MSc thesis

Computer Science

Analyses for Requirement Models

Elina Kettunen

January 16, 2020

FACULTY OF SCIENCE
UNIVERSITY OF HELSINKI

Tiedekunta – Fakultet – Faculty	Koulutusohjelma – Studieprogram – Study Program
Faculty of Science	Computer Science

Tekijä – Författare – Author

Elina Kettunen

Työn nimi – Arbetets titel – Title

Analyses for Requirement Models

Ohjaajat – Handledare – Supervisors

Prof. Tomi Männistö, Dr. Mikko Raatikainen

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
MSc thesis	16.01.2020	62

Tiivistelmä – Referat – Abstract

In software development, requirements define the expected behaviour of a system. Requirements can have relationships to other requirements and these relationships are called dependencies or constraints. Requirements are usually assigned to releases, which means that there is an order in which the requirements will be implemented. A single requirement model can be constructed from requirements and their properties, releases and dependencies.

Large software development projects can have tens of thousands of requirements and managing large-scale requirement models is challenging. In the OpenReq project, the aim was to develop better tools for requirements engineering. In this thesis, which is a part of OpenReq, the aim was to research analyses that have been done to requirement models or that could be done to requirement models. In OpenReq, requirement models consist of requirements as statements and their relationships. The research is conducted as a systematic literature review using snowballing as the search strategy. At the moment, the consistency check and diagnosis of inconsistencies are the only analyses performed on the OpenReq requirement models.

While there exist very little analyses for requirement models, we found several analyses, especially from the domain of feature modelling, which could be applicable to requirement models. The most promising of these analyses are detection of dead requirements and redundant constraints, but the original analyses must be adopted to consider the properties of requirement models such as releases. In the future, there should be research on how relevant these analyses are for real-life industrial requirement models.

ACM Computing Classification System (CCS)

Software and its engineering → Software creation and management → Designing software → Requirement analysis

Avainsanat – Nyckelord – Keywords

Requirement, requirements engineering, feature, release plan, consistency check

Säilytyspaikka – Förvaringställe – Where deposited

Helsinki University Library

Muita tietoja – Övriga uppgifter – Additional information

Software Systems specialisation line

Table of contents

1 Introduction

1.1 Background

Requirements in software engineering define the expected behaviour of a software system (Somerville 2016, p.102). For example, requirements are defined as statements (sentences) in software requirement specification (SRS), or documented as goals and tasks in goal modelling. Agile and lean software development have also introduced new concepts, such as epics, backlogs and business canvases. The requirements are usually assigned to a set *releases* each containing several requirements. The releases define a planned order in which the requirements are implemented. Requirements can have different properties, such as effort and cost.

Requirements engineering (RE) covers different activities such as elicitation, analysis, specification, validation and documentation of requirements (Somerville 2016, pp. 111-112). The management of requirements after they have been documented is one aspect of RE.

Requirements are related to each other and explicit relations are called *dependencies,* which describe the inter-requirement relationships. Requirements and their dependencies together form an entity that is in this study called a *requirement model.* Different operations with a requirement model, which characterize the model without altering it, are here called *analyses*.

Large projects can have tens of thousands of different requirements. The requirements can be stored, for example, in a dedicated requirement management tool, such as IBM's DOORS[1] (see Murphy *et al*. 2016), or in (shared) Word documents. Requirements can also be stored in issue trackers, such as Jira[2] and Bugzilla[3], especially in the case of open source development. In Jira, requirements are expressed as issues that can contain different issue types, such as tasks, bug reports and user stories. However, current requirement

[1] https://www.ibm.com/support/knowledgecenter/en/SSYQBZ_9.6.0/com.ibm.doors.homepage.doc/helpindex_doors.html
[2] https://www.atlassian.com/software/jira
[3] https://www.bugzilla.org/

management tools and issue trackers do not readily support managing and analysing large-scale requirement models.

In addition, in requirements engineering, one of the challenges is that there are often several stakeholders that have different, perhaps contradictory, ideas and expectations on the functionality of the system (Ruhe *et al.* 2003). Defining and managing a body of requirements based on these ideas and expectations, is challenging, especially with large projects that can cover numerous requirements from different stakeholders.

This thesis is a part of the OpenReq project. OpenReq[4] (2017-2019) is an international requirements engineering research project in which several universities and companies work together to develop better tools for organisations to manage software requirements (for a short overview, see Mäenpää *et al*. (2017)). The academic partners in the project are Hamburger Informatik Technologie- Center EV (HITeC, University of Hamburg (project leader), Germany), University of Helsinki (UH), Universitat Politecnica de Catalunya (UPC, Spain), and Technical University of Graz (TUGraz, Austria). The industrial partners are Siemens (Austria), vogella (Germany), ENG Engineering (Italy), The Qt Company (Finland), and Wind Tre S.p.a (Italy).

Requirement models as defined in OpenReq consist of requirements, their dependencies and properties, and planned releases. Our focus is on the cases where there are thousands of written requirements available. The product may already exist, and some of the requirements are bug reports. Requirements have been assigned to several releases. Specifically, the aim of this thesis is to discover analyses that could provide new, relevant information on large-scale, complex requirement models.

1.2 Problem definition

The research problem focuses on analyses that are being done or are applicable to requirement models, especially in the context of requirement models in the OpenReq project. The research is conducted as a literature review.

[4] https://openreq.eu/

1.2.1 Research questions

The research problem is refined into three research questions:

> **RQ1:** What analyses have been done to requirement models?
>
> **RQ2:** What analyses could be adopted from other domains to requirement models?
>
> **RQ3:** Which of these analyses would be applicable to OpenReq requirement models and how?

RQ1 refers to the existing analyses that have been done or proposed to be done to existing requirement models. The interest is on analyses that are done explicitly to requirement models.

RQ2 is mostly focused on the analyses of the *knowledge-based configurations* and *feature models*. Knowledge-based configuration is a research area focusing on mass-produced products, especially in traditional fields such as mechanical engineering. *Feature modelling* is the most popular variability modelling method in software product line research (see Section 2.2.2).

RQ3 addresses the analyses that could be meaningful and relevant in the OpenReq context with possible adaptations. In addition, as the analyses are applied in different contexts, their application is also elaborated.

1.3 Scope

In this study, the requirements are treated as existing "black box" entities that have a certain set of properties, but whose content and means of content representation are not studied. The process of creating a requirement model is not covered, which means that we are not interested in requirement prioritization, specification, scheduling, release planning or any activities that are related to constructing a requirement model.

We are primarily interested in a relatively mature requirement model and how it can be analysed. Therefore, certain agile and lean methods such as Minimum Viable Products in the Lean Startup (Ries 2011), in which the requirements are unsure, under constant change, and not well explicated, are not in the core focus.

We are interested in the requirement models in which requirements are defined as statements that have dependencies between them. Thus, goal modelling and goal-oriented languages, such as i* (Yu 1997) and KAOS (Dardenne *et al*. 1993), are out of the scope of this study.

Our point of focus is on the different analyses that can be done to requirement models, but we are not that concerned about how the analysis is done or what the most optimal algorithms would be. Transformations on the models, such as constructing an atomic set or different filters (see, e.g., Benavides *et al*. 2010), are not considered. Especially with feature models, there exist several different analyses focusing on the code metrics, but these are not relevant to requirement models, and thus are excluded from this study.

We concentrate on the role of the University of Helsinki, which has been involved in the OpenReq project in developing services that are meant to help with analysing a body of existing requirements. The concrete scenarios are the analyses of requirements as The Qt Company's Jira issues, which can be converted to OpenReq JSON format specified by OpenReq ontology. The OpenReq JSON is based on JavaScript Object Notation (JSON). It is very similar to Object Management Group's standard Requirement Interchange Format[5] (ReqIF), and it has been developed to provide a simple general requirement specification exchange format. Besides Qt's Jira issues, the UH's services can be utilized for other requirement models or *release plans* that follow OpenReq JSON format.

[5] https://www.omg.org/reqif/

2 Previous work

In this chapter, we first discuss the definitions of a requirement based on the previous work. Then we address concepts of requirement and feature models in the literature, while discussing what these concepts mean in the OpenReq project. We also briefly introduce the concepts related to release planning.

2.1 What a requirement is

In this section, we focus on how requirements have been defined and classified in the literature. We present the concept of a requirement in this thesis and in the OpenReq project.

2.1.1 Definition of requirement

There are several different kinds of definitions for a *requirement*. First of all, according to Mavin *et al.* (2017), the distinction between *goals* and requirements is important to recognize. Goals tell what the software product's stakeholders want, but on an idealized and aspirational level; goals are not necessarily measurable, and there are always conflicts between the goals. The authors see defining goals as a necessary early phase in the process of defining the software system requirements that describe the system's behaviour and properties.

In contrast to the goals, Mavin *et al.* (2017) see that requirements must be free of conflicts and there should be means to measure or test the requirements. However, often the difference between goals and requirements is not clear in the field of requirements engineering, if such a distinction is even made (Mavin *et al.* 2017). For example, Ruhe *et al.* (2003) see requirements as "*very volatile and prone to numerous changes*", i.e. the authors do not differentiate between goals and requirements.

In the standard dictionary of The Institute of Electrical and Electronics Engineers (IEEE), the term requirement is defined in the following manner (IEEE Std 610.12 1990):

> *"**(A)** A condition or capability needed by a user to solve a problem or achieve an objective.*

> ***(B)*** *A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.*
>
> ***(C)*** *A documented representation of a condition or capability as in definition (A) or (B).“*

In the same dictionary, a “raw requirement” is defined as

> “*an environmental or customer requirement that has not been analysed and formulated as a well-formed requirement”.*

In IEEE standards (ISO/IEEE Std 24765 2010), a goal is defined as

> **“*1.*** *an intended outcome. ISO/IEC TR 9126–4:2004, Software engineering – Product quality - Part 4: Quality in use metrics.4.2.*
>
> ***2.*** *intended outcome of user interaction with a product.*[6]*”*

or as “*An objective that is desirable to meet, but it is not mandatory to meet.*” (IEEE Std 1413.1 2002)

A textbook (Somerville 2016, p. 102) definition of requirement is:

> *“The requirements for a system are the descriptions of the services that a system should provide and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analysing, documenting and checking these services and constraints is called requirements engineering (RE).”*

[6] . *ISO/IEC 25062:2006, Software engineering – Software product Quality Requirements and Evaluation (SQuaRE)-Common Industry Format (GIF) for usability test reports. 4.8 NOTE [ISO 9241–11]*

The International Requirements Engineering Board[7] (IREB) has A Glossary of Requirements Engineering Terminology (Glinz 2017), where a requirement is defined as:

> *"1. A need perceived by a stakeholder.*
> *2. A capability or property that a system shall have.*
> *3. A documented representation of a need, capability or property."*

Many of the formal definitions described above emphasize the needs of users or customers when defining the term requirement. However, the system requirements are also considered and, for example, in IEEE Standards there is a distinction between a requirement and a vaguer goal. We also make a distinction between goals and requirements, as in our study requirements are clearly defined as individual entities, even though the exact content of a requirement might not be clearly stated. This means that every requirement is expected to have an individual identification label and a description, which distinguish it from other requirements.

2.1.2 Requirement in software development

In agile software development, which is a common software engineering approach nowadays, the concept of requirement is not clear and well specified (Cao & Ramesh 2008). It is typical that requirements are not predefined in a detailed documentation, but rather they emerge and then are clarified and refined during the development process. In the beginning of the project, there may be some high-level requirements engineering and, during the development cycles, the requirements become more detailed. Requirement prioritization also usually happens in iterations.

In Scrum method, which is an example of agile development, all requirements are included in a product backlog (Paetsch *et al.* 2003). A product backlog lists the features, functions and bugs in a priority order. For requirements of a single iteration (or sprint in Scrum), there is a sprint backlog containing a set of requirements, whose content should not be altered during the sprint. However, the customer can always reprioritize requirements for the next sprint.

[7] https://www.ireb.org/en

Requirements can be classified to high-level abstract requirements that are known as the user requirements, and more detailed and descriptive system requirements (Sommerville 2016, p. 102). User requirements are usually more like statements (broad or detailed) that describe the services the system provides to users and the constraints that restrict the services. System requirements describe in detail the software system's services, functions and operational constraints.

Software system requirements can be divided into functional and non-functional requirements (Sommerville 2016, pp. 105-111). Functional requirements describe how the system should behave in different situations such as in receiving inputs, and what kind of services the system should provide. Non-functional requirements describe constraints that affect the system's services or functions, and they can be further divided into several different categories such as product and external and organizational requirements. Examples of non-functional requirements are usability, efficiency and security requirements.

Sometimes the term *feature* is used to mean a product requirement. Feature is probably best known in the context of software product line engineering in which features are used to specify individual products (Benavides *et al.* 2010).

In the IEEE Standards dictionary (IEEE Std 610.12 1990), a *software feature* is defined as

> *"(1)*
>
> *(IEEE Std 829-1983 [5]) A distinguishing characteristic of a software item (for example, performance, portability, or functionality).*
>
> *(2)*
>
> *(IEEE Std 1008-1987 [10]) A software characteristic specified or implied by requirements documentation (for example, functionality, performance, attributes, or design constraints)."*

Kang (1990) was the first to define feature models, and in his definition a feature is an "*user-visible aspect, quality, or characteristic of a software system or systems*".

