Information and Software Technology 98 (2018) 44–58

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Feature interaction in software product line engineering: A systematic mapping study

Larissa Rocha Soares*[a,b], Pierre-Yves Schobbens[c], Ivan do Carmo Machado[a], Eduardo Santana de Almeida[a]

[a] Computer Science Department, Federal University of Bahia, Salvador, BA, Brazil
[b] RiSE - Reuse in Software Engineering, Salvador, BA, Brazil
[c] University of Namur, Belgium

ARTICLE INFO

Keywords:
Feature interaction
Software product lines
Systematic mapping

ABSTRACT

Context: Software product lines (SPL) engineering defines a set of systems that share common features and artifacts to achieve high productivity, quality, market agility, low time to market, and cost. An SPL product is derived from a configuration of features which need to be compounded together without violating their particular specifications. While it is easy to identify the behavior of a feature in isolation, specifying and resolving interactions among features may not be a straightforward task. The feature interaction problem has been a challenging subject for decades.

Objective: This study aims at surveying existing research on feature interaction in SPL engineering in order to identify common practices and research trends.

Method: A systematic mapping study was conducted with a set of seven research questions, in which the 35 studies found are mainly classified regarding the feature interaction solution presented: detection, resolution and general analysis.

Results: 43% of the papers deal with feature interaction at early phases of a software lifecycle. The remaining is shared among the other categories: source code detection, resolution and analysis. For each category, it was also identified the main strategies used to deal with interactions.

Conclusions: The findings can help to understand the needs in feature interaction for SPL engineering, and highlight aspects that still demand an additional investigation. For example, often strategies are partial and only address specific points of a feature interaction investigation.

1. Introduction

SPL comprise a family of related software systems based on a set of common features. In SPL engineering, reusable artifacts are developed to achieve decreased cost and time to market, and higher quality and market agility. Since an SPL product contains a configuration of features, for a product to be efficiently delivered, they have to work in harmony, i.e., the features have to be compatible with each other. Otherwise, feature interaction issues could arise and affect the product, leading to unexpected behaviors.

A feature interaction occurs when a feature behavior is influenced by the presence of another feature(s) [1]. Typically, the interaction is not easily identified from the analysis of each feature behavior separately. In order to certify that all the interactions are known and properly resolved, a strategy is to generate each valid product configuration. However, as soon as an SPL starts to grow in number of features, it becomes increasingly difficult to identify and resolve all the interactions. The number of feature interaction candidates is exponential in the number of features, and this statement is known as the feature interaction problem [1].

While it is easy to identify the behavior of a feature in isolation, specifying and resolving interactions with other features is not an easy task. An efficient approach has to combine many related issues, for example, with the number of features in the interaction, the amount of interactions, the different interaction types, human factors, and amount of available software artifacts and requirements.

Hence, the feature interaction problem has been a challenging subject for years [2]. Finding solutions that identify and resolve interactions are crucial. During the last decade, research strategies on SPL have been published in the literature regarding detection [3,4], resolution [5], and analysis [6,7] of feature interaction. Despite the existence of studies to map out available evidence on feature interaction

* Corresponding author.
E-mail address: larissars@dcc.com.br (L.R. Soares).

https://doi.org/10.1016/j.infsof.2018.01.016
Received 27 March 2017; Received in revised form 28 July 2017; Accepted 30 January 2018
Available online 01 February 2018
0950-5849/

for single systems development [2], there is a lack of understanding on common strategies, activities, artifacts and research gaps for interactions in SPL.

A systematic mapping study is a way to investigate the state-of-the-art and identify research topics that researchers and practitioners could address. Recently, a variety of systematic mapping studies have been conducted to different SPL fields in order to investigate, for example, SPL testing [8], adoption [9], agile methods [10], non-functional properties [11], and traceability [12].

This systematic mapping study aims at synthesizing existing research related to feature interaction solutions for SPL engineering. Hence, we categorize interactions according to seven research questions. The study focuses on the large amount of research that has been developed in the last years. We analyzed studies published between the years 2004 and 2016, although no lower limit has been set. As a result, 35 studies were found to be relevant and mainly classified regarding to the SPL development lifecycle stages and the feature interaction solution presented, either detection, resolution or general analysis.

The remainder of this paper is organized as follows. In Section 2, we discuss the main concepts related to feature interaction in SPL engineering. Section 3 presents the systematic mapping study protocol in details. Section 4 describes the classification scheme adopted in this study and results. Section 5 discusses the mapping study findings, implications of research, and directions. Section 6 presents the main threats to the validity of our study. Finally, Section 7 summarizes the research presented in this paper.

