

Chapter 2

Background

Abstract This chapter introduces fundamental basic concepts, notations, and terminologies for requirements engineering as well as software architecture that the proposed QuaDRA framework and its extension rely on. In addition, the UML profiles used throughout this book, life-cycle expressions used for describing the relation between the requirements in different methods of the QuaDRA framework as well as variability modeling used in the extension of QuaDRA are introduced. Finally, the description of the real-life case study smart grid used to illustrate the application of QuaDRA is presented.

2.1 Requirements Engineering

Understanding and describing the problem that the software has to solve in a precise way is the first thing to do when developing a software [68]. Requirements engineering (RE) as a sub-discipline of software engineering consists of requirements development and requirements management [240]. It covers a structured set of activities in discovering, documenting, and maintaining a set of requirements for a computer-based system [223]. The requirements of a software system consist of functional requirements and quality requirements (also known as non-functional requirements or NFRs).

We introduce definitions and descriptions of the quality requirements security and performance in Section 2.1.1. Problem frames as the basis for our problem-oriented requirements engineering is described in Section 2.1.2.

2.1.1 Quality Requirements

For the success of software projects, quality requirements are as critical as functional requirements as the software without considering the necessary quality properties may be too slow, unusable, or insecure [73, 40]. However, they are often neglected in practice or poorly described in requirement documents [251].

There is no consensus in the software engineering community regarding the definition of quality requirements (also known as non-functional requirements [184]). Often, they are referred to as “-ilities” such as reliability or “-ities” such as security. Nevertheless, there are quality requirements that end neither with “-ility” or “-ity” such as performance [73]. Chung & Sampaio do Prado Leite [73] introduce the notion of *satisficing* when talking about achieving quality requirements¹. It refers to the nature of quality requirements that cannot be addressed absolutely but in a “good enough sense”. The notion of satisficing reflects the sense of good enough.

Eliciting, modeling, and managing quality requirements is one of the important challenges in requirements engineering. Quality requirements are considered as the most expensive and complex ones to deal with [72, 93]. They tend to interfere, conflict, or contradict with each other. Achieving a particular type of quality requirements might hurt the achievement of other types of quality requirements [91, 72]. This negative impact reveals the need for making trade-offs between conflicting quality requirements to fulfill the overall software goal/purpose. Performance and security requirements represent such conflicting requirements. In the following, we describe these requirements and give definitions for them.

Security Requirements

Security is often an afterthought in designing software. Security requirements are not considered explicitly and therefore not integrated in the software architecture [211]. Hence, there is a need for explicitly and systematically addressing security as early as possible in the software development life cycle.

In the international standard ISO/IEC 25010 (SQuaRE) [130] which is the successor of ISO/IEC 9126-1, security is defined as one of the characteristics for product quality properties. It is divided into the five subcharacteristics *confidentiality*, *integrity*, *non-repudiation*, *accountability*, and *authenticity*. In this book, we fo-

¹ Quality requirements or non-functional requirements are treated as *softgoals* in the NFR Framework introduced by Chung & Sampaio do Prado [73].

cus on confidentiality, integrity, and authenticity which are defined in the standard ISO/IEC 25010 as follows:

Confidentiality is defined as the “*degree to which a product or system ensures that data are accessible only to those authorized to have access.*”

Integrity is defined as the “*degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.*”

Authenticity is defined as the “*degree to which the identity of a subject or resource can be proved to be the one claimed.*”

Performance Requirements

According to Bass et al. [45], quality requirements are not completely dependent on design or on implementation. Performance has partially architectural dependencies and partially non-architectural dependencies. For example, it depends on the amount of communication between components, on the allocation of the functionalities to each component, on the usage of shared resources, which are all architectural dependencies. On the other hand, performance depends also on the choice of algorithms to implement the functionalities and on how efficient the implementation of such algorithms is, which are both non-architectural dependencies.

Performance depends upon the load to the system and the resources available to process the load [47]. Therefore, for performance assessment, performance requirements and domain knowledge are used. Performance requirements describe the response time characteristics of the system-to-be. Domain knowledge represents assumptions on the system-to-be such as the workload and the constraints on resource usage.

In the international standard ISO/IEC 25010 (SQuaRE) [130] performance efficiency (performance hereafter) is defined as one of the characteristics for product quality properties. It is composed of the subcharacteristics *time behavior*, *resource utilization*, and *capacity*. Time behavior including *response time* and *throughput* is defined as “*the degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.*” In this book, we focus on response time.

2.1.2 Problem Frames

Problem frames are a means to describe software development problems. They were proposed by Michael Jackson [133], who describes them as follows:

“A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement.”

A problem frame is described by a *frame diagram*, which basically consists of *domains*, *interfaces* between them, and a *requirement*. Domains describe entities in the environment. Jackson distinguishes the domain types *biddable domains* that are usually people, *causal domains* that comply with some physical laws, and *lexical domains* that are data representations.

In *problem diagrams*, *interfaces* connect domains, and they contain *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by the name of that domain and “!”. In Fig. 2.1 the notation $MD!\{data\}$ (between *MeterData* and *SubmitMD*) means that the phenomenon *data* is controlled by the domain *MeterData* and observed by the machine *SubmitMD*.

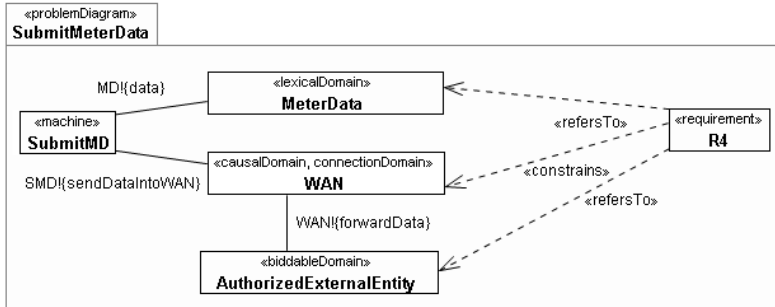


Fig. 2.1: Problem Diagram for submitting meter data to external entities

When we state a requirement, we want to change something in the world with the software to be developed. Therefore, each requirement constrains at least one domain. Such a constrained domain is the core of any problem description, because it has to be controlled according to the requirements. A requirement may refer to several other domains. The requirement *R4* in Fig. 2.1 constrains the domain *WAN*.

It refers to the domains *MeterData* and *AuthorizedExternalEntity*². The task is to construct a *machine* (i.e., software) that improves the behavior of the environment (in which it is integrated) in accordance with the requirements.

Requirements analysis with problem frames proceeds as follows: first the environment in which the machine will operate is represented by a *context diagram*. A context diagram consists of machines, domains and interfaces. Then, the problem is decomposed into subproblems, which are represented by *problem diagrams*. A problem diagram consists of a submachine of the machine given in the context diagram, the relevant domains, the interfaces between these domains, and a requirement. Figures 2.1 shows a problem diagram in UML notation.