If a clear distinction is made between requirements and features, features can be seen as already existing entities (solution domain) whereas requirements cover planned functionalities (problem domain) (Savolainen 2011, p. 12-14, 27). All authors do not clearly use this distinction between a feature and requirement, and sometimes the terms are used to describe the same thing.

In software engineering, requirements are dependent on the nature of a project and the stakeholders involved (Svahnberg et al. 2010). Requirements reflect the demands and expectations of the different stakeholders, and they should describe what the system is expected and supposed to do and how. Often there is an effort associated with each requirement; this effort can be measured, for example, in working hours needed to implement a particular requirement.

Effort and other time-related, technical or financial constraints related to effort are hard constraints that can restrict the time and order in which selected requirements can be implemented (Svahnberg *et al*. 2010). In addition, dependencies between requirements belong to hard constraints, as they tell, e.g., which requirements should be implemented before others, or if two requirements cannot be implemented at the same time.

Soft constraints include different kinds of value and risk factors, and they are affected by stakeholders' different statuses, as one stakeholder may have more influence on the requirement prioritisation or selection than another (Svahnberg *et al*. 2010). Relationships between selected requirements and the effort needed to implement them are not always clear to project managers and marketing (Ruhe *et al*. 2003).

Even though the difference between a feature and a requirement is not necessarily significant, in requirement modelling, or software engineering in general, there are aspects (e.g. release order) to consider that are not usually necessary in feature modelling. In this study, the term feature is used to describe already existing software product functionalities, and requirements describe characteristics that are being planned, already implemented or even discarded.

In this study, we do not consider the classification of requirements to categories such as functional and non-functional requirements, as the emphasis here is on the relationships between requirements. Even though there are different types of requirements and

considering these is important in creating a requirement model, in an already existing model the exact nature of an individual requirement does not matter as much as its dependencies with other requirements. However, the requirements can be hierarchically categorized to, for example, large entities such as epics that contain numerous smaller entities such as tasks.

To summarize, even though the distinct requirement types are not considered, we see requirements consisting of both user and system requirements, and in our requirement model it is assumed that the requirements have been described as individual statements. These statements can be expressed as sentences or longer explanations. We consider software bug reports as requirements, so in addition to describing what the system should do, requirements also describe what the system should not do.

2.1.3 Requirements in OpenReq

The concept of requirements in this thesis follows more precisely the definitions specified in OpenReq. All OpenReq partners have agreed to use a bespoke JSON format adhering to an ontology[8] realized by a JSON schema (hereafter called OpenReq JSON) to transfer data between services (Quer *et al*. 2018). There are several objects defined in the OpenReq JSON schema, including Project, Release, Requirement, and Dependency. These objects are used to store and transfer information on a software project, its requirements, releases and other related issues. The analyses that can be done to the requirement models (also called release plans) following the OpenReq format are the focus of this study.

Requirement in the OpenReq context can be divided into several different types, even though all requirements are saved in the same format. For example, in the case of the Qt Company's Jira issues, the requirements can be bug reports, issues and other types of requirements reported by the Jira users (Qt's engineers, Qt's customers etc). In OpenReq JSON, a requirement has properties such as an id, name, text, type, status and a creation date. Requirement can also have comments associated with them.

In OpenReq JSON, RequirementPart objects are used to store information on attributes of requirements. There can be numerous attributes or properties, but most of them are not

[8] https://github.com/OpenReqEU/openreq-ontology

relevant in requirement modelling as covered in this thesis. Only attributes such as the release of a requirement, priority and effort are considered.

In the case of the Qt Company, Jira contains tens of thousands of issues. Thus, the management and analysis of Qt Jira issues can be considered as a case of Very Large-Scale Requirements Engineering (VLSRE), as defined by Regnell *et al*. (2008).

In Qt Jira, the issues can have duplicates, i.e. the same issue is reported by several people with a different name and identification number, and there are various different types of dependencies between the issues/requirements. Duplicate is one of these dependency types. Various dependencies will be discussed in more detail in later sections.

2.2 Requirement and feature models

In this section we first describe requirement models and dependencies between requirements and then discuss feature models and their similarities and differences with requirement models. We also present the concept of requirement models in the OpenReq.

2.2.1 Requirement models and requirement dependencies

Goal modelling and use cases are a means in requirements engineering to model requirements and their interactions with actors, who represent different stakeholders. However, the concepts of these models are different from the requirement models on which we are concentrated. Goal modelling can be used in the early phases of requirements engineering, concentrating on intentional actors and their goals. Relationships between goals and actors are called dependencies, but these are different from the inter-requirement dependencies. There are different frameworks or goal-oriented languages (i*, KAOS) that are used for formal goal modelling (Dardenne *et al*. 1993, Yu 1997, Horkoff *et al*. 2016). However, the formal goal modelling approach is rarely used in the industry (Mavin *et al*. 2017).

Use case models, also known as use case diagrams, and behavioural models are examples of the different UML models that can be used with requirements (Sommerville 2016, pp. 144-146, 154-159). Use case models are used to model interactions between an external agent or an actor and the software system. One use case represents one user requirement, and the model presents the relationships between actors and requirements. Behavioural

models can be used to illustrate requirements, but they mainly describe the behaviour of the system when it is running. With both use cases and behavioural models, the intention is to describe the actions rather than requirements and the relationships between different requirements.

In the requirement models, which are the focus of this study, the relations of requirements to different actors or users are not considered.

Relationships between requirements are called dependencies or *interdependencies*[9]. Numerous dependency types have been described and classified in different dependency typologies. We found four taxonomies. The first was proposed by Pohl (1996) and the second by Carlshamre *et al*. (2001), who did empirical study on requirements and dependencies in five different companies. The third taxonomy or typology was proposed by Dahlstedt and Persson (2005), who aimed to synthesize the existing knowledge. The fourth was proposed by Zhang *et al*. (2014), who combined empirical research with a synthesis.

Dependencies can be classified as structural, constraint and cost or value dependencies (Dahlstedt & Persson 2005). Structural dependencies include dependency types such as *similar_to* (sic) (a requirement is similar to another requirement) and *refined_to* (a requirement refines properties of another requirement). Constraint dependencies, include *requires*, *excludes* and *conflicts_with*. The cost or value dependencies include dependencies that refer to a requirement increasing or decreasing cost or value of implementing another requirement.

Zhang *et al*. (2014) carried out their study in the context of change propagation analysis that can be used to determine how potential changes can affect other parts of a software. Dependencies between requirements act as the basis for change propagation analysis, but according to Zhang *et al*. only a few studies have evaluated how the different dependency types can be applied in real-world projects, and how they help change propagation analysis and dependency identification.

[9] Interdependencies would be the accurate term for relationships between individual requirements, but for the sake of clarity the term dependency is used to describe these relationships in this study.

When dependencies are clearly defined, finding and recognizing different dependencies in requirements is easier (Zhang *et al*. 2014). However, in practice the number of dependencies can make the identification and management of dependencies difficult (Carlshamre *et al*. 2001). Even though there are usually many requirements without any dependencies to other requirements, those requirements that have dependencies to others can have several. In one industrial survey by Carlshamre *et al*. (2001), the researchers noticed that relatively few requirements (20%) were associated with approximately 75% of all dependencies.

In their case study, Zhang *et al*. (2014) found that clearly defined dependency types help to find more dependencies in requirements, analyse quality of requirements and identify the system's key dependent functions. However, certain dependency types have ambiguous definitions, their semantics have been described only on a very general level, or the definitions overlap too much. Problems like these make dependency identification more difficult. Therefore, the authors proposed a new dependency typology with more precise semantics based on their empirical case study on dependency detection.

Dependencies can be seen as a specific issue of requirements traceability, which refers to the ability to trace a requirement's life and development inside the system to other artefacts, such as even to implementing source code (Dahlstedt & Persson 2005). However, even though the issues related to requirements traceability are considered important in supporting software development, they are beyond the scope of this study. Different types of dependencies are considered only in the context of a requirement model, and the focus is on the constraint dependencies *requires* and *excludes*.

2.2.2 Feature models

As noted earlier, a requirement is much like a feature and therefore it is meaningful to also elaborate feature modelling here. Feature models have been researched extensively (Raatikainen *et al*. 2019). The concept of a feature model was first introduced by Kang *et al.* (1990) and they are used to specify the variability of software product lines. There are several different languages for describing feature models, such as basic feature models, cardinality-based feature models, and extended feature models (Benavides *et al.* 2010).

Typically, a feature model is a model of existing features that can be used to construct a product, or, in case of a software product line, product variants. Finding the right configuration for the model is one of the main interests in software product line feature modelling (Benavides *et al*. 2010, Thüm *et al.* 2009).

Feature models consist of a hierarchical set of individual features and different types of relationships or dependencies between these features (Benavides *et al*. 2010, Galindo *et al*. 2018, Tiihonen *et al*. 2016). The relationships can be divided into parent-child relationships (the relationship between a compound (parent) and its subfeatures or subrequirements (children)) constituting a tree hierarchy and cross-tree or cross-hierarchy relationships or constraints. Tree hierarchies are typical in feature models, and child features can also be called subfeatures.

Parent-child relationships can be mandatory (child appears in the product if its parent does), optional or alternative (only one of the children can be selected when the parent is selected) (Benavides *et al*. 2010, Galindo *et al*. 2018, Ochoa *et al*. 2018). There is also the Or-relationship where one or more of the child features can be selected when the parent is selected. In more general terms, parent-child relationships can be provided with cardinality for which aforementioned relationships are shorthand notations.

Constraint dependencies, or cross-tree constraints usually refer to relationships where one feature includes or excludes another. For example, if a feature A includes a feature B, the feature B must be included in the same model (Benavides *et al*. 2010, Galindo *et al*. 2018, Ochoa *et al*. 2018).

The size of the feature models varies greatly, and automation is necessary especially for the analysis, including configuring, of large-scale feature models, since the manual analysis would be too time-consuming and difficult. There are several different methods for the automated analysis of feature models, and these have been reviewed, e.g., by Benavides *et al*. (2010), El-Sharkawy *et al*. (2018), and Galindo *et al*. (2018). Before the analysis, feature models (and also requirement models) are typically translated to another representation such as constraint programming, propositional logic or description logic (Benavides *et al*. 2010, Galindo *et al*. 2018). Then a specific algorithm or an off-the-shelf solver is used to analyse the feature model, for example to check its

consistency or detect errors in the model. The obtained result is used in product configuration or derivation.

Lettner *et al*. (2019) introduced a framework for automated analysis of multi-layered feature models. Multi-layered feature models can be used, for example, with mobile software development, where the developers must consider both platform and application features that each represent their own layer.

One problem noted in several review papers on feature models is that almost all research has been done to feature models that have only a limited number of features (Afzal *et al*. 2016, Benavides *et al*. 2010). Another problem mentioned in reviews is the lack of evaluations or validations of different analysis operations for variability or feature models (El-Sharkawy *et al*. 2019, Galindo *et al*. 2018).

Benavides *et al.* (2010) recognise two main types of feature models that are used in the experiments: realistic and generated feature models. Realistic feature models encompass those that either model real-world domains or are simplified versions of them (Benavides *et al*. 2010). Especially in the automotive industry, feature models can grow considerably large, having up to 10,000 features (Batory *et al*. 2006). As large-scale feature models consisting of hundreds or thousands of features are not typically published as a whole, researchers have used specially developed algorithms to automatically generate large models to test the scalability of their analysis methods (see e.g. Segura *et al*. 2014).

Since 2006, the trend has been to use automatically generated rather than realistic feature models (Benavides *et al*. 2010). The automatically generated models[10] can be modified according to the researchers' needs to contain considerably larger amounts of features with more complex relationships than available realistic models.

Compared to feature models, requirement models have a clear temporal and development aspect rather than only selecting and configuring requirements (Svanhberg *et al*. 2010). In a requirement model, requirements are assigned to releases, which are not present in feature models. Since requirements are often not yet concrete existing entities, but rather planned features and system properties, it is usually important to determine in which

[10] An example repository of feature models can be found at http://www.splot-research.org/, where there are several generated feature models available for research.

order the requirements will be released, in addition to finding the right configuration of the requirement model. For example, in a requirement model a *requires* dependency between requirements A and B, i.e. A requires B, could mean that the requirements should be either in the same release, or B should be in in one of the releases preceding the release of requirement A.

2.2.3 Requirement model and dependencies in OpenReq

In the OpenReq project, requirement models are also called release plans. There is no specific object for representing an entire requirement model in OpenReq JSON, but rather a requirement model consists of project, and arrays of requirements, dependencies and releases objects. Each release has a list of requirements assigned to it, and a capacity, which tells the potential maximum combined effort of requirements assigned to a release.

In OpenReq, the dependencies can be divided into parent-child relationships and cross-tree constraints like described above. However, the alternative relationship is not really used in parent-child relationships. This means that in the OpenReq context, the situations, when only one child of two or more alternatives could be selected, do not occur unless further restricted by the cross-tree constraints. Cross-tree constraints include dependency types like *requires* and *excludes*. Parent-child relationship is called *decomposition*. There are also other dependency types such as *similar* and *duplicates*, which refer to two requirements resembling each other or being exactly the same. A more comprehensive description of dependencies in OpenReq can be found in the ontology, which is available on GitHub.

