# Problem-Based Requirements Interaction Analysis*

Azadeh Alebrahim, Stephan Faßbender, Maritta Heisel, Rene Meis

paluno - The Ruhr Institute for Software Technology – University of Duisburg-Essen
firstname.lastname@paluno.uni-due.de

**Abstract. [Context]** The ability to address the diverse interests of different stakeholders in a software project in a coherent way is one fundamental software quality. These diverse and maybe conflicting interests are reflected by the requirements of each stakeholder. **[Problem]** Thus, it is likely that aggregated requirements for a software system contain interactions. To avoid unwanted interactions and improve software quality, we propose a structured method consisting of three phases to find such interactions. **[Principal ideas]** For our method, we use problem diagrams, which describe requirements in a structured way. The information represented in the problem diagrams is translated into a formal Z model. Then we reduce the number of combinations of requirements, which might conflict. **[Contribution]** The reduction of requirements interaction candidates is crucial to lower the effort of the in depth interaction analysis. For validation of our method, we use a real-life example in the domain of smart grid.

**Keywords:** Requirements interactions, problem frames, feature interaction, Z notation

## 1 Introduction

Nowadays, for almost every software system various stakeholders with diverse interests exist. These interests give rise to different sets of requirements. The combination of these sets leads to unwanted *interactions* among the requirements. Such interactions among requirements cannot be detected easily.

In general, the deviation between the intended behavior and structure as formulated by single requirements of a stakeholder and the overall behavior and structure of the resulting system- or software-to-be is called requirement *inconsistency* [1,2]. Such inconsistencies can stem from different sources. The *first source* is the different understanding of terms and different views on the system-to-be of different stakeholders. Missing or misleading information also adds to this class of inconsistencies [1], [3]. A *second source* are inconsistencies which stem from the transformation between different kinds of representations and models [1]. *Another important source* are interactions between requirements which lead to an unexpected behavior. For functional requirements this source is already known as feature interaction for a long time, e.g. in the

* Part of this work is funded by the German Research Foundation (DFG) under grant number HE3322/4-2 and the EU project Network of Excellence on Engineering Secure Future Internet Software Services and Systems (NESSoS, ICT-2009.1.4 Trustworthy ICT, Grant No. 256980).

domain of telecommunication [4,5]. For interactions, one can distinguish between unwanted and desirable interactions. The strongest type of interactions are conflicts in which requirements deny each other, and dependencies where one requirement can be only fulfilled when another requirement is also fulfilled. Between these extrema, there are different shades of negative or positive influences [2], [6]. For this paper, we assume that inconsistencies in terms of the first and second source are solved and we will focus on conflicts. But our method also allows to find other kinds of interactions.

Requirements engineering is concerned with describing the problem that the software has to solve in a precise way [7]. The problem is located in the environment in which the machine will be integrated and not in the computer [8]. Therefore, reasoning about the requirements involves reasoning about the environment and the assumptions made about it [7]. Zave and Jackson define the three terms *requirements (R)*, *domain knowledge (D)*, and *specification (S)* in their extensive work [9]. The requirements describe the desired system after the machine is built. The domain knowledge represents the relevant parts of the problem world. The specifications describe the behavior of the machine in order to meet the requirements. These three descriptions are related through the entailment relationship $D, S \models R$, expressing that the specification within the context of the domain knowledge should satisfy the requirements.

As a basis for requirements analysis, we use the problem frames approach [8] based on the work of Zave and Jackson [9]. It suggests to decompose the overall software problem into simple subproblems. Each subproblem is related to one or more requirements. The solutions of the subproblems will be composed to solve the overall software problem. The composition will only be successful if there is a consistent set of requirements (subproblems). Therefore, the identification of interactions and inconsistencies in the requirements analysis is essential to avoid costly modifications later on in the software development life cycle and to improve the overall software quality.

In this paper, we propose a formal and structured method composed of three phases to identify interactions among functional requirements involving the environment. We start with a full set of requirements representing subproblems. In all three phases, we narrow down the set of combinations of requirements which might interact. The narrowing process is formally defined using the Z notation [10,11] for each step of our method. The formal Z specification is the basis for the tool support of our method. We developed our specifications using the Community Z Tools[1]. After the final phase, the remaining candidates have to be analyzed in detail to choose appropriate measures to avoid the interactions. The analysis of the remaining candidates is out of the scope of this paper, but our method reduces the amount of candidates and thus the effort necessary for further analysis.

The rest of the paper is organized as follows: Section 2 gives a brief overview of the problem frames our method relies on. As running example, we introduce a sun blind control system in Section 3. Our method to detect interacting requirements is described and applied to the sun blind control system in Section 4. We validate our method by using a real-life example of smart grids in Section 5. Section 6 presents related work, while Section 7 concludes the paper and suggests recommendations for future work.

[1] http://czt.sourceforge.net/

## 2 Background

We use the problem frames [8] approach proposed by Jackson to build our requirements interaction method on. Jackson introduces the concept of *problem frames*, which is concerned with describing, analyzing, and structuring software problems. According to Jackson, the computer and the software represent the solution, called *machine*. The requirements and the environment are called *system*.

A problem frame represents a class of software problems. It is described by a *frame diagram*, which 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. *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 "!". For instance, the shared phenomena *openCommand*, *closeCommand*, and *stopCommand* in Fig. 1 are observable by the domains *UserOpenControl* and *User*, but controlled only by the domain *User*. When we state a requirement, we want to change something in the world with the machine (i.e., 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. Hence, a constrained domain triggers the need for developing a new machine which provides the desired control. A requirement may refer to several domains in the environment of the machine.

