# Distributed Frames: Pattern-Based Characterization of Functional Requirements for Distributed Systems

Roman Wirtz(✉), Maritta Heisel, and Marvin Wagner

University of Duisburg-Essen, Duisburg, Germany
roman.wirtz@uni-due.de

**Abstract.** In the connected world, the complexity of software-based systems increases. Many of those systems consist of different subsystems which are connected with each other via a network. The decomposition into those subsystems requires a detailed analysis and documentation of their functional requirements. Documenting and managing such requirements in a consistent manner is a challenge for software engineers. The requirements for each subsystem cannot be considered in isolation, but it is necessary to state the relations between the functional requirements, too. In previous work, we proposed a method that allows systematically identifying and documenting functional requirements for distributed systems. The method is model-based and makes use of Jackson's problem frames approach which defines patterns for reoccurring software development problems. We now extend his approach with patterns for problems specifically for distributed systems which we call Distributed Frames. Using a pattern description template, we provide different examples of such frames. To exemplify the application of those patterns, we show how they can be embedded into our requirements elicitation method.

**Keywords:** Requirements engineering · Distributed systems · Model-based · Functinonal requirements · Requirements analysis

## 1 Introduction

In the connected world, software-based systems are often realized as distributed systems. Tanenbaum defines a distributed system as a system whose components are located on different connected computers [15]. Those components communicate via messages to achieve a common goal.

The complexity of distributed systems confronts software engineers with new problems during the entire software development process. Especially in one of the earliest phases of software development, namely requirements engineering, it is a challenge for engineers to capture all aspects of a distributed system under development. Although the different components may be deployed independently of each other in different environments, the functionalities of the components highly depend on each other. Thus, it does not suffice to elicit and document requirements for each component independently. In addition, the connection between the components is often remote and, hence, is not reliable.

For further analysis, e.g. with regard to privacy or security, it is of essential importance to document the dependencies and interfaces between the subsystems in a consistent and systematic manner. For example, an attacker may inject malicious code on the client-side which will then affect stored data on the server-side.

Our aim is to assist software engineers in performing a detailed and systematic elicitation and documentation of functional requirements for distributed systems.

In previous work [18], we proposed a model-based method called *RE4DIST* (**R**equirements **E**ngineering for **DIST**ributed Systems). The method is based on Jackson's problem frames notation which we extended to model functional requirements for distributed systems. The extension allows making the connection between the subsystems and the relations between the corresponding requirements explicit. The method starts with the decomposition based on the system's context, and it ends up with a model of functional requirements to be fulfilled by the subsystem.

In the present paper, we follow Jackson's pattern-based approach and introduce so-called *Distributed Frames*. Each distributed frame is a pattern that describes a common problem class for a distributed system, i.e, specific types of its functional requirements. An instance of a frame describes a concrete instance of a functional requirement for a distributed system. The pattern-based approach allows making knowledge about requirements reusable. Furthermore, by instantiating the pattern, software engineers can document the elicited requirements in a consistent manner. To specify a distributed frame, we propose a template-based format. It describes the frame itself, textual patterns for the functional requirement, and typical examples for the frame's application context.

To exemplify the application of distributed frames, we extend our *RE4DIST* method with regard to the frames. The extension allows the elicitation and documentation of functional requirements by instantiating an appropriate frame.

The remainder of the paper is structured in the following way: In Sect. 2, we introduce a notation for Michael Jackson's problem frames and five basic frames. In Sect. 3, we present the underlying requirements model in form of an Ecore metamodel [14]. Section 4 contains the template for specifying distributed frames and some examples. We describe the extension of our method in Sect. 5, and using a small case study, we exemplify the application of the method in Sect. 6. We discuss related work in Sect. 7 and conclude the paper in Sect. 8 with a brief summary and an outlook on future research directions.

## 2 Problem Frames

We first introduce the notation for problem frames and problem diagrams, followed by the introduction of some basic frames.

### 2.1 Notation

To model functional requirements, we make use of the problem frames approach as introduced by Michael Jackson [10]. We consider two types of diagrams, context diagrams and problem diagrams, which both consist of domains, phenomena, and interfaces.

Machine domains (💻) represent the piece of software to be developed.

Problem domains represent entities of the real world. There are different types of these domains: biddable domains with an unpredictable behavior, e.g. persons (☺), causal domains(⚙) with a predictable behavior, e.g. technical equipment, and lexical domains (🗎) for data representation. A domain can take the role of a connection domain (⧉) which serves as a connection between two other domains.

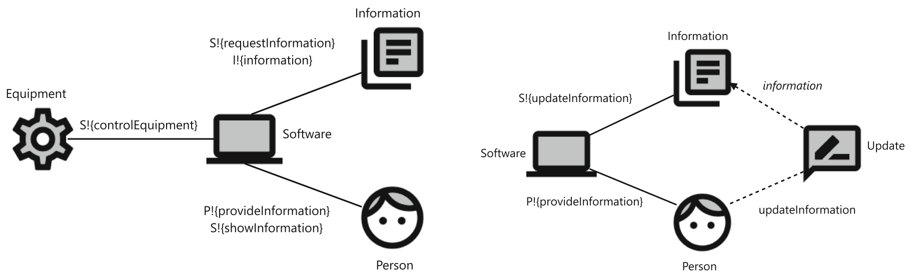
Interfaces between domains consist of phenomena. There are (i) symbolic phenomena, representing some kind of information or a state, (ii) causal phenomena, representing commands, actions and the like, and (iii) events. Each phenomenon is controlled by exactly one domain and can be observed by other domains. A phenomenon controlled by one domain and observed by another is called a shared phenomenon between these two domains. Interfaces (solid lines) contain sets of shared phenomena. Such a set contains phenomena controlled by one domain, indicated by *D!*{...}, where *D* stands for an abbreviation of the controlling domain.

A context diagram describes where the problem, i.e. software to be developed, is located and which domains it concerns. It does not contain any requirements. We show an example of such a diagram in Fig. 1a. It contains four domains and the corresponding interfaces. There are *Software* 💻, *Equipment* ⚙, *Information* 🗎, and *Person* ☺.

A problem diagram is a projection of the context. It contains a functional requirement (represented by the symbol 🖹) describing a specific functionality to be developed. A requirement is an optative statement that describes how the environment should behave when the software is installed.

Some phenomena are *referred to* by a requirement (dashed line to the controlling domain), and at least one phenomenon is *constrained* by a requirement (dashed line with arrowhead and italics). The domains and their phenomena that are *referred to* by a requirement are not influenced by the machine, whereas we build the machine to influence the *constrained* domain's phenomena in such a way that the requirement is fulfilled.

In Fig. 1b, we show a small example describing a functional requirement for updating some information which is a projection of the context given in Fig. 1a. A *Person* ☺ provides information to *Software* 💻 to be updated. We make use of a lexical domain *Information* 🗎 to represent a database. The functional requirement *Update* 🖹 refers to the phenomenon *updateInformation* and constrains the phenomenon *information*.