The dependencies between issues in the Qt Jira are called links. It is possible to explicitly define these links between issues, but often they are not used and possible dependencies are only mentioned in the description or comments of the issue. Thus, the dependencies reported by engineers or users may be deficient, incomplete or not explicit. Also, getting the semantics of a dependency right might be difficult in some cases. Typically, the hierarchy is at a maximum of four levels deep, e.g., in Jira the depth is three levels.

2.3 Requirement prioritization and release planning

Requirement prioritization aims to assess the importance of individual requirements and rank them according to specified priorities (Riegel & Doerr 2015). In a software project, there are initially often more requirements than can be met during the project. Different types of constraints, mainly money and time related, mean that there should be ways to determine the most relevant requirements for the project. Different requirement prioritization techniques have been reviewed, e.g., by Riegel and Doerr (2015) and Thakurta (2016).

In incremental software development, each system release adds or improves features that are valuable for the customers. Defects found in the previous releases are fixed in some of the following releases according to the feedback from customers or other stakeholders (Ruhe & Saliu 2005). In Software Release Planning (SRP) the aim is to find "the best combination of features to implement in a sequence of releases" (Ameller *et al*. 2016). The focus is on deciding what features (or requirements) to select and assign the selected features to releases in order to create an optimal sequence of releases so that constraints related to resources, technologies and risk are satisfied (Amellar *et al*. 2016, Ruhe & Saliu 2005).

In IEEE standards, *release* is defined as "*A particular version of a configuration item that is made available for a specific purpose (for example, test release)"* (IEEE/EIA Std 12207.0 1996).

And a *release plan* (IEEE Std 828 2012) is

> *"a plan that describes what portions of system functionality will be implemented in which releases and the rationale for each release. It includes or provides reference to a description of release contents, release schedule, release impacts and release notifications."*

The standards (ISO/IEC/IEEE Std 24765 2010) also describe *release management* as

> *"management of the activities surrounding the release of one or more versions of software to one or more customers, including identifying, packaging, and delivering the elements of a product"*

Thus, release planning can be seen as one of the activities of release management.

According to Ruhe & Saliu (2005), a good release plan contains the most optimal blend of features assigned to the right releases, and thus satisfies the demands of the most relevant stakeholders. A good release plan should also consider available resources and the dependencies that exist between different features.

Strategic release planning is also called road-mapping, and there are various models that have been developed to address this problem. Release planning models are actually more precisely methods or techniques. Svahnberg *et al*. (2010) reviewed different strategic release planning models, and found out that there were only a few real available options regarding models, methods and tools are for practitioners. One challenge was to find fully validated release planning models, since only a few of the release planning models have been tested in full scale industrial scenarios. Issues related to scalability are also considered important for strategic release planning models.

Release planning models use several different types of algorithms to answer the next release problem, i.e. what the optimal requirements selected for the next release are (Ameller *et al*. 2016). These algorithms include, e.g., genetic and evolutionary algorithms, and the knapsack problem. The basic idea in general is to feed requirements including their properties, dependencies and various different constraints or properties (resource, quality, time etc) to the model, and using the algorithm the model outputs, e.g, the next release, user stories and/or a list of requirements. Some of the release planning models place more emphasis on the different stakeholders and their values than others. In these cases, the values can be given as one input type to models.

In agile software development, a release consists of iterations, and in agile planning there are several iteration plans in addition to a higher level release plane. Agile release planning aims to account more for the changing requirements during the software development process (Szoke 2011). A conceptual model to support agile iteration planning and release scheduling has been proposed by Szoke (2011).

To summarize, requirement prioritization aims to give each requirement a priority, which is based on the views of different stakeholders. In release planning, the requirements are assigned to releases based on the results of the prioritization process. In our study, however, we assume that the requirement model has been specified, meaning that the

prioritization and release assignment have already been carried out. The task is not to make more release assignments per se. However, it is still possible to change or correct the release assignments if deemed necessary.

3 Material and methods

The study was conducted as a systematic literature review (Kitchenham & Charters 2007) by using *snowballing* as described by Wohlin (2014). Systematic literature reviews have become a popular research method in software engineering research, and they provide the means to systematically research literature on a selected topic. In snowballing, the search for papers for inclusion and review begins by selecting a start set of papers based on the relevance of the said papers. Snowballing is done iteratively using either backward or forward snowballing, or both. The basic idea in each iteration of the snowballing is to use first the start set to discover new papers, which in turn are used as the next step for finding further relevant papers. The process goes on in iterations until no further interesting papers are found. In backward snowballing, the new papers are found reading the start set and selecting relevant papers from the references lists, whereas in forward snowballing the new papers are found among the papers citing the start set papers.

3.1 Search strategy

In this study, the start set consisted of 14 papers that are listed in Table 1. The papers in this set were selected based on the expertise of the research group. The start set consisted of six systematic literature reviews, two systematic mapping studies, and five original research papers. We read each start set paper and selected the relevant looking papers based on the location of the interesting reference in the text: for example, the paper was cited when describing a potentially interesting analysis, or there was an interesting title in the full reference (in the case of literature reviews also in their primary study listings).

If the paper contained a clear description of the analysis and an algorithm, more information about the analysis was searched from the papers referencing that particular paper (forward snowballing).

Table 1. Papers in the start set. SLR = Systematic Literature Review, SMS = Systematic Mapping Study

ID	Reference	Title	SLR	SMS	Original research
S1	Afzal *et al*. 2016	Intelligent software product line configurations: A literature review	x		
S2	Ameller *et al*. 2016	A Survey on Software Release Planning Models	x		
S3	Benavides *et al*. 2010	Automated analysis of feature models 20 years later: A literature review	x		
S4	El-Sharkawy *et al*. 2019	Metrics for analyzing variability and its implementation in software product lines: A systematic literature review	x		
S5	Felfernig *et al*. 2004	Consistency-based diagnosis of configuration knowledge bases			x
S6	Felfernig *et al*. 2012a	An efficient diagnosis algorithm for inconsistent constraint sets			x
S7	Felfernig *et al*. 2014a	Conflict Detection and Diagnosis in Configuration			
S8	Galindo *et al*. 2018	Automated analysis of feature models: Quo vadis?		x	
S9	Mavin *et al*. 2017	Does Goal-Oriented Requirements Engineering Achieve its Goal?			x
S10	Ochoa *et al*. 2018	A systematic literature review on the semi-automatic configuration of extended product lines	x		
S11	Ruhe *et al*. 2003	Trade-off analysis for requirements selection			x
S12	Ruhe & Saliu 2005	The art and science of software release planning			x
S13	Svahnberg *et al*. 2010	A systematic review on strategic release planning models	x		
S14	Thakurta 2017	Understanding requirement prioritization artifacts: a systematic mapping study		x	

3.2 Inclusion and exclusion criteria

The articles were selected for inclusion in this study based on the defined inclusion and exclusion criteria. The detailed list of these criteria can be found in Table 2. The main criterion for inclusion was to identify analyses that could be relevant for requirement models. As the emphasis was on the variety of analyses, different algorithms designed for a single analysis were not considered (cf. Section 1.3). This means that even though many papers presented new or improved algorithms to detect, for example, inconsistency in feature models, these papers were not considered, as the consistency check as an analysis was already covered in the start set. Respectively, several analyses for feature models are

related to code metrics; for example, there are analyses that aim to assess the complexity of the features and constraints based on the code (S4). However, for requirement models such as code-based metrics are not relevant, since often requirements are not available as code. Thus, analyses dealing with code metrics were not considered in this study.

Table 2. Detailed inclusion and exclusion criteria.

Study inclusion criteria
1. **The paper is available in full text.** 2. **The paper is peer reviewed.** 3. **The paper is written in English.** 4. **The article describes or reviews analyses for feature models, release models or other models that are relevant for requirement models**
Study exclusion criteria
1. **The analysis method presented in the paper is just a variation of a method previously covered.** 2. **The paper deals with requirement prioritization.** 3. **The analysis method deals with code metrics.** 4. **The analyses focus on the changes of the requirement model over time.**

3.3 Search process and results

In the following, we describe the search iterations that resulted in the 24 included papers (Figure 1).

3.3.1 First iteration

In the start set, there was one paper (S6), which was included in the result set from the start. Thus, it is included in the final number of 24 papers, i.e. S6=P3.

Backward snowballing: In the first iteration, initially 240 papers with potentially interesting titles were selected based on the 14 papers in the start set (set A, see Figure 1). Of these papers, the duplicates were eliminated and a total of 84 papers were selected for further inspection based on their abstracts. Of these papers, 72 were eliminated based on the inclusion and exclusion criteria, leaving a total of 12 papers (set B) for the next iteration. These 12 papers were read in full and included in the study.

Forward snowballing: Forward snowballing was done to all start set papers, except for S3, because the paper S8 is a follow-up of this paper and we felt that it would be adequate to select only the most recent paper for forward snowballing. In forward snowballing, duplicates and already handled papers were ignored in the beginning, and in total 13 papers were selected for closer inspection based on their titles. Of these 13 papers a set of four (set C) were selected for the next iteration based on the inclusion and exclusion criteria.

3.3.2 Second iteration

Forward snowballing was done to all 17 papers (sets B and C) that had been selected during the first iteration. During this second iteration from the set B (12 papers), forward snowballing was done to all papers except for P1, because S3 and S8 are follow-ups of this paper. Forward snowballing of 11 papers from the set B produced a set of 25 interesting papers based on their abstracts. Of these 25 papers, a set of 7 papers (set D) was selected for the next iteration. From the set C (4 papers), forward snowballing did not produce relevant papers.

3.3.3 Third iteration

Forward snowballing from the set D (7 papers) did not produce relevant papers for this study. Therefore, we adjudged to have reached a saturated level of evidence and ended the search phase.

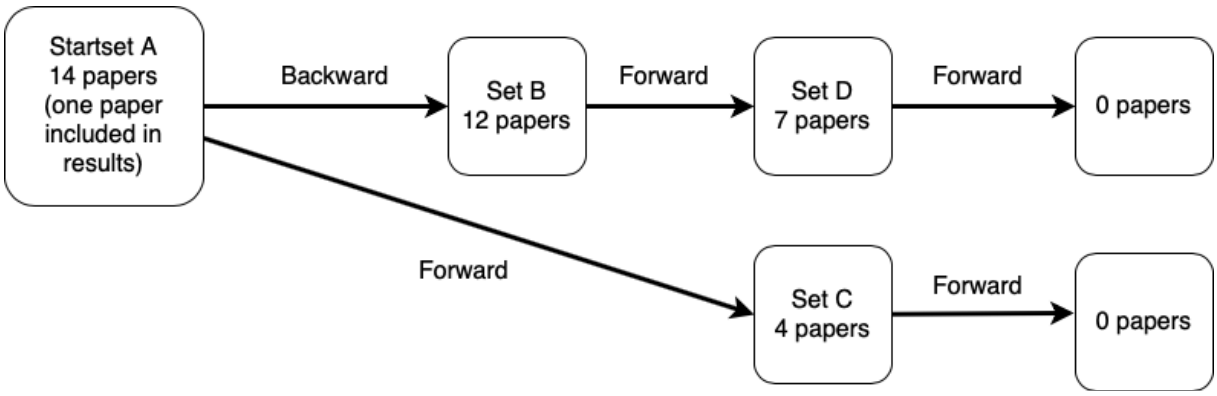

Figure 1. Visualisation of the snowballing process and included papers in the set resulting from each iteration.

3.3.4 Final set of papers

The final set consisted of 24 papers that are hereafter referred to by acronyms P1, P2 etc (Table 3).

Table 3. Papers included in the results.

ID	Reference	Title	Publication type
P1	Benavides *et al*. 2005	Automated reasoning on feature models	Journal
P2	Felfernig & Zehentner 2011	CoreDiag: eliminating redundancy in constraint sets	Workshop
P3	Felfernig *et al*. 2012a	An efficient diagnosis algorithm for inconsistent constraint sets	Journal
P4	Felfernig *et al*. 2012b	Resolving anomalies in configuration knowledge bases	Conference
P5	Felfernig *et al*. 2013	Towards anomaly explanation in feature models	Workshop
P6	Felfernig *et al*. 2014b	Redundancy Detection in Configuration Knowledge	Book chapter
P7	Felfernig & Schubert 2011	Personalized diagnoses for inconsistent user requirements	Journal
P8	Felfernig *et al*. 2018	Anytime diagnosis for reconfiguration	Journal
P9	Hemakumar 2008	Finding contradictions in feature models	Workshop
P10	Junker 2001	QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms	Workshop
P11	von der Massen & Lichter 2003	Requiline: a requirements engineering tool for software product lines	Workshop
P12	von der Massen & Lichter 2004	Deficiencies in feature models	Workshop
P13	Perez-Morago *et al*. 2015	Efficient Identification of Core and Dead Features in Variability Models	Journal
P14	Polat Erdeniz *et al*. 2018	LearnDiag: A direct diagnosis algorithm based on learned heuristics	Conference
P15	Reinfrank *et al*. 2015a	A goal-question-metrics model for configuration knowledge bases	Workshop
P16	Reinfrank *et al*. 2015b	Intelligent supporting techniques for the maintenance of constraint-based configuration systems	Workshop

Table 3. Continued.

