applications. The next chapter introduces our vision to the use of such a repository in an MDE approach. The details about the formalization of the TERESA process is studied in WP3.

3.1 Motivations

The TERESA approach promotes a model-based approach coupled with a repository of modeling artifacts mainly, S&D patterns, S&D and Resource properties models, Process model to define an engineering discipline for trust that is adapted to resource constrained embedded systems. In such a vision, the S&D patterns derived from (resp. associated with) domain specific models aim at helping the application developer to integrate application S&D building blocks. This is the reason why we advocate the use of a model-based repository, where patterns are clearly related to domain models.

Patterns are used to specify architecture and design aspects. Usually, they refer to templates which describe solutions for commonly occurring problems. Unfortunately, most of S&D 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 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 correct application of a pattern because the description does not consider the effects of interactions, adaptation and combination. This makes 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 security issues) remains unsolved.

The main goal of the $WP4$ is to define a modeling and development framework to support the specifications, the definitions and the packaging of a set of modeling artifacts to assist the developers of trusted applications for resource constrained embedded systems. Security and dependability patterns are not only defined from a platform independent viewpoint

(i.e. they are independent from the implementation), they are also expressed in a consistent way with domain specific trust models. Consequently, they will be much easier to understand and validate by application designers in a specific area. That is:

- S&D experts can make patterns publically available
- S&D patterns can be used by RCES developers in other companies
- S&D patterns can be derived from and associated with domain specific models
- S&D patterns help application developers to integrate application building blocks with S&D building blocks

The resulting documentation and repository prototype as well as a number of guidelines will facilitate 1) the populating of the repository with further security and dependability or S&D patterns, and 2) the transformation of the S&D patterns into platform dependent specifications. This will be completed by the following concerns:

1. S&D pattern modeling framework to get a common representation of patterns for several domains in the context of RCES (1) to capture the essence of the pattern, (2) to provide enough detail to enable the usability of the pattern by a non-specialist, (3) to provide sufficient information to be validated, (4) to provide sufficient explanation to enable the usability of the pattern in other domains as well as the domain in which the pattern was defined.
2. Repository of integrated models (S&D patterns, Resource models, trust models, engineering process models, ..)
3. Repository access tool allowing application designers to capitalize on the MDE even if they are not experts in modeling
4. Patterns are formally validated when they are saved in the repository. Same, during the pattern integration, formal validation helps us to guarantee the correctness of this step.
5. When a pattern has been formally validated, implementations with automatic derived guidelines for platform dependent implementation of the patterns will be available.

As described in the next sections, TERESA promotes a new discipline for system engineering using a pattern as its first class citizen: *Pattern-based System Engineering (PBSE)*. First, we introduce a set of definitions and concepts that will prove useful in understanding our approach. Next, we provide an overview of the PBSE approach and an example of a process of building patterns.

3.2 Concepts and Definitions

This section provides a set of definitions and concepts that might be useful for the understanding of our approach. This extends the definitions and concepts given in Section 2.2.

Definition 3.1 (domain)**.** *A domain can be defined as the scope of a system in terms of its concerns and as methods and mechanisms employed in the development of a system.*

In the context of TERESA project, a domain represents all the knowledge including protocols, processes, methods, techniques, practices, OS, HW systems, measurement and certification related to the specific domain. With regard to the artifacts used in this project we will identify the first classes of the domain to specialize such artifacts. For instance, the specification of a pattern at domain independent point of view is based on the software design constructs. The specialization of such a pattern for a domain uses a domain protocol to implement the pattern solution (see example of secure communication pattern given in Section and Section).

Domain independent and domain specific modeling artifacts are required in TERESA. As the objective is to reuse the domain independent model artifacts for several industrial partners and also let them be able to customize those domain independent artifacts with their domain knowledge or requirements to produce their own domain specific artifacts. Thus, the how' to support these concepts should be captured in the specification languages (see metamodels presented in Chapter 5).

In the following, we propose to use two dimensions to classify patterns views and by abuse we use the term pattern instead of pattern view. The first one is based on the system process model viewpoint among specification, analysis, design, and implementation; the second one allows to take into account our context: domain independent and domain specific.

- Domain Pattern View. A domain pattern is a pattern whose form is described by means of terms and concepts from an application domain.
- System (conceptual) Pattern View. A system pattern expresses a fundamental structural organization or schema for pattern systems (ex. using SysML, ...).
- *Architectural Pattern View.* An architectural pattern expresses a fundamental structural organization or schema for software pattern systems as the first analysis of the system pattern. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them (ex. using SySML, UML, ...).
- *Module Pattern View.* A module pattern provides a scheme for detailed design of the architectural view of a software pattern artifacts system using means of software design constructs, for example objects, classes, components, inheritance, aggregation and use-relationship (ex. using UML, ...).

- *Implementation Pattern View.* An implementation view is a low-level view specific to a programming language. It describes how to implement the pattern artifacts defined at the module level and the relationships between them using the features of the given programming language constructs (ex. using C, Java, ...).

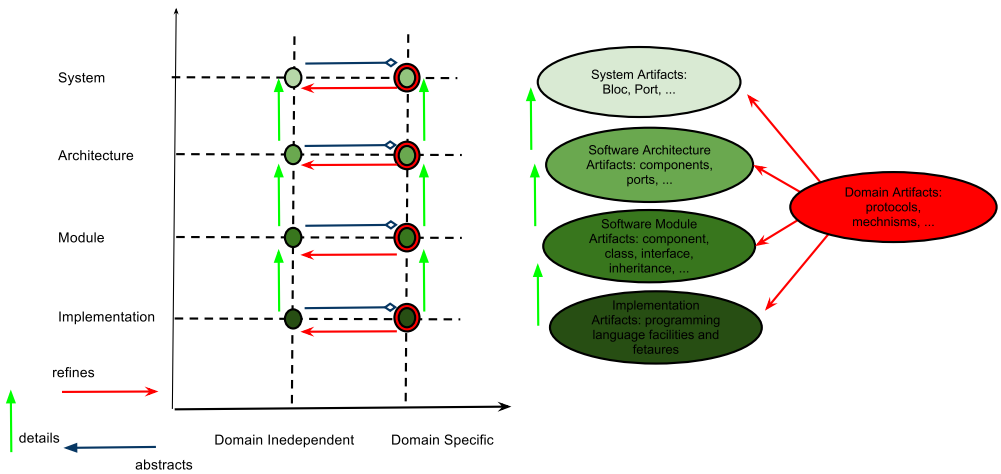


Figure 3.1: An overview of the TERESA Patterns Types

Using these definitions, domain patterns are restricted to an application domain. As shown in Figure 3.1, we can define $4(n+1)$ types of patterns where n is the number of domain applications. On the one hand, as a vertical transformation following the engineering system process model we can define 04 types (or levels). Each level complements, or elaborates (details) upon the previous higher level using system and software engineering design constructs until implementation using specific implementation programming language constructs. On the other hand, we define a reference pattern as a domain independent pattern and as a horizontal transformation following the domain concept we define a domain specific view for each level and for each application domain using terms and concepts from such a domain.

3.3 Pattern-based Lifecycle Process Models

PBSE addresses challenges similar to those studied in software engineering: many of the methods, tools and principles of software engineering used in these systems will be used in the same way or adapted/extended in PBSE. Closely related to our vision is the Component Based Software Engineering (CBSE) [48], as shown in Figure 3.2 [1]. Therefore, PBSE focuses on patterns and from this viewpoint addresses two kind of processes: the process of *pattern development* and *system development with patterns*. The latter means building systems from pre-existing patterns. For that, in the TERESA vision these two processes are separated; and the patterns (resp. the other modeling artifacts) should have already

[1] This Figure is an adaptation of the one presented in [48].

been developed and stored in the repository. Moreover, we defined additional processes, mainly validation; which is associated with the pattern development, pattern finding and evaluation process and pattern integration process that are associated with the system development one. Beyond this, the main concern of the *pattern development* is design pattern for reuse and the one of the *system development with patterns* is the finding of the adequate pattern and evaluating them with regard the system under-development requirements.