**(a)** Context diagram [18] **(b)** Problem diagram [18]

**Fig. 1.** Examples.

**Table 1.** Basic problem frames.

| Name | Domain types referred to | Domain types constrained |
|---|---|---|
| Required behavior | – | *C* ⚙ |
| Commanded Behavior | *B* ☺ | *C* ⚙ |
| Information display | *C* ⚙ | *C* ⚙ |
| Simple workpiece | *B* ☺ | *L* 🗐 |
| Transformation | *L* 🗐 | *L* 🗐 |

*Legend:* ***C*** *- causal,* ***L*** *- lexical,* ***B*** *- biddable*

The icons we use in our diagrams differ from Jackson's notation. We adopted icons from Google's Material Design[1] to provide intuitive views for the diagrams [18].

## 2.2 Problem Frames

Jackson distinguishes between five problem frames which we will consider in the following sections for further analysis concerning distributed systems. In Table 1, we provide an overview for these frames.

**Required Behavior.** Some parts of the physical environment shall be controlled. The task is to build a machine that imposes that control.

**Commanded Behavior.** An operator can issue commands to control some part of the physical environment. The machine to be built shall receive the commands and shall impose the control accordingly.

**Information Display.** The machine shall obtain some information from the environment continuously and present it at the required place and in the required form.

**Simple Workpieces.** The task is to build a machine that allows users to process some information, e.g. to edit, delete, or copy it.

[1] Google Material - https://material.io (last access: March 15, 2019).

**Transformation.** The machine to be built shall take some machine-readable information as input and transform it into the required output.

In Sect. 4, we provide these frames for distributed systems.

## 3 Meta Model

To model functional requirements, we make use of Jackson's problem frames approach (cf. Sect. 2) for which we introduce a metamodel in the following. We extend that model with additional elements to capture specific aspects of distributed systems. For instantiating and maintaining the model, we developed a graphical editor. We decided to build that tool based on the *Eclipse Modeling Framework (EMF)* [14]. EMF is open source and offers a wide range of products for model-based development. For example, we use Eclipse Sirius[2] to provide a graphical editor for the application of our method.

### 3.1 Model Elements

**Domains.** A domain can be a connection domain, which is indicated by an appropriate attribute. We distinguish between *Machine* and *Problem Domains* as proposed by Jackson. Besides, we introduce the domain type *Distributed System*. A problem domain can be a *Causal Domain*, *Biddable Domain* or *Lexical Domain*. For expressing the relation between different machines, we introduce the domain type *Remote Machine*. The domain acts as a placeholder for a subsystem and therefore references exactly one machine domain. We show the relevant part of the model in Fig. 2.

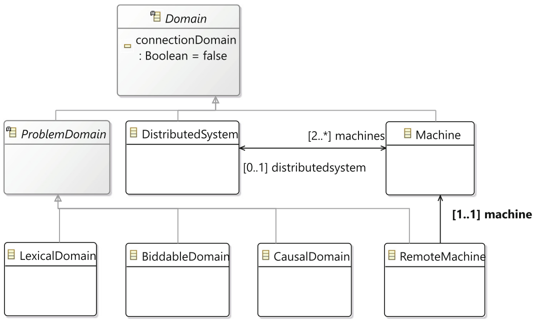


**Fig. 2.** Metamodel - domains.

**Interfaces.** In Fig. 3, we show the part of the model to describe interfaces. A *Domain Interface* connects at least two domains and contains a set of *Phenomena*. A phenomenon is controlled by exactly one domain. To describe the realization of interfaces in more detail, we adapt the so-called attack vector from the

[2] Eclipse Sirius - https://www.eclipse.org/sirius/ (last access: November 12, 2019).

*Common Vulnerability Scoring System* (CVSS) [6]. An attack vector predefines values to describe how an attacker accesses a vulnerable component. We introduce an *AccessVector* to describe how domains interact with each other. The vector distinguishes the following four values: *Network* describes remote connections through different networks, e.g. connections via the internet, *adjacent* stands for local network connections, *local* means access to domains not connected to the internet, e.g. some user interfaces, and *physical* describes physical connection to domains, e.g. sensors.

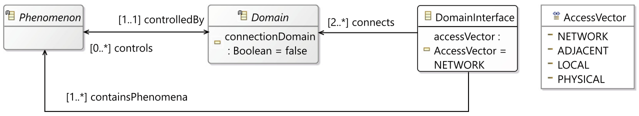


**Fig. 3.** Metamodel - interfaces.

**Requirements.** Figure 4 shows the part of the model to describe requirements. A *Requirement* is a special kind of *Statement*. It can be distributed, which means that it concerns more than one machine. Each statement has at least one *StatementReference* for at least one *Phenomenon*. A reference can either be a *ConstrainsReference* or a *RefersToReference*. For each requirement, we also make the machines explicit that are related to the specific requirement.

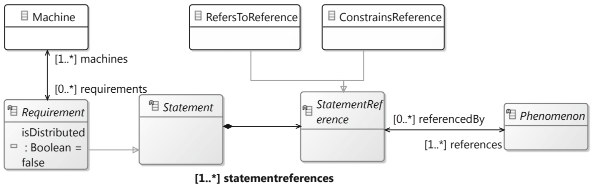


**Fig. 4.** Metamodel - requirements.

**Diagrams.** Within the model, it is possible to define different views on specific elements using diagrams (see Fig. 2). As mentioned in Sect. 2, Jackson distinguishes between context diagrams and problem diagrams. We introduce two new types of context diagrams. A *GlobalContextDiagram* describes the overall context of the distributed system. A *SubContextDiagram* is derived from it and describes the context for a specific subsystem (Fig. 5).

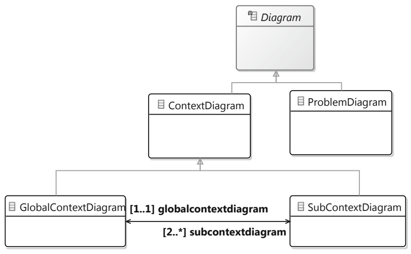


**Fig. 5.** Metamodel - diagrams.

## 4 Distributed Frames

In Sect. 2, we introduced five basic problem frames. In the context of our previous work, we now propose distributed frames which are a special kind of problem frames applicable for distributed systems. In contrast to a problem frame, a distributed frame does not only consider a single system but different subsystems. Each frame describes a pattern that allows the characterization of reoccurring problems in the context of functional requirements for distributed systems. We first introduce a description format for distributed frames followed by several examples of frames.

### 4.1 Description Format

We provide a template to specify distributed frames in a consistent way, for which we give an overview in Table 2. The template consists of some basic information and a frame description.

**Table 2.** Frame description format.

| **Basic Information** | |
|---|---|
| Name | Short and descriptive name for the frame. |
| Description | Short informal description about the frame and the context for which it is applicable. |
| Known uses | List of typical examples where the pattern can be applied. |
| **Frame Description** | |
| *Sender* | |
| Frame Diagram | Diagram which contains the relevant domains and interfaces on the sender side. |
| Textual pattern | Textual pattern for the relevant part of the functional requirement on the sender side. |
| *Receiver* | |
| Frame Diagram | Diagram which contains the relevant domains and interfaces on the sender side. |
| Textual pattern | Textual pattern for the relevant part of the functional requirement on the receiver side. |

**Basic Information.** We provide a short informal description that summarizes the distributed frame and briefly describes the context for which it is applicable. The textual description of the functional requirement to be satisfied by the distributed system is also part of that informal description. The requirement will later be decomposed for the involved subsystems. Last, we list typical examples of scenarios as known uses for the application of the frame.