We use problem frames in this book as a basis for requirements engineering. The use of problem frames has the following benefits:

- It takes the surrounding environment of the software into consideration [133].
- It allows decomposing the overall software problem into simpler subproblems, thus reducing the complexity of the problem. The reason is that the complexity of each single problem diagram is independent of the size of the system. Moreover, the number of problem diagrams increase linearly even for large systems [133].
- It enables us to check for inconsistencies in different parts of the model due to its semi-formal structure [115].
- It makes it possible to annotate problem diagrams with quality requirements and additional information such as domain knowledge, particularly when considering quality requirements [17].
- It allows us to obtain detailed information from the structure of problem diagrams. Such information enables us to perform interaction analysis and optimization, whereas other requirements engineering approaches such as scenario-based approaches and use cases do not contain detailed information for such analyses [6].
- It not only helps to understand the software problem, but also supports in solving that problem. The structure of the problem diagrams and the properties of the involved domains facilitate the development of corresponding architecture components that reflect the problem characteristics. Hence, software architectures can be derived from requirement models expressed as problem diagrams [69].

² This example is taken from the case study smart grid which we introduce later on in this chapter.

2.2 Software Architecture Concepts

In this section, we provide an overview of the main concepts and definitions of software architecture and architecture terminology (Section 2.2.1 and Section 2.2.2), architectural patterns and quality-specific patterns (Section 2.2.3 and Section 2.2.4), Viewpoint models (Section 2.2.5), architecture description languages (Section 2.2.6), and architecture evaluation (Section 2.2.7).

2.2.1 Definition of Software Architecture

The need for having a software architecture (SA) discipline has been recognized in the sixties, seventies, and eighties with growing complexity of software systems [192, 80]. But the formal work in the area of software architecture began in the 1990s [187, 80].

Hofmeister et al. [126] describes software architecture as a blueprint of a system bridging the system requirements and implementation. It does not provide a comprehensive refinement of the system, but an abstraction of the system to manage complexity.

It is generally acknowledged that there is no common agreement on the definition of software architecture [104, 222, 35]. More than 150 definitions of the software architecture from the literature and from practitioners are collected by the Software Engineering Institute (SEI) at Carnegie-Mellon University³ [78]. One of the most used definitions for software architecture is provided by Bass et al. [44]:

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationship among them.”

Rozanski and Woods [204] describe two key parts of this definition, namely *system structures* and *externally visible properties* more detailed. Two types of system structure exist for software architecture:

- The *static structure* forms the design-time organization of the software. It includes the elements of the software and their relationships.
- The *dynamic structure* describes the run-time elements of the software and their interactions.

Externally visible properties of the system are manifested in two different ways, namely *externally visible behavior* and *quality properties*:

³ <http://www.sei.cmu.edu/architecture/>

- *Externally visible behavior* specifies what the system does. It defines the functional interactions between the system and its environment.
- *Quality properties* specify how the system does it. They are non-functional properties of the system that are externally visible such as performance and security.

To a software problem, there might be more than one possible solution, known as *candidate architectures*. According to Rozanski and Woods [204]:

“A candidate architecture for a system is a particular arrangement of static and dynamic structures that has the potential to exhibit the system’s required externally visible and quality properties.”

In this book, we explore the solution space to identify candidate architectures with respect to quality requirements. The candidate architectures or alternative architectures stem from various architectural patterns having different impact on quality requirements or quality strategies help satisficing quality requirements.

Rozanski and Woods [204] define an *Architecture Description (AD)* as

“a set of products that documents an architecture in a way its stakeholders can understand and demonstrates that the architecture has met their concerns.”

Products in this context is referred to *architectural models, scope definition, constraints, and principles*.

2.2.2 Difference between Architecture and Design

In the previous section, we gave an overview of existing definitions of software architecture. In this section, we discuss how architecture is different from design. This difference matters for this book as we have been developing a method including requirements analysis and software architecture. Therefore, we need to know for our method

- where are the boundaries to design
- which decisions are “architectural” and which are “non-architectural”

Perry & Wolf [192] clearly distinguish in their definitions between architecture and design. They define *architecture* as follows:

“Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework to satisfy the requirements and serve as a basis for the design.”

Design is defined by Perry & Wolf [192] as:

“Design is concerned with the modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements.”

Hence, the design phase consists of two levels: high-level structure and low-level structure of the software⁴. Software architecture (or architectural design) is concerned with the design and implementation of the high-level structure of the software [156, 174], whereas detailed design (or non-architectural design) is concerned with the design and implementation of the low-level structure of the software. From the perspective of the architecture, detailed design is part of the realisation [194, pp 116,117].

According to Hofmeister et al. [126], software architecture is placed after requirements and domain analysis and before detailed design, coding, integration, and testing. This provides an approximate order of executing the tasks, it however does not mean that the analysis phase must be finished before the design phase begins. Overlaps and iterations between tasks exist. As described in Section 2.1, in the requirements engineering phase the requirements of the system are elicited, analyzed, and managed. It results in requirements that provide the key input to the software architecture design. Requirements may need to be changed according to the software architecture tasks. The software architecture guides the implementation tasks, including detailed design, coding, integration, and testing.

According to Clements et al. [78], decisions that are concerned with satisfying functional and quality requirements can be seen as “architectural decisions”. Decisions that result in element properties that are not visible are “design decisions” and not “architectural decisions”. Typical examples for design decisions are the choice of data structures and algorithms.

Rozanski & Woods [204] state that “a concern, problem, or system element is *architecturally significant* if it has a wide impact on the structure of the system or on its important quality properties such as performance, scalability, security, reliability, or evolvability.” Whether something is architecturally significant is a subjective decision which is driven by the judgement of the architect, its skill and expertise, and the circumstances of each individual system [204].

⁴ Also called coarse-grained design and fine-grained design

2.2.3 Architectural Patterns

The software architecture is to a large extent influenced by its quality requirements [56, 44, 126]. It has to fulfill the defined functional requirements as well as the desired quality requirements [56, 44, 126]. Developing such a software architecture that achieves its quality requirements is one of the most demanding tasks [39]. Architectural patterns in general contribute to the satisfaction of desired quality requirements.

Architectural styles have been investigated for many years in different areas of computer science [175]. According to Bass et al. [44], an architectural style is “*a specialization of element and relation types, together with a set of constraints on how they can be used.*”

We use the term *architectural pattern* as a synonym for *architectural style* as suggested by Bass et al. [44] and Hofmeister et al. [126]. The idea of software patterns stems originally from Christopher Alexander, a professor of building architecture, who published a series of books about patterns, pattern language, and catalog of patterns in building architecture [98].