ID	Reference	Title	Publication type
P17	Reiterer *et al*. 2014	WeeVis	Book chapter
P18	Rincón *et al*. 2014	An ontological rule-based approach for analyzing dead and false optional features in feature models	Journal
P19	Rincón *et al*. 2015	Method to identify corrections of defects on product line models	Journal
P20	Salinesi & Mazo 2012	Defects in Product Line Models and how to Identify them	Book chapter
P21	Trinidad *et al*. 2006	Isolated features detection in feature models	Workshop
P22	Trinidad *et al*. 2008	Automated error analysis for the agilization of feature modelling	Journal
P23	Trinidad *et al*. 2009	Abductive reasoning and automated analysis of feature models: how are they connected?	Workshop
P24	Zhang *et al*. 2011	Feature model validation: a constraint propagation-based approach	Conference

3.4 Data analysis

All papers considered in this study were initially listed either in an Excel worksheet or in a Word document together with a short description of its relevance. From the relevant papers, the data was extracted based on the research questions, as is summarised in the next chapter.

4 Results

In this chapter, we present the findings organized by the research questions. However, we first present an example of a requirement model, which will be used to demonstrate possible errors encountered in requirement modelling throughout this chapter. Then we examine analyses that have been done to requirement models (RQ1) and analyses from feature modelling that could be used with requirement models (RQ2). Finally, we concentrate on those analyses that could be relevant in the context of OpenReq requirement models (RQ3).

Feature and requirement model analyses can use both deductive and abductive reasoning to gain information about the models (P23). In deductive reasoning, the question can be, for example, if the feature or requirement model is valid (i.e. consistent), and in abductive reasoning the question would be why the model is invalid. In diagnosis, abductive reasoning is used to determine which components of the system are incorrect and why.

4.1.1 Example requirement model

A requirement model as considered in this thesis or feature model can be represented as a configuration task, which can be defined as different types of knowledge representations such as constraint satisfaction problems (CSP), binary decision diagrams (BDD), and SAT or description logics (P2, P13). A model consists of a set of requirements that have their own requirement domains. These requirement domains define the states of the requirements and in the simplest case the requirement domain can consist of two Boolean values (*true* or *false*). This means that the requirement can either be included (*true*) or not included (*false*) in the product.

In addition to the requirements and their domains, the model also includes *constraints* that are used to describe the dependencies between requirements. Typically, a requirement model, like a feature model, is presented as a tree structure by decomposition constraint (dependency). The nodes (requirements or features) can also be connected by cross-branch constraints (dependencies) such as *requires* or *excludes* (cf. Section 2.2.3). The root of the tree is the product itself.

These constraints can be divided into two groups: customer requirements and requirement model constraints. Customer requirements are also known as user

requirements (see Section 2.1), and they describe the demands or needs of a user or customer. For example, one such requirement could be simply that the customer wants to have a web page for her/his company. A further example of this type of a requirement would be that the web page must have a form for sending data (for example, a webpage user's name and email address) to a database server. It is not necessary for the customer to be aware of all requirement model constraints, which define dependencies between the individual requirements. For example, one type of a constraint would be that a form on a webpage requires a certain type of a library to function properly.

The example requirement model presented below does not represent an actual product, but is meant to illustrate dependencies between requirements. Dependencies are represented by lines and arrows.

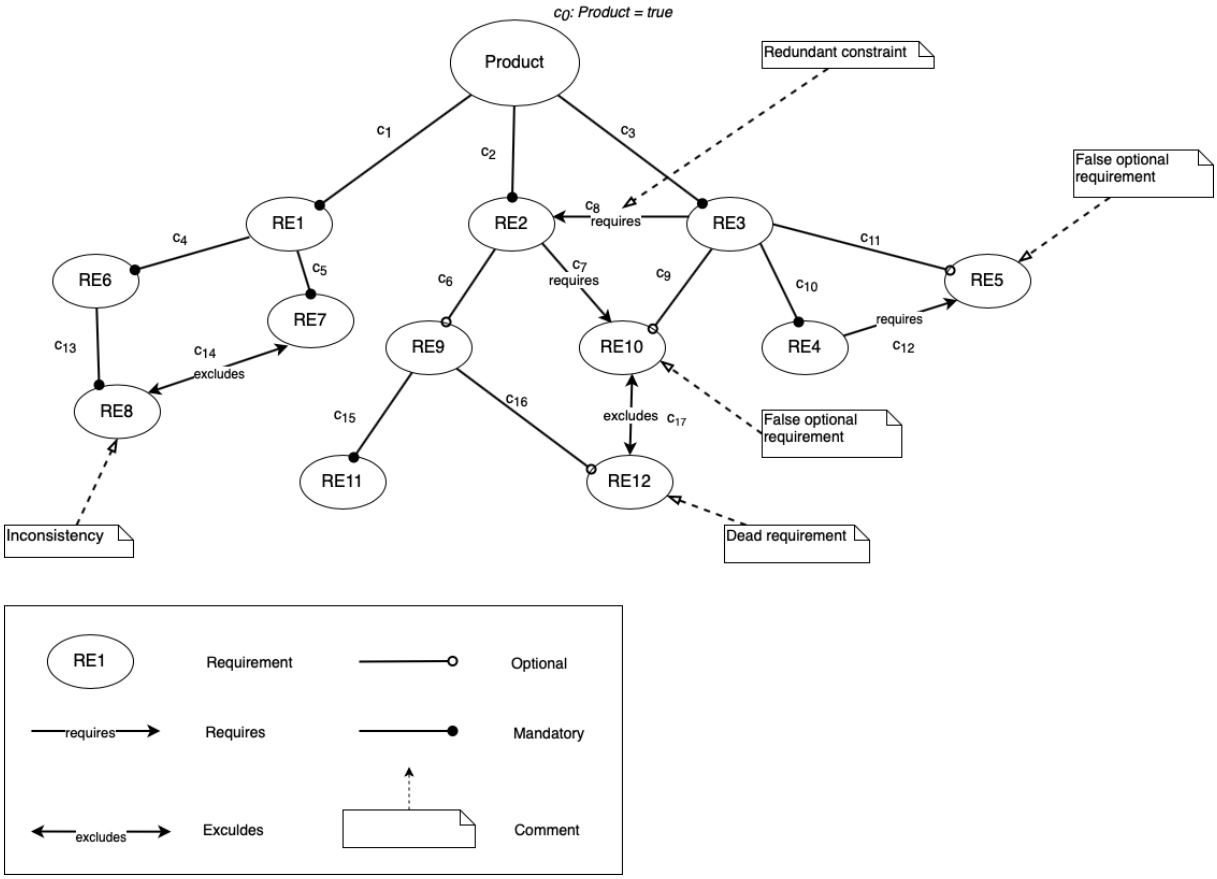

Figure 2. An example of an inconsistent requirement model. The model is based on a notation introduced by Czarnecki and Kim (2005).

Formally, the example model used in this study (Figure 2) can be defined as a CSP that is a triple *(R, D, C)*, where *R*[11] represents a set of requirements $R = \{r_1, r_2, \ldots, r_n\}$, *D* is the set of corresponding requirement domains $D = \{dom(r_1), dom(r_2), \ldots, dom(r_n)\}$ where $(dom(r_i) = \{ true, false \}$, and $C = CR \ U \ CF$ is a set of constraints that restrict the possible configurations, which can be created based on the model. A requirement model configuration means a complete assignment of requirements $r_i \in R$. In the example model, the product is the root of the model, but it could also be defined as r_0. The set *C* consists of a set *CR* containing possible customer or user requirements, and of a set *CF* which contains requirement model constraints. The set *CF* is also often referred to as the configuration knowledge base (C_{KB}) of the model (P2, P5). In the following text and code examples, CF and C_{KB} refer to the same constraint set.

The example requirement model (Figure 2) can be defined as the following CSP *(R, D, C)*:

- *R = {product, RE1, RE2, …, RE12}*
- *D = {dom(product) = {true, false}, dom(RE1) = {true, false}, dom(RE2) = {true, false}, …, dom(RE12) = {true, false}}*
- *C= CR U CF*
  - *CR: { c_0: product = true, c_{18}: RE9 = true }*
  - *CF = {*

    c_1: product ↔ RE1,

    c_2: product ↔ RE2,

    c_3: product ↔ RE3,

    c_4: RE1 ↔ RE6,

    c_5: RE1 ↔ RE7,

    c_6: RE9 — RE2,

    c^7: RE2 → RE10,

    c_8: RE3 → RE2,

    c_9: RE10 — RE3,

    c_{10}: RE3 ↔ RE4,

    c_{11}: RE5 — RE3,

[11] In the original papers that handle feature models R is marked as $F = \{f_1, f_2, \ldots, f_n\}$.

c_{12}: $RE4 \rightarrow RE5$,

c_{13}: $RE6 \leftrightarrow RE8$,

c_{14}: $\neg RE7 \vee \neg RE8$,

c_{15}: $RE9 \leftrightarrow RE11$,

c_{16}: $RE12$ — $RE9$,

c_{17}: $\neg RE10 \vee \neg RE12$ }

Notation:

RE1 ↔ RE2: RE2 is a mandatory child of RE1

RE1 — RE2: RE1 is an optional child of RE2

RE1 → RE2: RE1 requires RE2

¬RE1 ∨ ¬RE2: RE1 excludes RE2 (and vice versa)

In this model, the dependencies are understood as global, meaning that an *excludes* dependency between two requirements (e.g. Requirement RE7 and Requirement RE8) prevents these two requirements from being in the same product.

In the user requirements *CR*, the constraint c_{18}: $RE9 = true$ means that the user wishes to include Requirement RE9 in the product, even though it has been defined as optional in the model. In a realistic requirement model, this would mean that Requirement RE9 has been assigned to a release to be implemented.

This model is illustrated in Figure 2 using a feature diagram-like notation, and it represents requirements and their dependencies. There are several issues or errors in the model, and these will be handled in the other sections of this chapter.

As there are no established means to represent releases, requirements' efforts and priorities, these are not considered in the example model. However, these could be presented as additional constraints in a requirement model. There are also other requirement properties besides release, effort and priority, but these three are the most commonly appearing. In the following sections, the main focus is on the requirement model errors that stem from the dependencies (constraints) between requirements.

4.2 RQ1: What analyses have been done to requirement models (RM)?

Based on the literature studied, it appears that there is very little research that is focused on the requirement models as defined in OpenReq. There is a large amount of literature on release planning models (see e.g. S2, S13). However, such release planning models (cf. Section 2.3) are a means to construct a release plan from existing requirements by assigning requirements to releases, but we are interested in already existing models instead of constructing them.

However, some articles handling requirements also mention analyses done to the requirement models. For example, like feature models, requirement models can be checked for inconsistencies.

One tool that was being developed to manage both feature and requirement models is RequiLine. It was published as a prototype that can be used to manage feature and requirement models and perform queries regarding, for example, the dependencies between features and requirements. RequiLine can also perform consistency checks on the models. Consistency check in RequiLine means detecting contradicting dependencies and possible errors in the model format (P11). The exact technical details of the algorithm for the consistency check were not mentioned in the article that described the prototype of RequiLine. We were not able to find further information regarding this tool.

Consequently, even though there are some mentions of consistency check on requirement models, it is more common in feature modelling. In the following section, we specify consistency check as it is applied in the context of feature models.

4.2.1 Consistency check

If a requirement model considered as a feature model is inconsistent, there must exist at least two constraints between selected mandatory features or requirements that contradict each other. A simple case would be that a requirement A requires B, but B excludes A.

Inconsistent feature models are also called *void*, as they describe no product at all. Essentially this means that the root feature is a dead feature, which means that

inconsistent or void feature models can be considered a special case of *dead features* (cf., Section 4.3.2). (P22)

Formally, a feature or requirement model is inconsistent (void) if *inconsistent(CF U {c_0})*, where *CF* contains the feature or requirement model constraints, and c_0 represents the product. Thus, no customer requirements are necessarily considered.

The example requirement model described in Figure 2 represents an inconsistent requirement model. In the model the constraints c_4, c_5, c_{13} and c_{14} lead to an inconsistency. Requirement RE1 has a mandatory child (c_4) Requirement RE6 that in turn has a mandatory child (c_{13}) Requirement RE8. However, Requirement RE8 excludes (c_{14}) Requirement RE7, which in turn is a mandatory child (c_5) of Requirement RE1. This leads to a contradiction, which causes the inconsistency.

4.3 RQ2: What analyses could be adopted from other domains to requirement models?

We discovered several different analyses that are used in feature modelling and knowledge-based configuration. As stated in Introduction and Section 3.2, all code metrics related analyses of feature models were excluded from this study, together with model transformations.

There are analyses that could be used with requirement models but were considered to be too vague or not relevant enough. One such analysis was determining the *number of products* in a feature model. A feature model can represent several potential products depending on the different combinations of features. With software product lines, determining the number of products can provide information on the flexibility and complexity of a SPL (P1). However, we were unable to find compelling evidence of the usefulness of this analysis in industry, and hence decided to exclude it. Another similar and only SPL-related analysis is *False SPL*, which means a model that describes only one valid product, as opposed to a real SPL, which describes several products (P19, P20).