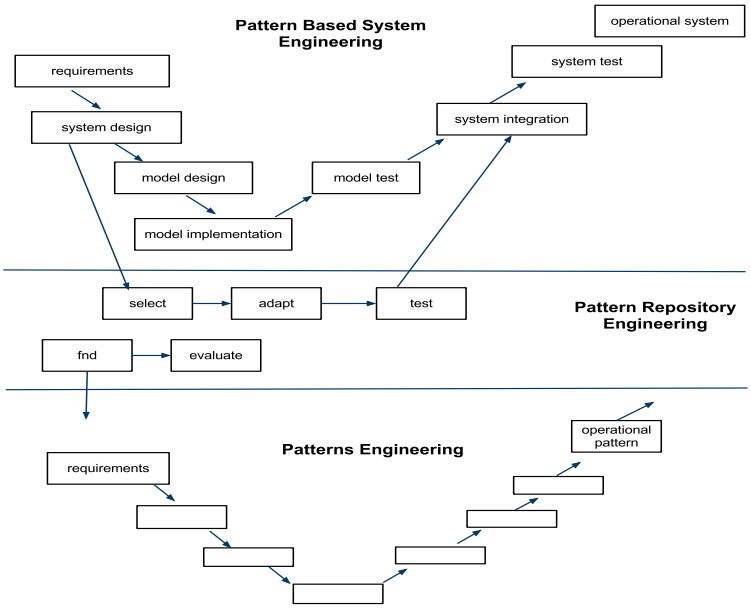


Figure 3.2: Overview of pattern based development

The top part of Figure 3.2 illustrates an example of a V-Model adopted for the PBSE approach. The V-Model is the process model widely used in many organizations and particularly by our industrial partners. In the Deliverable D3.2, the process is described in more details and illustrated through the example of industry control systems in the railway domain. The study is focused on the safety process viewpoint.

The process shown in Figure 3.2 is a simple and an overview of the reality. In such a process, we avoid many problems and build on several assumptions mainly the patterns selected are sufficiently appropriate with regard to the units design (architecture and module), thus the required adaptation involves less efforts and expertise than the units' design. The downside of this approach is that in the very early stage of the development, mainly during the requirements and design phases, the requirements engineers and the system architects have to be aware of existing patterns. These observations yield the following: the need of new process formalism to capture the Pattern-based Lifecycle Process Models. This task is studied in depth in Workpackage 3.

3.4 Building ReUsable Modeling Artifacts - S&D Patterns

The process of building patterns is also based on the V-model (see the bottom part of Figure 3.2) with some modifications to capture the patterns; in addition to the demands on the pattern functionality, a pattern is built to be used and reused. The later implies generality, extensibility and flexibility. These characteristics significantly change the pattern life. For more details on these characteristics, the reader is referred to the deliverable D4.1 about pattern lifecycle requirements. In the following, we present the basic activities of the lifecycle pattern process model.

- *Requirement analysis and specification*: In this phase we deal with the activity of analysing the possibilities of realizing the solutions that will fulfill the requirements. The pattern's services, constraints and goals are defined. This means that the output of this phase establishes what the pattern is supposed to do. In our case this implies security, dependability and resource requirements as functional requirements and portability and generality as non-functional requirements.
- *Pattern system and software design*: The goal of this phase is to define the system and software architecture of the pattern. Then, a detailed design (or a module design) refines such an architecture. Software design establishes a set of design elements to specify a pattern in the form of artifact abstractions and their relationships.
- *Implementation and unit testing*: This phase is responsible of the formalization of the design solution in an executable way using programming language as a set of unit constructs. The unit testing follows the implementation.
- *Pattern system integration*: The goal of this phase is the integration of the pattern artifacts.
- *Pattern system verification and validation*: The correctness of the complete pattern is verified and validated with respect to its requirements. The patterns should be verified in isolation in different configurations, either formally or by testing.
- *Operation support and maintenance*: Overall the phases of the pattern process, the documentation is generated as a delivery for better understanding of the pattern. The maintenance activities include some steps that are similar to the integration phase: A new or modified artifact is deployed into the pattern system. For that, a pattern will be modified or a new version of a pattern will be created.

To foster reuse, we introduced the concept of *Domain Modeling View*. Particularly an S&D pattern at domain independent level exhibits an abstract solution without specific knowledge on how the solution is designed and implemented with regard to the application domain. Thus a domain independent pattern can easily be integrated into the overall abstract system specification. The benefit of this structure is to offer a common language for different domain applications. Following the MDE process, the domain independent model of patterns is then refined towards a domain specific level, taking into account domain artifacts, concrete elements such as mechanisms to use, devices that are available,

etc. Consequently, an S&D pattern at domain specific level contains the respective information.

4 TERESA Framework

As introduced in the deliverable D7.1, Model Driven Engineering (MDE) based solutions seem very promising to meet the TERESA needs. The idea promoted by MDE is to use models at different levels of abstraction for developing systems. In other words, models provide input and output at all stages of system development until the final system itself is generated. The advantage of having an MDE process is that it should clearly define each step to be taken, forcing the developers to follow the defined methodology. MDE allows to increase software quality and to reduce the software systems development life cycle. That is, a same model is used for all businesses and, thus, the consistency is ensured by construction. Moreover, from a model, it is possible to automatize some steps by model refinements and generate code for all or parts of the application.

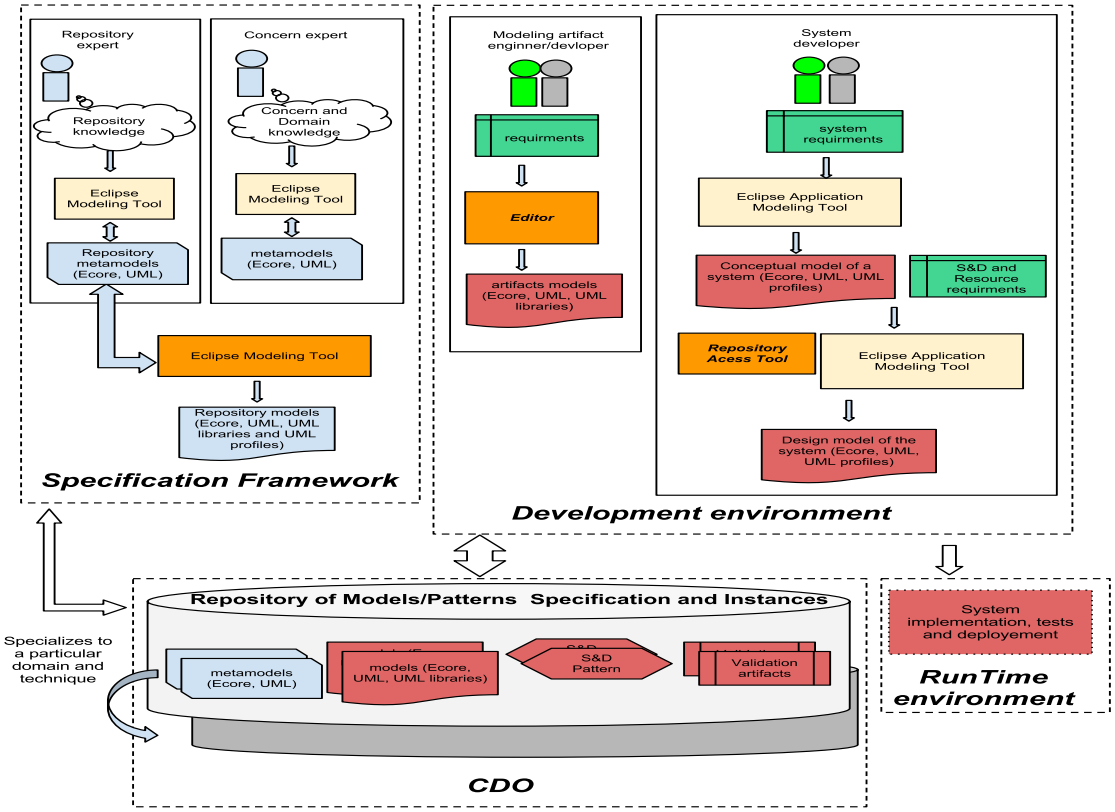


Figure 4.1: The proposed framework for TERESA applications

In the following sections, we present an overview of the TERESA framework based on a set of modeling artifacts for embedded system engineering (see Figure 4.1). In the next two chapters, we present the specification languages for the TERESA modeling artifacts and the repository respectively.

4.1 Specification Framework

The core of the framework is the definition of the conceptual modeling artifact models. Such languages are obtained by using a metamodeling approach, as illustrated in the next chapter. As we shall see, in our context we deal with the following concerns: (1) process, (2) S&D pattern, (3) S&D property and constraint, (4) resource property and constraint and (5) repository.