Architectural patterns [62, 215, 33] describe the high-level structure and behavior of software systems. They represent well-proven generic solutions to problems that arise recurrently at the architectural design level. An architectural pattern has three essential parts: a *problem* definition, a description of the problem’s *context*, and a corresponding *solution* to the problem [109, 33]. Besides satisfying functional requirements, architectural patterns aim at satisfying several quality requirements. Applying an architectural pattern results in *consequences* regarding the fulfillment of quality requirements. Positive consequences are documented as *benefits* whereas the negative consequences are labeled as *liabilities*. Patterns may have different variants that extend their functionality and/or come with different benefits and liabilities.

In the literature, there is no consensus on the classification of patterns, regarding their philosophy, the way of describing patterns, and the granularity of architectural patterns. For example, *interpreter* is a classical design pattern introduced by Gamma et al. [103]. It, however, can be treated like an architectural pattern, since it is a central and externally visible component [33]. Hence, there is no single catalog of architectural patterns to be used by software architects. We decided to select the patterns from Buschmann et al. [62], which are among the best set of the existing architectural pattern collections.

2.2.4 *Quality-specific Mechanisms and Tactics*

Architecture tactics or *Tactics* are established and proven strategies that can be used to help fulfill a particular quality requirement [44, 204]. From an architectural view a tactic may affect the overall architecture only slightly or, in some cases, an implemented tactic may not be visible in the architecture at all. For example, a client/server architecture could be augmented by a “Heartbeat” tactic to address availability [44]. It enables the server to know which clients are still alive; however, this modification is neither an architectural pattern nor is its implementation guaranteed to modify existing architectural views.

Introduce concurrency is an example for a performance tactic. It proposes to process the requests in parallel by processing different event streams on different threads for processing different sets of activities. This tactic describes a coarse-grained solution to help achieve response time requirements. Such a tactic can be mapped to more fine-grained mechanisms such as *master-worker*. The same holds for security tactics. For example, *maintain data confidentiality* can be achieved by the fine-grained mechanism *encryption*. To this end, we make use of such fine-grained mechanisms instead of tactics in the QuaDRA framework. These mechanisms are briefly described in the following.

2.2.4.1 Security patterns and mechanisms

Encryption is an important means to achieve confidentiality. A plaintext is encrypted using a secret key and decrypted either using the same key (symmetric encryption) or a different key (asymmetric encryption). One advantage of symmetric encryption is that it is faster than asymmetric encryption. The disadvantage is that both communication parties must know the same key, which has to be distributed securely or negotiated. In asymmetric encryption, there is no key distribution problem, but a trusted third party is needed that issues the key pairs.

RBAC Verifying permission is a frequently recurring problem in security relevant systems. Hence, it has been treated in several access control patterns for the design phase [248, 211]. Access control patterns define security constraints regarding access to resources. Role-Based Access Control (RBAC) provides access to resources based on functions of people in an environment, known as roles, and the kind of permission they have, known as rights.

Digital signature is an important means for achieving integrity and authenticity of data. Using the digital signature, the Sender produces a signature using the private key and the data. The receiver ensures that the data is created by the known sender using the public key.

MAC is an important means for achieving integrity and authenticity of data. Message Authentication Code (MAC) uses a secret key and the data to generate a MAC. The verifier uses the same secret key to detect changes to the data.

2.2.4.2 Performance patterns and mechanisms

Load Balancer is a mechanism that is used to distribute computational load evenly over two or more hardware components. The load balancing pattern consists of a component called Load Balancer, and multiple hardware components that implement the same functionality. The load balancer can be realized as a hardware or a software component [96].

Master Worker makes it possible to serve requests in parallel, similarly to load balancing. In contrast to load balancing that uses hardware components, the master-worker pattern provides a software solution. It consists of a software component called Master and two or more other software components, called Worker. The task of the master is to divide the request into parallel tasks and to forward them to the workers, which manage the smaller tasks [96].

First Things First ensures that the most important tasks will be processed if not every task can be processed. The problem that this pattern aims at solving is that a temporary overload of inbound requests is expected. This situation may overwhelm the processing capacity of a specific resource. The First Things First pattern uses the strategy of prioritizing tasks and performing the important tasks with high priority first. In the case of a permanent overload, applying this pattern would cause the starving of low-priority tasks [220].

Flex Time reduces the load of the system by spreading it temporally. That is, it moves the load to a different period of time where the inbound requests do not exceed the processing capacity of the resource. The problem that this pattern solves is that an overload of the system is expected. The inbound requests exceed the processing capacity of a specific resource. Flex Time is only applicable when some tasks can be performed at a different period of time [220].

2.2.5 Viewpoint Models

As the architecture of a software system is a complex construct, it cannot be described in one single model. There are several representations of one or more structures and abstraction levels for software architecture, each of which describes a separate concern of the architecture [204]. However, ISO/IEC/IEEE 42010 [131],

which replaced IEEE Recommended Practice for Architectural Description of Software Intensive Systems [129], provides no commitment what structures (commonly called *views* [204]) are required for software architecture. This ambiguity in defining a software architecture and its constituents makes the understanding and communication between the involved groups of stakeholders inefficient and error-prone [222].

Common architectural view models summarized from the literature [174] are *Kruchten's 4+1 view model* [156], *SEI viewpoint model* [80], *Siemens 4 view model* [126], and *Raozanski & Woods view model* [204, 245]. Table 2.1 shows the views of each view model classified into *requirement view*, *design view*, and *realization view*. These view models can be extended with further views if required, for example for representing quality requirements.

Table 2.1: Overview of common view models

View model	Requirement view	Design view	Realization view
4+1	use case view	logical view	development view
		process view	physical view
Siemens	-	conceptual view	code view
		module view	execution view
SEI	-	functional view	code view
		concurrency view	development view
			physical view
Raozanski & Woods	context view	functional view	development view
		information view	deployment view
		concurrency view	operational view

2.2.6 Architecture Description Languages vs UML

As the architecture description of a software system is essential for communication among stakeholders and for being a basis for later phases of software development, it should be unambiguous. Informal box and arrow diagrams are used by most of the architects, which are highly ambiguous [187]. Hence, there have been some attempts in the software engineering research community to specify design specific languages, called *Architecture Description Languages (ADLs)* [80]. ADLs are a means for representing the architecture of a software system in a formal way [187]. Some prominent ADLs are *Rapide* [170], *Darwin* [171], *UniCon* [214], etc. How-

ever, the ADLs did not become very popular among the practitioners except for a few in a specific domain [187].

In contrast, the *Unified Modeling Language (UML)* [235] is being widely adopted to describe architectural constructs. UML is originally not constructed to support architecture descriptions, since it does not support architectural concepts (for example layers) and the successive refinement of design from the architectural abstractions [80]. UML lacks formal semantics and is therefore a source of ambiguity and inconsistency [187]. However, UML has received much attention from practicing architects as its facilities can be tailored to describe architectures. The following reasons might contribute to the popularity of UML [187]:

- Providing a graphical representation of the software architecture. Most of the ADLs are textual and less appealing to the software architects.
- Supporting multiple views which are important to the software architecture.
- Many tools are available for UML. ADLs lack supporting tools.
- UML is a general-purpose modeling language in contrast to most of the ADLs that are constructed for domain-specific applications.