We describe problem frames using UML class diagrams, extended by stereotypes as proposed by Hatebur and Heisel [12]. All elements of a problem frame act as placeholders which must be instantiated to represent concrete problems. In doing so, one obtains a *problem diagram* that belongs to a specific class of problems. Figure 1 shows a problem diagram in UML notation. It describes that the *UserOpenControl* machine pulls up, lowers or stops the sun blind on behalf of user commands *openCommand*, *closeCommand*, or *stopCommand*. The requirement R4 constrains the *SunBlind* domain. This is expressed by a dependency with the stereotype ≪`constrains`≫. It refers to the *User*, as expressed by a dependency with the stereotype ≪`refersTo`≫.

## 3 Running Example

We demonstrate our approach using a sun blind control system. A sun blind is made up of metallic fins which are attached to the outer side of the window. Additionally, we have a sun sensor which measures the sun intensity, a wind sensor which measures the wind speed, and a display which is suitable to display the current sun intensity and wind speed. The sun blind is sensitive to sun and wind. A machine shall be built that lowers the sun blind on sunshine and pulls it up on strong wind. For individual settings it shall be possible to control the sun blind manually, too. The following requirements are given:

(R1) If there is sunshine for more than one minute, the sun blind will be lowered.
(R2) If there is no sunshine for more than 5 minutes, the sun blind will be pulled up.

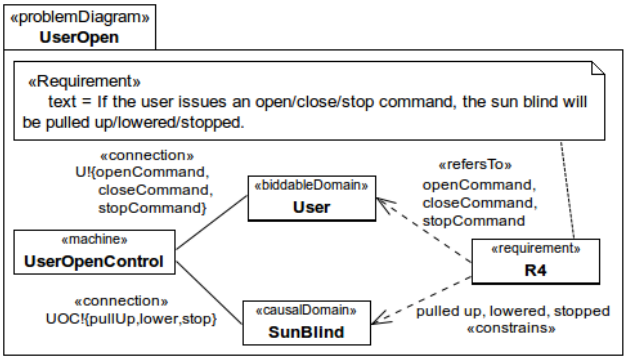


**Fig. 1.** Problem diagram for the requirement R4

(R3) If there is strong wind for more than 10 seconds, the sun blind will be pulled up, to avoid destruction of the sun blind.
(R4) If the user issues an open/close/stop command, the sun blind will be pulled up/-lowered/stopped.
(R5) If the user interacts with the sun blind, then sunshine and no sunshine are ignored within the next 4 hours.
(R6) If the user deactivates the holiday mode, then the sun blind is turned on.
(R7) If the user activates the holiday mode, the sun blind is pulled up and turned off.
(R8) Sunshine intensity and wind speed shall be displayed on the weather display.

We modeled the requirements as problem diagrams, which are used as input for our method. The problem diagram for the requirement R4 is shown in Fig. 1. The other requirements are modeled in a similar way. Throughout the paper we will refer to this example to describe the proposed method.

## 4 Interaction detection method

Our method starts with a set of problem diagrams. Based on the information provided by these problem diagrams, the structure-based pruning (phase one) takes place and removes all requirements for which the structure of problem diagrams already implies that they will not interact. The result is a first set of interaction candidates. In the second phase, those candidates can be further reduced using the information if requirements have to be satisfiable in parallel (phase two). The sets of requirements that have to be satisfiable in parallel have to be known beforehand and these are an external input to our method. The remaining interaction candidates are finally reduced in the last phase (phase three) by checking whether the conjunction of preconditions of possibly interacting requirements is satisfiable. Our formalization is built on the following sets and relations that can be derived from the given problem diagrams.

*Requirement* is the set of all requirements occurring in at least one problem diagram.
*Domain* is the set of all domains occurring in at least one problem diagram.

*Phenomenon* is the set of all phenomena which are referred to or constrained by at least one requirement in a problem diagram.

$constrains : Requirement \times Domain \rightarrow \mathbb{P}\, Phenomenon$ is the function that assigns to a pair of requirement $r$ and domain $d$ the set of phenomena $P$ that $r$ constrains on $d$.

$refersTo : Requirement \times Domain \rightarrow \mathbb{P}\, Phenomenon$ is the function that assigns to a pair of requirement $r$ and domain $d$ the set of phenomena $P$ that $r$ refers to on $d$.

Please note that *constrains* and *refersTo* are total functions. If a requirement does not constrain or refer to a domain, then the value of the respective function is an empty set of phenomena. In Z notation [10,11], we define the above sets and relations as follows:

$$[Requirement, Domain, Phenomenon]$$

$$constrains : Requirement \times Domain \rightarrow \mathbb{P}\, Phenomenon$$
$$refersTo : Requirement \times Domain \rightarrow \mathbb{P}\, Phenomenon$$

### 4.1 Phase One: Structure-Based Pruning

In phase one, we make use of the structure of the problem diagrams. The steps for selecting the requirements which are candidates of a requirements interaction are described as follows:

**Step One: Initial Setup** First, we define a Z schema *Interaction* consisting of three variables, which we will use to describe the actual state of our method. The sets *RelevantDomain* and *RelevantRequirement* contain all domains and requirements which are considered to be relevant for an interaction. The function *MinReqInteraction* returns for each relevant domain the minimal sets of requirements that may interact with each other. A set of requirements is considered as minimal interacting if each strict subset of it does not contain a possible interaction [2].