4.2 Development Environment

The development environment associated to the repository is based on Specification Framework for defining modeling artifacts, and is composed of tools for modeling artifact instantiation and tools for the selection of a modeling artifact, by accessing the repository, which modeling artifact can be used to satisfy the system developer specification.

4.3 Repository

Once this specification language has been defined, it is possible to develop a repository in which to store both TERESA modeling artifacts specifications and instances. The development of such a repository leverages the metamodeling techniques and the availability of MDE tools which allow to generate a skeleton of the repository conforms to its structure and the modeling artifacts. The technology on which the repository is based is Eclipse CDO/EMF. For more details on the implementation issue, the reader should refer to Deliverable D4.3. the deliverable D4.3.

4.4 Runtime Environment

With regard to the run-time environment, we refer to system implementation and execution platforms as the ones offering their services to support the modeling artifact execution. This part is studied in the WP6.

5 Modeling Artifacts Specification

In the TERESA vision, a *pattern* is the key element for building trusted applications for resource constrained embedded systems. That is, we depict in Figure 5.1 a snapshot of the modeling artifacts organized around the pattern. In the following sections, we present the specification of TERESA modeling artifact using metamodeling technique. As we shall see, the models are presented in the form of UML Class Diagrams.

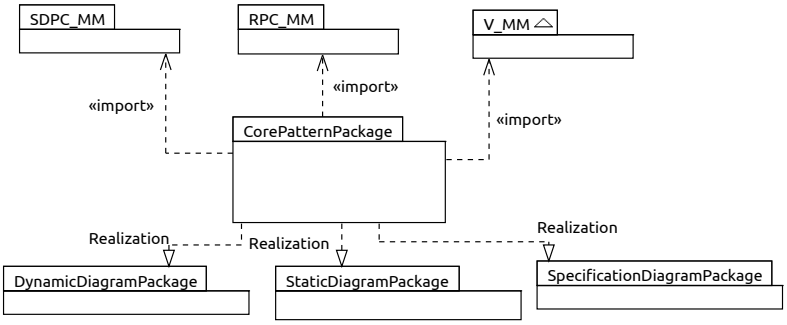


Figure 5.1: Pattern for Security & Dependability MetaModel Dependencies

In the following we present the fundamental concepts of the modeling artifacts composing our repository. Then, we present in depth the repository specification in the next chapter.

- *Repository Centric Process MetaModel (RCP_MM.)*
- *Pattern for Security & Dependability MetaModel (P4SD_MM.)*
- *Security&Dependability Property and Constraint MetaModel (SDPC_MM.*
- *Resource Property and Constraint MetaModel (RPC_MM.)*
- *Validation MetaModel (V_MM.)* contains the validation artifacts for the pattern validating process.

5.1 Repository Centric Process Modeling (RCP_MM)

The process model is studied in Workpackage 3 and the results are presented in the deliverable D3.2.

5.2 S&D Pattern Modeling (P4SD_MM)

The goal of the current section is to propose model-based S&D patterns to get a common representation of patterns for several domains in the context of trusted embedded system applications. The solution envisaged here is based on meta-modeling techniques to encode S&D patterns at an even greater level of abstraction. Therefore, a pattern can be stored in a repository and can be loaded according to the desired S&D properties. As a result, S&D patterns will be used as bricks to build trusted applications through a model driven engineering approach.

One of the major concerns in designing secure and dependable systems is to determine at which level of abstraction security and dependability concerns should be placed. The supporting research includes specification, modeling, implementation mechanisms, verification... to name only a few. For example, distributed systems are organized into separate layers following some reference model, e.g. applications, middleware and the operating system services. Combining the layered organization of target applications, domain specific systems and S&D patterns modeling leads roughly to what is shown in Fig. 5.2.

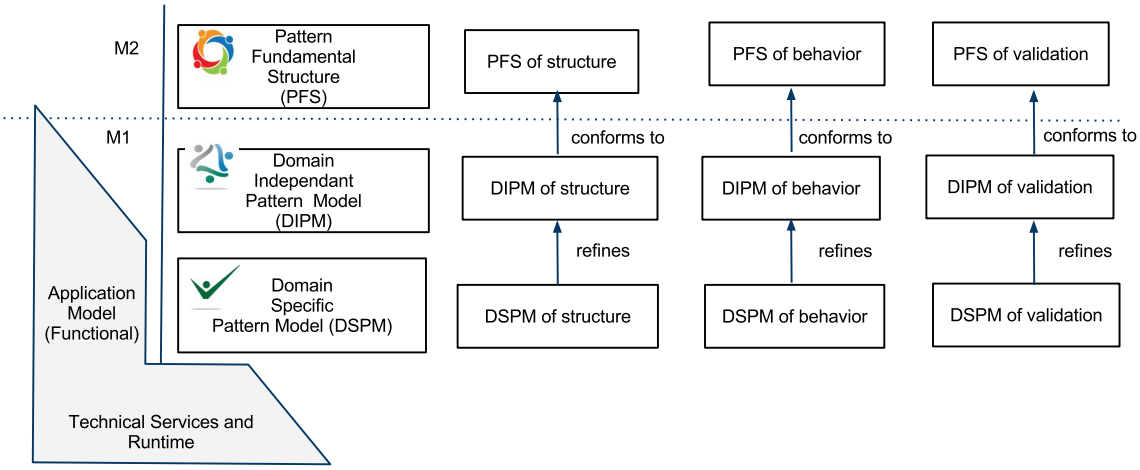


Figure 5.2: S&D Pattern Modeling Framework Structure

The framework must cope with S&D, Resource properties, Constraints, and domain specific properties. For this purpose, the proposition presented in this part of the deliverable is based 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 S&D applications in the context of embedded systems (since combining S&D and domain specific artifacts introduces a great complexity), and secondly to overcome the lack of formalism of the classical pattern form (e.g. textual). The benefit of this structure is to offer a common modeling language for several domains in the context of trusted embedded systems.

The following subsection describes an example in order to enhance the issues identified in the document. Then, the first level of abstraction, namely PFS, will be described.

5.2.1 Motivating Example: Secure Communication Pattern

The essence of Fig. 5.2 is to promote the separation of general-purpose services from implementations. In our context, this structure highlights the separation of general-purpose of the pattern from its related mechanisms. This is an important issue to understand the use of patterns for security and dependability and, in particular, the notion of *trust*. In which layer security mechanisms are placed depends on the trust a client has in how *secure* the services are in some particular layer. As example of a widely used pattern we choose the Secure Communication Pattern referred to in the following as SCP. Messages passing across any public network can be intercepted. The problem is how to ensure that the data is secure in transit, i.e. how to guarantee data authenticity. This is one of the goals of the SCP.

However, SCP are slightly different with regard to the application domain. A system domain may have its own mechanisms and means, protocols that can be used to implement this pattern range from SSL, TLS, Kerberos, IPSec, SSH, to WS-Security. In summary, they are similar in the goal, but different in the implementation issues. So, the motivation is to handle the modeling of S&D patterns by following abstraction. As an example we use SSL mechanism [1] as a concrete implementation of the SCP.

The SSL mechanism is composed of two phases: The *SSL Handshake* that establishes a secure channel, and the *SSL Record* in which this channel can be used to exchange data securely. The client initiates the SSL handshake by providing the server with a random number and information about the cryptographic algorithms it can handle. The server replies by choosing the actual algorithm to use, requiring the client to authenticate itself (this is optional and used in our example) and by sending a random number of its own and its certificate issued by some CA trusted by both the server and the client.

There are first two messages. For authenticating itself, in the final handshake message the client includes its own certificate, a signature on all handshake messages generated with the respective private key, and a third random number encrypted using the server's public key contained in the server's certificate. After having verified the certificates and signature, both client and server use the exchanged random numbers to generate session keys for generating and verifying message authentication codes (MACs) and for encrypting and decrypting messages.

Since the key used by the client for generating a MAC / encrypting a message is used by the server only for MAC verification / decryption and vice versa, and since they are based on one random number confidential for the client and the server, the keys establish a channel that provides authenticity and confidentiality for both client and server.

[1] SSL or its update named TLS proposed in RFC 2246

5.2.2 Pattern Fundamental Structure (PFS)

PFS is a metamodel defining a new formalism (i.e., language) for describing S&D patterns, and constitutes the base of our pattern modeling language. PFS describes all the artifacts (and their relations) needed to represent S&D patterns in the context of trusted embedded systems applications. Here we consider patterns as building blocks that expose services (via interfaces) and manage S&D and Resource properties (via features) yielding a unified way to capture meta-information related to a pattern and its context of use. Such a pattern is specified by means of a domain-independent generic representation and a domain-specific representation.