**Frame Desciption.** We distinguish between the sender side and the receiver side. Since our approach is applicable for any type of distributed system, we do not use the notion of client/server side here. For each side, we provide a frame diagram using the notation as described in Sect. 2.1. A frame diagram contains the domain types, connecting interfaces, and requirement references for the frame. By instantiating the frame diagram in the concrete context, one can create a problem diagram. Besides the frame diagram, we propose a textual pattern that describes the functional requirement in natural language. ⟨...⟩ indicates a variable in the textual pattern that needs to be filled.

## 4.2 Frame Specifications

In the following, we give specifications for distributed frames. Table 3 provides an overview. It contains the name of the distributed frame (DF) and the constrained and referred to domain types on the sender and receiver side.

## 4.3 Basic Frames

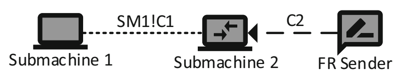
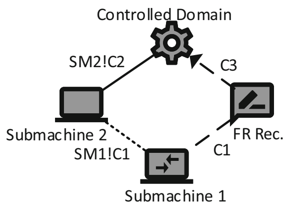
In Sect. 2.2, we described five basic frames that have been defined by Jackson [10]. We specify those frames for distributed systems using our previously defined format.

**Table 3.** Distributed frames overview.

| | **Sender** | | **Receiver** | |
|---|---|---|---|---|
| **Name** | **Domain types referred to** | **Domain types constrained** | **Domain types referred to** | **Domain types constrained** |
| | | ***Basic Frames*** | | |
| Required behavior (DF) | – | *RM* | *RM* | *C* |
| Commanded Behavior (DF) | *B* | *RM* | *RM* | *C* |
| Information display (DF) | *C* | *RM* | *RM* | *C* |
| Simple workpiece (DF) | *B* | *RM* | *RM* | *L* |
| Transformation (DF) | *L* | *RM* | *RM* | *L* |
| | | ***Additional Frames*** | | |
| Query (DF) | *B* | *RM*, *CON* | *RM*, *L* | *RM* |
| Update (DF) | *B* | *RM*, *CON* | *RM* | *RM*, *L* |

*Legend:* ***C*** *- causal,* ***L*** *- lexical,* ***B*** *- biddable,* ***RM*** *- remote machine*

**Table 4.** Frame description for *Required Behavior (DF)*.

| ***Sender*** | ***Receiver*** |
|---|---|
|  |  Controlled Domain<br>SM2!C2<br>C3<br>Submachine 2<br>FR Rec.<br>SM1!C1<br>C1<br>Submachine 1 |
| A control command ⟨C1⟩ can be sent to ⟨Submachine 2⟩. | The ⟨Controlled Domain⟩ is controlled with ⟨C2⟩ according to the command ⟨C1⟩ issued by ⟨Submachine 1⟩. |

## Required Behavior (DF)

***Description.*** A subsystem can control domains of its physical environment. Another subsystem can issue commands to control those domains. The task is to build a distributed system in which the machine of one subsystem can control domains in the physical environment of another subsystem remotely.

***Known Uses***

- Smart home services are often deployed as a cloud application. Previously defined commands are sent from the cloud service to the customer's home to control the equipment.
- Traffic light controllers can be connected to send control commands, e.g. to prioritize a tram.

***Frame Description.*** In Table 4, we show the frame description for the frame *Required Behavior (DF)*. There is a frame diagram for each sender and receiver side. The corresponding textual patterns are given below. In the frame diagrams we use abbreviations for the phenomena annotated at the interfaces. *Y* stands for symbolic phenomena, *C* for causal phenomena, and *E* stands for events (cf. Sect. 2).

## Commanded Behavior (DF)

***Description.*** A subsystem can control domains of its physical environment. Operators can issue commands via another subsystem to control those domains. The task is to build a distributed system in which an operator can control domains in the physical environment of another subsystem remotely.

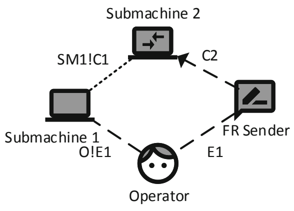
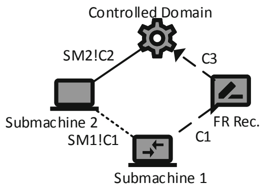
***Known Uses***

- Smartphone applications can be used to control equipment, such as TVs or smart home equipment.

– Sound and light equipment for concerts can be configured remotely.
– Vehicles can be maintained remotely by the manufacturer.

***Frame Desciption.*** In Table 5, we show the frame description for the frame *Commanded Behavior (DF)*.

**Table 5.** Frame description for *Commanded Behavior (DF)*.

| ***Sender*** | ***Receiver*** |
|---|---|
|  Submachine 2<br>SM1!C1<br>C2<br>Submachine 1<br>O!E1<br>FR Sender<br>E1<br>Operator |  Controlled Domain<br>SM2!C2<br>C3<br>Submachine 2<br>SM1!C1<br>FR Rec.<br>C1<br>Submachine 1 |
| The ⟨Operator⟩ can cause the event ⟨E1⟩ to trigger ⟨C2⟩ of ⟨Submachine 2⟩. | The ⟨Controlled Domain⟩ is controlled with ⟨C2⟩ according to the command ⟨C1⟩ issued by ⟨Submachine 1⟩. |

### Information Display (DF)

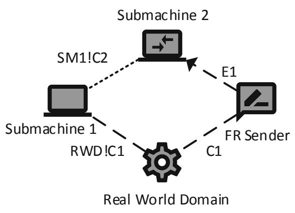
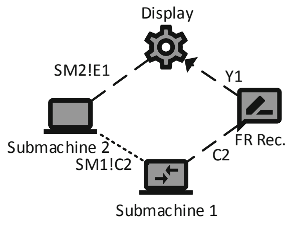
***Description.*** One subsystem continuously receives information from the physical environment. Another subsystem has some display in its environment. The task is to exchange the received information between the subsystems and to display them.

***Known Uses***

– A vehicle sends sensor data to the driver's smartphone where the data is displayed.
– Traffic monitors at train stations show the estimated arrival time of a train.

***Frame Description.*** In Table 6, we show the frame description for the frame *Information Display (DF)*.

**Table 6.** Frame description for *Information Display (DF)*.

| ***Sender*** | ***Receiver*** |
|---|---|
|  Submachine 2<br>SM1!C2<br>E1<br>Submachine 1<br>FR Sender<br>RWD!C1<br>C1<br>Real World Domain |  Display<br>SM2!E1<br>Y1<br>FR Rec.<br>Submachine 2<br>C2<br>SM1!C2<br>Submachine 1 |
| The ⟨Real World Domain⟩ can send information to ⟨Submachine 2⟩ with the command ⟨C1⟩. | With the command ⟨C2⟩, ⟨Submachine 1⟩ can show information ⟨Y1⟩ at the ⟨Display⟩. |

## Simple Workpieces (DF)

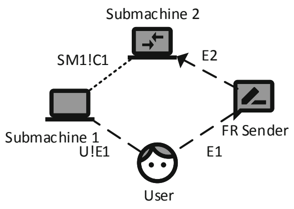
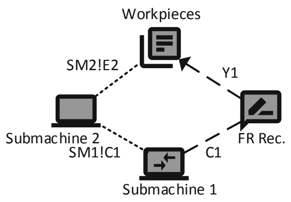
***Description.*** A user can use a subsystem to manipulate some data which is remotely accessible at another subsystem. The task is to transmit the commands to the subsystem where the data is available and to manipulate the data accordingly.