*Interaction*

$$RelevantDomain : \mathbb{P}\, Domain$$
$$RelevantRequirement : \mathbb{P}\, Requirement$$
$$MinReqInteraction : Domain \nrightarrow \mathbb{P}\,\mathbb{P}\, Requirement$$

---

$$\mathrm{dom}\ MinReqInteraction = RelevantDomain$$
$$\bigcup(\bigcup(ran\ MinReqInteraction)) = RelevantRequirement$$
$$\forall d : RelevantDomain \bullet \forall R : MinReqInteraction(d) \bullet$$
$$\quad \#R \geq 2 \land \forall Q : MinReqInteraction(d) \mid R \neq Q \bullet \neg\, Q \subseteq R$$

Initially, we assume that all pairs of requirements which constrain the same domain possibly cause an interaction on it. Formally, we define the initial interaction schema *Init* as follows:

*Init*

$$Interaction$$

---

$$RelevantDomain = Domain$$
$$\forall d : Domain \bullet MinReqInteraction(d) =$$
$$\quad \{r_1, r_2 : Requirement \mid r_1 \neq r_2 \land$$
$$\qquad constrains\ (r_1, d) \neq \varnothing \land constrains\ (r_2, d) \neq \varnothing \bullet \{r_1, r_2\}\}$$

**Table 1.** Initial requirements interaction table

| Requirement / Domain | SunSensor (CausalDomain) | SunBlind (CausalDomain) | WindSensor (CausalDomain) | User (Biddable-Domain) | WeatherDisplay (CausalDomain) |
|---|---|---|---|---|---|
| R1 | *sunshine* | **lowered** | | | |
| R2 | *no sunshine* | **pulled up** | | | |
| R3 | | **pulled up** | *heavy wind* | | |
| R4 | | **pulled up, lowered, stopped** | | *openCommand* , *closeCommand* , *stopCommand* | |
| R5 | | **on, off** | | *openCommand, closeCommand, stopCommand* | |
| R6 | | **off, pulled up** | | *activateHoliday* | |
| R7 | | **on** | | *deactivate-Holiday* | |
| R8 | *sun intensity* | | *wind speed* | | **displayed sunshine intensity, displayed wind speed** |

We will visualize our method using so called requirement interaction tables (see Table 1). In these tables, we represent the functions *constrains* and *refersTo* with the restricted domain $RelevantRequirement \times RelevantDomain$. We highlight the phenomena $P$ of a cell $(r, d)$ in bold font if requirement $r$ constrains phenomena $P$ of the domain $d$. If $r$ refers to phenomena $P$, then they are written in italic font. The initial interaction table for our running example is given in Table 1. We start with 21 possible combinations of interacting requirements because we have to assume that each combination of the seven requirements constraining the sun blind causes an interaction.

**Step Two: Reducing Relevant Domains** We check for each column, and therefore for each domain, if the domain is constrained at least by two requirements (at least two cells with bold entries). If this is not the case, then the domain is not relevant. The reason is that interactions only occur on domains which are constrained by at least two requirements. Formally, we can define this step with the following Z operation schema.

$$
\begin{array}{|l}
\hline
P1S2 \\
\Delta Interaction \\
\hline
RelevantDomain' = \{d : Domain \mid \exists\, r_1, r_2 : Requirement \mid r_1 \neq r_2 \bullet \\
\quad constrains\ (r_1, d) \neq \varnothing \land constrains\ (r_2, d) \neq \varnothing\} \\
MinReqInteraction' = RelevantDomain' \lhd MinReqInteraction \\
\hline
\end{array}
$$

From Table 1 we can see that no requirements interactions can occur on the sun sensor, wind sensor and user because these domains are only referred to by the requirements in the problem diagrams. Since there is only one requirement constraining the weather display, we also do not expect any requirements interactions on it. On the

**Table 2.** Requirements interaction table after step 2 of phase 1

| Requirement / Domain | SunBlind (CausalDomain) |
|---|---|
| R1 | **lowered** |
| R2 | **pulled up** |
| R3 | **pulled up** |
| R4 | **pulled up, lowered, stopped** |
| R5 | **on, off** |
| R6 | **off, pulled up** |
| R7 | **on** |

sun blind domain, we expect requirements interactions from the table because every requirement besides R8 constrains this domain. Hence, after the second step of phase one, we only identify domain *SunBlind* as relevant for interactions. When we apply the operation schema $P1S2$ on the initial interaction schema ($Init \fatsemi P1S2$), we get the interaction table shown in Table 2.

**Step Three: Reducing Relevant Requirements** Third, we have to check for each phenomenon of a relevant domain if it is interacting with a combination of phenomena of the interaction table which refer to or constrain the same domain. A set of phenomena is interacting if it is not possible to observe them or different characteristics of them at the same time. Please note that different characteristics of a phenomenon could not be observable at the same time. In such cases, we consider these phenomena as self-conflicting. For each combination, we have to decide whether we can reject the assumption that there is an interaction or not. If we cannot reject this assumption for sure, we have to consider this combination of phenomena as interacting. We are interested in the interactions of phenomena of a domain because these are the source of interactions between requirements that refer to or constrain them. We define the set of the minimal interacting sets of phenomena for each domain using the function $MinPhenInteraction : Domain \rightarrow \mathbb{P}\,\mathbb{P}\,Phenomenon$. A set $P \in MinPhenInteraction(d)$ contains a number of interacting phenomena of the domain $d$ - at least two - and each strict subset $Q \subset P$ is free of interactions [2]. This is expressed in the Z notation as follows:

$$
\begin{array}{|l}
MinPhenInteraction : Domain \rightarrow \mathbb{P}\,\mathbb{P}\,Phenomenon \\
\hline
\forall\, d : Domain \bullet \forall\, P : MinPhenInteraction(d) \bullet \\
\quad P \neq \varnothing \wedge \forall\, Q : MinPhenInteraction(d) \mid P \neq Q \bullet \neg\, Q \subseteq P
\end{array}
$$