2. Background

2.1. Software product lines (SPL)

The variety of features is what mainly differentiate a system from another. SPL can be described as a family of systems created and developed from a set of features. SPL engineering explores the commonalities and manages variabilities among related products, in which it is possible to establish a common platform on top of software assets that can be systematically reused and assembled into different products.

A feature is “a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option” [13]. Based on the selection of features, software engineers can configure distinct products satisfying a range of common and variable features, which comprising both functional and non-functional properties [14].

In SPL engineering, features are usually classified as [15]: (i) mandatory, a feature must be selected whenever its parent feature is selected; (ii) optional feature, a feature may or may not be selected; (iii) *OR* feature group, one or more features in the group must be selected; and (iv) *XOR* (alternative) feature group, when one and only one of the features in the group must be selected.

There are several paradigms to develop an SPL, such as: feature-oriented software development (FOSD), aspect-oriented software development, and component based software engineering [13]. Among these, FOSD is an emerging paradigm that enable customization, and synthesis of software products. The FOSD paradigm has been used in the SPL development in order to take advantage of the systematic application of features in all phases of software lifecycle.

2.2. Feature-oriented software development (FOSD)

The software development process based on FOSD relies on the concept of a feature to analyze, design, as well as to implement software systems. The FOSD paradigm corresponds to a collection of methods, tools, languages and formalisms connected by features. Basically, FOSD involves four phases: (i) domain analysis (FDA), (ii) domain design and specification (FDD), (iii) domain implementation (FDI), and (iv) product configuration and generation (FPC) [13]. Since the feature concept is spread through all those phases, approaches for reducing feature interactions problems are also discussed over them.

Feature modeling is the main activity of the FDA phase, whose objective is to identify variabilities and commonalities in a given domain. The architecture of an SPL is designed in the FDD phase, through either formal or informal specifications and modeling languages. In FDI, features are developed to meet its specifications. Finally, the FPC phase is responsible for generating a software product according to user’s requirements. FOSD encourages an automatic software generation based on tools that support a valid features selection. There are several commercial and academic tools available to assist software engineers in finding a valid selection and support a product development, such as: Gears [16], pure::variants [17], and FeatureIDE [18].

2.3. Feature interaction

Feature interaction has been widely discussed in the telecommunications domain. During the 80’s [2], many solutions covering different lifecycle stages emerged in that community. A widespread feature interaction example is related to the features *call waiting* and *call forwarding* of a telephone system. When both are present at the same time, the system behavior is ill-defined. *Call waiting* allows managing two interleaved calls – one is suspended while another one is being answered. *Call forwarding when busy* requires to specify a phone number to forward new calls that arrive when the phone is busy. If these features are used together, and a new call is received when the phone is busy, it does not know how to proceed: it can either forward or suspend the new call. Feature interaction can cause unexpected situations, and may even go against the system specification [19].

Although the feature interaction problem came to light in the telecommunications area, it has been recognized as a general problem in different fields, such as: in software engineering to predict, detect and resolve interactions [20]; cyber-physical systems to reduce CO emissions [21]; automotive contradictory physical forces that lead to unsafe behavior [22]; and comprehension of feature interaction for the Internet of Things [23].

In our work, we focus on software engineering, which proved to be effective for software production on a large scale [24]. While an SPL may comprise a huge number of features, the number of possible interactions is even exponentially bigger. A single feature in an SPL product may interact with another feature or a set of features, creating a high-degree complex interaction. Moreover, a feature behavior may be, for example, dependent on presence conditions (e.g., the busy condition on the telephone example aforementioned) and input configurations (e.g., user entries, unpredictable variables’ values), which can further increase the complexity of detecting interactions. Thus, even if a product behaves as expected most of the time, in specific conditions it might present unexpected interactions on either data or control flow.

Solutions to the feature interaction problem can find a very wide application, including in the domains of smart home systems [25], automotive systems [26], email systems [6], embedded medical devices [27], antivirus products [28], databases [3,29], web systems [7], network [6], among others.

2.3.1. Feature interaction in SPL engineering

In SPL engineering, feature interaction is usually defined by means of a feature *behavior*, i.e., changes in the behavior of the features involved in interaction, which do not occur when the features are used in isolation. Feature interaction can also manifest in *non-functional* attributes, for instance when features in combination have an influence on a specific attribute, such as performance [29]. Also, feature interaction can be seen through the *user’s* point of view. For example, for a system to be perfectly configured and ready to be delivered, the software engineer must guarantee that all the feature interactions respect the user’s intentions.