***Known Uses***

- Some technical equipment allows editing the configuration remotely.
- Using a web service, a user can edit his/her data.
- Collaboration tools allow data manipulation in the cloud.

***Frame Description.*** In Table 7, we show the frame description for the frame *Simple Workpieces (DF)*.

**Table 7.** Frame description for *Simple Workpieces (DF)*.

| ***Sender*** | ***Receiver*** |
|---|---|
|  Submachine 2<br>SM1!C1<br>E2<br>Submachine 1<br>FR Sender<br>U!E1<br>E1<br>User |  Workpieces<br>SM2!E2<br>Y1<br>Submachine 2<br>FR Rec.<br>SM1!C1<br>C1<br>Submachine 1 |
| The ⟨User⟩ can cause the event ⟨E1⟩ to trigger the event ⟨E2⟩ of ⟨Submachine 2⟩. | Information ⟨Y1⟩ of the ⟨Workpieces⟩ can be edited by the ⟨Submachine 1⟩ via the command ⟨C1⟩. |

## Transformation (DF)

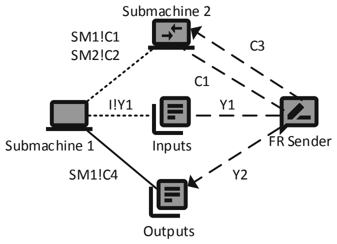
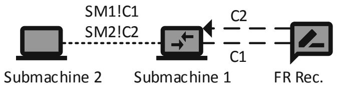
***Description.*** Data of one subsystem shall be transformed. For transforming the data, another subsystem shall be used. The task is to develop a system that allows transmitting data from one system to another to transform it. Afterwards, the transformed data shall be stored in the subsystem where it originates.

***Known Uses***

- There are several online converters that allow submitting data that can be downloaded afterward, i.e. an image to PDF converter.
- There are online tools that allow to encrypt and decrypt data.

***Frame Description.*** In Table 8, we show the frame description for the frame *Transformation (DF)*.

**Table 8.** Frame description for *Transformation (DF)*.

| *Sender* | *Receiver* |
|---|---|
|  Submachine 2<br>SM1!C1<br>SM2!C2<br>C3<br>C1<br>!!Y1<br>Y1<br>Submachine 1<br>Inputs<br>FR Sender<br>SM1!C4<br>Y2<br>Outputs |  SM1!C1<br>SM2!C2<br>C2<br>C1<br>Submachine 2<br>Submachine 1<br>FR Rec. |
| Some information ⟨Y1⟩ can be transmitted to ⟨Submachine 2⟩ while triggering the command ⟨C3⟩. The transformed information is stored at ⟨Outputs⟩ (⟨Y2⟩). | ⟨Submachine 1⟩ can transform some information with the command ⟨C1⟩ which is returned afterwards. |

### 4.4 Additional Frames

Besides the basic frames, we show the specifications of two additional frames, namely *Query* and *Update* [3,16]. We adapt those frames for distributed systems and provide their specifications in the following.

## Query (DF)

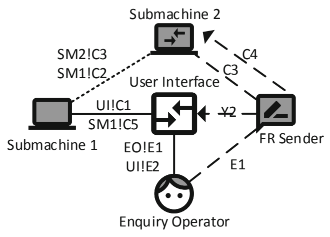
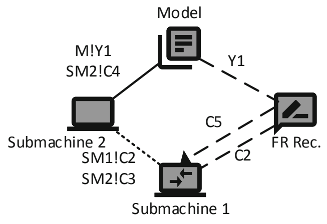
***Description.*** Users want to request data from a remotely accessible resource. The requested information shall be displayed to them.

### ***Known Uses***

– Requesting a website.
– Smartphone applications that retrieve information from an external resource.
– Reading data from network-attached storages (NAS).

***Frame Description.*** In Table 9, we show the frame description for the frame *Query (DF)*.

**Table 9.** Frame description for *Query (DF)*.

| ***Sender*** | ***Receiver*** |
|---|---|
|  Submachine 2<br>SM2!C3<br>SM1!C2<br>C4<br>C3<br>User Interface<br>UI!C1<br>SM1!C5<br>Y2<br>FR Sender<br>Submachine 1<br>EO!E1<br>UI!E2<br>E1<br>Enquiry Operator |  Model<br>M!Y1<br>SM2!C4<br>Y1<br>C5<br>FR Rec.<br>Submachine 2<br>SM1!C2<br>SM2!C3<br>C2<br>Submachine 1 |
| To query some information, the ⟨Enquiry Operator⟩ can cause the event ⟨E1⟩ to trigger the command ⟨C4⟩ for ⟨Submachine 2⟩. ⟨Submachine 2⟩ provides the information via ⟨C3⟩ which is then displayed at the ⟨User Interface⟩. | ⟨Submachine 1⟩ can query some information ⟨Y1⟩ from the ⟨Model⟩ with the command ⟨C2⟩. |

## Update (DF)

***Description.*** Users want to manipulate data that is available at a remotely accessible resource. In contrast to the frame *Simple Workpiece (DF)*, there is feedback for the users.

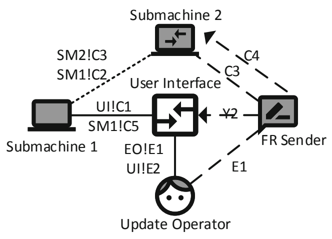
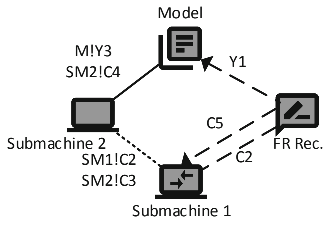
### ***Known Uses***

– Websites where users can enter or edit some information to be stored.
– Uploading data to a NAS with a progress bar as feedback.

***Frame Description.*** In Table 10, we show the frame description for the frame *Update (DF)*.

In the next section, we extend the *RE4DIST* method with our proposed frames.

**Table 10.** Frame description for *Update (DF)*.

| ***Sender*** | ***Receiver*** |
|---|---|
|   Submachine 2 SM2!C3 SM1!C2 C4 C3 User Interface UI!C1 SM1!C5 Y2 Submachine 1 FR Sender EO!E1 UI!E2 E1 Update Operator |   Model M!Y3 SM2!C4 Y1 C5 Submachine 2 FR Rec. SM1!C2 C2 SM2!C3 Submachine 1 |
| To update some information, the ⟨Update Operator⟩ can cause the event ⟨E1⟩ to trigger the command ⟨C4⟩ for ⟨Submachine 2⟩. ⟨Submachine 2⟩ provides a feedback via ⟨C3⟩ which is then displayed at the ⟨User Interface⟩. | ⟨Submachine 1⟩ can update some information ⟨Y1⟩ at the ⟨Model⟩ with the command ⟨C2⟩. |

## 5 Pattern-Based Requirements Documentation

Our method to elicit and document functional requirements for distributed systems (DS) consists of six steps. In Fig. 6, we provide an overview of the steps and the corresponding input and output of each step. For each step, we present examples of validation conditions (VC) to ensure that errors occurring during the application of our method can be identified as early as possible. In addition, we briefly describe the tool which supports the application of our method. In Sect. 6, we provide a case study which exemplifies our method.