We have to define the function *MinPhenInteraction* manually for the given problem frames model. For this step it is sufficient to define it for the relevant domains of the interaction schema after step two of phase one ($(Init \fatsemi P1S2).RelevantDomain'$). For the precondition analysis in phase three, we also need the interacting requirements on the other domains. For our running example, we identify the following sets of minimal interacting phenomena.

$$
\begin{array}{l}
Sunblind, SunSensor, WindSensor, User : Domain \\
lowered, pulledUp, stopped, on, off, sunshine, noSunshine : Phenomenon \\
strongWind, openCommand, closeCommand, stopCommand : Phenomenon \\
\hline
MinPhenInteraction(Sunblind) = \\
\quad \{\{lowered, pulledUp\}, \{lowered, stopped\}, \{pulledUp, stopped\}, \{on, off\}\} \\
MinPhenInteraction(SunSensor) = \{\{sunshine, noSunshine\}\} \\
MinPhenInteraction(WindSensor) = MinPhenInteraction(User) = \varnothing
\end{array}
$$

Using the minimal interacting sets of phenomena, we can update the function *MinReqInteraction* from the interaction schema, which maps a relevant domain to the sets of requirements that possibly interact on the domain. We distinguish two cases. First, two requirements are interacting on a domain if there is a self-conflicting phenomenon where both requirements refer to or constrain. Second, if we can define a bijection between a set of at least two requirements and a set of interacting phenomena, with the property that if the bijection maps a requirement to a phenomenon then the requirement also refers to or constrains it, then these requirements may interact with each other. Formally, we define the following operation schema:

$$
\begin{array}{l}
P1S3 \\
\hline
\Delta Interaction \\
\hline
MinReqInteraction' = \lambda\, d : RelevantDomain \bullet \\
\quad \{r_1, r_2 : Requirement \mid r_1 \neq r_2 \land \exists\, p : Phenomenon \bullet \\
\qquad \{p\} \in MinPhenInteraction(d) \land \\
\qquad p \in constrains\ (r_1, d) \cup refersTo\ (r_1, d) \land \\
\qquad p \in constrains\ (r_2, d) \cup refersTo\ (r_2, d) \bullet \{r_1, r_2\}\} \cup \\
\quad \{R : \mathbb{P}\, Requirement \mid \exists\, P : MinPhenInteraction(d) \bullet \exists\, F : R \rightarrowtail\!\!\!\!\!\rightarrow P \bullet \\
\qquad \forall\, r : R;\ p : P \bullet F(r) = p \Rightarrow p \in constrains\ (r, d) \cup refersTo\ (r, d)\} \\
RelevantDomain' = RelevantDomain \setminus \mathrm{dom}(MinReqInteraction' \rhd \{\varnothing\})
\end{array}
$$

Based on the updated function *MinReqInteraction*′, also the interaction table is reduced by the above operation schema. All requirements that are not in one of the interacting sets can be left out because they are irrelevant. Furthermore, all domains for which no set of possible interacting requirements exists are also irrelevant.

With the above definition of *MinPhenInteraction*(*Sunblind*), we get the following sets of possibly interacting requirements for our running example:

$$
\begin{array}{l}
R1, R2, R3, R4, R5, R6, R7 : Requirement \\
\hline
(\mu\, S : Init \fatsemi P1S2 \fatsemi P1S3).MinReqInteraction'(Sunblind) = \\
\quad \{\{R1, R3\}, \{R1, R4\}, \{R1, R2\}, \{R1, R6\}, \{R2, R4\}, \{R3, R4\}, \{R4, R6\}, \\
\qquad \{R5, R6\}, \{R5, R7\}, \{R6, R7\}\}
\end{array}
$$

In this step, we have reduced the 21 initial combinations of possibly interacting requirements to 10.

### 4.2 Phase Two: Check for Parallel Requirements

We now investigate whether the possibly interacting requirements have to be satisfiable at the same time. In the case that they do not have to be satisfiable all at the same time,

we do not expect interactions among them. Hence, we need a set of sets of requirements *parallelReq* as input for this phase. This set has to be set up manually and shall contain the maximal sets of requirements that have to be satisfiable at the same time, i.e. if we add a new requirement to a set, then there is a requirement in the set that has not to be satisfiable at the same time with the new requirement.

$$parallelReq : \mathbb{P}\,\mathbb{P}\,Requirement$$

$$\bigcup parallelReq \subseteq (\mu\, S : Init \fatsemi P1S2 \fatsemi P1S3).RelevantRequirement'$$