As shown in Figure. 5.3, the pattern representation we propose provides a clear and flexible structure. Such a structure is already used with success in CBSE (Component-Based Software Engineering). CBSE is a discipline that is known to be reliable in the area of software engineering and that is considered as to be a good solution to optimize software reuse and dynamic evolution while guaranteeing the quality of the software [49]. Moreover, the modularity enables to tame the complexity of large systems.

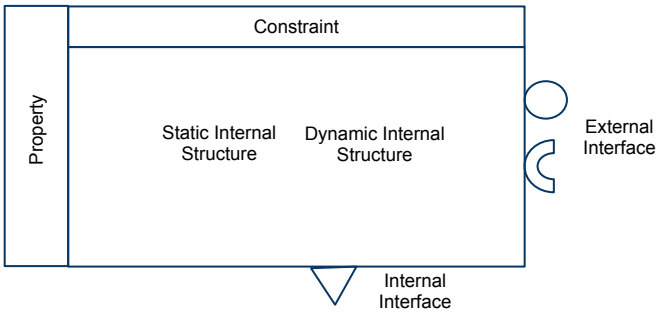


Figure 5.3: Representation of an S&D Pattern

The following paragraph details the *CorePatternPackage* (see Figure 5.1) composed of the principle classes of our meta-model to specify an S&D pattern, as described with UML notations in Figure. 5.4.

- *IPattern.* represents a modular part of a system that encapsulates a solution of a recurrent problem.

  An *IPattern* is modeled throughout the development life cycle and successively refined into deployment and run-time. An *IPattern* may be manifested by one or more artifacts, and in turn, that artifact may be deployed to its execution environment. A deployment specification may define values that parametrize the pattern's execution. The *IPattern* has some fields that define its identity. These fields are based on the GoF [5] information and are defined as follows:

  – *name.* is the name of the pattern,

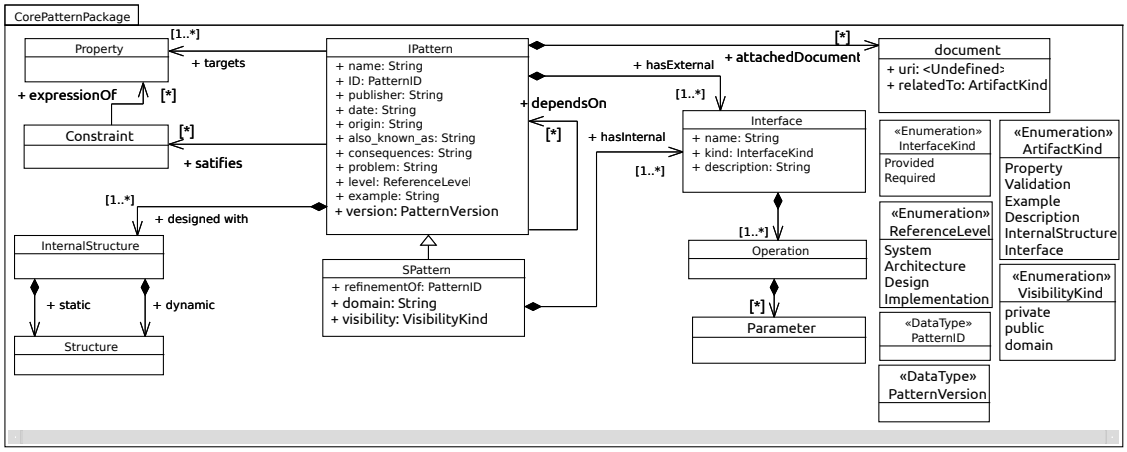


Figure 5.4: Pattern for Security and Dependability Metamodel

- *id.* defines the identity of the pattern
- *version.* the version of the pattern,
- *date.* date of creation,
- *publisher identity.* is the identity of the publisher(s) – login, PW, name, organization, email,
- *origin.* corresponds to the origin of the pattern – Industrial, Academic,
- *also known as.* gives a list of names under which the pattern is also known,
- *related patterns.* describes the dependencies with other patterns,
- *consequences.* corresponds to the set of consequences which occur once the pattern is integrated,
- *problem.* describes the problem solved by the pattern,
- *level.* is the level of conception – Architecture, Design, Implementation,
- *example.* provides an example.

• *Interface. IPattern* interacts with its environment with *Interfaces* which are composed of *Operations*. So, larger pieces of a system's functionality may be assembled by reusing patterns as parts in an encompassing pattern or an assembly of patterns, and wiring together their required and provided interfaces. More precisely, 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. Finally, two kinds of interfaces are considered:

  - *External interface.* allows implementing interaction with regard to:
    * *integrate* an *IPattern* into an application model. These interfaces are realized by the *IPattern*.
    * *compose IPatterns* together.
  - *Internal interface.* allows 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*.

- *Property.* is a particular characteristic of a pattern. A *Property* is either an *SDProperty* or a *RProperty* (see deliverable D3.2). So, each property of a pattern will be validated at the time of the pattern validating process and the result will be compiled as a set of constraints (see the *RPC_MM* and *SDPC_MM*) which have to be satisfied by the platform. This artifact will simplify and enhance the selection/search activities during the pattern-based engineering process.
- *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 have several possible implementations.
- *Static Diagram Package.* is the diagram that allows to design the static internal structure of a pattern.
- *Dynamic Diagram Package.* is the diagram that allows to design the dynamic internal structure of a pattern.
- *Specification Diagram Package.* represents the main diagram for designing a pattern.
- *SPattern.* inherits from *IPattern*. It is used to build a pattern at DSPM. A *SPattern* has *Internal Interfaces* in order to interact with the platform.

An important point concerns the completeness of the internal pattern representation. For this purpose, we have proposed an extension mechanism which will be specified in the future. Thus, if a domain specific artifact is missing, the repository supports the extension mechanism which allows the user to add the needed artifact.

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).

We use UML notations and instantiation mechanisms to represent the pattern at the different levels of the pattern process. The following two figures depict the example of the Secure Communication Pattern (SCP), presented in Section 5.2.1, at system and architecture level respectively. Then, in the two next sections the required artifacts for the module design level will be pointed out while following the two abstraction levels (i.e.,

DIPM and DSPM) through the same example. Note, however, that for simplicity's sake, many functions of this use case have been omitted.

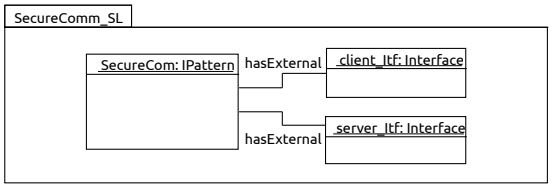


Figure 5.5: Secure Communication Pattern at system level

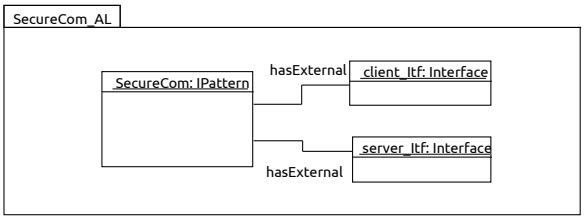


Figure 5.6: Secure Communication Pattern at architecture level

5.2.3 Domain Independent Pattern Model (DIPM)

This level focuses on pattern domain independent artifacts. 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 S&D patterns in RCES applications. In our case study, the DIPM of the SCP consists of two entities communicating through a secure channel. The SCP is defined as follows:

Properties. At this level, we identify two S&D properties: authenticity and trust.

External Interfaces. Let *SCPExtDI* denote the external interface to expose the SCP functionalities through function calls:
$Send(P, Q, ch(P, Q), m), Receive(P, Q, ch(P, Q), m)$, with $P, Q \in \{C, S\}$,
$ch(C, S) = ch(S, C)$ denoting the communication channel of client and server, and m a message.

Internal Structure. The behavior of SCP can be modeled by a UML Sequence Diagram describing secure date exchange between client and server through *SCPExtDI* interface.