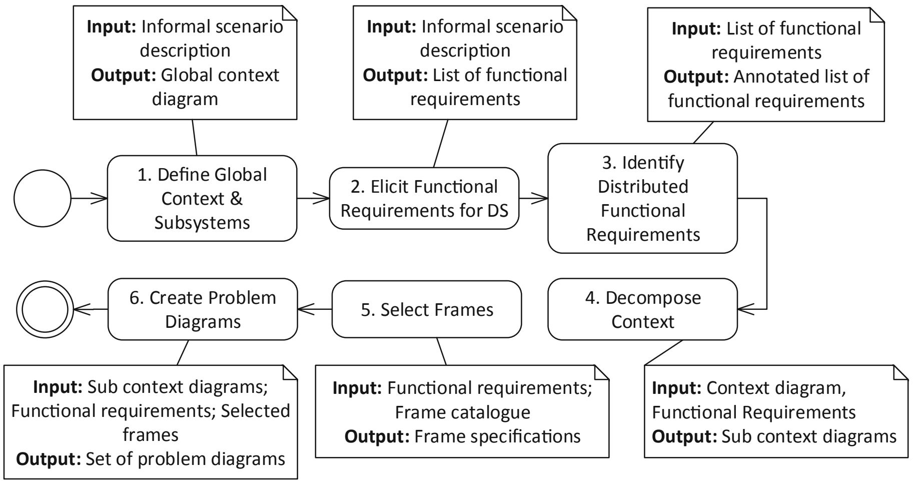

Input: Informal scenario description Output: Global context diagram
Input: Informal scenario description Output: List of functional requirements
Input: List of functional requirements Output: Annotated list of functional requirements
1. Define Global Context & Subsystems
2. Elicit Functional Requirements for DS
3. Identify Distributed Functional Requirements
4. Decompose Context
5. Select Frames
6. Create Problem Diagrams
Input: Sub context diagrams; Functional requirements; Selected frames Output: Set of problem diagrams
Input: Functional requirements; Frame catalogue Output: Frame specifications
Input: Context diagram, Functional Requirements Output: Sub context diagrams

**Fig. 6.** Method overview.

## 5.1 Step 1: Define Global Context and Subsystems

The goal of the first step is to get an understanding of the global context in which the distributed system will operate. We consider an informal scenario description as the initial input. Based on this input, we identify problem domains in the context of the distributed system.

We document the results in a context diagram as described in Sect. 2. There is exactly one distributed system domain (represented by the symbol ⊟ ) in the context diagram which covers all subsystems that shall be developed. Since existing systems do not need to be developed, we describe them by means of causal domains. Using interfaces, we describe the communication between the distributed system and the environmental domains.

For the distributed system, we identify those subsystems that shall be developed. There are at least two subsystems. The subsystems do not necessarily differ from each other. For example, in a peer-to-peer system, the subsystems realized as peers can have the same functional requirements. We represent the subsystems as machine domains with aggregations to the distributed system in the context diagram.

*Validation Conditions.* Based on the description of the step, we define four validation conditions (VC).

- VC1. There is exactly one distributed system in the global context diagram.
- VC2. A distributed system consists of at least two subsystems.
- VC3. All subsystems have been identified and have been documented in the context diagram.
- VC4. All problem domains of the context have been identified, e.g. stakeholders and technical equipment.

*Tool Support.* As mentioned in Sect. 3, we make use of an Ecore model for our tool. To define the initial context and subsystems, we provide a graphical editor based on *Eclipse Sirius*[3]. The editor assists software engineers in creating the initial context diagram and ensures the semantic rules provided by the model.

Our tool supports the automatic validation of *VC1* and *VC2*. The other two conditions have to be validated manually, but we ask the user of the tool to confirm the validation before proceeding to the next step.

## 5.2 Step 2: Elicit Functional Requirements for DS

Based on the informal scenario description and the global context diagram, we identify the functional requirements that the distributed system shall satisfy. For each functional requirement, we define a unique name and a proper description of the expected functionality, and we document both textually.

[3] Eclipse Sirius - https://www.eclipse.org/sirius/ (last access: March 12, 2019).

*Validation Conditions.* For the second step of our method, we define two validation conditions.

VC5. Each functional requirement has a unique name and a valid description.
VC6. Each functional requirement has been identified and has been documented.

*Tool Support.* Our tool provides a table to list all functional requirements one by one. To this table, one can add new requirements using a wizard, and all requirements will be stored in the model to be reusable in further steps.

The first validation condition can partially be checked via the model, whereas the second one has to be confirmed by the user of our tool before proceeding to the next step.

### 5.3 Step 3: Identify Distributed Functional Requirements

Due to different environments in which the subsystems may be realized, e.g. a mobile application in contrast to a server application, different teams will be involved in developing a distributed system. A requirement can be distributed, i.e. it requires the interaction between different subsystems to be satisfied.

In the present step, we mark distributed requirements to assign them to the responsible development team. In addition, we document dependencies of subsystems for satisfying requirements. For each requirement, we decide about its type and assign a set of responsible subsystems. A requirement that concerns at least two subsystems has to be considered as distributed, and in a distributed system there is at least one requirement concerning several subsystems.

*Validation Conditions.* We define two validation conditions for the third step of our method.

VC7. Only requirements concerning at least two subsystems have been classified as distributed.
VC8. At least one requirement has been defined as distributed.

*Tool Support.* To specify the type of requirement, our tool presents the list of requirements to the user where he/she can select the type. For distributed requirements, we provide a dialog to select the related subsystems. Using references to the corresponding machine domains, our tool documents the results in the model and updates the list of requirements.

Both stated validation conditions can be validated automatically using our tool.

### 5.4 Step 4: Decompose Context

In the first step of our method, we described the global context of the distributed system. As mentioned earlier, different teams will be involved in developing a distributed system. In the present step, we break down the global context in smaller units, one for each subsystem. Again, we make use of context diagrams which we call *Sub-Context Diagram* to document the results, one for each subsystem.

Such a sub-context diagram consists of the machine domain for the subsystem and the relevant problem domains. To express the relation between the subsystems, we introduce new elements to the context diagram, namely remote machines (represented by the symbol 🖳 ) and remote interfaces (dotted line). For each related subsystem with which communication exists, we add a remote machine domain and the corresponding remote interface.

The interfaces between machine and problem domains are taken from the global context definition, but the remote interfaces describing the communication between subsystems do not exist there and hence, need to be added.

The set of sub-context diagrams helps developers in focusing on the context of a concrete subsystem. However, we still document the relation to other subsystems.

*Validation Conditions.* To validate the application of the fourth step, we define the following five conditions:

VC9. There is one context diagram for each subsystem.
VC10. Each domain of the initial context diagram is contained in at least one context diagram of a subsystem.
VC11. Interfaces between machine and remote machine have been marked as *remote*.
VC12. Each context diagram contains all related subsystems represented by means of remote machine domains.
VC13. Only problem domains directly connected to the subsystem or via a connection domain are part of the context diagram.