2.2.7 Architecture Evaluation

Finding errors during requirements analysis or early design and correcting them is less costly than finding the same errors during testing. An architecture represents the results of early design decisions. Architecture evaluation helps finding those errors early to avoid failure. An architecture evaluation determines how suitable the architecture is with respect to a set of goals and how problematic with respect to another set of goals. The results of an architecture evaluation are information and insights about the architecture [80]. Architecture Trade-off Analysis Method (ATAM) is one of the well-known methods for evaluating architectures [204]. ATAM consists of nine steps categorized in four groups *presentation* (Steps 1 - 3), *investigation and analysis* (Steps 4 - 6), *testing* (Steps 7 and 8), and *reporting* (Step 9). The steps are summarized as follows:

1. **Present the ATAM:** ATAM is described to the assembled participants by the evaluation leader.
2. **Present the business drivers:** The business goals motivating the development effort and the primary architectural drivers (for example high security) are described by the project manager.
3. **Present the architecture:** The architecture is described by the architect focusing on how business drivers are addressed.

4. **Identify the architectural approaches:** The architect identifies architectural approaches.
5. **Generate the quality attribute utility tree:** Quality attributes comprising system utility (performance, security, etc.) are elicited, specified down to the level of scenarios, and prioritized.
6. **Analyze the architectural approaches:** Architectural approaches addressing scenarios identified in the previous step are elicited and analyzed. In this step, architectural risks, nonrisks, sensitivity points, and trade-off points⁵ are identified.
7. **Brainstorm and prioritize scenarios:** Scenarios are prioritized involving all the stakeholders.
8. **Analyze the architectural approaches:** This step re-applies Step 6 using the highly ranked scenarios from the previous step. In this step, additional architectural approaches, risks, nonrisks, sensitivity points, and trade-off points might be identified.
9. **Present the results:** The information collected during the ATAM steps is presented to the assembled stakeholders by the ATAM team.

2.3 UML Profiles

UML is a widely used notation to express analysis and design artifacts. Therefore, we use the *UML profile for problem frames* [115] and the *Architecture profile* [70] that extend the UML meta-model to support problem-oriented requirements analysis as well as the representation of quality-based software architecture with UML. These profiles can be used to create the diagrams for the problem frames approach. The description of UML4PF is given in Section 2.3.1 while the Architecture profile is described in Section 2.3.2. In addition, we introduce the dependability profile [114] in Section 2.3.3 that we use for annotating security requirements. The MARTE profile [233] used for annotating performance requirements is described in Section 2.3.4.

⁵ The terms risk, nonrisk, sensitivity point, and trade-off point are defined in Chapter 11 (see Section 11.5 on page 367) when applying ATAM.

2.3.1 UML profile for Problem Frames

Hatebur and Heisel proposed a UML profile for problem frames [115] that extends the UML meta-model. It allows one to express Jackson's original notation in UML. Côté et al. [81] developed an Eclipse-Plugin, called UML4PF, that facilitates representing the different diagrams occurring in the problem frame approach in UML. The developed plug-in contains a number of validation conditions in terms of OCL expressions [236] to check the consistency of model elements within one single diagram as well as between different diagrams.

Diagram types

Five kinds of diagrams exist in the UML profile for problem frames, namely the *context diagram*, *problem frame*, *problem diagram*, *domain knowledge diagram*, and *technical context diagram*. To represent these diagrams the corresponding stereotypes `«ContextDiagram»`, `«ProblemFrame»`, `«ProblemDiagram»`, `«DomainKnowledgeDiagram»`, and `«TechnicalContextDiagram»` have to be applied. These stereotypes extend the meta-class *Package* in the UML meta-model, as illustrated in Fig. 2.2. The context diagram and the technical context diagram are special cases of a domain knowledge diagram.

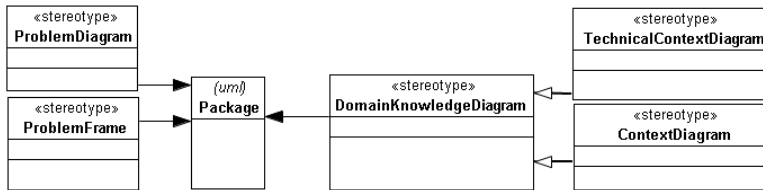


Fig. 2.2: Diagram types

Domain types

Domains are represented by classes (extending the meta-class *Class*) with the stereotypes `«Domain»` and `«Machine»`. More specific stereotypes are defined for different types of domains such as `«BiddableDomain»`, `«Causal-`

Domain», and «LexicalDomain». To describe the problem context, a *connection domain* («ConnectionDomain») between two other domains may be necessary. Connection domains establish a connection between other domains by means of technical devices. Examples are video cameras, sensors, or networks. This kind of modeling allows one to add further domain types, such as «DisplayDomain» (introduced in [82]), being a special case of a causal domain. Domain types are shown in Fig. 2.3.

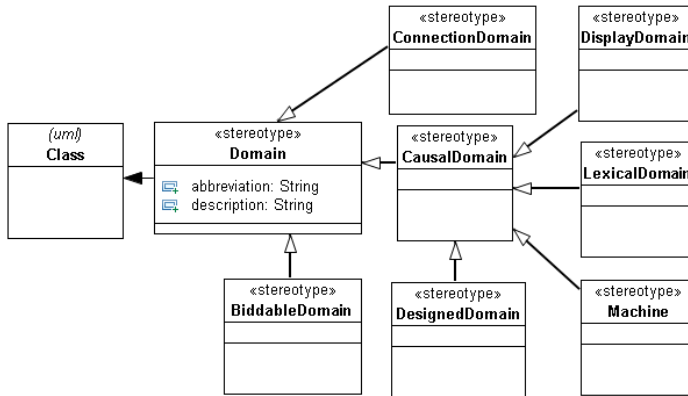


Fig. 2.3: Domain types

Statement types

As depicted in Fig. 2.4, domain knowledge («DomainKnowledge») and requirements («Requirement») are special kind of statements. Using the attribute *description* of the stereotype «Requirement», a requirement can be textually described. Assumptions («Assumption») and facts («Fact») represent special kinds of domain knowledge.