For our running example, we see that R6 and R7 are exclusive requirements that do not have to be satisfiable at the same time with others. Furthermore, R4 and R5 have not to be satisfiable at the same time because R5 refers to the user interaction of R4 as precondition. The requirements R1, R2, and R3 that are concerned with the observation of the environment have all to be satisfiable at the same time, together with R4 or R5. Hence, we get the following sets of parallel requirements:

$$parallelReq = \{\{R1, R2, R3, R4\}, \{R1, R2, R3, R5\}\}$$

We can now use the set of sets of parallel satisfiable requirements to reduce the sets of minimal interacting requirements. A set of requirements is only interacting if all requirements of it have to be satisfiable at the same time. We get the following operation schema:

*P2*

$$\Delta Interaction$$

$$MinReqInteraction' = \lambda\, d : RelevantDomain \bullet \{R : MinReqInteraction(d) \mid \exists\, P : parallelReq \bullet R \subseteq P\}$$
$$RelevantDomain' = RelevantDomain \setminus \mathrm{dom}(MinReqInteraction' \rhd \{\varnothing\})$$

For our running example, we get the following reduced set of sets of minimal interacting requirements.

$$(\mu\, S : Init \fatsemi P1S2 \fatsemi P1S3 \fatsemi P2).MinReqInteraction'(Sunblind) = \{\{R1, R3\}, \{R1, R4\}, \{R1, R2\}, \{R2, R4\}, \{R3, R4\}\}$$

Beginning from the 21 combinations of possibly interacting requirements, we have now reduced the number of relevant combinations to 5.

### 4.3 Phase Three: Precondition-based Pruning

We now have a reduced set of sets of possibly interacting requirements $(Init \fatsemi P1S2 \fatsemi P1S3 \fatsemi P2).MinReqInteraction'$. For each of these sets, we investigate whether there is a system state that fulfills the preconditions of all requirements of this set. We only consider those parts of the preconditions that are not influenced by the software to be built, i.e. phenomena of domains that are only referred by requirements. We only consider those parts of the precondition because two requirements with contradicting preconditions can interact in the case that one requirement establishes the precondition

of the other requirement, such that the postconditions of both requirements could not be satisfied.

As argued in Section 1, requirements have to be expressed in terms of the environment. Therefore, they are normally written according to the general textual pattern: *"If the environment is like this, then it shall be changed like that."* Hence, a requirement has a pre- and a postcondition, both talking about phenomena of the environment [13]. We formalize the textual description of each relevant requirement to a formula $pre \Rightarrow post$. The formula *pre* describes the system state in terms of the referred to and controlled phenomena of the requirement when the requirement has to be fulfilled, and the formula *post* describes the system state to be achieved by the requirement.

For example, the requirement R3 states "If there is strong wind for more than 10 seconds, the sun blind will be pulled up, [...]", and we can express it with the formula $strong\ wind \Rightarrow pulled\ up$.

To determine whether a set of requirements is satisfiable, we have to define the function $precondition : Requirement \rightarrow \mathbb{P}\,\mathbb{P}\,Phenomenon$ that returns the phenomena of only referred domains (see above) occurring in the precondition of the requirement in disjunctive normal form (DNF). E.g., a precondition $(a \land b) \lor c \lor (d \land e \land f)$ is represented as $\{\{a,b\},\{c\},\{d,e,f\}\}$. We assume that the preconditions of all requirements in isolation are free of interaction. Otherwise the requirement is not satisfiable and can be left out.

$$
\begin{array}{|l}
precondition : Requirement \rightarrow \mathbb{P}\,\mathbb{P}\,Phenomenon \\
\hline
\forall\, d : Domain;\ r : Requirement \bullet \\
\quad \forall\, I : MinPhenInteraction(d);\ P : precondition(r) \mid \#I \geq 2 \bullet \neg\, I \subseteq P \land \\
\quad\quad \forall\, p : P \bullet \exists_1\, d : ran(dom(constrains \rhd \varnothing)) \bullet p \in refersTo(r,d)
\end{array}
$$

We define the precondition function as following for our running example.

$$
\begin{array}{|l}
\hline
precondition(R1) = \{\{sunshine\}\} \\
precondition(R2) = \{\{noSunshine\}\} \\
precondition(R3) = \{\{strongWind\}\} \\
precondition(R4) = \{\{openCommand\},\{closeCommand\},\{stopCommand\}\}
\end{array}
$$

For each relevant domain $d \in (Init \fatsemi P1S2 \fatsemi P1S3 \fatsemi P2).RelevantDomain'$, we now consider each set of possibly interacting requirements $R \in (Init \fatsemi P1S2 \fatsemi P1S3 \fatsemi P2).MinReqInteraction'(d)$ and combine the preconditions of the requirements in $R$ by conjunction and restore the disjunctive normal form using the generic function *dunion*.

$$
\begin{array}{|l}
[X] \\
\hline
dunion : \mathbb{P}\,\mathbb{P}\,\mathbb{P}\,X \rightarrow \mathbb{P}\,\mathbb{P}\,X \\
\hline
dunion = \lambda\, S : \mathbb{P}\,\mathbb{P}\,\mathbb{P}\,X \bullet \\
\quad \{T : \mathbb{P}\,X \mid \exists f : S \rightarrow \mathbb{P}\,X \mid \forall s : S \bullet f(s) \in s \bullet T = \bigcup(\mathrm{ran}\, f)\} \\
\hline
\end{array}
$$

For our running example, we get the following combined preconditions in disjunctive normal form.