*Tool Support.* Our tool automatically creates a sub-context diagram for each subsystem. It automatically adds related machines based on the requirement classifications taken from step three and the remote interfaces in-between. We also provide a wizard to select relevant problem domains, phenomena, and interfaces from the initial context. A graphical editor allows adjusting the generated diagrams. To ensure consistency between all steps, we make use of model references to the results of the previous steps.

Except for the last one, our tool allows to automatically evaluate the validation conditions. For the last step, it asks the user to confirm the manual validation.

### 5.5 Step 5: Select Frames

In the fifth step, we select suitable frames to describe the functional requirements. There are two types of frames: (i) problem frames (cf. Sect. 2.2 and [4]) and (ii) distributed frames (see Sect. 4). For non-distributed requirements, we consider problem frames and for distributed requirements, we consider distributed frames. Since the requirements have been documented in natural language, the selection requires manual effort. The specifications we provide for distributed frames help engineers in selecting appropriate frames, e.g. by considering the described context. A functional requirement is not necessarily restricted to a single frame. In some cases, it could be necessary to combine frames.

In case no suitable frame or combination exists, a new frame has potentially been identified. That frame has to be documented in the frame catalog using our template. This way, knowledge can be captured for further development projects.

*Validation Conditions.* There are three validation conditions for the fifth step:

VC14. For each distributed requirement, at least one distributed frame has been selected.
VC15. For each non-distributed requirement, at least one problem frame has been selected.
VC16. New frames have been added to the catalogue.

*Tool Support.* Currently, our tool does not support any frame specification (see Sect. 8). Therefore, the frame selection requires manual interaction based on the catalogue of frames and identified requirements.

### 5.6 Step 6: Create Problem Diagrams

The final step of our method is the creation of problem diagrams for the functional requirements we identified in the second step. For requirements not being classified as distributed, we create problem diagrams as proposed by Michal Jackson [10] based on the sub-context diagram for the responsible subsystem. To specify an interface in more detail, it is possible to add connection domains, e.g. a user interface.

To create problem diagrams, we instantiate the frame diagrams of the frames we selected in the previous step. To specify the interfaces between domains in more detail, we annotate its type according to the access vector as introduced in Sect. 3.

For requirements being classified as distributed, we create one problem diagram per involved subsystem. Those diagrams contain the relevant problem domains taken from the sub-context diagram and remote machines for subsystems related to the functional requirement. To connect machine and remote machines, we again make use of remote interfaces. The textual requirement description can be created by instantiating the corresponding textual pattern.

A distributed requirement is characterized by the communication between machine and remote machine for its satisfaction. Therefore, the requirement *refers to* or *constrains* at least one phenomenon of a remote machine. *Refers to* means that the remote machine triggers an event of the machine to be considered, and *constrains* means that the machine to be considered triggers an event of the remote machine. The annotated phenomenon describes that event.

*Validation Conditions.* For the final step of our method, we define four validation conditions.

VC17. Each functional requirement is contained in at least one problem diagram.
VC18. For each distributed requirement, there is a problem diagram for each involved subsystem.
VC19. A distributed requirement *refers to* or *constrains* at least one phenomenon of a remote machine.
VC20. The problem diagram is an instance of the corresponding frame diagram.

*Tool Support.* Using our tool, users can generate problem diagrams for each requirement and each subsystem, respectively. The initial structure of the diagrams can be generated automatically, i.e. requirement and machine. In addition, we provide a wizard that assists users of the tool in selecting relevant problem domains and interfaces from the model, and in adding connection domains. Again, we use references to existing model elements to ensure consistency between all diagrams.

Our tool can evaluate all validation conditions automatically, except the last one since the frame specifications are currently not part of the model.

### 5.7 Final Output

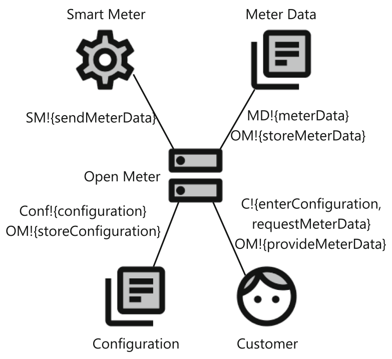
The final output of our method is a set of diagrams for each subsystem. The set consists of a context diagram for the subsystem and problem diagrams which describe the functional requirements to be satisfied by the subsystem. The set allows independent development of each system while still preserving dependencies to other subsystems. Since we document the results in one model, changes will be propagated throughout all method steps and diagrams.

## 6 Example

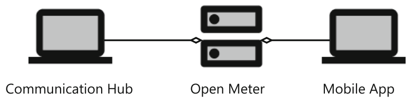
In the following, we apply our method to a part of a smart grid case study. The diagrams and tables we show in the following have been created with our tool.

### 6.1 Informal Scenario Description

For the present paper, we focus on a small part of the overall scenario that concerns the customer's home. The initial scenario description is as follows: The communication hub is the central gateway, for which software shall be developed. Smart meters measure the customer's power consumption. They transmit the data in given intervals to the communication hub where the data is stored. In addition, a customer can connect to the communication hub using a mobile application on a smartphone or tablet. Customers can configure the mobile application to connect to their communication hub and can then request a list of stored meter data.



(a) Global context diagram [18]



(b) Subsystems [18]

**Fig. 7.** Case study - global context diagram & subsystems.

## 6.2 Step 1: Define Global Context and Subsystems

Our distributed system is called *Open Meter* 🖥, for which we present the global context diagram in Fig. 7a. We identified the stakeholder *Customer* ☺, who is able to enter a *Configuration* 🗐 for the mobile application and who can request previously stored meter data. We consider a *Smart Meter* ⚙ as existing technical equipment. Measured data will be stored persistently in the database which we call *Meter Data* 🗐.

In Fig. 7b on the right hand-side, we also provide an overview of the different subsystems that shall be developed. Our distributed system consists of two subsystems: The *Communication Hub* 💻 will be realized as an embedded system for the gateway at customers' home. The *Mobile App* 💻 will be realized as software for smartphones and tablets.

## 6.3 Step 2: Elicit Functional Requirements for DS

For our scenario, we consider three functional requirements:

**Enter Configuration.** Customers can configure the mobile application to connect to the communication hub.

**Request Meter Data.** Customers can request a list of their meter data via the mobile application.

**Store Meter Data.** In given intervals, smart meters send the measured data to the communication hub, where it is stored persistently.

## 6.4 Step 3: Identify Distributed Functional Requirements

Next, we identify those requirements that concern more than one subsystem.

**Enter Configuration.** Customers enter the configuration locally in the mobile application. There is no communication with other systems and therefore, the requirement is not considered as distributed.

**Request Meter Data.** To request the meter data, customers use their mobile application to access the communication hub. The communication hub then returns the stored data. Both subsystems are involved in that process, and therefore we consider the requirement as distributed.