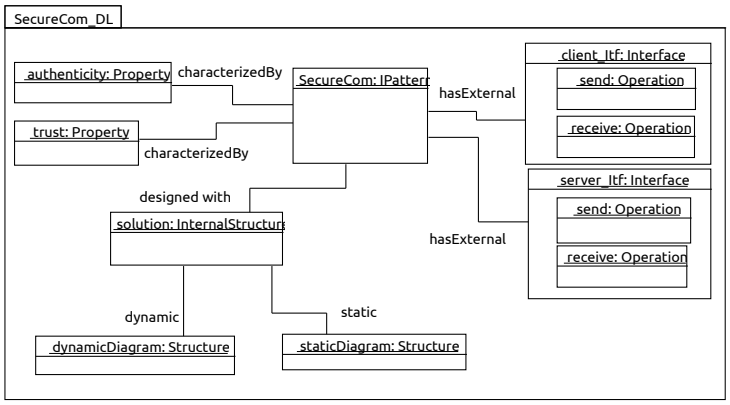


Figure 5.7: Secure Communication Pattern at design level (domain independent)

5.2.4 Domain Specific Pattern Model (DSPM)

The objective of the specific design level is to specify the S&D patterns for a specific application domain. This level offers artifacts at a low level of abstraction with more precise *information* and *constraints* about the target domain. This modeling level is a refinement of the DIPM that considers the specific characteristics and dependencies of the application domain. Different DSPM can refine the same DIPM for different domains. When using the SSL protocol as a mechanism related to the application domain to refine the SCP at DSPM, we introduce the following artifacts:

Properties. In addition to the refinement of the two S&D properties identified in the DIPM, at this level we identify some related RECS properties, e.g. the size of the cryptographic key.

External Interfaces. Let *SCPExtDS* denote the external interface. This is a refinement of the DIPM external interface:
- $send(C, S, mac_C(m), m)$: the client sends m and the corresponding MAC (Message Authentication Code) to the server.
- $recv(S, C, mac_C(m), m)$: the server receives m and corresponding MAC.

Internal Interfaces. Let *SCPIntDS* denote the internal interface as a subset of functions related to the use of SSL to refine the SCP pattern:
- $genRand(P, R); P \in \{C, S\}$: P generates a random number.
- $sign()$: the client signs the SSL handshake messages.
- $verifyCert()$: the client and server certificate, respectively, is verified.
- $extractPubKey()$: the server's public key is extracted from the certificate.
- $genMac(C, macKey_C, m, mac_C(m))$: the client generates the message authentication code (MAC) for a message using its own SSL shared secret for MAC generation.
- $verify(S, macKey_S, m, mac_C(m))$: the server verifies, using its shared secret for verification, that the message authentication code for m is correct and originates from the client.

Internal Structure. The behavior of SCP can be modeled by a UML Sequence Diagram following the SSL protocol described in Section 5.2.1, involving *SCPExtDS* and *SCPIntDS* interfaces.

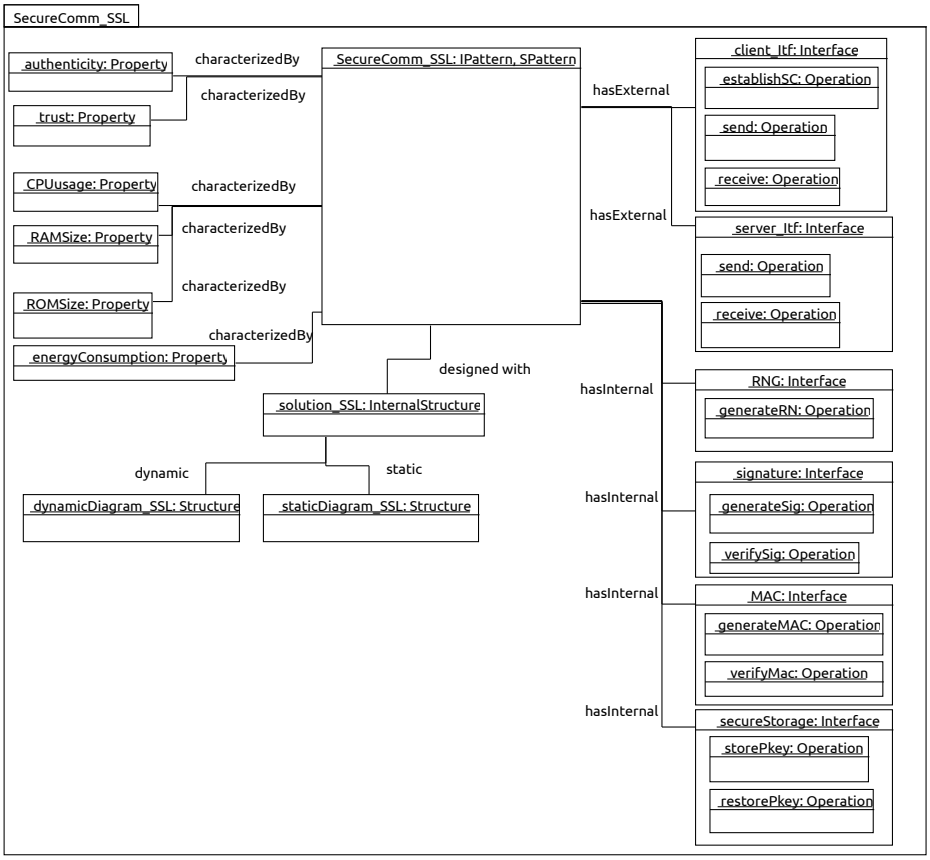


Figure 5.8: Secure Communication Pattern at design level (domain specific)

5.3 S&D Property and Constraint Modeling (SDPC_MM)

The Trust, Security, Dependability and Constraint model is studied in both Workpackages 3 and 4. The results are presented in Deliverables D3.2 and D4.3.

5.4 Resource Property and Constraint Modeling (RPC_MM)

The resource and constraint model is studied in Workpackages 3 and 4. The results are presented in Deliverables D3.2 and D4.3.

5.5 Validation Modeling (V_MM)

The *V_MM* describes the validation artifacts for the pattern validating process. The key element of *V_MM* is the Building Block (see the deliverable D5.1.). Such an artifact may be represented as a black box to allows the proof of some properties. A *VBuildingBlock* takes as input a set of *VProperties* and produces validated new *VProperties*. The principle is based on the following aspect: an homomorphism must be found between a property at domain independent level (*VDIProperty*) and a property at domain specific level (*VDSProperty*). For more details validation issues, the reader is referred to the deliverables of Workpackage 5 and to [50].

6 Repository Specification (MARM)

The ultimate goal of TERESA is to propose a common process for several domains by using a repository of modeling artifacts, and this section aims at providing the TERESA repository structure specification. The specification of the TERESA repository is based on the requirements that are detailed in the deliverable D4.1. Please note that a synthesis about the correspondence of each requirement from the deliverable D4.1 with a Repository's artifact of the Deliverable D4.2 is proposed in the conclusion of this deliverable.

The repository presented here is a model-based repository of modeling artifacts. It constitutes one of the most important key elements in the engineering process for resource constrained embedded systems. Concretely, the TERESA repository is a data structure that stores specification and instantiation of modeling artifacts coupled with a set of tools to manage/visualize/export these artifact in order to use them in engineering processes.

6.1 Overview

This section presents an overview of the repository system architecture as shown in Figure 6.1. We identified several roles. The modeling expert interacts with the repository to specify the modeling artifacts and the domain expert interacts with the repository in order to instantiate and then to store these artifacts. The repository manager is responsible for the repository administration. Finally, the *Application Developer* uses the *Application Developer Interface* to select the modeling artifact for building an application. The system is composed of several components:

- Repository storage,
- Repository management,
- Modeling artifact specification,
- Modeling artifacts instantiation,
- Repository access Tool.