$$
\begin{array}{l}
dunion(precondition(\!| \{R1, R2\} |\!)) = \{\{sunshine, noSunshine\}\} \\
dunion(precondition(\!| \{R2, R3\} |\!)) = \{\{sunshine, strongWind\}\} \\
dunion(precondition(\!| \{R1, R4\} |\!)) = \{\{sunshine, openCommand\}, \\
\quad \{sunshine, closeCommand\}, \{sunshine, stopCommand\}\} \\
dunion(precondition(\!| \{R2, R4\} |\!)) = \{\{noSunshine, openCommand\}, \\
\quad \{noSunshine, closeCommand\}, \{noSunshine, stopCommand\}\} \\
dunion(precondition(\!| \{R3, R4\} |\!)) = \{\{strongWind, openCommand\}, \\
\quad \{strongWind, closeCommand\}, \{strongWind, stopCommand\}\}
\end{array}
$$

A set of interacting requirements can now be rejected if all phenomena sets in the combined disjunctive normal form contain a set of interacting phenomena, because then there is no system state that leads to an interaction between the requirements. Formally, we can specify this step using the following operation schema.

*P3*

$$
\begin{array}{l}
\Delta Interaction \\
\hline
\forall d : RelevantDomain \bullet MinReqInteraction'(d) = \\
\quad \{R : MinReqInteraction(d) \mid \exists P : dunion(precondition(\!| R |\!)) \bullet \\
\quad\quad \forall I : \bigcup(\mathrm{ran}\, MinPhenInteraction) \bullet \neg I \subseteq P\}
\end{array}
$$

From the above equations, we see that R1 and R2 have interacting phenomena in their combined precondition because there cannot be *sunshine* and *noSunshine* at the same time. Hence, we remove $\{R1, R2\}$ from the set of sets of interacting requirements. For all other requirement sets, we can find system states that satisfy the combined preconditions because there can be sunshine and strong wind at the same time and the user can issue commands independently from the actual weather.

The remaining sets of interacting requirements have to be further analyzed to determine if the requirements are interacting and which measures (such as prioritization) shall be chosen to cope with the interactions.

From our running example we see that our method reduced the 21 initial combinations of requirements that may interact to 4 combinations.

## 5 Validation

The proposed method was used for analyzing requirements of a real-life example in the domain 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, which allows financial, informational, and electrical transactions. The gateway represents the central communication unit between a household and the grid in a smart metering system. It is responsible for collecting, processing, storing, and communicating meter data, and for controlling a household, e.g., energy supply.

As information sources, we considered diverse documents such as "Application Case Study: Smart Grid" and "Smart Grid Concrete Scenario" provided by the industrial partners of the EU project NESSoS[2]. As sources for functional requirements for such a

[2] http://www.nessos-project.eu/

**Table 3.** Effort spent for conducting the method and resulting reduction

| | | Method Step | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Modeling of Problem Diagrams | Initial Setup | Reducing Relevant Domains | Set up MinPhen | Reducing Relevant Requirements | Check for Parallel Requirements | Precondition-based Pruning |
| **Effort** | ∅ Per Item | 28 min. per Problem Diagram | 2.5 min. per Problem Diagram | 0.5 min. per Domain | 9.5 min. per Domain | 0.5 min. per Requirement | 6.85 min. per Problem Diagram | 8 min. per Requirement |
| | Number of Items | 27 Problem Diagrams | 27 Problem Diagrams | 19 Domains | 4 Domains | 27 Requirements | 27 Problem Diagrams | 5 Requirements |
| | *Total* | *12.5 person hours* | *1.125 person hour* | *0.16 person hour* | *0.6 person hour* | *0.225 person hour* | *3 person hours* | *0.67 person hour* |
| **Potential Interactions** | % of Initial | 100% | 100% | 49% | 49% | 2.8% | 2.8% | 1.7% |
| **Left** | % of Remaining | 100% | 100% | 49% | 100% | 5.8% | 100% | 60% |

gateway, we considered "Requirements of AMI (Advanced Multi-metering Infrastructure]" [14] provided by the EU project OPEN meter[3]. We refined the 13 minimum uses cases as described by this document to 27 requirements and modeled them as problem diagrams using the UML4PF tool [15].

The general effort of preparing the problem diagrams and executing our method is shown in Table 3. The steps of our method are added as columns. The rows are divided into 2 main parts. First, the effort per item and the total effort regarding the number of items. Second, the reduction of possible interactions within one step with regards to the total number of interactions or with regards to the interaction left by the previous step. This way, Table 3 provides the information about the effort and the resulting reduction for each step.

The problem diagrams modeling the 27 requirements given by the 13 minimum use cases served as an input to Phase 1, Step 1, resulting in 351 possible requirements interactions. The initial requirements interaction table consisted of 19 domains and 27 requirements. A number of 64 phenomena were documented as relevant, because the requirements mentioned them. In the second step, the number of domains on which an interaction could happen was reduced to 4, and 7 requirements were removed from the set of candidates, which could cause an interaction. At this point, the number of possible interactions was already reduced by more than fifty percent to 171 (see Table 3). The involved number of possibly involved phenomena was cut down to 19. Three of the phenomena were identified as possibly interacting phenomena. As a result, only 1 domain and 5 requirements remained after Step 3. Thus, at the end of Phase 1, we already reduced the number of possible interactions to 10, which makes a reduction by more than 95 percent (see Table 3). Since all of the requirements left may have to be fulfilled in parallel, no further reduction was possible in Phase 2. While checking the preconditions in Phase 3, one more requirement could be rejected to be a candidate for an interaction. In the end, 4 requirements, sources for 6 possible interactions, had to be analyzed in depth.