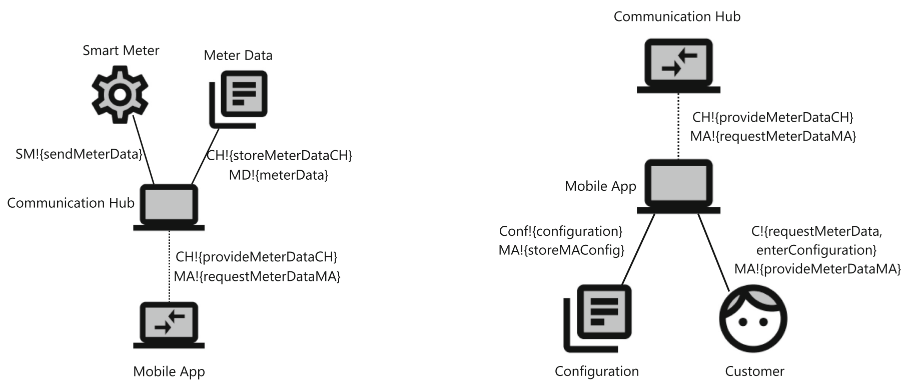
**Store Meter Data.** Smart meters connect to a communication hub. There is no interaction with other subsystems.

## 6.5 Step 4: Decompose Context

Our scenario contains two subsystems, *Communication Hub* and *Mobile Application*. Hence, it is necessary to define one sub-context diagram for each.

**Communication Hub.** Figure 8a shows the context diagram for the *Communication Hub* . The domain *Meter Data* represents the database where the communication hub stores the measured data persistently, and a *Smart Meter* sends the measured data. Since the *Mobile App* is also part of the distributed system, it is represented as a remote machine. The interface between both subsystems is unreliable and therefore marked as a remote interface.

**Mobile Application.** For the *Mobile App* , we develop the context diagram in Fig. 8b. It consists of the *Customer* who uses the application, a *Configuration* , and the *Communication Hub* , which is again connected to the machine with a remote connection. There are phenomena to enter the configuration and to request meter data.



**(a)** Sub-context diagram for *Communication Hub* [18]

**(b)** Sub-context diagram for *Mobile Application* [18]

**Fig. 8.** Case study - sub-context diagrams.

## 6.6 Step 5: Select Frames

For the three requirements of our scenario, we select the following frames:

**Enter Configuration.** The requirement is non-distributed one. Since the customer (biddable domain) can enter the configuration (lexical domain), we select the problem frame *Simple Workpieces* [10].

**Request Meter Data.** The requirement is also not distributed. A smart meter (causal domain) stores the meter data (lexical domain). We choose the problem frame *Model Building* [10].

**Store Meter Data.** Store meter data is a distributed requirement. Therefore, we select a distributed frame. The requirement fits to the frame *Query (PF)*, since a customer (biddable domain) can request data from a remote resource (lexical domain).

In the next step, we create the diagrams according to the corresponding frame diagram.

## 6.7 Step 6: Create Problem Diagrams

There are three functional requirements in our scenario for which we present the corresponding problem diagrams in the following.

**Enter Configuration.** Since the requirement *Enter Configuration* is not a distributed requirement, there is only one problem diagram. It is an instance of the problem frame *Simple Workpiece* and consists of the *Customer*, the *Mobile App*, and the *Configuration*. In addition, we decided to make the *User Interface* of the mobile application explicit.

The interface between customer and user interface is physical (P). The interfaces between user interface and mobile application, and between mobile application and configuration are both local (L).

The requirement *Enter Configuration* constrains the phenomenon of the *Configuration* and refers to the phenomenon of the *Customer*. We show the problem diagram in Fig. 9.

**Store Meter Data.** We show the problem diagram for the requirement *Store Meter Data* in Fig. 10. It is an instance of problem frame *Model Building* and consists of the *SmartMeter*, the *Communication Hub*, and the *Meter Data*.

Since a smart meter uses the local network to communicate with the communication hub, the interface is classified as adjacent (A). Between communication hub and meter data, there is a local interface.

The requirement constrains the phenomenon of the *Meter Data* and refers to the phenomenon of the *Smart Meter*.

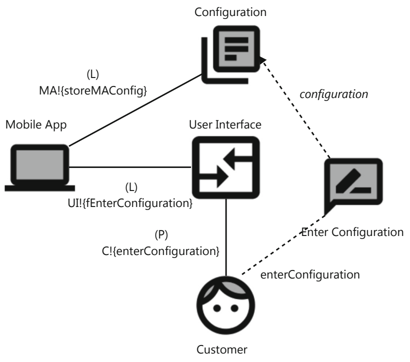


**Fig. 9.** Case study - problem diagram for *Enter Configuration* [18].

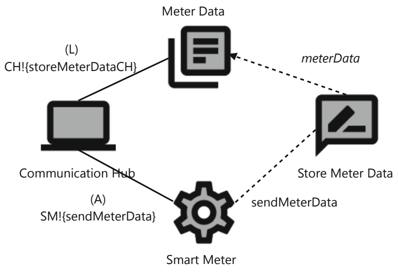
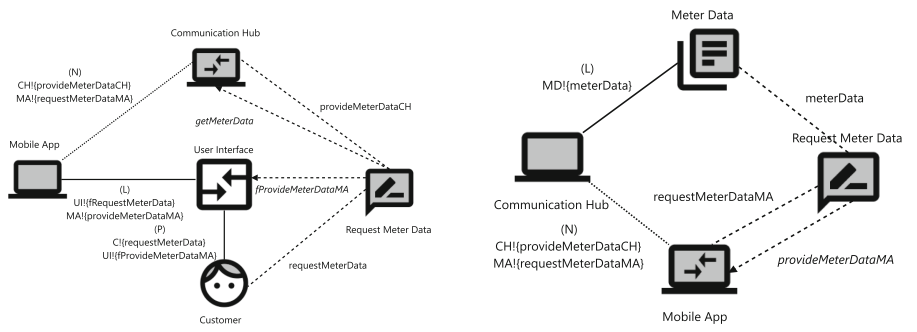


**Fig. 10.** Case study - problem diagram for *Store Meter Data* [18].

**Request Meter Data.** We identified the requirement *Request Meter Data* as distributed, because it concerns both subsystems. Therefore, we create problem diagrams for the *Communication Hub* and for the *Mobile App*. They are an instance of the problem frame *Query (DF)*.

**Mobile Application.** In Fig. 11a, we show the problem diagram for the requirement *Request Meter Data* with regard to the *Mobile App*. It contains the machine, the *Customer* who initiates the request, the *User Interface*, and the remotely connected *Communication Hub*.
Between customer and user interface, we again consider a physical interface (P), and between user interface and mobile app, there is a local interface (L). Since mobile application and communication hub can communicate via the internet, the interface is annotated with network (N).
The requirement refers to the phenomenon *enterConfiguration* of the *Customer* and to the phenomenon *provideMeterDataCH* of the remote machine. It constrains the phenomenon *getMeterData* representing the event to retrieve the data from the database, and the phenomenon *fProvideMeterDataCH* of the *User Interface* representing the feedback for the customer.



**(a)** Problem diagram for *Mobile App* [18]

**(b)** Problem diagram for *Communication Hub* [18]

**Fig. 11.** Problem diagram for *Request Meter Data* - Query (DF).

**Communication Hub.** We show the problem diagram for the *Communication Hub* in Fig. 11b. It consists of the machine, the *Meter Data*, and the remotely connected *Mobile App*.
The types of interfaces are the same as in the previous diagrams.
The requirement refers to the phenomenon of the *Meter Data* and to the phenomenon *requestMeterDataMA* of the *Mobile App*. In addition, the requirement constrains the phenomenon *provideMeterDataMA*, since the *Communication Hub* initiates the event to provide the meter data to the customer.

The created diagrams which have been documented in the model can now be used for further analysis, e.g. with regard to security (cf. Sect. 8).