Interface types

In problem diagrams, *interfaces* connect domains. For representing interfaces, we use associations with the stereotype «connection» (extending the meta-class

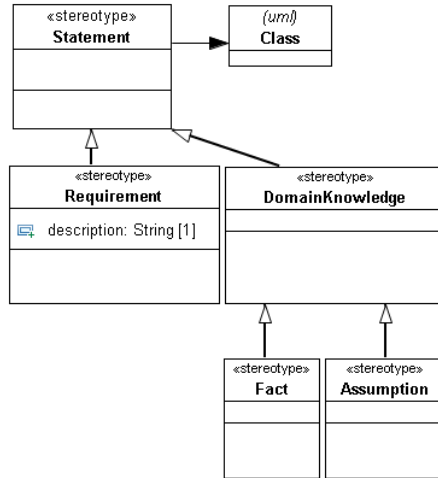


Fig. 2.4: Statement types

Association). Using the attribute *description* of the stereotype **«connection»**, a textual description to an interface can be given. For annotating the interfaces in a more precise way, more specific connections such as **«call_return»** and **«stream»** are available as shown in Fig. 2.5.

Dependency types

Each requirement constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype **«constrains»**. A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype **«refersTo»**. These dependencies extend the meta-class *Dependency* of the UML meta-model.

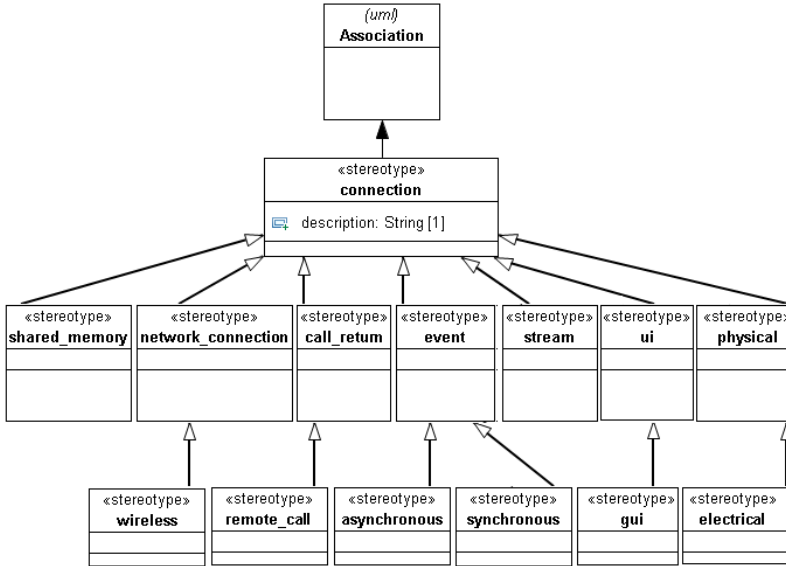


Fig. 2.5: Interface types

UML4PF Tool Support

For supporting requirements analysis with problem frames the tool UML4PF [81] is developed as an Eclipse plug-in⁶. It contains the UML profile for problem frames which allows creating problem diagrams as class diagrams in UML. For creating problem diagrams, we use Papyrus⁷ as the graphical editor, which is available as an Eclipse plug-in, open-source, and EMF-based. Nevertheless, any other EMF-based editor can be used for creating the different diagram types.

UML4PF maintains a set of validation conditions expressed in OCL⁸ which can be validated using another Eclipse plug-in for OCL. The components of UML4PF are shown in Fig. 2.6. Boxes highlighted in gray denote components that UML4PF re-uses and those in white represent those components particularly created for UML4PF. The features of UML4PF can be summarized as follows:

Requirements Editor supports adding new requirements in a textual form.

⁶ <http://www.eclipse.org/>

⁷ <https://eclipse.org/papyrus/>

⁸ <http://www.omg.org/spec/OCL/2.0/>

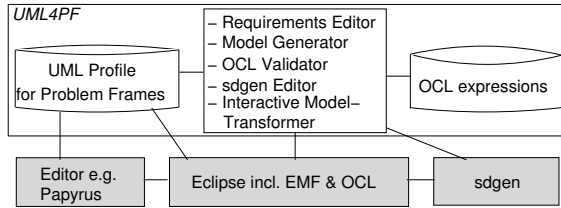


Fig. 2.6: Components of the UML4PF Tool (taken from [81])

Model Generator automatically generates model elements.

OCL Validator checks OCL expressions for validity and consistency of the requirement models.

sdgen Editor supports editing sequence diagrams.

Interactive Model Transformer supports creating software architectures using interactive model transformations.

2.3.2 Architecture Profile

We describe the structural view of software architectures by composite structure diagrams consisting of components and connectors.

Component types

For modeling components in the composite structure diagrams, the UML meta-class *Class* is extended by the stereotype **«Component»**. For each machine in the context diagram, one or more architectures are developed. The stereotypes **«Initial_architecture»**, **«Implementable_architecture»**, and **«Layered_architecture»** indicate different stages of the software architecture development (see Fig. 2.7). Furthermore, the stereotypes **«Hardware»** and **«Software»** are introduced for representing hardware and software components.

There are different stereotypes that can be used for the machine domain. If the machine domain represents a distributed system, one uses the stereotype **«distributed»**. By a local system such as a single computer, the stereotype **«local»** is used as shown in Fig. 2.8. It offers the attributes *Multiprocessor* for stating whether the system is a multiprocessor system, *MemorySpeed* for giving

the memory speed, and *OS* for describing the operating system. The stereotype `«process»` expresses a process on a certain platform. A process can be described by the attributes *Multiprocessor* and *usedOS*. The stereotype `«task»` represents a single task within a process with the attribute *usedOS* for describing the used operating system.

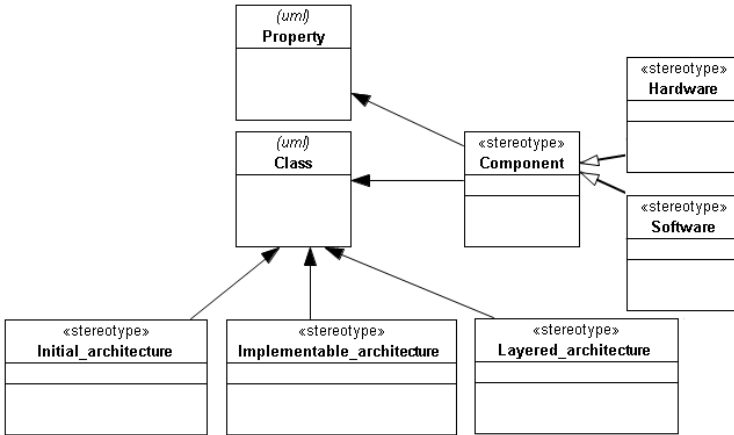


Fig. 2.7: Technical component types

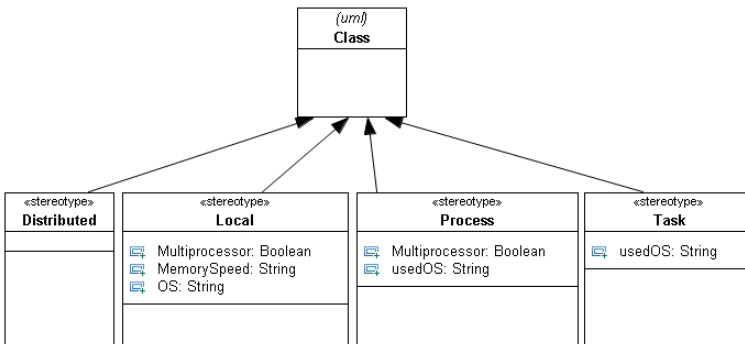


Fig. 2.8: Component types

Connector types

For modeling connectors in software architectures, we use the same stereotypes that we used for interfaces in the UML profile for problem frames. The stereotypes for connectors, however, extend the meta-class *Connector* instead of the meta-class *Association* for interfaces.