The literature presents many definitions for feature interaction in

the SPL engineering context. Those definitions usually approach three major aspects: feature behavior, non-functional attributes, and user point of view. The main definitions are described next:

- Abal et al. [30]: "Features in a configurable system interact in non-trivial ways, in order to influence each others functionality. When such interactions are **unintended**, they induce bugs that manifest themselves in certain configurations but not in others".
- Apel et al. [1]: "A feature interaction occurs when the **behavior** of one feature is influenced by the presence of another feature (or a set of other features)".
- Apel and Kästner [13]: "A feature interaction is a situation in which two or more features exhibit **unexpected behavior** that does not occur when the features are used in isolation".
- Atlee et al. [4]: "feature interaction is a discrepancy between a features **behavior** in isolation versus its behavior in the presence of other features."
- Hall [31]: "the desired behavior of a feature combination may violate the ideal of individual feature modularity. However, combining independently designed and validated features often leads to **undesirable behaviors** as well. When this occurs, it is termed a feature interaction."
- Mosser et al. [5]: "feature interaction as the identification of a mismatch between the **intention of the user** and the obtained product."
- Schobbens et al. [32]: "A recurrent problem is the one of feature interaction: adding new features may modify the operation of already implemented ones. When this modification is **undesirable**, it is called a **feature interference**."
- Schuster et al. [3]: "Feature interactions describe the common observation that two or more features (functionally) interact so that the **behavior** of the underlying program may be changed".
- Siegmund et al. [29]: "An interaction occurs when a particular feature combination has an **unexpected** influence on **performance**."

In summary, a feature interaction can be described as a situation in which a feature influences another feature in a positive or negative way. Without an earlier analysis, interactions would only be discovered during feature composition, and even with early detection, it still needs resolution, which usually requires domain knowledge.

Although most definitions present interactions as unintended and undesirable properties, not every interaction is harmful to the system. Sometimes, features are combined to cooperate and accomplish their tasks. Priorities and overridings are some of the strategies used by developers to specify the intended resolution of a feature interaction [26]. However, a solution for a feature interaction issue may require an special attention, i.e., it should neither violate the specifications of the features involved in the interaction nor the specifications of the other system's features. Configuring products with distinct features that present a correct interaction and offer stable and functional services is essential to ensure the development of reliable SPL.

2.3.2. Classification

Feature interaction has been categorized under different aspects, such as: (i) according to the software development stage in telecommunications [33]; (ii) based on how they can be detected [31]; and (iii) more recently regarding the order and visibility of an interaction [1].

In the former, Ohta and Harada [33] represented telecommunication services specification as finite state machines (FSM) and described the interaction problem from the standpoint of the FSM. Hall [31] classified interactions in three categories: (type I) the features dictate contradictory behaviors; (type II) there is no immediate contradiction, but an intended property of one participating feature will eventually be violated, and (type III) other unwanted interaction. Hall provided approaches and tools to detect categories I and II, and gave a support to understand category III.

Next, Plath and Ryan [34] refine Hall's type II, according to the feature to which the violated property belongs to. They also introduce lack of commutativity between features as interaction type IV. Later, Apel et al. [1] classified feature interactions by means of two dimensions: order and visibility. Whereas the order reflects the number of the features involved in the interaction, the visibility describes feature interactions as either *external* (if they impact the user-visible behavior of the system) or *internal* (if it breaks internal properties of the system or requires specific interaction code). In this study, we follow Apel et al. [1] classification.

External feature interactions can be classified in two categories: functional and non-functional interactions [1]. The former corresponds to interactions that violate the functional specification of a composed system; and the latter refers to interactions that influence non-functional properties of a composed system, such as performance, reliability and security.

Internal feature interactions can be classified into *structural* or *operational* interactions [1]. Features interact *structurally* when *coordination code* is necessary to deal with the problem caused by the interaction. A coordination code represents an additional piece of code responsible for resolving the interaction, which is supplementary to the code combination of the individual features involved. Coordination code can be surrounded by preprocessor directives involving several features, or written in lifters [35] or derivatives [36]. Features interact *operationally* when the data flows or control flows differ from the combination of the flows of the features involved.

3. Mapping study process

A systematic mapping study aims at presenting an overview of a research area, providing its amount of studies, publication frequency over the years, results and trends [37]. It is also used to provide a visual sampling and a classification of studies, besides identifying publishing forums and research gaps. According to Kitchenham et al. [38], the mapping study's goal is to survey the available knowledge about a topic.

A systematic mapping study process comprises three main phases: planning, conducting and documenting [38]. The planning phase encompasses the development of a protocol, i.e., a framework that includes all the tasks of the mapping study and serves as a guide in the other two phases. In order to design our mapping study protocol, we conducted meetings and brain-storming sessions with SPL researchers, and all this information supported the process and protocol specifications.