## 7 Related Work

In the following, we present related work that follows similar approaches or that may complement our work.

Haley argues that the problem frames notation does not allow to specify a *limited to many relation* between interfaces [7]. Therefore, the author suggests using cardinalities on interfaces. Cardinalities would extend our notation to be more precise in specifying the relations between the different subsystems, e.g. to state the number of concurrent instances.

The same author introduces so-called projection domains to document relations between different units of distributed architectures [8]. The approach neither provides detailed documentation of the context for each subsystem nor a method to systematically identify overlapping requirements.

Gol Mohammadi et al. propose a framework to combine goal-oriented requirements engineering with problem frames [11]. The proposed framework allows extending problem and context modeling approaches with soft-goals, e.g. for security. Using the framework in our method is a promising way to improve the context definition.

To decompose the requirements of a distributed system, Penzenstadler defines a catalog of criteria [12]. There are criteria for context, functionalities, and design of software. The presented catalog may help to further describe the subsystems we identified with our method. Therefore, it may complement our work.

Beckers and Faßbender describe a pattern-based approach for capturing quality requirements like performance [1] in distributed systems. Since we focus on functional requirements, the proposed pattern and our method can complement each other.

There are many design patterns that have been identified in the context of distributed systems (e.g., [2,5,9]). Some of those patterns capture aspects like security, as well. Currently, our method only addresses requirements engineering. By mapping distributed frames to appropriate design patterns, we can assist the design phase during software development.

Finally, Ramachandran and Mahmood discuss the state of the art in requirements engineering for distributed computing [13]. The authors put a special focus on cloud computing which became very popular in the last years. Currently, we do not focus on any specific type of distributed system. Their work may solve as an input to further analyze distributed frames in the context of cloud computing.

## 8 Conclusion

**Summary.** In this paper, we presented a pattern-based approach to characterize functional requirements for distributed systems. Following Jackson's problem frames approach, we introduced the so-called *Distributed Frames*. We first introduced a common template format to specify them. Next, we presented several examples of such frames using our template.

In addition, we provided an extension of our *RE4DIST* method that takes our distributed frames into account. The extension allows a pattern-based documentation of functional requirements, and functional requirements can systematically be described by instantiating a suitable frame. Our proposed Ecore model ensures consistency and traceability between the different steps of the method.

Finally, we exemplified the extended method and the application of our patterns based on a small case study.

**Future Work.** We plan to extend our tool which we developed in previous work [18]. We will embed the pattern catalog into the tool to support the selection and instantiation of appropriate frames. A frame instance can then be stored in the Ecore model which we presented in Sect. 3.

Currently, our distributed frames are only a small set of relevant patterns for requirements. We will go on with identifying additional frames, and we plan to make the catalog publicly available so that others can contribute, as well. Furthermore, we plan to develop a pattern system. In this system, each distributed frame can be further refined, for example, to capture specific aspects for Peer-to-Peer systems.

Due to unreliable connections between the different subsystems and continuous exchange of information, security and privacy are of special importance for distributed systems. With our method, we allow making those connections explicit. In previous work, we mapped security incidents to functional requirements [17]. We will extend the mapping with regard to distributed frames, and we will investigate in more detail how relevant threats can be identified automatically.

## References

1. Beckers, K., Faßbender, S.: Peer-to-peer driven software engineering considering security, reliability, and performance. In: 7th International Conference on Availability, Reliability and Security, pp. 485–494, August 2012. https://doi.org/10.1109/ARES.2012.26
2. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture, 4th edn. Wiley SEries in Software Design Patterns, Wiley (2007). http://www.worldcat.org/oclc/314792015
3. Choppy, C., Heisel, M.: Une approache à base de patrons pour la spécification et le développement de systèmes d'information. Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL (2004)
4. Côté, I., Hatebur, D., Heisel, M., Schmidt, H., Wentzlaff, I.: A systematic account of problem frames. In: Hvatum, L.B., Schümmer, T. (eds.) Proceedings of the 12th EuroPLoP, Irsee, Germany, 4–8 July 2007, pp. 749–768. UVK - Universitaetsverlag Konstanz (2007). http://hillside.net/europlop/europlop2007/workshops/D3.pdf
5. Fernandez-Buglioni, E.: Security Patterns in Practice: Designing Secure Architectures Using Software Patterns, 1st edn. Wiley Publishing, New York (2013)
6. FIRST.org: Common Vulnerability Scoring System v3.1: Specification Document (2019)
7. Haley, C.B.: Using problem frames with distributed architectures: a case for cardinality on interfaces. In: Proceedings of the 2nd International Software Requirements to Architectures Workshop (STRAW 2003), May 2003. http://oro.open.ac.uk/3394/
8. Haley, C.B., Laney, R.C., Nuseibeh, B.: Using problem frames and projections to analyze requirements for distributed systems. In: Proceedings of the 10th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ 2004), June 2004. http://oro.open.ac.uk/3393/
9. Hendrikx, K., Olivié, H.J., Duval, E.: Design patterns for distributed information systems. In: Dyson, P., Devos, M. (eds.) Proceedings of the 4th EuroPLoP, Germany, 7–11 July 1999, pp. 47–56. UVK - Universitaetsverlag Konstanz (1999). http://web.archive.org/web/20031011072203/. http://www.argo.be/europlop/Papers/Final/Hendrickx.ps
10. Jackson, M.A.: Problem Frames - Analysing and Structuring Software Development Problems. Pearson Education (2000). http://www.pearsoned.co.uk/Bookshop/detail.asp?item=100000000004768
11. Mohammadi, N.G., Alebrahim, A., Weyer, T., Heisel, M., Pohl, K.: A framework for combining problem frames and goal models to support context analysis during requirements engineering. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E., Xu, L. (eds.) CD-ARES 2013. LNCS, vol. 8127, pp. 272–288. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40511-2_19

12. Penzenstadler, B.: DeSyRe: decomposition of systems and their requirements: transition from system to subsystem using a criteria catalogue and systematic requirements refinement. Ph.D. thesis, Technical University Munich (2010). http://mediatum.ub.tum.de/node?id=999357
13. Ramachandran, M., Mahmood, Z. (eds.): Requirements Engineering for Service and Cloud Computing. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51310-2
14. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional, Boston (2009)
15. Tanenbaum, A.S., Steen, M.V.: Distributed Systems: Principles and Paradigms, 2nd edn. Prentice-Hall Inc., Upper Saddle River (2006)
16. Wentzlaff, I., Specker, M.: Pattern-based development of user-friendly web applications. In: Proceedings of the 2nd International Workshop on Model-Driven Web Engineering (MDWE 2006), Palo Alto, USA. ACM (2006)
17. Wirtz, R., Heisel, M.: A systematic method to describe and identify security threats based on functional requirements. In: Zemmari, A., Mosbah, M., Cuppens-Boulahia, N., Cuppens, F. (eds.) CRiSIS 2018. LNCS, vol. 11391, pp. 205–221. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-12143-3_17
18. Wirtz, R., Heisel, M.: RE4DIST: model-based elicitation of functional requirements for distributed systems. In: van Sinderen, M., Maciaszek, L.A. (eds.) Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019, Prague, Czech Republic, 26–28 July 2019, pp. 71–81. SciTePress (2019). https://doi.org/10.5220/0007919200710081