2.3.3 Dependability Profile

We use the UML profile for dependability proposed by Hatebur and Heisel [114] to annotate problem diagrams with security requirements.

Modeling confidentiality

For modeling a confidentiality requirement, the stereotype **«Confidentiality»** has to be applied. It is a specialization of the stereotype **«Dependability»**, as shown in Fig. 2.9, which extends the meta-class *Class* in the UML meta-model. The stereotype **«Confidentiality»** states that the confidentiality of the domain which is constrained in the problem diagram should be preserved by the *stakeholder* and its disclosure should be prevented from the *attacker*. The constrained domain is a causal domain. The attackers should be described in detail. The objective, skills, equipment, knowledge, preparation time, and the attack time have to be described. For describing the attackers, the stereotype **«Attacker»** (not shown in Fig. 2.9) has to be used which is a special biddable domain.

Modeling integrity

For modeling an integrity requirement, the stereotype **«Integrity»** has to be applied which is similarly to the stereotype **«Confidentiality»**, a specialization of the stereotype **«Dependability»**, as shown in Fig. 2.9. The stereotype **«Integrity»** states that the data or service of the domain which is constrained in the problem diagram (*constrainedByFunctional*) must be either correct or the domain which is influenced by a violation (*influencedViolation*) must perform an action (*actionIfViolation*).

Modeling authenticity

An authenticity requirement can be modeled using the stereotype **«Authenticity»**. It is a specialization of the stereotype **«Dependability»** (see Fig. 2.9). The stereotype **«Authenticity»** states that access to the influenced domain (*influenced*) must be permitted for known domains (*known*) and must be denied for unknown domains (*unknown*).

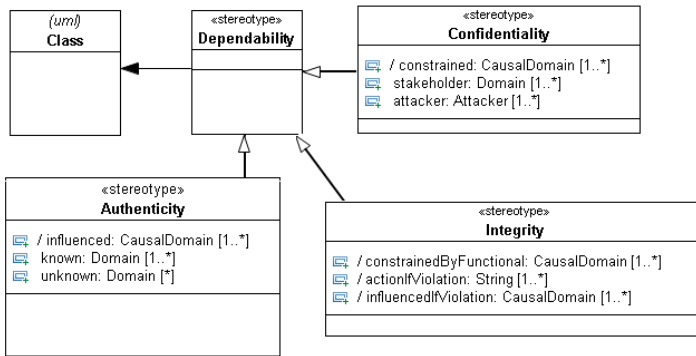


Fig. 2.9: Relevant stereotypes of dependability profile

2.3.4 MARTE Profile

The UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [233] adopted by OMG consortium extends the UML modeling language to support modeling of performance and real-time concepts. MARTE replaced the UML profile for Schedulability, Performance, and Time specification (SPT) [232]. The MARTE profile consists of the three main packages *MARTE foundations*, *MARTE design model*, and *MARTE analysis model*, shown in Fig. 2.10.

The package *MARTE foundations* contains elements to be reused by two other packages. It consists of the sub-packages for defining core elements (*CoreElements* package), modeling non-functional properties (*NFP* package), time prop-

erties (*Time* package), generic resource modeling (*GRM* package), and resource allocation (*Alloc* package).

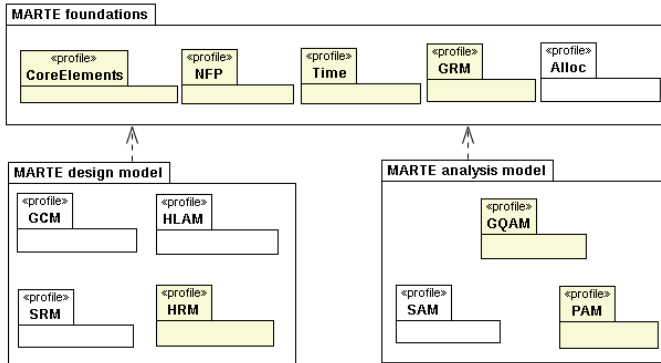


Fig. 2.10: Package structure of the MARTE profile

The packages *MARTE design model* and *MARTE analysis model* are structured for designing systems and annotating system properties for analysis purposes. The package *MARTE design model* contains the sub-packages *Generic Component Model package (GCM)* that supports the modeling of component-based systems, *High-Level Application Modeling package (HLAM)* for modeling of high-level features, *Software Resource Modeling (SRM)*, and *Hardware Resource Modeling (HRM)* for detailed modeling of software and hardware resources.

The package *MARTE analysis model* contains the sub-packages *Generic Quantitative Analysis Modeling (GQAM)*, *Schedulability Analysis Modeling (SAM)*, and *Performance Analysis Modeling package (PAM)*. The package *GQAM* provides generic concepts for analysis modeling that are further specialized by the packages *PAM* for analysis of performance properties and *SAM* for analysis of schedulability properties.

To model performance requirements and domain knowledge, we use the packages *CoreElements*, *NFP*, *Time*, *GRM*, *HRM*, *GQAM*, and *PAM*. There is an open-source implementation of the MARTE specification based on Eclipse⁹ provided by Papyrus UML¹⁰, which we use to annotate our requirements analysis models with performance analysis properties.

⁹ <http://www.eclipse.org/>

¹⁰ <https://www.eclipse.org/papyrus/>

2.4 Life-Cycle Expressions

The problem frames approach decomposes the overall problem into smaller sub-problems that fit to problem frames. We use lightweight *life-cycle expressions* to describe the relation between the requirements of the corresponding problem diagrams to be achieved to solve the overall problem. They are used in different methods of the QuaDRA framework. The life-cycle expressions can be built using the following syntax.

$$LC ::= R \mid (LC) \mid [LC] \mid LC^* \mid LC^+ \mid \\ LC; LC \mid LC \mid LC \mid LC \parallel LC$$

The syntactical elements have the following semantics. Each requirement R represents a life-cycle expression. Round braces are used to define the evaluation order of the expression for a clear precedence of the operators. Let L and M be life-cycle expressions, then

- $[L]$ is the life-cycle, where L is optionally executed.
- L^* is the life-cycle, where L is executed 0 or more times.
- L^+ is the life-cycle, where L is executed at least once.
- $L; M$ is the life-cycle, where at first L is executed and then M .
- $L \mid M$ is the life-cycle, where either L or M is executed.
- $L \parallel M$ is the life-cycle, where L and M are executed concurrently.