Fig. 1 shows the tasks performed in the mapping study process and the outcomes of each phase. In the first, the protocol and research questions are defined; the second corresponds to the execution of the mapping and it is responsible for searching and screening the papers. Two strategies were used to search for papers: automatic search and manual search, and they were based on a set of inclusion and exclusion criteria. In the final phase, we defined a classification scheme and extracted data mainly based on the research questions. The results regarding a detailed analysis of each primary study is presented as a systematic mapping study. In the next sections, all the process tasks are detailed.

3.1. Research questions

As previously stated, the objective of our mapping study is to identify common practices on feature interactions, research trends, open issues and topics for future research. Thus, we focused on identifying **how do the existing approaches deal with feature interactions in SPL**, which is the general question that drives the research. Hence, five more specific questions were derived, and a summary is

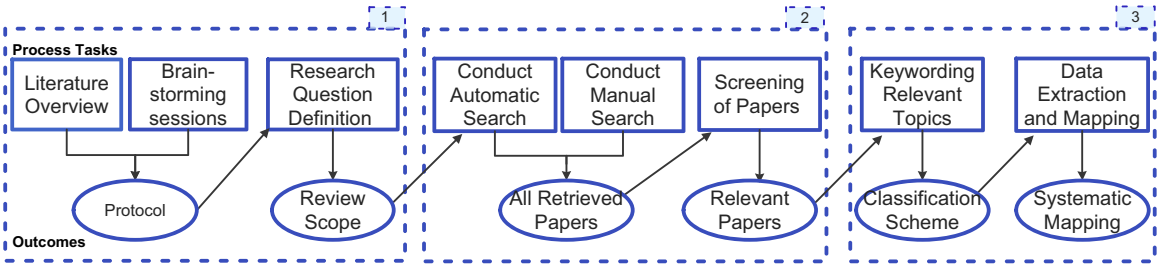

Fig. 1. The systematic mapping process, adapted from Petersen et al. [37].

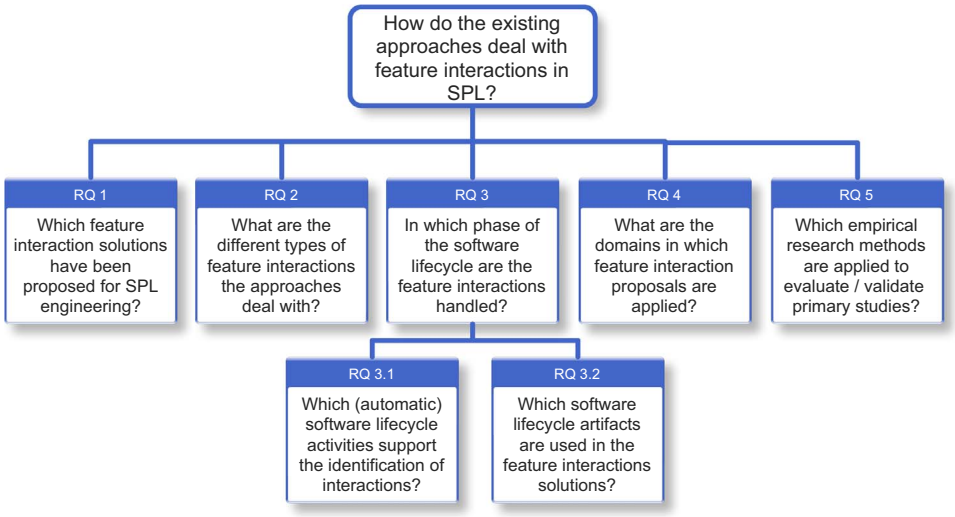

Fig. 2. Research questions of the systematic mapping study.

shown in Fig. 2.