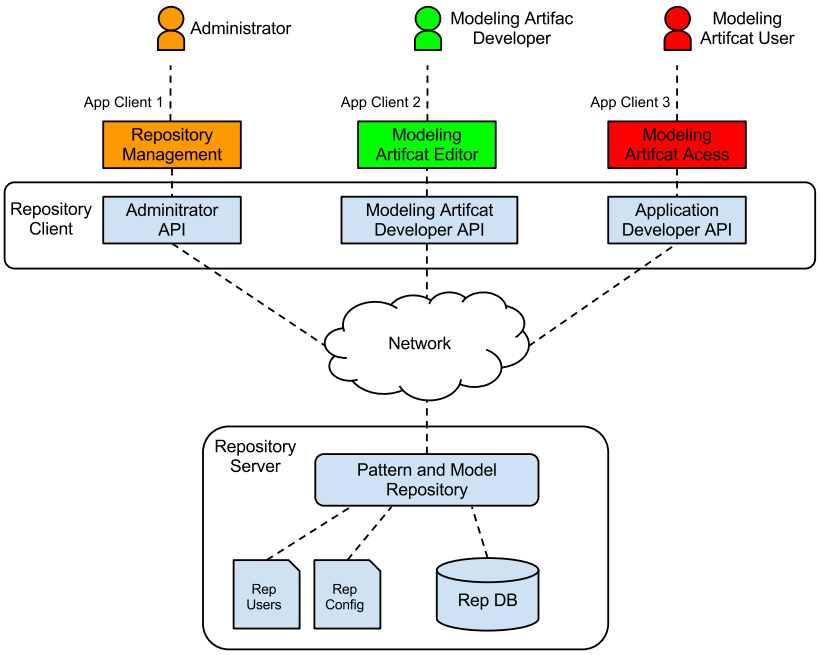


Figure 6.1: Overview of the Repository System

As we shall see, to build our repository system, we advise the use of a well known architectural style: Multitiered Architectures as an alternative client-server organizations. We quote Figure 6.2 from [51] and for our context we will use the organization *(c)*.

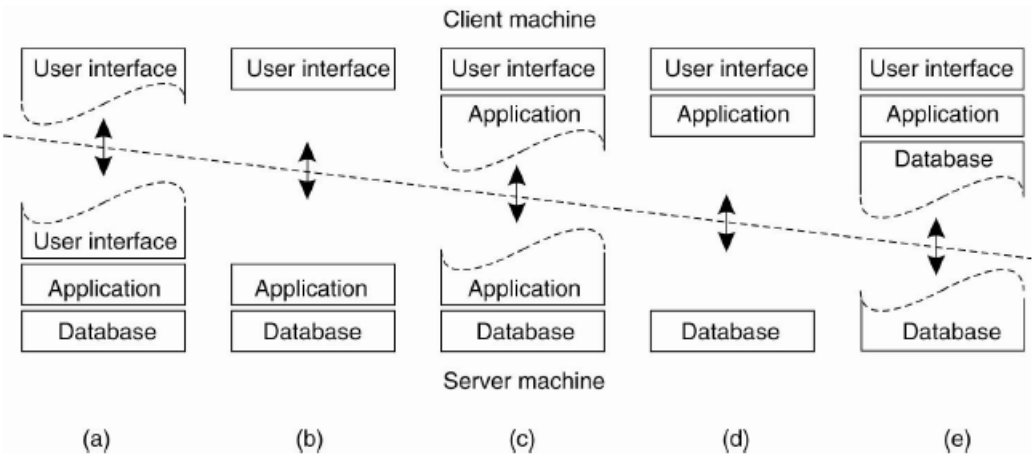


Figure 6.2: Repository System Architecture

Before presenting in depth the specification of the repository, we present in Figure 6.3 a package diagram to show a global view of both the repository structure and its interaction part. The repository is composed of two main parts: the first one is dedicated to store and mange data in the form of *Compartments*, the second one is about the *Interfaces* in order to display patterns and models and to manage interactions between users and the repository. The following sections detail the repository package diagram.

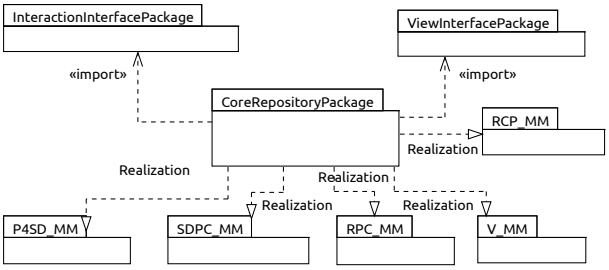


Figure 6.3: Overview of the Repository Specification

6.2 Core Repository Package

A repository exposes a set of *Interfaces* and is designed with a set of *Compartments*. In this section we focus on the description of the compartments (see Figure 6.4). The interfaces are studied in the next sections.

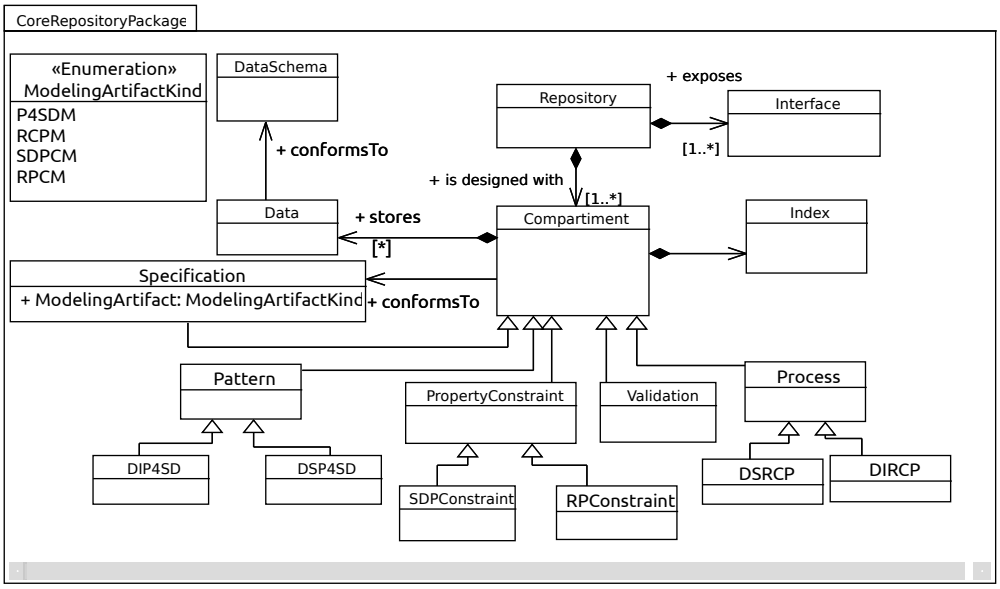


Figure 6.4: Core Repository Package

On the other hand, the *Compartment* deals with the data and the data structure for processing the data. So, the *Compartment* is composed of three main elements:

- *Data.* corresponds to the concrete data stored into the repository.
- *DataSchema.* is the structure that specifies how the data is organized.
- *Index.* is a data structure that eases the artifact access into the repository.

In the context of TERESA, we have identified five kinds of *Compartments*:

- *Specification.* is used to store the specification of the TERESA modeling artifacts, mainly process, pattern, property, constraint and validation.
- *Process.* to serialize the instances of processes at both domain independent and domain specific (DIRCP, DSRCP),
- *Pattern.* to serialize the instances of the patterns at both domain independent and domain specific (DIP4SD, DSP4SD)
- *PropertyConstraint.* to serialize the model library of the resource, S&D properties and constraints (SDPC, RPC),
- *Validation.* to serialize of validation artifacts.

6.3 Interaction Interface Package

The *InteractionInterface* Package depicts the *API* of the repository. It describes a set of interactions between the repository and the "external world". The main goal of the repository is to share expertise interacting with existing engineering process in order to build trusted applications for resource constrained embedded systems. That is, we propose in the following an *API* as a refinement of the repository system architecture presented above (see Figure 6.1).