2.5 Variability Modeling

In software product line engineering (SPLE), orthogonal variability modeling (OVM) describes an approach to capture a product line's variability. In contrast to other approaches, which integrate variability into existing design artifacts, (e.g. using UML profiles, cf. [256]) OVM explicitly captures variability in distinct models. Using traceability links, elements from OVM models can be connected to arbitrary design or development artifacts or elements within these artifacts, e.g. requirements, a state within a UML state machine, or implemented classes [194].

OVM comprises a set of model elements that allow for modeling variability. The central model element is the abstract *variation point* (VP). A VP defines a place where single products may differ. Since a VP is an abstract model element, an instance must either be an *internal* VP or an *external* VP. Internal VPs are visible only to the developers, whereas external VPs are visible to every stakeholder. This

visibility concept allows for creating views that contain only elements that are relevant for non-developers.

Since an OVM model defines the variability of an entire SPL, it provides a concept to derive products. Several model elements (including VPs) support a selection concept. A single product is defined through all elements that have been selected. To indicate a choice for the developer, selectable VPs may be *optional*. In contrast, if a VP is considered essential, it is declared *mandatory*. A mandatory VP must be selected for every product.

While VPs define where products may differ, *variants* define how they differ. Variants and VPs are linked through *variability dependencies* (VD), where a variant has to be associated with at least one VP (in turn, a VP must be associated with at least one variant). Similar to VPs, variability dependencies may be either *optional* or *mandatory*. If a VP is selected and is associated with a variant through an optional VD, this very variant may be selected. However, if the association is a mandatory one, the variant must be selected.

To ensure flexibility in the product derivation, OVM offers the possibility to define *alternate choices*. An alternate choice groups a set of variants that are associated with the same VP through optional dependencies and defines a minimum and a maximum value. In product derivation, a number of n with $minimum \leq n \leq maximum$ variants have to be selected if their corresponding VP has been selected.

Since in practice relationships and interactions between variants and VPs can be observed, OVM allows for defining these relationships through *variability constraints*. Variability constraints can be set up between two variants, two VPs, or a variant and a VP. OVM provides two types of variability constraints: *requires* and *excludes*. The *requires* constraint is directed from a source to a target element and requires the target to be selected if the source has been selected. The *excludes* constraint is undirected and prevents selecting one element if the other element has been selected.

2.6 Case Study Smart Grid

To illustrate the application of our framework, we use the real-life case study of smart grid. As sources for real functional and quality requirements, we consider diverse documents such as “*Application Case Study: Smart Grid*” and “*Smart Grid Concrete Scenario*” provided by the industrial partners of the EU project NES-

SoS¹¹, the “*Protection Profile for the Gateway of a Smart Metering System*” [155] provided by the German Federal Office for Information Security¹², “*Smart Metering Implementation Programme, Overview Document*” [106] and “*Smart Metering Implementation Programme, Design Requirements*” [105] provided by the UK Office of Gas and Electricity Markets¹³, and “*D1.2 Report on Regulatory Requirements* [201]” and “*Requirements of AMI (Advanced Multi-metering Infrastructure)*” [200] provided by the EU project OPEN meter¹⁴.

The smart grid case study is suitable for illustrating the applicability of the methods proposed in the QuaDRA framework due to the following reasons:

Consideration of quality requirements: In the smart grid case study, different kinds of quality requirements have to be taken into account. We list them in the following:

Security: For instance, a smart grid involves a wide range of data that should be treated in a secure way. Additionally, introducing new data interfaces to the grid (smart meters, collectors, and other smart devices) provides new entry points for attackers. Therefore, special attention should be paid to security concerns.

Performance: The number of smart devices to be managed has a deep impact on the performance of the whole system. This makes performance of smart grids an important issue.

Considering these different kinds of quality requirements in the smart grid case study allows us to illustrate:

- the elicitation and modeling of quality requirements (*Phase 1: context elicitation & problem analysis*, Chapter 4)
- the selection of architectural patterns (*Phase 2: architectural pattern selection*, Chapter 5)
- the capturing of quality-related domain knowledge and its integration in the requirement models (*Phase 3: domain knowledge analysis*, Chapter 6),
- the exploration of quality-specific solution alternatives (*Phase 5: quality solution identification & analysis*, Chapter 8).

Consideration of stakeholders: Due to the fact that different stakeholders with diverse and partially contradicting interests are involved in the smart grid, the

¹¹ <http://www.nessos-project.eu/>

¹² www.bsi.bund.de

¹³ <http://www.ofgem.gov.uk>

¹⁴ <http://www.openmeter.com/>

requirements for the whole system contain conflicts or undesired mutual influences. Therefore, the smart grid is a very good candidate to illustrate

- the applicability of our method for detecting interactions among functional and quality requirements (*Phase 4: requirement interaction analysis*, Chapter 7),
- the resolution of interacting requirements by generating requirement alternatives, selecting, and applying quality-specific solution alternatives (*Phase 6: quality solution selection & application*, Chapter 9),
- the derivation of architecture alternatives (*Phase 7: quality-based software architecture alternative derivation & evaluation*).

We give a description of smart grids in Section 2.6.1. Section 2.6.2 presents the functional requirements of the smart grid case study that we use throughout this work. The relevant security and performance requirements are given in Sections 2.6.3 and 2.6.4.

2.6.1 Description of Smart Grids

To use energy in an optimal way, smart grids make it possible to couple the generation, distribution, storage, and consumption of energy. Smart grids use information and communication technology (ICT) which allows for financial, informational, and electrical transactions.

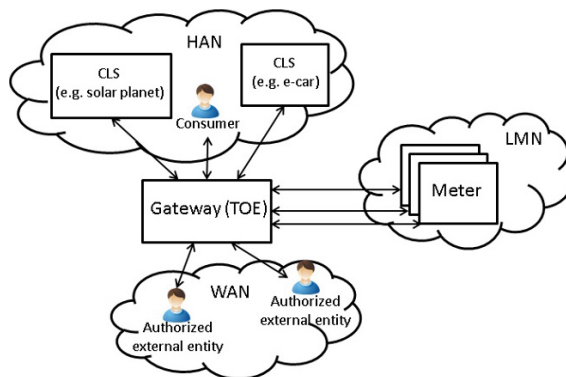


Fig. 2.11: The context of a smart grid system

Figure 2.11 shows the simplified context of a smart grid system based on the protection profile [155]. We first define the terms specific to the smart grid domain taken from the protection profile:

Gateway represents the central communication unit in a *smart metering system*.