Another analysis we considered not relevant for our requirement models is *unnecessary refinement* detection. Here the idea is to detect features or variables that are always bound together, i.e. they always have the same assignment (P15, P16). For example, in a feature

model of a car there is always the wheel, so whenever *car=true*, *wheel=true*, in which case the wheel as a separate feature could be considered an unnecessary refinement. However, even though unnecessary refinements may perhaps add some complexity to feature models, with requirement models the idea is to present all requirements that must be implemented in a product and in which order. In this case with a requirement model of a car, it would be necessary to have a wheel listed as one mandatory requirement among others.

We considered analyses related to anomalies detection especially interesting, so these will be covered in this section.

4.3.1 Diagnosis of inconsistencies

Reiter's theory of diagnosis (Reiter 1987) is one of the most widely used frameworks used to work with the diagnosis problem. However, we use the definition presented by Felfernig *et al*. (2015) who define diagnosis "*as a recommendation of a set of knowledge base elements (in our case constraints) that represent a source of an inconsistency, i.e. should be analysed within the scope of testing and debugging operations*." In the following sections, we use the term diagnosis to refer to a set of problematic constraints or dependencies in a feature or requirement model.

As stated in Section 4.2.1, a requirement model can be tested for inconsistency. Often, when a model is found to be inconsistent, we are also interested in the reasons for the inconsistencies, i.e. the diagnosis. Several different algorithms have been developed to provide diagnoses of inconsistent feature models (e.g. P3, P9, P10). These analyses perform a consistency check and in a case of an inconsistent model determine which constraints in the model cause this inconsistency.

One algorithm that is used to get a diagnosis proposal is *FastDiag* (Algorithm 1). It is a *divide and conquer*-based diagnosis algorithm where a set C represents all constraints being searched for diagnosis, and AC is the set containing all constraints in a feature or requirement model. The idea is to divide the set C into two subsets C_1 and C_2 and test if the first subset becomes consistent. If this happens, the diagnosis is further searched from the second subset and the first subset can be omitted. If the first subset does not become consistent, the diagnosis is searched recursively from both subsets of the set C.

Eventually, inconsistent constraints are collected into two sets D_1 and D_2, which will be combined and returned as a diagnosis. The algorithm assumes that constraints have been listed in the sets in ascending priority order. It finds a diagnosis solution, but it may not be the optimal solution. (P3, P5)

For simplicity, here and in all the following pseudocodes the methods *isConsistent()* and *isInconsistent()* refer to the consistency check explained above. Function *isConsistent()* returns *True* if a feature/requirement model is consistent, and respectively, *isInconsistent* returns *True* if the model is inconsistent.

Algorithm 1. Original FastDiag (C, AC)

```
C: set of constraints being searched for diagnosis
AC: all constraints in the model, C_KB U C_R
D: set containing eventually the diagnosis (constraints), at first empty
q: length of the constraint set C

func FastDiag(C, AC)
    if isEmpty(C) or isInconsistent(AC-C) return ∅
    else return FD(∅, C, AC);

func FD(D, C, AC)
    if D != ∅ and isConsistent(AC) return ∅;
    if singleton(C) return C;
    k= q/2;
    C_1 = {c_1…c_k};
    C_2 = {c_k+1…c_q};
    D_1 = FD(C_1, C_2, AC-C_1);
    D_2 = FD(D_1, C_1, AC-D_1);
    return (D_1 U D_2);
```

FastDiag is used for consistency diagnoses in the Wiki technologies-based WeeVis environment that is designed to support the development and maintenance of configuration knowledge bases (P17).

4.3.2 Dead feature or requirement detection

Dead features mean features that are in a feature model, but cannot appear in any product of a Software Product Line. In a model, a requirement or feature is dead when it has been, for example, defined as an optional child of another requirement, but the dead requirement cannot ever be present in a consistent model due to constraints with other requirements. This means that a model containing dead features is not necessarily in itself

inconsistent if the dead requirements have not been defined as mandatory or selected by the users. (P5, P9, P22, P24) Dead features have also been called isolated features (P21).

Dead requirements can be detected by defining each requirement as mandatory in the model, and checking if this leads to an inconsistent model (P5, P9, P13, P18, P22, P24).

Formally, a feature f_i is (fully) dead when *inconsistent(CF U {c_0} U {f_i = true})*, i.e. the feature f_i is not included in any of the possible consistent model configurations (P5). There can also be conditionally dead features, which are present in some but not in all configurations (P5, P9).

For conditionally dead features apply both *inconsistent(CF U {c_0} U {f_i = true})* and *consistent(CF U {c_0} U {f_i = true})*, which means that the feature is dead in only some of the configurations (P5).

A very simple example of a dead feature would be a case when a feature A has a mandatory child feature B, but due to a modelling error there is an *excludes* dependency between these features. In this case, both features would be dead features. Other common cases of dead features have been listed in P22.

A case for a conditionally dead feature may arise, for example, when one feature has two optional children that exclude each other. This situation would basically be similar to the children having an alternative dependency to their parent. Therefore, a conditionally dead feature per se is not necessarily an error in a model.

The example requirement model (Figure 2) shows one example of a dead requirement. In this case, the requirement RE12 is an optional child (c_{16}) of the requirement RE9, but there exists an excludes-type dependency (c_{17}) between requirements RE12 and RE10. Because RE10 is required (c_7) for the mandatory requirement RE2, it must be in the product, whereas the optional requirement RE12 can never be in the same product.

The inconsistency check can be used to detect dead features or requirements; basically, one could just run the check for each feature f_i marked as f_i=true, and collect those features that cause the model to become inconsistent into a list of dead features. However, if one wishes to have a diagnosis of the reasons for a dead feature, an algorithm such as *FastDiag*

can be used to detect constraints associated with dead features or requirements (P5). This means that the input would be *FastDiag(C=CF, AC=CF U {c_0} U {f_i = true})*. *FastDiag* would then output a set of constraints that lead to the feature or requirement f_i being dead.

Another algorithm for dead feature detection (Algorithm 2) was introduced in P16. Here dead features are called dead domain elements, and the idea is to return a set containing all dead domain elements. The algorithm checks each domain for each variable to see if any of the domain assignments leads to an inconsistency.

Algorithm 2. DeadDomainElements(C, V)

```
C: constraints in the knowledge base
V: features in the knowledge base
D: set containing eventually all dead domain elements
for each v_i in V do
  for all dom_j ∈ dom(v_i) do
    C' = C U {v_i=dom_j}
    if isInconsistent(C') then
      D = D + {v_i=dom_j}
    end if
  end for
end for
return D
```

4.3.3 False optional feature or requirement detection

A feature that is present in all products of a SPL, even though it has not been defined as mandatory, is called a *false optional feature* or a *false variable feature* (P5, P19, P24). Basically, a false optional feature has been defined as an optional child of another feature, but other constraints between the false optional feature and other features make the false optional feature actually mandatory in the model.

False optional features can be detected by checking if the model is inconsistent when the parent feature is present and the optional child feature is not included in the model (P5, P24).

Formally, a feature f_{opt} is a false optional feature if *inconsistent(CF U {c_0} U {f_{parent} = true AND f_{opt} = false}* (P5).

In the example requirement model, Requirements RE5 and RE10 are false optional requirements (Figure 2). Requirement RE5 has been defined as an optional child (c_{11}) of

Requirement RE3, but Requirement RE3 has a mandatory child (c_{10}) Requirement RE4, and Requirement RE4 requires (c_{12}) Requirement RE5. Requirement RE10 has been defined as an optional child (c_9) of Requirement RE3, but mandatory Requirement RE2 requires (c_7) Requirement RE10. Thus, both Requirements RE5 and RE10 are required to be present in the consistent requirement model.

A list of false optional requirements could be constructed in a similar manner as with dead requirements, using the consistency check for all requirements defined as optional. Any consistency check algorithm could be used but because we already described *FastDiag*, which also provides a diagnosis, we use it as an example of an algorithm that could be used for diagnosis of false optional requirements (P5). The input would in this case be *FastDiag(C=CF, AC=CF U {c_0} U {f_{parent} = true AND f_{opt} = false})*.

4.3.4 Full mandatory feature or requirement detection

Features which are present in all consistent solutions of a feature model, are called *full mandatory* or *core features*. This means that any model not having one of these features is inconsistent. (P5, P13) In feature models, full mandatory features are classified as anomalies, even though they do not represent real errors. According to Felfernig *et al*. (P5), detecting full mandatory features would help to assess whether all of them are necessarily mandatory, i.e. if there could be more variability in the model.

Formally, a feature f_i is a full mandatory feature when *inconsistent(CF U {c_0} U {f_i = false})*, i.e. feature f_i must be included in all possible consistent model configurations (P5).

In the example requirement model (Figure 2), for example Requirements RE1, RE2, RE3 and RE6 are full mandatory requirements.

Full mandatory features can be detected in the same manner as the dead features. Basically, the only difference is that, with full mandatory features in the inconsistency check, the feature f_i is defined as being not included, i.e. f_i = *false*, whereas with dead features f_i = *true*. Also, in this case the algorithm *FastDiag* could be used to provide a list of constraints leading to full mandatory features.

Like dead features, full mandatory features can also be defined as full mandatory domain elements. The algorithm to detect these is the same as the Algorithm 2 *DeadDomainElements*, except that now $v_i = dom_j$ is changed to $v_i != dom_j$ (see P16).

4.3.5 Redundancy detection

In constraint sets, redundant constraints refer to constraints that can be deleted from the constraint set or knowledge base without changing the semantics of the remaining constraint set (P2). Basically, this means that in the model there are overlapping constraints, which describe the same thing more than once.

Formally redundant constraints can be described by defining an initial constraint set $S = \{c_1, c_2, \ldots, c_n\}$ that contains a redundant constraint c_i. If c_i is removed from S, then $(S-\{c_i\}) \cup S^C$ is inconsistent (S^C is negation of S) (P2).

In the example requirement model (Figure 2), there is a *requires* dependency (c_8) from Requirement RE3 to Requirement RE2. This constraint can be considered redundant, because both Requirement RE2 and Requirement RE3 have already been listed as mandatory requirements for the product. Alternatively, Requirement RE2 being mandatory could be considered a redundancy, as it could be optional if the *requires* constraint is kept.

According to Felfernig *et al*. (P5), the redundancy rate with feature models can vary considerably depending on how many engineers are developing the model. The models with only one or few developers tend to have a lower degree of redundant constraints (expressed as a percentage of redundant constraints in the whole constraint set) than models with several different developers.

Different redundancy detection algorithms can be applied depending on whether the expected redundancy degree is high or low. In the case in which the number of redundant constraints is expected to be higher, the *CoreDiag* algorithm (together with *CoreD*) is one proposed solution. (P2) With a lower degree of redundancy, the *FMCore* algorithm is more efficient (P5).

CoreDiag

The *CoreDiag* algorithm (Algorithm 3) uses the *CoreD* algorithm (Algorithm 4) to calculate a maximal set of redundant constraints from a configuration knowledge base C_{KB}, which contains the domain-specific constraints (i.e. the constraints in the feature/requirement model; *CF*) (P2). The algorithms are based on the *divide and conquer* principle; *CoreD* divides the constraint set *C* into two subsets C_1 and C_2 and determines if one of the subsets already contains a minimal core. Minimal core means that the constraints in it preserve the semantics of C_{KB}, meaning that removing any of these constraints would lead to an inconsistency in the model. If one of the sets already contains the minimal core, the other is not considered. The algorithm *CoreDiag* returns a set of all redundant constraints. This set is the complement of the minimal core, i.e. the non-redundant constraints.

Even though in the original paper Felfernig and Zehentner (P2) describe *CoreDiag* performing the analysis to the whole configuration knowledge base, in practice the analysis can also be done to a constraint set *S*, which can represent either a subset of C_{KB} or the whole configuration knowledge base, $S \subseteq C_{KB}$. For the sake of clarity, in the following pseudocodes of the algorithms *CoreDiag* and *FMCore* the set $S = \{c_1, c_2, \ldots, c_n\}$ is used to mark the constraint set containing redundant constraints to be analysed.

In the worst case, the number of needed consistency checks in *CoreDiag* is $2c * log_2(n/c) + 2c$, where *c* is the number of constraints in the minimal core, and *n* the size of the set *S* (P2). In the best case, the complexity is $log_2(n/c) + 2c$, which requires that all constraints in the minimal core are positioned in one branch of the search tree in *CoreD*. *CoreDiag* performs better if the number of constraints in the minimal core is low.

CoreDiag has been implemented in the WeeVis environment (P17).