As shown in Figure 6.5, the *API* exposes the following interfaces:

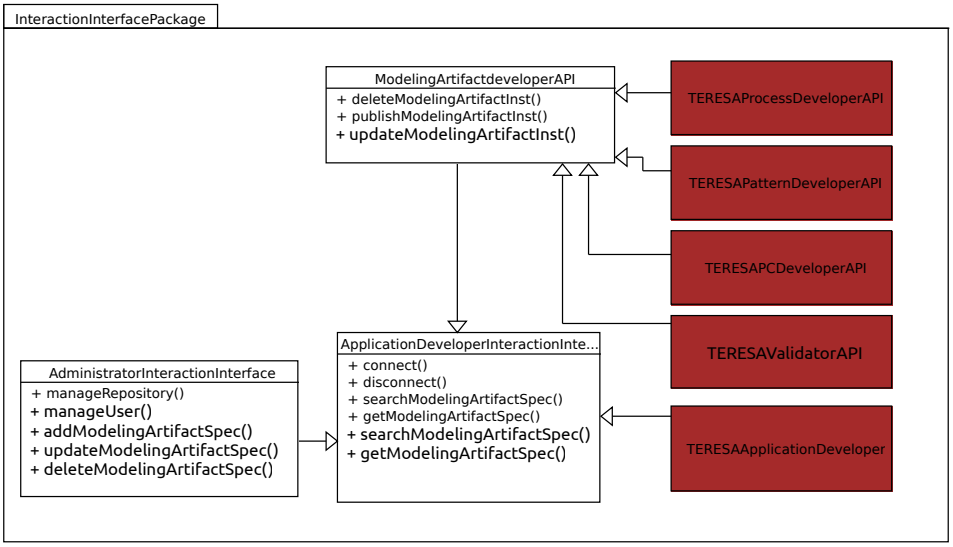


Figure 6.5: Interaction Interface Package

- *Classified by activities:*
  - *ModelingArtifactDeveloperAPI.* to manage instances of the modeling artifacts,
  - *Application.* offers a set of operations mainly connection/disconnection to the repository, search/selection of specifications and instances of the modeling artifacts,
- *Classified by Roles:*
  - *TERESAProcessDeveloperAPI.* specializes the ModelelingArtifactDeveloperAPI for a process model,
  - *TERESAPatternDeveloperAPI.* specializes the ModelelingArtifactDeveloperAPI for a pattern,
  - *TERESAPCDeveloperAPI.* specializes the ModelelingArtifactDeveloperAPI for a property/constraint library,
  - *TERESAValidatorAPI.* specializes the ModelelingArtifactDeveloperAPI for validation artifact,
  - *TERESAApplicationDeveloperAPI.* specializes the ApplicationDeveloperInteractionInterfaceAPI for a TERESA system development process,
  - *AdministratorAPI.* is the management of the repository,

6.4 View Interface Package

The *VisualizationInterface* allows to display the content of the repository, as a *Human Machine Interface*, including the specification and the instantiation of the modeling artifacts presented in the previous sections. The proposed views should take into account user roles (e.g. Administrator, Developer, MDE expert, S&D expert, etc.). The content of the package is described in depth in the Deliverable 4.5 and 4.6.

7 Feedback Template (Completeness of the modeling framework)

In this part we propose a set of measures to evaluate the completeness of the proposed metamodels. In other words, we study *the correctness and completeness of the pattern, repository and properties* specification language. As we shall see, the evaluation template concerns the concepts used to capture those of the domain. The template is proposed to our industrial partners and the results will be analyzed in the next deliverable (D4.3 and D4.4).

We propose the following:

- *Concept.* concept name,
- *Definition and Usage.* concept definition from the domain independent view point,
- *Domain relevance.* the link between the concept and the domain concepts,
- *Domain acceptance.* a feedback as an evaluation about the completeness of a concept as a generic representation of a set of domain concept instances,
- *Recommendations.* a set of recommendations to enhance the concept in terms of its representation of the related domain concepts.

The following tables depict the main concepts used in the specification of TERESA modeling artifacts.

Concept	**Definition & Usage**	**Domain Relevance**	**Acceptance (%) -1 if not**	**Recommendations**
Pattern				
Interface				
Property				
Constraint				
Internal Structure				

Table 7.1: Pattern Feedback

Concept	Definition & Usage	Domain Relevance	Acceptance (%) -1 if not	Recommendations
System				
Entity				
Property				
Constraint				
S&D Property				
Trust Property				

Table 7.2: S&D MetaModel Feedback

Concept	Definition & Usage	Domain Relevance	Acceptance (%) -1 if not	Recommendations
System				
Property				
Constraint				
Resource Property				

Table 7.3: Resource MetaModel Feedback

Concept	Definition & Usage	Domain Relevance	Acceptance (%) -1 if not	Recommendations
Repository				
Compartment				
Data				
DataSchema				
Index				

Table 7.4: Repository MetaModel Feedback

8 Conclusion

Security and dependability are not building blocks added to an application at the end of the life cycle. It is necessary to take into account these concerns from the requirement to the integration phases. In this part, we have proposed an approach for S&D pattern development and packaging that follows the MDE paradigm. Our approach is based on meta-modeling techniques that allow to specify S&D patterns at different levels of abstraction. An S&D pattern at domain independent level allows the application developer to identify S&D requirements and select a respective abstract solution without specific knowledge on how the solution is designed and implemented. Thus a DIPM pattern can easily be integrated into the overall abstract system specification. The benefit of this structure is to offer a common language for different domain applications. So far, this common language encompasses four industrial sectors, namely, home control, industry control, automotive, and metering [52]. Following the MDE process, the domain independent model of patterns is then refined towards a domain specific level, taking into account domain artifacts, concrete elements such as mechanisms to use, devices that are available, etc. Consequently, an S&D pattern at domain specific level contains the respective information. On the other hand, we proposed an operational architecture of the repository, based on a well known architectural style: Multitiered Architectures as an alternative client-server organizations.

As a side remark, note that the goal is to obtain an even higher level abstraction to represent S&D patterns to capture several facets of security and dependability in the different domain of embedded system applications, not an implementation of a specific solution. The key is then to show that the major sectors of trusted embedded systems applications dealing with security and dependability are covered by our approach. This result leads to some anticipated issues about general techniques to model S&D patterns. This result is of particular interest to build a trusted computing engineering discipline that is suited to a number of sectors in embedded systems.

An important point concerns the completeness of the internal pattern representation which will be specified in the future. Thus, if a domain specific artifact is missing, the repository supports the extension mechanism which allows the user to add the needed artifact. The next steps of this work consist in integrating all the presented results in a more global process with the pattern life cycle (i.e., create, update, store patterns) and the integration of a pattern in an application (see $WP3$). All patterns are stored in a repository. Thanks to it, it is possible to find a pattern regarding to S&D criteria. Patterns and the application are formally validated regarding to S&D properties. At last, guidelines will be provided during the pattern development and the application development (i.e., help to select a suitable pattern with respect to the constraint and the specificity of the target application and/or platform).

As presented in [53], pattern validation follows these two abstraction levels, i.e. we validate a DIPM pattern and possible DSPM instantiations independently. However, the additional final validation step proves that the latter is indeed a refinement of the former which in turn proves that the overall application system indeed satisfies the S&D requirements initially specified by the application developer. This process may significantly reduce the cost of engineering the system, since it enables to address S&D issues early in the system development process while at the same time relieving the developer from the technical details. Yet an important task remains to be performed when integrating an S&D pattern into an application: It has to be ensured that the assumptions used for proving the correctness of the DSPM pattern are indeed satisfied by the particular environment of the application. While so far these assumptions are specified in terms of formal security properties, future steps consist in deriving them into environment constraints through the external model libraries discussed in the deliverable D3.2 (i.e., S&D and Resource properties).

Note, however, that the ultimate goal is to propose guidelines based on an MDE approach for a system development process to build trusted applications for resource constrained embedded systems. That is, we will discuss our vision on how the repository of modeling artifact will be used in an MDE approach following the TERESA Applications Framework presented in Chapter 4.

With regard to the implementation issues, we target the development of a tool suite to support the development and the management of the modeling artifacts discussed in this work. In the context of TERESA project, two versions will be provided as

1) a framework to specify TERESA modeling artifacts using Eclipse technologies (EMF, GMF, CDO, Papyrus), and

2) a repository of integrated models (patterns, S&D models, ..) and a repository access tool to allow application designers to capitalize on the MDE even if they are not experts in modeling. We plan to use CDO technology (http://www.eclipse.org/cdo/).

This first version will be concluded by the deliverable D4.3 and the second one by the deliverable D4.4. The second version will complete the first one with regard to the completeness of the modeling artifacts specifications and instantiations on the one hand and with regard to the tooling on the other hand. Further, the second one will be proposed in the context of MDE development approach for TERESA applications.

9 Bibliography

[1] R. Zurawski, "Embedded systems." CRC Press Inc, 2005.

[2] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 461–491, 2004.

[3] D. Schmidt, "Model-driven engineering," *in IEEE computer*, vol. 39, no. 2, pp. 41–47, 2006.

[4] J. Bézivin, "Towards a precise definition of the omg/mda framework," in *Proceedings of ASE'01*. IEEE Computer Society Press, 2001, pp. 273–280.

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

[6] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language*, ser. Center for Environmental Structure Series. New York, NY: Oxford University Press, 1977, vol. 2.

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

[8] D. Riehle and H. Züllighoven, "Understanding and using patterns in software development," *TAPOS*, vol. 2, no. 1, pp. 3–13, 1996.