It is responsible for collecting, processing, storing, and communicating *meter data*.

Meter data refers to meter readings measured by the meter regarding consumption or production of a certain commodity.

Meter represents the device that measures the consumption or production of a certain commodity and sends it to the gateway.

Authorized external entity could be a human or IT unit that communicates with the gateway from outside the gateway boundaries through a *Wide Area Network* (WAN). The roles defined as external entities that interact with the gateway and the meter are *consumer*, *grid operator*, *supplier*, *gateway operator*, *gateway administrator*, ... (for the complete list of possible external entities see the protection profile [155]).

WAN (Wide Area Network) provides the communication network that interconnects the gateway with the outside world.

LMN (Local Metrological Network) provides the communication network between the meter and the gateway.

HAN (Home Area Network) provides the communication network between the consumer and the gateway.

LAN (Local Area Network) provides the communication network that interconnects domestic equipment or metrological equipment¹⁵.

Consumer refers to the end user or producer of commodities (electricity, gas, water, or heat).

2.6.2 Functional Requirements

The functionality of the smart grid is described as use cases. The use cases given in the documents of the open meter project are divided into the three categories *minimum*, *advanced*, and *optional*. Minimum use cases are necessary to achieve the goals of the system, whereas advanced use cases are of high interest, but might not be absolutely required, and optional use cases provide add-on functions. As treating all 20 use cases would go beyond the scope of this work, we decided to

¹⁵ In protection profile, LAN is referred to as hypernym for LMN (Local Metrological Network) and HAN (Home Area Network).

consider only the use case *Meter Reading for Billing*. This use case is concerned with gathering, processing, and storing meter readings from smart meters for the billing process. The considered use case belongs to the category *minimum*.

The protection profile [155, p.18] states that “*the Gateway is responsible for handling Meter Data. It receives the Meter Data from the Meter(s), processes it, stores it, and submits it to external parties.*” Therefore, we define the requirements *R1-R3* to receive, process, and store meter data from smart meters. The requirement *R4* is concerned with submitting meter data to authorized external entities. The gateway shall also provide meter data for consumers for the purpose of checking the billing consistency (*R5*). Requirements with their descriptions are listed in Table 2.2.

Table 2.2: Requirements for smart metering

Requirement	Description	Related functional requirement
R1	Smart meter gateway shall receive meter data from smart meters	-
R2	Smart meter gateway shall process meter data from smart meters	-
R3	Smart meter gateway shall store meter data from smart meters	-
R4	Smart meter gateway shall submit processed meter data to authorized external entities	-
R5	The gateway shall provide meter data for consumers for the purpose of checking the billing consistency	-
R6	The gateway shall provide the protection of integrity when receiving meter data from a meter via the LMN	R1
R7	The gateway shall provide the protection of confidentiality when receiving meter data from a meter via the LMN	R1
R8	The gateway shall provide the protection of authenticity when receiving meter data from a meter via the LMN	R1
R9	Data shall be protected from unauthorized disclosure while persistently stored in the gateway	R3
R10	Integrity of data transferred in the WAN shall be protected	R4
R11	Confidentiality of data transferred in the WAN shall be protected	R4
R12	Authenticity of data transferred in the WAN shall be protected	R4
R13	The gateway shall provide the protection of integrity when transmitting processed meter data locally within the LAN	R5
R14	The gateway shall provide the protection of confidentiality when transmitting processed meter data locally within the LAN	R5
R15	The gateway shall provide the protection of authenticity when transmitting processed meter data locally within the LAN	R5
R16	Data shall be protected from unauthorized disclosure while temporarily stored in the gateway	R1

R18	The time to retrieve meter data from the smart meter and publish it through the WAN shall be less than 5 seconds (together with R20, R22, R24)	R1
R19	The time to retrieve meter data from the smart meter and publish it through the HAN shall be less than 10 seconds (together with R21, R23, R25)	R1
R20	The time to retrieve meter data from the smart meter and publish it through the WAN shall be less than 5 seconds (together with R18, R22, R24)	R2
R21	The time to retrieve meter data from the smart meter and publish it through the HAN shall be less than 10 seconds (together with R19, R23, R25)	R2
R22	The time to retrieve meter data from the smart meter and publish it through the WAN shall be less than 5 seconds (together with R18, R20, R24)	R3
R23	The time to retrieve meter data from the smart meter and publish it through the HAN shall be less than 10 seconds (together with R19, R21, R25)	R3
R24	The time to retrieve meter data from the smart meter and publish it through WAN shall be less than 5 seconds (together with R18, R20, R22)	R4
R25	The time to retrieve meter data from the smart meter and publish it through the HAN shall be less than 10 seconds (together with R19, R21, R23)	R5

2.6.3 Security Requirements

To ensure security of meter data, the protection profile [155, pp. 18, 20] demands protection of data from unauthorized disclosure while received from a meter via the LMN (*R7*), temporarily or persistently stored in the gateway (*R9*, *R16*), transmitted to the corresponding external entity via the WAN (*R11*), and transmitted locally within the LAN (*R14*). The gateway shall provide the protection of authenticity and integrity when receiving meter data from a meter via the LMN, to verify that the meter data has been sent from an authentic meter and has not been altered during transmission (*R6*, *R8*). The gateway shall provide the protection of authenticity and integrity when sending processed meter data to an external entity, to enable the external entity to verify that the processed meter data has been sent from an authentic gateway and has not been changed during transmission (*R10*, *R12*, *R13*, *R15*).

2.6.4 Performance Requirements

The report “*Requirements of AMI*” [200, p. 199–201] demands that the time to retrieve meter data from the smart meter and publish it through the WAN shall be less than 5 seconds. Since we decompose the whole functionality, from retrieving meter data to publishing it, into requirements *R1*–*R4*, we also decompose this performance requirement into the requirements *R18* (related to *R1*), *R20* (related to *R2*), *R22* (related to *R3*), and *R24* (related to *R4*). The requirements *R18*, *R20*, *R22*, and *R24* shall be fulfilled in a way that in total they do not need more than 5 seconds.

Further, the report “*Requirements of AMI*” states that for the benefit of the consumer, actual meter readings are to be provided to the end consumer device through HAN. It demands that the time to retrieve meter data from the smart meter and publish it through HAN shall be less than 10 seconds. Similar to the previous requirement, we decompose this requirement into the requirements *R19* (related to *R1*), *R21* (related to *R2*), *R23* (related to *R3*), and *R25* (related to *R5*). These requirements together shall be fulfilled in less than 10 seconds.

Bridging the Gap between Requirements Engineering
and Software Architecture

A Problem-Oriented and Quality-Driven Method

Alebrahim, A.

2017, XXVI, 500 p. 141 illus., Softcover

ISBN: 978-3-658-17693-8