The analysis revealed that the requirements left caused 2 interactions. One of the original use cases in [14] described a process where the energy provider is able to

[3] http://www.openmeter.com/

disconnect a household from the grid by ordering the gateway to cut off the electricity supply. One reason could be unpaid bills. On the other hand, the provider can order the gateway to reconnect the household. A second use case describes that the customer is able to define a power consumption threshold. If the threshold is reached by the actual power consumption, the household is also cut off the grid by the gateway. But for this case, the consumer is allowed to override the cut-off manually, reconnecting the household. The two use cases, and therefore also the requirements, did not refer to each other, allowing the customer to override a cut-off ordered by the provider. Or the other way round, the provider could reconnect a household which was taken off the grid on demand of the customer. Hence, we found 2 real interactions.

To sum up, the effort to investigate requirements for interactions in depth was reduced by more than 95 percent. For the interactions left over to the in-depth analysis, the precision was 33 percent (2 real interactions / 6 possible interactions), which is acceptable considering the overall reduction. For calculating the recall, we made a full in depth analysis of all requirements and found no additional interactions which makes a perfect recall of 100 percent. In general, when looking for interactions, it is favorable to have a high recall rather than having a high precision. The reason is that missing one real interaction makes any effort reduction worthless.

For the smart grid case study, especially the effort spent for phase one payed off (see Table 3). Phase 2 and 3 resulted only in a minor reduction of possible interactions. This result should be subject to further research, as it may depend on the special structure of the smart grid case study. But overall, the effort of executing our method is reasonable with regards to the reduction.

## 6 Related Work

Although the problem of interaction between requirements has been known for a long time, there exist only few approaches dealing with this problem.

Egyed and Grünbacher [16] introduce an approach based on software quality attributes and traces between requirements. They assume that two requirements are conflicting only if their quality attributes are conflicting and there is a dependency between them. The authors do not consider the case of conflicting requirements due to their functionality and not their quality.

The approach proposed by Alférez et al. [17] finds candidate points of interaction. The authors first analyze the dependencies between use cases to identify potential candidates of conflict. Then they determine whether the detected use cases are related to more than one feature. In contrast to our method, it is not formally defined. Furthermore, this approach is based on use cases, whereas we rely on problem frames.

Kim et al. [18] propose a process for detecting and managing conflicts between functional requirements expressed in natural language. After identifying, documenting, and prioritizing requirements using goals and scenarios in the first phase, the requirements are classified through the requirements partitioning criteria in the second step. In the third phase, conflicts are detected using a syntactic method to identify candidate conflicts and a semantic method to identify actual conflicts. Step four manages the detected conflicts according to the priorities. Similar to our method, this process reduces

the scope of requirements to be considered by performing a syntactic analysis. The semantic analysis is performed manually by the analyst to check and answer a list of questions. As opposed to our method, this method is not formally specified.

In contrast to our problem-based method, Hausmann et al. [19] introduce a use case-based approach to detect potential inconsistencies between functional requirements. A rule-based specification of pre- and postconditions is proposed to express functional requirements. The requirements are then formalized in terms of graph transformations that enable expressing the dependencies between requirements. Conflict detection is based on the idea of independence of graph transformations. The authors provide tool support to represent the results of the analysis. Similar to our method, the results of the conflict detection method have to be analyzed further manually. Our method detects a set of interaction candidates that need to be analyzed further for real interactions. This approach detects dependencies that represent errors or conflicts to be decided by the modeler. This is due to the incomplete nature of use cases.

Lamsweerde et al. use different formal techniques for detecting conflicts among goals based on KAOS [2]. One technique to detect conflicts is deriving boundary conditions by backward chaining. Boundary conditions refer to combination of circumstances causing inconsistency in among different goals. Every precondition yields a boundary condition. The other technique is selecting a matching generic pattern. Our method for finding conflicts among requirements can be seen as complementary to this approach that provides techniques for detecting goal conflicts and resolving them. However, to use our method in connection with this approach, requirements as refinement of goals have to be modeled as problem diagrams.

Heisel and Souquières [13] developed a formal and heuristic method to detect requirement interactions. Each requirement consists of a pre- and a postcondition. The authors analyze whether the postconditions are contradictory by sharing common preconditions. They also determine postcondition interaction candidates by looking for incompatible postconditions. As opposed to our approach, the authors formalize the whole set of requirements, which is costly and time-consuming. Our approach utilizes the structure of problem diagrams to reduce the effort for the formalization.

An approach to detect feature interactions in the software product line (SPL) is proposed by Classen et al. [20]. The authors link feature diagrams used in the SPL to the problem frames approach by redefining the notions of *feature* and *feature interaction* based on the entailment relationship $D, S \models R$ [8,9]. This enables the authors to consider the environment in addition to the requirements, similar to our method. To detect feature interactions, four algorithms are presented based on a set of consistency rules. This work is complementary to our work. Using our approach, the sets of requirements and domains that have to be considered for interactions can be reduced and therefore the modeling and formalization effort is reduced.