Algorithm 3. CoreDiag(S)

```
S: the set of constraints
S^c: the complement of S

S^c = {¬c1 V ¬c2 V,…, V ¬cn}
return(S – CoreD(S^c, S^c, S))
```

Algorithm 4. CoreD(B, D, C)

```
B: consideration set
D: constraints added to B
C: set of constraints being checked

if D != ∅ and isInconsistent(B) return ∅;

if singleton(C) return(C)

k= q/2
C1 = {c1…ck}
C2 = {ck+1…cq}
D1 = CoreD(B U C2, C2, C1)
D2 = CoreD(B U D1, D1,C2)

return (D1 U D2)
```

FMCore

The *FMCore* algorithm (Algorithm 5) presented by Felfernig *et al*. (P5) is a version of the earlier *Sequential* algorithm that can be used to detect redundant constraints (P2). The difference between these two algorithms is the way that they check the inconsistency: in the original *Sequential* algorithm, the inconsistency check is done to S - $\{c_i\}$ U $\{S^c\}$, whereas in the *FMCore* algorithm the set S^c is replaced with $\{\neg c_i\}$. However, in Felfernig *et al*. (P6), the algorithm which is basically *FMCore* is called *Sequential*. In this version the algorithm returns a set containing redundant constraints, instead of a set containing non-redundant constraints. Otherwise, it is the same as the *FMCore* as described in P5.

The purpose of the *FMCore* algorithm is to construct and return a set that contains only non-redundant constraints (P5). This set S_{temp} contains in the beginning of the algorithm all constraints (S), and a consistency check is done to all constraints $c_i \in S$. The idea is that redundant constraints do not change the semantics of S, meaning that S - $\{c_i\}$ U $\{S^c\}$ remains inconsistent. As the algorithm runs, all redundant constraints are removed from the set S_{temp} and eventually this set forms the minimal core (the original constraint set S without any redundant constraints).

Algorithm 5. FMCore(S)

```
S: the set of constraints
S^c: the complement of S
S_temp: the set containing (eventually) all non-redundant constraints

S_temp = S;
for each c_i in S_temp do
  if isInconsistent((S_temp - {c_i}) U {¬c_i}) then
    S_temp = S_temp - {c_i};
  end if
end for
return S_temp
```

Both in the best and the worst cases the number of consistency checks made by the *FMCore*/*Sequential* algorithm is *n*, which is the number of constraints in the set *S*. This means that the algorithm's time complexity is *O(n)*. The algorithm finds the actual minimal core, which contains all non-redundant constraints (P5, P6).

Felfernig *et al*. (P5) tested the performance of *FMCore* with different feature models, the largest containing 172 features and 205 constraints with a redundancy rate of 0.71. With the largest model, *FMCore*'s runtime was 3261 ms, but, interestingly, with one smaller model (72 features, 96 constraints, redundancy rate 0.64), the runtime was considerably longer, 5070 ms. Felfernig *et al*. (P6) show that *FMCore*/*Sequential* started performing worse than *CoreDiag* when the redundancy rate rose to 0.5.

The original *Sequential* algorithm (P2) and *CoreDiag* have been implemented in the ICONE configuration knowledge base development and maintenance environment (P4).

There are differing views on the detrimental effect of redundant constraints on feature models. On one hand, the redundant constraints considerably increase the effort for calculating a solution for a knowledge base and make the knowledge base more difficult to maintain and develop. Thus, recognizing and eliminating them is considered important (P2, P16). On the other hand, in many cases adding redundant constraints to the model increases its readability (P12). Redundancies can also be intentionally added to emphasize selected relationships between features, and other researchers do not even consider redundant constraints feature model errors (P22).

4.4 RQ3: Which of these analyses would be applicable to OpenReq requirement models and how?

In RQ3, we elaborate the analyses found earlier in the context of the OpenReq project and especially the requirement model as defined in OpenReq. In the project, the University of Helsinki has been developing microservices that have been used to check the consistency of a requirement model (also called a release plan), and to provide a diagnosis of inconsistent requirements or dependencies. There are other analyses that could be used to gain more information on OpenReq requirement models, and these will be discussed in this section, together with the consistency check (Table 4).

It should be noted that in contrast to the feature models, in requirement models there is the temporal aspect of releases. This aspect is not considered in the analyses of feature models, but in requirement models releases can be considered as constraints that should be taken into account when configuring the CSP. Basically, requirements that have been assigned to releases are the same as user requirements in feature models.

In the OpenReq ontology, *requires* and *excludes* dependencies can be defined as either local or global (see Section 4.2.1). However, in our experience the local case is more rarely in practise than the global one.

Table 4. Analyses for requirement models.

Analysis	Description
Inconsistency detection	Determine if a RM is inconsistent
Diagnosis of inconsistencies	Find reasons for inconsistencies
Dead requirement detection	Find requirements that cannot be in any product
False optional requirement detection	Find requirements that have been marked as optional, but are actually mandatory
Full mandatory requirement detection	Find requirements that are mandatory in all model configurations
Redundancy detection	Find redundant constraints

4.4.1 Consistency check

As stated in Section 4.2.1, the consistency check has not been well defined to consider requirement properties such as releases. However, the consistency check is fundamental

to other analyses presented in Section 4.3, and here we present the issues related to the consistency check in the context of requirement models.

In a requirement model, the contradictory dependencies between requirements in a same model or release lead to an inconsistent model. As opposed to feature models, in requirement models inconsistencies can also arise when, for example, a requirement A requires a requirement B, and if B has not been assigned to the same release as A, or to one of the releases preceding A's release, the model is inconsistent. This inconsistency does not disappear even if B would be assigned to a later release, since A requiring B means that B must be released before or at the same time as A. Naturally, after both have been released, the product is practically consistent, at least regarding these requirements, because it is not relevant in which of the implemented releases requirements were.

An inconsistency can also occur regarding other properties of a release. For example, the summed effort of requirements assigned to a same release can exceed the capacity of the release. For example, if the release number one (capacity 8) has three requirements that each have an effort of 3 assigned to it, the release model is inconsistent.

In a requirement model, inconsistencies can appear within an individual release or between different releases. For instance, there might be a case, where a release contains contradicting requirements, which makes it inconsistent. Other releases might be consistent, but one inconsistent release makes the whole model inconsistent. In another case, there might be a requirement D in one release that requires a requirement C, but since C has been assigned to a later release, the requirement model is inconsistent. In this case, the dependency *requires* is understood globally, i.e. the necessary requirement must be in the same or in a preceding release. The same applies for the *excludes* dependency, as in a global case two excluding requirements cannot be in the same project. In some situations, the *excludes* dependency can be defined as local, which means that two requirements with an *excludes* dependency between them cannot be in the same release, but can be assigned to separate releases.

In the case of Qt's Jira issues, there are rules regarding their assignments to Fix Versions (essentially releases). The first rule is that all child Jira issues, which have the same or a higher priority than their parent issues, must not be assigned in a later Fix Version

(release). The second rule is that if an issue A requires an issue B, the issue B must not have a lower priority than A, nor can B be in a later Fix Version (release) than A. Both of these rules must be considered when analysing the consistency of a requirement model of Qt issues.

As Qt does not use effort as a requirement property, inconsistencies due to effort and capacity issues are not possible. Also, since there is no *excludes* dependency, contradicting dependencies will not cause inconsistencies. The inconsistencies in Qt requirement models will arise from violations to the two rules explained above. Thus, when analysing Qt issues, we must modify the analyses normally used in feature modelling to consider releases and priorities. At the same time, we should make sure that the consistency check and other analyses discussed in the following sections also work for standard OpenReq JSON conforming requirement models that use properties such as effort and capacity.

4.4.2 Diagnosis of inconsistencies

At the moment, the requirement model consistency check provided by the microservices developed at the University of Helsinki uses an adaptation of the *FastDiag* algorithm (P3) for the diagnosis of inconsistent models. The algorithm is used with a CSP solver (Choco Solver[12], Prud'homme *et al*. 2017), and it has been modified to consider priorities, releases, release capacities and requirement effort in addition to just requirements and dependencies. In practice, *FastDiag* performs two diagnosis proposals: it suggests either dependencies or requirements that are behind the inconsistency.

The original *FastDiag* algorithm does not consider releases and their priorities or inconsistencies that result in in requirements' efforts exceeding the capacities of releases. It should be noted that not all companies use the effort as an attribute of a requirement (e.g. the Qt Company). Checking a model for a capacity exceeding also does not either necessarily require defining the model as a CSP. However, since CSP solvers can be easily used to detect capacity exceedings, they are also used for this purpose in the OpenReq project.

In the *FastDiag* algorithm, all constraints are expected to be in a priority order from the lowest to highest priority. If the input is, for example, constraints $\{c_1, c_2, c_3\}$, c_3 has the

[12] www.choco-solver.org

highest priority. In a case when c_2 and c_3 contradict each other, e.g., when c_2 is A requires B and c_3 is A excludes B, *FastDiag* lists only c_2 as an inconsistent constraint, since it has a lower priority than c_3. This may be problematic, as there are cases when it would be better if the algorithm would point out all constraints related to the inconsistency. The same issue applies to requirements, since they are also prioritized and the algorithm only concentrates on the lower priority requirements.

4.4.3 Detecting dead requirements

Dead requirements represent clear errors in defining a requirement model, and detecting them is one analysis applicable to OpenReq requirement models. Conditionally dead requirements may occur when there are two alternative requirements, and one of them gets assigned to a release, rendering the other one dead. This conditionally dead requirement may remain in the model, but since it cannot be assigned to any release, it later becomes obsolete.

In a requirement model, a requirement can be considered dead within one release or within several releases. In both cases, the requirement model itself becomes inconsistent if there is an attempt to include the dead requirement. In the example requirement model (Figure 2), if one release would contain Requirements RE2 and RE10, and the customer would like to also include Requirement RE9 and with it Requirement RE11, this release would be consistent. However, adding Requirement RE12 to the release would create an inconsistency within this release, and also within the whole model. Even if Requirement RE12 would be assigned to another release, the whole requirement model would be inconsistent, and even though the original release, with only the Requirements RE2, RE9, RE10 and RE11, would still be consistent.

The case discussed above illustrates a situation where a requirement is dead because of inter-requirement dependencies. These kinds of cases are similar to the ones discussed in articles on detecting dead features in feature models (see Section 4.3.2). With requirement models, there might be a situation where the combined effort of requirements exceeds the capacity of a release they have been assigned to (see Section 4.2.1). In this case, one or several of the requirements could also be considered dead, since they cannot be in a consistent requirement model due to the capacity exceeding. This kind of a situation is possible with OpenReq requirement models, and the *FastDiag* algorithm

with a CSP solver could also be used to find dead requirements in cases when the requirements are dead due to releases and efforts.

In another similar but probably not very practical case, the requirements' efforts exceed the capacity of all available releases which they can be assigned to, e.g., in terms of project length. Also, this case would lead to several dead requirements. In practice, this is the scenario of requirements prioritisation, when it is decided which requirements are implemented.

The *FastDiag* algorithm could be used to detect dead requirements in a similar manner to that explained in Section 4.3.2. The algorithm should be modified to also consider releases. For checking all requirements, *FastDiag* is required to be run for all individual requirements. This could be too time-consuming with larger models, but the process could be perhaps speeded up by checking only requirements marked as optional.

It is noteworthy that in Qt Jira, there is no *excludes* dependency or effort that is required for dead requirements to arise. Therefore, dead requirement detection is not meaningful.

Algorithm 6. DeadAndFullMandatoryReqs(C, V)

```
C: constraints in the knowledge base
V: requirements in the knowledge base
D: set containing eventually all dead domain elements
F: set containing eventually all full mandatory domain elements

for each v_i in V do
  for all dom_j ∈ dom(v_i) do
    C' = C U {v_i=dom_j}
    if isInconsistent(C') then
      D = D + {v_i=dom_j}
    end if
    C'' = C U {v_i!=dom_j}
    if isInconsistent(C'') then
      D = D + {v_i!=dom_j}
    end if

  end for
end for
return D
```

If we use the Algorithm 6 *DeadDomainElements* to detect all dead requirements, we can modify the algorithm to also detect full mandatory requirements on the same run. This

new algorithm is described above, but some modifications might be needed because of releases, unless these can be incorporated into the CSP solver models.

4.4.4 Detecting false optional requirements

False optional requirements represent errors in a requirement model and they could be detected from OpenReq requirement models. In the case of Qt's Jira issues, a false optional requirement could be defined in a situation where a requirement A (priority 1) has a child requirement B (priority 2), but a requirement C (priority 1) requires the requirement B. All requirements have been assigned to the same release. In this case, the requirement B is a false optional requirement, and its priority should be marked higher. If A and C would be in one release and B in a later release, the requirement model would be inconsistent.

The *FastDiag* algorithm could be used to detect false optional requirements in the same way as described in Section 4.3.3.

4.4.5 Detecting full mandatory requirements

Full mandatory requirements have not been really defined in requirement modelling but they could be understood as requirements that all have the same high priority. For example, some Qt issues have several children, and there could be cases when all children have been marked as having the same priority as their parent, even though some of them could perhaps be prioritized lower. Detecting full mandatory requirements could point out such cases in which some requirements have been assigned too high a priority, and thus full mandatory requirements could discover the lack of variability in a requirement model.

The algorithm *DeadAndFullMandatoryReqs* (Algorithm 6) could be used to detect full mandatory requirements.

4.4.6 Detecting redundant constraints

Detecting redundant constraints means the same in requirements models as in feature models. However, with requirement models, redundant constraints can be used to emphasize the order in which the requirements should be implemented. For example, in the example requirement model (Figure 2), the redundant *requires* dependency between Requirement RE3 and Requirement RE2 could mean, that Requirement RE2 must be implemented before Requirement RE3.