[9] P. A. Bernstein and U. Dayal, "An overview of repository technology," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. Morgan Kaufmann Publishers Inc., 1994, pp. 705–713.

[10] J. Jürjens, "Umlsec: Extending uml for secure systems development," in *Proceedings of the 5th International Conference on The Unified Modeling Language*, ser. UML '02. London, UK: Springer-Verlag, 2002, pp. 412–425.

[11] T. Lodderstedt, D. Basin, and J. Doser, "Secureuml: A uml-based modeling language for model-driven security," in *Proceedings of the 5th International Conference on The Unified Modeling Language*, ser. UML '02. London, UK: Springer-Verlag, 2002, pp. 426–441.

[12] S. Bernardi, J. Merseguer, and D. Petriu, "Adding dependability analysis capabilities to the marte profile," in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, ser. MoDELS '08. Springer-Verlag, 2008.

[13] OMG, "Omg. a uml profile for marte: Modeling and analysis of real-time embedded systems,beta 2." http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf, June 2008.

[14] B. Hamid, A. Radermacher, A. Lanusse, C. Jouvray, S. Gérard, and F. Terrier, "Designing fault-tolerant component based applications with a model driven approach." in *The IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems(SEUS)*, ser. Lecture Notes in Computer Science. Springer, 2008, pp. 9–20.

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

[16] V. D. Giacomo, M. Felici, V. Meduri, D. Presenza, C. Riccucci, and A. Tedeschi, "Using security and dependability patterns for reaction processes," in *Proceedings of the 2008 19th International Conference on Database and Expert Systems Application*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 315–319. [Online]. Available: http://portal.acm.org/citation.cfm?id=1440465.1441170

[17] N. Yoshioka, H. Washizaki, and K. Maruyama, "A survey of security patterns," *Progress in Informatics*, no. 5, pp. 35–47, 2008.

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

[19] M. Tichy, D. Schilling, and H. Giese, "Design of self-managing dependable systems with uml and fault tolerance patterns," in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, ser. WOSS '04. New York, NY, USA: ACM, 2004, pp. 105–109. [Online]. Available: http://doi.acm.org/10.1145/1075405.1075426

[20] J.-M. J. A. L. Guennec, G.n Sunyé, "Precise modeling of design patterns," in *In Proceedings of UML'00*. Springer-Verlag, 2000.

[21] S. G. D-K Kim, R. France and E. Song, "A uml-based meta-modeling language to specify design patterns," in *Patterns," Proc. Workshop Software Model Eng. (WiSME) with Unified Modeling Languages*, 2004.

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

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

[24] 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)*. IEEE Computer Society, September 2008, pp. 287–291.

[25] P. Sriplakich, X. Blanc, and M. Gervais, "Supporting transparent model update in distributed case tool integration," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06. New York, NY, USA: ACM, 2006, pp. 1759–1766.

[26] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger, "Towards a semantic infrastructure supporting model-based tool integration," in *Proceedings of the 2006 international workshop on Global integrated model management*, ser. GaMMa '06. New York, NY, USA: ACM, 2006, pp. 43–46.

[27] C. Mayr, U. Zdun, and S. Dustdar, "Reusable architectural decision model for model and metadata repositories," in *FMCO*, 2008, pp. 1–20.

[28] R. B. France, J. M. Bieman, and B. H. C. Cheng, "Repository for model driven development (remodd)." in *MoDELS Workshops'06*, 2006, pp. 311–317.

[29] C. M. Lloyd, J. R. Lawson, P. J. Hunter, and P. F. Nielsen, "The cellml model repository," *Bioinformatics/computer Applications in The Biosciences*, vol. 24, pp. 2122–2123, 2008.

[30] "ebxml: Oasis registry services specification v2.5," 2003.

[31] "freebxml: Oasis ebxml registry reference implementation project," http://ebxmlrr.sourceforge.net/, 2007.

[32] N. Milanovic, R.-D. Kutsche, T. Baum, M. Cartsburg, H. Elmasgünes, M. Pohl, and J. Widiker, "Model&metamodel, metadata and document repository for software and data integration," in *MoDELS*, 2008, pp. 416–430.

[33] C. Sapia, M. Blaschka, and G. Höfling, "Grammi: Using a standard repository management system to build a generic graphical modeling tool," in *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8 - Volume 8*, ser. HICSS '00. IEEE Computer Society, 2000, pp. 8058–.

[34] T. Holmes, U. Zdun, and S. Dustdar, "Morse: A model-aware service environment," 2009.

[35] "Netbeans community: Metadata repository (mdr)," http://mdr.netbeans.org/, 2009.

[36] "Eclipse: Eclipse cdo (ccopyright2011)," http://wiki.eclipse.org/CDO, 2011.

[37] "Standard: Iso/iec 13335," 2004.

[38] R. Birukou, E. Blanzieri, P. Giorgini, and M. Weiss, "Facilitating pattern repository access with the implicit culture framework," in *Proceedings of "EuroPLoP 2007"*, 2007.

[39] S. R. S. R. Kodituwakku and P. Bertok, "Pattern categories: A mathematical approach for organizing design patterns," in *Pattern Languages of Programs 2002. Revised papers from the Third Asia-Pacific Conference on Pattern Languages of Programs, (KoalaPLoP 2002)*, ser. CRPIT, J. Noble, Ed., vol. 13. Melbourne, Australia: ACS, 2003, p. 63.

[40] R. Girardi and A. N. Lindoso, "An ontology-based knowledge base for the representation and reuse of software patterns," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1, pp. 1–6, 2006.

[41] R. Johnson and M. Hfiz, "Security patterns and their classification schemes," Technical Report, 2006.

[42] "Iso/iec 13335-1. information technology - guidelines for the management of it security - part 1: Concepts and models for it security," Technical Report, 1996.

[43] "Iso 7498-2. . information processing system - open systems interconnections - basic reference model - part 2: Security architecture," Technical Report, 1989.

[44] J. A. Zachman, "A framework for information systems architecture," *IBM Syst. J.*, vol. 26, September 1987.

[45] D. Trowbridge, W. Cunningham, M. Evans, L. Brader, and P. Slater, "Describing the enterprise architectural space," *MSDN*, 2004.

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

[47] "Serenity, system engineering for security & dependability," www.serenity-project.org, 2006.

[48] I. Crnkovic, M. R. V. Chaudron, and S. Larsson, "Component-based development process and component lifecycle," in *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*. IEEE Computer Society, 2006, p. 44.

[49] A. W. Brown and K. C. Wallnau, "The current state of CBSE," *IEEE Software*, vol. 15, no. 5, pp. 37–46, 1998.

[50] B. Hamid, S.Gurgens, C. Jouvray, and N. Desnos, "Enforcing S&D Pattern Design in RCES with Modeling and Formal Approaches," in *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), wellington, 16/10/2011-21/10/2011*, J. Whittle, Ed., vol. 6981. Springer, octobre 2011, pp. 319–333.

[51] G. Tel, *Introduction to distributed algorithms*, 2nd ed. Cambridge University Press, 2000.

[52] 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*. ACM, 2010.

[53] C. J. B. Hamid, S. Gürgens and N. Desnos, "Enforcing s&d pattern design in rces with modeling and formal approaches," in *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), wellington, 16/10/2011-21/10/2011*, J. Whittle, Ed., vol. 6981. Springer, octobre 2011, pp. 319–333.

Terminology and abbreviation

- *EFP.* Extra Functional Properties
- *NFP.* Non Functional Properties
- *RCES.* Resource Constrained Embedded Systems
- *S&D.* Security and Dependability
- *PFS.* Pattern Fundamental Structure (meta-model)
- *DIPM.* Domain Independent Pattern Model
- *DSPM.* Domain Specific Pattern Model
- *RCP_MM.* Repository Centric Process Metamodel
- *P4SD_MM.* Pattern for Security & Dependability MetaModel
- *SDPC_MM.* Security&Dependability Property and Constraint Metamodel
- *RPC_MM.* Resource Property and Constraint Metamodel
- *V_MM.* Validation Metamodel
- *SDPCM.* Security Dependability Property and Constraint Model
- *RPCM.* Resource Property and Constraint Model
- *P4SDM.* Pattern for Security and Dependability Model
- *RCPM.* Repository-Centric Process Model
- *MARM.* Modeling Artifacts Repository Model