## 7 Conclusions and Future work

In this paper, we investigated how to identify requirements interaction using a problem-based method. We described a structured method to identify requirements interactions between functional requirements. The method is formalized using Z and this specifi-

cation serves as basis for the tool support of the proposed method. For the first phase, we explained how to identify candidates for an interaction among a set of requirements modeled as problem diagrams. In the second phase, we showed how to reduce this set of candidates further using the information whether requirements have to be fulfilled in parallel or not. In the third phase we further reduced the possibly parallel requirements by checking their precondition. For the paper, we explained our method using a running example. For validation, we applied the method to a real example in the smart grid domain. The main contributions are:

- A re-usable requirements interaction detection method which provides structured guidance for a software engineer.
- A significant reduction of the initial set of requirements to be analyzed in depth which makes the use of heavy weight analysis methods, such as formal methods, practicable.
- A formal basis which enables tool-support that eases the execution of our method.[4]
- Identifying interactions among more than two requirements.

Considering the scalability of our method, we experienced a less than linear increase in effort for a rising number of requirements regarding the pruning steps. The reason is that the size of the requirements interaction table depends on the number of requirements and the number of domains. Even for a large amount of requirements, in most cases, the number of involved domains remains stable. And even for a large table, each decision can be done based on the information of one entry of the table. The table itself can be generated automatically and tool support is existent or developed right now for each pruning step. Hence, for the pruning the most effort stems from the modeling of the requirements themselves (see Table 3). But this is a necessary and unavoidable step when analyzing requirements in a structured way. By applying our method to a real-life case study in this paper, we showed the feasibility and usefulness of the method.

For the problem frames approach itself, we experienced a linear increase in effort for each additional requirement (see Table 3). Thus, the scalability is acceptable. Note that our method can be applied to any other requirements notation as long as it provides all involved domains in the environment, the constrains or refers relations between functional requirements and domains, and the phenomena for the constrains and refers relations. This also means that we do not use all information given by problem diagrams. But from our point of view the full problem frames approach is a natural solution for collecting this information as it is system centric, considers the environment with its domains, and provides a structured method to deal with functional requirements and refine them in the needed way.

For the future, we plan to add support for considering quality requirements. Additionally, we strive for extending the tool support. For example, we will implement the possibility to express the concurrence of the fulfillment of requirements in graphical way, e.g. using UML interaction overview diagrams. This enables the automatic generation of sets of parallel requirements. Finally, we plan to work on a method which will guide the process of interactions resolving and the according modification of the requirements.

[4] Tool-support for generating the initial requirements table is available under http://www.uml4pf.org/rit/rit.html.

## References


1. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering. FOSE '07, Washington, DC, USA, IEEE Computer Society (2007) 37–54
2. van Lamsweerde, A., Letier, E., Darimont, R.: Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Trans. Softw. Eng. **24**(11) (1998) 908–926
3. Sommerville, I., Sawyer, P., Viller, S.: Viewpoints for requirements elicitation: A practical approach. In: Int. Conf. on RE: Putting Requirements Engineering to Practice, IEEE Computer Society (1998) 74–81
4. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. Comput. Netw. **41** (2003) 115–141
5. Cameron, E.J., Velthuijsen, H.: Feature interactions in telecommunications systems. Comm. Mag. **31**(8) (August 1993) 18–23
6. Robinson, W.N., Pawlowski, S.D., Volkov, V.: Requirements interaction management. ACM Comput. Surv. **35** (2003) 132–190
7. Cheng, B.H.C., Atlee, J.M.: Research Directions in Requirements Engineering. In: Future of Software Engineering. FOSE '07, IEEE Computer Society (2007) 285–303
8. Jackson, M.: Problem Frames. Analyzing and structuring software development problems. Addison-Wesley (2001)
9. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. **6** (1997) 1–30
10. Wordsworth, J.: Software development with Z - a practical approach to formal methods in software engineering. International computer science series. Addison-Wesley (1992)
11. Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989) http://spivey.oriel.ox.ac.uk/mike/zrm/zrm.pdf.
12. Hatebur, D., Heisel, M.: A UML profile for requirements analysis of dependable software. In Schoitsch, E., ed.: Proc. of the Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP). LNCS 6351, Springer (2010) 317–331
13. Heisel, M., Souquières, J.: A heuristic algorithm to detect feature interactions in requirements. In: Language Constructs for Describing Features. Springer (2000) 143–162
14. OPEN meter project: Requirements of AMI. Technical report, OPEN meter project (2009)
15. Côté, I., Hatebur, D., Heisel, M., Schmidt, H.: UML4PF – a tool for problem-oriented requirements analysis. In: Proc. of the Int. Conf. on Requirements Engineering (RE), IEEE Computer Society (2011) 349–350
16. Egyed, A., Grunbacher, P.: Identifying requirements conflicts and cooperation: How quality attributes and automated traceability can help. IEEE Softw. **21**(6) (November 2004) 50–58
17. Alférez, M., Moreira, A., Kulesza, U., Araújo, J.a., Mateus, R., Amaral, V.: Detecting feature interactions in SPL requirements analysis models. In: Proc. of the 1st Int. Workshop on Feature-Oriented Software Development. FOSD '09, ACM (2009) 117–123
18. Kim, M., Park, S., Sugumaran, V., Yang, H.: Managing requirements conflicts in software product lines: A goal and scenario based approach. Data Knowl. Eng. **61** (2007) 417–432
19. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In: Proc. of the 24th Int. Conf. on Software Engineering. ICSE '02, ACM (2002) 105–115
20. Classen, A., Heymans, P., Schobbens, P.Y.: What's in a feature: a requirements engineering perspective. In: Proc. of the Theory and practice of software, 11th int. conf. on Fundamental approaches to software engineering. FASE'08/ETAPS'08, Springer (2008) 16–30