With OpenReq requirement models, we see no need to automatically delete redundant constraints, but it could be useful if the analysis of a model would provide a list of (possibly) redundant constraints. If we would use the *FMCore*/*Sequential* algorithm (P4), it would be best if the algorithm returned a set containing the detected redundant constraints instead of the minimal core, which contains the non-redundant constraints. This version of the algorithm, published in P6, is presented below (Algorithm 7).

Algorithm 7. Sequential(S)

```
S: the set of constraints
S^c: the complement of S
S_temp: the set containing (eventually) all non-redundant constraints

S_temp = S;
for each c_i in S_temp do
  if isInconsistent((S_temp - {c_i}) U {¬c_i}) then
    S_temp = S_temp - {c_i};
  end if
end for
return S - S_temp
```

In this way, the diagnosis of a model would contain information on possible redundancies, and the engineer could review the results and decide if any of the constraints deemed redundant should be removed from the requirement model.

5 Discussion

Requirement models in OpenReq context are in many ways similar to feature models, but there are also several differing aspects. Requirement models have the temporal aspect of releases and requirement priorities and in some cases, issues related to efforts, and all these must be considered in the analyses of the models.

As the aim in the OpenReq project is to analyse very large-scale requirement models, all analyses discovered in this study should be conductible on models with tens of thousands of requirements. Concrete real-life examples of these type of models can be constructed from The Qt Company's Jira issues.

5.1 Answers to research questions

Regarding RQ1, i.e. analyses that have been done to requirement models, we found only a little evidence from the literature on analyses of requirement models similar to the OpenReq requirement model. Since release planning-related analyses were excluded, as they concentrate on algorithms for constructing a model rather than analysing existing models, outside the OpenReq project there was only one case (P11) that clearly mentioned consistency checking of requirement models. It is possible that requirement models may not be considered separate from feature models, since the distinction between a feature and a requirement is not always clear. This might partly explain the lack of analyses for explicit requirement models.

Another possible reason for the lack of analyses for mature requirement models could be that, in industry, requirements and their dependencies are not always that clearly defined. If requirements are only defined in shared Word documents, constructing a requirement model based on them can be laborious to do manually and challenging to automate. Also, in agile software development, requirements tend to change considerably, which makes exact model construction challenging (cf. Section 2.1.2)

Even though there are similarities between feature models and requirement models, with requirement models there is a clear temporal aspect of releases, which is absent from most feature models. That is why feature model analyses are not applicable to OpenReq requirement models without some modifications. The releases bring in consecutive

releases, their capacity, and the effort of requirements. There are cases when developers work on a product for several years by gradually adding features to intermediate configurations to achieve the final desired product configuration (multi-step configuration problem, see e.g., Czarnecki *et al*. 2005, White *et al*. 2009), but this process appears to be closer to release planning than the problem of analysing a mature requirement model.

As covered in RQ2, i.e. analyses that could be adopted from other domains to requirement models, many of the anomaly detection-focused analyses used in feature modelling could be applicable in the requirement model context. Most of the found analyses handled both the detection and diagnosis of model anomalies and errors. We found no evidence of these type of analyses having been done to requirement models, but they should be applicable to OpenReq models, for example, by making changes to the current version of the *FastDiag* algorithm.

In RQ3, we concentrated on the analyses that could be applicable to OpenReq requirement models. Of the analyses first introduced in Section 4.3, the most relevant for OpenReq style requirement models would most likely be the detections of dead requirements and redundancies.

Even though dead features are mentioned frequently in the studied feature modelling papers, we are not fully certain how problematic they are in practice with requirement models. For example, as The Qt Company does not use effort with their requirements, nor do they have an *excludes* dependency, it is practically impossible to have dead requirements in the models constructed from Qt Jira issues.

As requirement models have not been extensively researched, it is difficult to say how prevalent dead requirements are in real industry requirement models. Even if dead requirements are not an issue with The Qt Company, other companies use effort as a requirement attribute, and thus may have dead requirements in their models. Therefore, implementing dead feature detection for OpenReq requirement models would help the requirements engineering community to research this issue.

There may be other differences between requirement and feature models. For example, the dependency *excludes* is widely reported as an important constraint in feature

modelling, but we do not know how much it is used in practise in requirement modelling. Carlshamre *et al*. (2000) reported not finding any conflicting requirements in their case study of five industrial sets of requirements. We know that the Qt Company does not use *excludes*, and it would be interesting to know how relevant this dependency type actually is in industry. It is possible that, in industry, conflicts are eliminated at an early stage of the requirements management process as noted in Carlsharmre *et al*. (2000).

As the redundancy rates with OpenReq requirement models have not been studied yet, we do not know for certain whether *FMCore*/*Sequential* or *CoreDiag* would be a better choice for the algorithm. However, we assume the redundancy rate to be relatively low, as the problem with Jira data is that there are not enough links rather than too many links (Tiihonen *et al*. 2019), in which case *FMCore*/*Sequential* would be more appropriate.

As false optional requirements have not been studied, we do not know how common or relevant they are in real-life requirement models, e. g., with Qt data. They are not major errors in a requirement model, but detecting them could perhaps help to make the models simpler or point out errors in dependency assignments.

Full mandatory requirements do not represent actual errors in a requirement model and we are not sure if detecting them would be relevant in the OpenReq context. Detecting full mandatory requirements would inform about the variability of the requirement model, but the practical relevancy of this would need to be studied. Detecting them should work with OpenReq models, so it could be possible to test their relevance.

5.2 Validity and reliability

There are potential threats to validity, which may have affected the results of this study. Threats related to the construct validity of the study concern the acquisition of the data. In this study, the data were acquired by snowballing from a preselected start set of articles. Snowballing was selected as a method for searching the literature because it helped to limit the search field. Even though there were 14 articles in the start set, the articles on the next round (both backward and forward snowballing) came from only five[13] of the start set articles. A different start set might have produced different results,

[13] S1, S3, S5, S6, S7.

and it is possible that some relevant analyses were not discovered, which affects the reliability of the study.

From the start set, the interesting next round articles were selected initially only by looking at their titles, so some relevant studies may have been overlooked. Thus, the outcome of the selection process could be different if this study would be replicated by some other researcher. However, we do not believe that relevant analyses, at least from a domain of feature modelling, were missed, since the start set contained several comprehensive literature reviews.

One potential issue is that all selected analyses and algorithms are based on the research of the same research group (Felfernig *et al*.). This was mainly due to the fact that the algorithm *FastDiag* was already used in the OpenReq project, and modifying it could be more convenient than trying to incorporate a completely new algorithm. However, the ideas of analyses such as dead feature or redundancy detection were initially encountered in papers by other groups.

Regarding the external validity of the study, even though the focus has been on the OpenReq requirement models, and especially on The Qt Company's data, there are aspects that could be generalized for wider use. The OpenReq JSON, which has been created during the project, is very similar to ReqIF format, which means that the analyses that can be done to OpenReq models, could also be conducted on requirement models using ReqIF. Of course, there should be modifications to algorithms, but the basis of analyses remains the same.

5.3 Future work

The aim of this study was to research analyses that could be relevant to requirement models, but we were not concerned about finding the best possible algorithms for these analyses. In the future research, however, it could be beneficial to study different algorithms to find the most efficient ways to analyse large-scale requirement models. This would mean looking more closely into other model representations (e.g. Binary Decision Diagrams, see P13), and into algorithms such as, for example, *PersDiag* (P2), *FlexDiag* (P8) and *LearnDiag* (P14). For example, besides the *FastDiag* algorithm, there are other ways to detect dead features, for example, a requirement model could be defined as a Binary

Decision Diagram and analysed using the algorithm proposed in P13. This algorithm was tested with several feature models, including randomly generated models with up to 5000 features, and it was found to be significantly faster than previous SAT or BDD-based procedures. It is possible to modify the University of Helsinki microservices to use other model representations, such as BDDs, but at the moment this is not considered necessary. However, since OpenReq is interested in large-scale requirement models, at some point it might be necessary to pay attention to efficiency issues, if the current and planned algorithms prove to be too time-consuming.

As dead requirements and redundancies appear to be the most promising potential analyses, implanting them in the microservices of the University of Helsinki should be the first step in adding more analysis tools for OpenReq requirement models. It would be important to test various real-life requirement models to determine the relevancy of anomalies such as dead requirements and redundancies. This would also help to determine whether the automatic removal of anomalies is beneficial.

We did not find studies on the algorithms mentioned in Chapter 4 being used to analyse large-scale real-life feature models. In the future, the research on requirement models should aim to use real industry models, such as ones based on Qt data, to analyse not only the prevalence of anomalies, but also the performance of available algorithms on models with tens of thousands of requirements. For example, we do not know how actual models behave in practice or how complex they are to analyse. As stated in Section 2.2.2, most research on feature models is nowadays done to generated feature models. In addition to studying large, real-life requirement models based on Qt data, for comparison it would be important to analyse other requirement models that use, for example, effort as a requirement property or *excludes* as a dependency. It is possible, that other types of constraints are also used in industry and studying their effect on the requirement models could be relevant.

6 Conclusions

In this thesis, we studied analyses that have been done or could be done to requirement models. We searched for literature on relevant analyses and discovered that little research has been done to the analyses of mature requirement models. However, there are several analyses, especially in the domain of feature modelling, which could be applied to detect anomalies and errors in requirement models. Feature models do not consider releases, which are important in requirement models, but many analyses could still be extended for requirement modelling. We identified inconsistency detection, diagnosis of inconsistencies, dead requirement detection, false optional requirement detection, full mandatory requirement detection and redundancy detection. However, more research is needed to establish the relevancy of analyses such as dead features or requirements, and redundant constraints. Dead requirements can be detected in requirement models, but we do not know how relevant dead requirements actually are in industry, as the *excludes* dependency may not be widely used in real-life requirement models. Redundant constraints can add complexity to requirement models and detecting them could be beneficial. However, it is questionable whether automatically deleting errors and anomalies such as redundant constraints, is beneficial, and more research is needed. False optional or full mandatory requirements could be detected, but their relevancy in requirement modelling should be studied.

Acknowledgements

The project OpenReq is funded by European Union's Horizon 2020 Research and Innovation programme under grant agreement No 732463.

Primary studies for the literature review

P1. Benavides, D., Trinidad, P. & Ruiz-Cortés, A. 2005, "Automated reasoning on feature models", *Advanced Information Systems Engineering, CAiSE 2005, Lecture Notes in Computer Science*, vol. 3520, pp. 491–503.

P2. Felfernig, A. & Zehentner, C. 2011. "CoreDiag: eliminating redundancy in constraint sets", In: *22nd International Workshop on Principles of Diagnosis (DX'2011), Murnau, Germany*, pp. 219–224.

P3. Felfernig, A., Schubert, M. & Zehentner, C. 2012a, "An efficient diagnosis algorithm for inconsistent constraint sets", *Artificial Intelligence for Engineering Design Analysis and Manufacturing: AIEDAM,* vol. 26, no. 1, pp. 53–62.

P4. Felfernig, A., Reinfrank, F. & Ninaus, G. 2012b, "Resolving anomalies in configuration knowledge bases", *Foundations of Intelligent Systems. ISMIS 2012, Lecture Notes in Computer Science*, 7661, pp. 311–320.

P5. Felfernig, A., Benavides, D., Galindo, J. & Reinfrank, F. 2013, "Towards anomaly explanation in feature models", *15th International Configuration Workshop CEUR Workshop Proceedings*, pp. 117–124.

P6. Felfernig, A., Reinfrank, F., Ninaus, G. & Blazek, P. 2014b, "Redundancy Detection in Configuration Knowledge", in *Knowledge-Based Configuration: From Research to Business Cases*, pp. 157–166.

P7. Felfernig, A. & Schubert, M. 2011, "Personalized diagnoses for inconsistent user requirements", *Artificial Intelligence for Engineering Design, Analysis and Manufacturing: AIEDAM,* vol. 25, no. 2, pp. 175–183.

P8. Felfernig, A., Walter, R., Galindo, J.A., Benavides, D., Erdeniz, S.P., Atas, M. & Reiterer, S. 2018, "Anytime diagnosis for reconfiguration", *Journal of Intelligent Information Systems,* vol. 51, no. 1, pp. 161–182.

P9. Hemakumar, A. 2008, "Finding contradictions in feature models", in: *First International Workshop on Analyses of Software Product Lines (ASPL'08),* pp. 183–190.

P10. Junker, U. 2001, “QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms”, in: *Proc. IJCAI-01, Workshop on Modelling and Solving Problems with Constraints,* Seattle, WA.

P11. von der Massen, T. & Lichter, H. 2003, “Requiline: a requirements engineering tool for software product lines”, in: F. van der Linden (ed.), *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE), Lecture Notes in Computer Sciences*, vol. 3014, Springer-Verlag, Siena, Italy.

P12. von der Massen, T. & Lichter, H. 2004, “Deficiencies in feature models”, in: T. Männisto, J. Bosch (eds.), *Workshop on Software Variability Management for Product Derivation—Towards Tool Support*.

P13. Perez-Morago, H., Heradio, R., Fernandez-Amoros, D., Bean, R. & Cerrada, C. 2015, "Efficient Identification of Core and Dead Features in Variability Models", *IEEE Access,* vol. 3, pp. 2333–2340.

P14. Polat Erdeniz, S., Felfernig, A. & Atas, M. 2018, “LearnDiag: A direct diagnosis algorithm based on learned heuristics”, *Lecture Notes in Computer Science*, 11117, pp. 190–197.

P15. Reinfrank, F., Ninaus, G., Peischl, B. & Wotawa, F. 2015a, "A goal-question-metrics model for configuration knowledge bases", *17th International Configuration Workshop, CEUR Workshop Proceedings*, pp. 123–130.

P16. Reinfrank, F., Ninaus, G., Wotawa, F. & Felfernig, A. 2015b, "Intelligent supporting techniques for the maintenance of constraint-based configuration systems", *17th International Configuration Workshop, CEUR Workshop Proceedings*, pp. 31–38.

P17. Reiterer, S., Felfernig, A., Blazek, P., Leitner, G., Reinfrank, F. & Ninaus, G. 2014, "WeeVis" in *Knowledge-Based Configuration: From Research to Business Cases*, pp. 297–307.

P18. Rincón, L.F., Giraldo, G.L., Mazo, R. & Salinesi, C. 2014, "An ontological rule-based approach for analyzing dead and false optional features in feature models", *Electronic Notes in Theoretical Computer Science,* vol. 302, pp. 111–132.

P19. Rincón, L., Giraldo, G., Mazo, R., Salinesi, C. & Diaz, D. 2015, "Method to identify corrections of defects on product line models", *Electronic Notes in Theoretical Computer Science*, vol. 314, pp. 61–81.

P20. Salinesi, C. & Mazo, R. 2012, "Defects in Product Line Models and how to Identify them", in Elfaki, A. (ed), *Software Product Line - Advanced Topic, InTech editions*, pp. 97–122.

P21. Trinidad, P., Benavides, D. & Ruiz-Cortés, A. 2006, "Isolated features detection in feature models", *CAiSE Forum, CEUR Workshop Proceedings*.

P22. Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A. & Toro, M. 2008, "Automated error analysis for the agilization of feature modeling", *Journal of Systems and Software,* vol. 81, no. 6, pp. 883–896.

P23. Trinidad, P., Benavides, D. & Ruiz-Cortés, A. 2009, "Abductive reasoning and automated analysis of feature models: how are they connected?", *Variability Modelling of Software-Intensive Systems*, pp. 145–153.

P24. Zhang, G., Ye, H. & Lin, Y. 2011, "Feature model validation: a constraint propagation-based approach", *10th International Conference on Software Engineering Research and Practice*, Las Vegas, Nevada.

References

Afzal, U., Mahmood, T. & Shaikh, Z. 2016, "Intelligent software product line configurations: A literature review", *Computer Standards & Interfaces,* vol. 48, pp. 30–48.

Ameller D., Farré C., Franch X. & Rufian G. 2016, "A Survey on Software Release Planning Models", *Product-Focused Software Process Improvement, PROFES, Lecture Notes in Computer Science*, vol 10027., pp. 48–65.

Batory, D., Benavides, D. & Ruiz-Cortes, A. 2006, "Automated analysis of feature models: Challenges ahead", *Communications of the ACM,* vol. 49, no. 12, pp. 45–47.

Benavides, D., Segura, S. & Ruiz-Cortes, A. 2010, "Automated analysis of feature models 20 years later: A literature review", *Information Systems,* vol. 35, no. 6, pp. 615–636.

Cao, L. & Ramesh, B. 2008, "Agile requirements engineering practices: An empirical study", *IEEE Software,* vol. 25, no. 1, pp. 60–67.

Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B. & Natt och Dag, J. 2001, "An industrial survey of requirements interdependencies in software product release planning", *Proceedings of the IEEE International Conference on Requirements Engineering,* pp. 84–91.

Czarnecki, K., Helsen, S. & Eisenecker, U. 2005, "Staged configuration through specialization and multilevel configuration of feature models", *Software Process Improvement and Practice*, vol. 10, no. 2, pp. 143–169.

Czarnecki, K. & Kim, P. 2005, "Cardinality-based feature modeling and constraints: a progress report", in: *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*.

Dahlstedt, Å.G. & Persson, A. 2005, "Requirements interdependencies: State of the art and future challenges" in *Engineering and Managing Software Requirements*, pp. 95–116.

Dardenne, A., van Lamsweerde, A. & Fickas, S. 1993, "Goal-directed requirements acquisition", *Science of Computer Programming,* vol. 20, no. 1-2, pp. 3–50.

El-Sharkawy, S., Yamagishi-Eichler, N. & Schmid, K. 2019, "Metrics for analyzing variability and its implementation in software product lines: A systematic literature review", *Information and Software Technology,* vol. 106, pp. 1–30.

Felfernig, A., Friedrich, G., Jannach, D. & Stumptner, M. 2004, "Consistency-based diagnosis of configuration knowledge bases", *Artificial Intelligence,* vol. 152, no. 2, pp. 213–234.

Felfernig, A., Schubert, M. & Zehentner, C. 2012a, "An efficient diagnosis algorithm for inconsistent constraint sets", *AI Edam-Artificial Intelligence for Engineering Design Analysis and Manufacturing,* vol. 26, no. 1, pp. 53–62.

Felfering, A., Reiterer, S., Reinfrank, F., Ninaus, G. & Jeran, M. 2014a, "Conflict Detection and Diagnosis in Configuration", Knowledge-Based Configuration, pp. 73–87.

Felfernig, A., Reiterer, S., Stettinger, M. & Tiihonen, J. 2015, "Intelligent techniques for configuration knowledge evolution", *ACM International Conference Proceeding Series*, pp. 51–58.

Galindo, J.A., Benavides, D., Trinidad, P., Gutiérrez-Fernández, A-M. & Ruiz-Cortés A. 2018, "Automated analysis of feature models: Quo vadis?", *Computing* pp. 1–47.

Glinz, M. 2017, "A Glossary of Requirements Engineering Terminology", Version 1.7, *International Requirements Engineering Board IREB*, 130 pp.

Horkoff, J., Aydemir, F.B., Cardoso, E., Li, T., Maté, A., Paja, E., Salnitri, M., Mylopoulos, J. & Giorgini, P. 2016, "Goal-Oriented Requirements Engineering: A Systematic Literature Map", *IEEE 24th International Requirements Engineering Conference, RE 2016*, pp. 106–115.

IEEE 610.12 1990, *Standard Glossary of Software Engineering Terminology*.

IEEE 1413.1 2002, *IEEE Guide for Selecting and Using Reliability Predictions Based on IEEE 1413*.

IEEE Standard 828 2012, *IEEE Standard for Configuration Management in Systems and Software Engineering*.

IEEE/EIA Standard 12207.0 1996, *Industry Implementation of International Standard ISO/IEC 12207:1995 (ISO/IEC 12207) Standard for Information Technology -- Software Life Cycle Processes*.

ISO/IEC/IEEE 24765 2010, *Systems and software engineering — Vocabulary*.

Kang, K., Cohen, S., Hess, J., Novak, W. & Peterson, S. 1990, "Feature-oriented domain analysis (FODA) feasibility study", *Technical Report CMU/ SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990*.

Karlsson, J., Wohlin, C. & Regnell, B. 1998, "An evaluation of methods for prioritizing software requirements", *Information and Software Technology,* vol. 39, no. 14–15, pp. 939–947.

Kitchenham B. A. & Charters S. 2007, "Guidelines for performing systematic literature reviews in software engineering", *Version 2.3, EBSE Technical Report, EBSE- 2007-01*, Keele University.

Lettner, M., Rodas, J., Galindo, J.A. & Benavides, D. 2019, "Automated analysis of two-layered feature models with feature attributes", *Journal of Computer Languages,* vol. 51, pp. 154–172.

Mavin, A., Wilkinson, P., Teufl, S., Femmer, H., Eckhardt, J. & Mund, J. 2017, "Does Goal-Oriented Requirements Engineering Achieve its Goal?", *IEEE 25th International Requirements Engineering Conference (Re),* pp. 174–183.

Murphy, T., Revang, M. & Wurster, L. 2016, "Market Guide for Software Requirements Definition and Management Solutions", Gartner.

Mäenpää, H., Raatikainen, M., Tiihonen, J., Kojo, T. & Männistö, T. 2017, ”Intelligent, community-driven requirements engineering: OpenReq Finland”, The 13th International Conference on Open Source Systems (OSS 2017), Buenos Aires, Argentina, poster.

Ochoa, L., Gonzalez-Rojas, O., Juliana, A.P., Castro, H. & Saake, G. 2018, "A systematic literature review on the semi-automatic configuration of extended product lines", *Journal of Systems and Software,* vol. 144, pp. 511–532.

Paetsch, F., Eberlein, A. & Maurer, F. 2003, "Requirements engineering and agile software development", *Proceedings of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE*, pp. 308–313.

Prud’homme, C., Fages, J-G. & Lorca, X. 2017, “Choco Documentation”, TASC – LS2N CNRS UMR 6241, COSLING S.A.S.

Quer, C., Franch, X., Palomares, C., Falkner, A., Felfernig, A., Fucci, D., Maalej, W., Nerlich, J., Raatikainen, M., Schenner, G., Stettinger, M. & Tiihonen, J. 2018, “Reconciling practice and rigour in ontology-based heterogeneous information systems construction”, *The Practice of Enterprise Modeling, Lecture Notes in Business Information Processing* 335, pp. 205–220.

Raatikainen, M., Tiihonen, J. & Männistö, T. 2019, "Software product lines and variability modeling: A tertiary study", Journal of Systems and Software, vol. 149, pp. 485–510.

Regnell, B., Svensson, R.B. & Wnuk, K. 2008, “Can we beat the complexity of very large-scale requirements engineering?”, *Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, 5025*, pp. 123–128

Reiter, R. 1987, “A theory of diagnosis from first principles”, *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95.

Riegel, N. & Doerr, J. 2015, “A systematic literature review of requirements prioritization criteria”, *Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science*, 9013, pp. 300–317

Ries, E. 2011, The Lean Startup, Crown Publishing Group, Random House, Inc., New York, pp. 320.

Ruhe, G., Eberlein, A. & Pfahl, D. 2003, "Trade-off analysis for requirements selection", *International Journal of Software Engineering and Knowledge Engineering,* vol. 13, no. 4, pp. 345–366.

Ruhe, G. & Saliu, M. 2005, "The art and science of software release planning", *IEEE Software,* vol. 22, no. 6, pp. 47–53.

Savolainen, J. 2011, Product Line Management Techniques with Requirement and Feature Models, Aalto University publication series, Doctoral dissertations 114/2011, 89 pp.

Segura, S., Parejo, J.A., Hierons, R.M., Benavides, D. & Ruiz-Cortés, A. 2014, "Automated generation of computationally hard feature models using evolutionary algorithms", Expert Systems with Applications, vol. 41, no. 8, pp. 3975–3992.

Sommerville, I. 2016, "Software Engineering", 10th edition, Pearson Educated Limited, Harlow, England, 810 pp.

Svahnberg, M., Gorschek, T., Feldt, R., Torkar, R., Bin Saleem, S. & Shafique, M.U. 2010, "A systematic review on strategic release planning models", *Information and Software Technology,* vol. 52, no. 3, pp. 237–248.

Szoke, A. 2011, "Conceptual scheduling model and optimized release scheduling for agile environments", *Information and Software Technology,* vol. 53, no. 6, pp. 574–591.

Thakurta, R. 2016, "Understanding requirement prioritization artifacts: a systematic mapping study", *Requirements Engineering,* vol. 22, no. 4, pp. 491–526.

Thüm, T., Batory, D. & Kästner, C. 2009, "Reasoning about edits to feature models", *Proceedings - International Conference on Software Engineering*, pp. 254–264.

Tiihonen, J., Raatikainen, M., Myllärniemi, V. & Männistö, T. 2016, "Carrying ideas from knowledge-based configuration to software product lines", *International Conference on Software Reuse, Lecture Notes in Computer Science* 9679, pp. 55–62.

Tiihonen, J., Raatikainen, M., Myllyaho, L., Lüders, C.M. & Männistö, T. 2019, "Coping with inconsistent models of requirements", *Configuration Workshop,* CEUR Workshop Proceedings.

White, J., Doughtery, B., Schmidt, D., & Benavides D. 2009, "Automated reasoning for multi-step software product-line configuration problems", *Proceedings of the Software Product Line Conference*, pp. 11–20.

Wohlin, C. 2014, "Guidelines for snowballing in systematic literature studies and a replication in software engineering", *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, ACM International Conference Proceeding Series*.

Yu, E. 1997, "Towards modelling and reasoning support for early-phase requirements engineering", *Proceedings of ISRE '97: 3rd IEEE International Symposium on Requirements Engineering*, vol. 97, pp. 226–235.

Zhang, H., Li, J., Zhu, L., Jeffery, R., Liu, Y., Wang, Q. & Li, M. 2014, "Investigating dependencies in software requirements for change propagation analysis", *Information and Software Technology,* vol. 56, pp. 40–53.