- *RQ 1. Which feature interaction solutions have been proposed for SPL engineering?*The solutions presented by the research papers represent the different approaches to handle feature interaction in an SPL development project. They involve detection, resolution and management of feature interactions. In this question, we aim to investigate these existing solutions, which can indicate the aspects addressed by both research and industry communities, besides potential open rooms for improvement.
- *RQ2. What are the different types of feature interactions the approaches deal with?*To answer this question, we used the Apel et al. [1] visibility classification. This classification is related to the context of a feature interaction, i.e., interactions can be can be either externally-visible or internally-visible.
- *RQ 3. In which phase of the software lifecycle are the feature interactions handled?*FOSD is a paradigm that favors the systematic application of a feature in all phases of the software lifecycle [13]. FOSD aims at facilitating the structuration, reuse, and variation of software in a systematic and uniform way. In this question, we intend to identify in which lifecycle phase the feature interaction solution takes place. In addition, feature interaction solutions can be handled in many different activities of those phases, as for example, during the domain design phase or product configuration. In addition, the following sub-questions were derived:
  - *RQ 3.1. Which software lifecycle activities support the identification of interactions?*The process to identify and resolve interactions usually involves several activities in different software lifecycle phases. For example, the domain analysis phase may include domain scoping, feature modeling and automated reasoning; and the domain design and specification phase, may include architecture modeling and a formal specification [13]. Furthermore, in some cases, part of these activities are automated.
  - *RQ 3.2. Which software lifecycle artifacts are used in the feature interactions solutions?* Different artifacts can be used to support the feature interaction solutions during each phase of the software lifecycle. Examples of those artifacts are: requirements and feature model, project plan, business case and risk assessment.
- *RQ 4. What are the domains in which the feature interaction approaches are applied?* SPL can be applied in many different domains, such as, medical, financial and automotive embedded systems. Investigating the domains in which feature interactions approaches have been applied can indicate: (i) domains where the study on feature interactions are essential; and (ii) domains overlooking this research topic.
- *RQ 5. Which empirical research methods are commonly employed to assess the primary studies?*It is important to analyze the research rigor in feature interactions proposals for SPL engineering. In this question, we aim to investigate the applied empirical research methods (e.g. case study, controlled experiment, quasi-experiment and formal validation). It allows assessing the maturity of evaluation and validation in the area.

3.2. Search strategy

The strategy of searching and selecting the primary studies consisted of three main phases: automatic search, manual search and full-text reading, as shown in Fig. 3. For the first phase, a search query was executed in the digital libraries listed in Table 1. They are key publisher-specific resources [38] and cover almost all important conferences, journals and workshops research studies in the Software Engineering field. The search covered all studies published up to the first quarter of 2016.

The search query was composed of a set of keywords defined

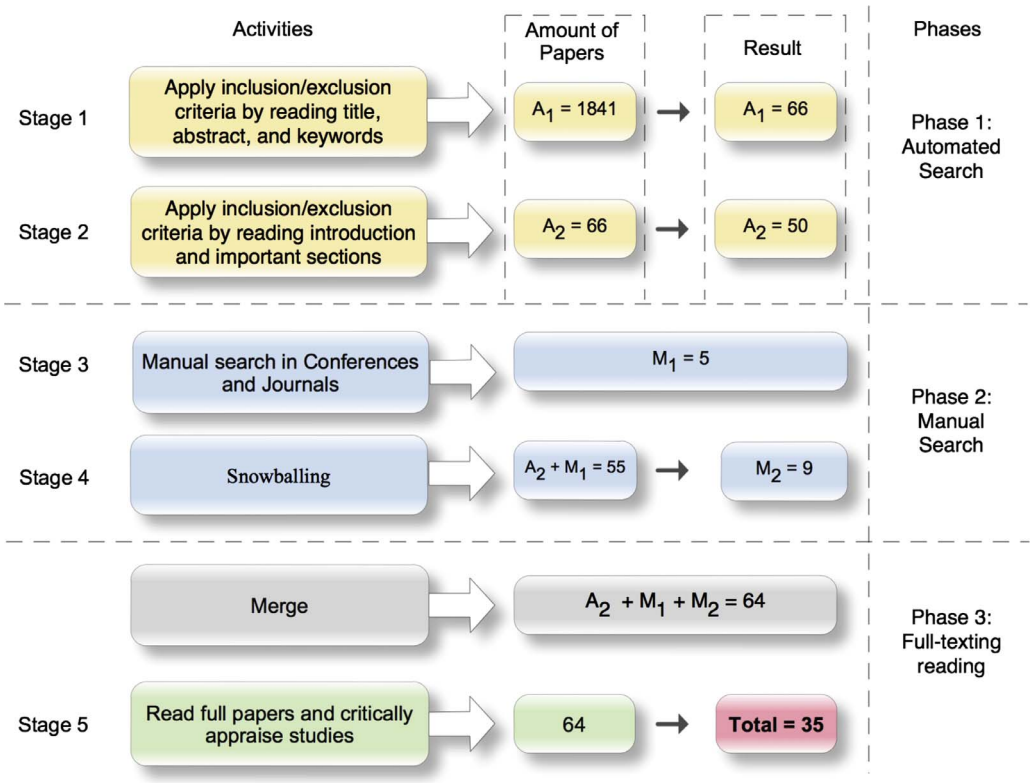

Fig. 3. Search and selection process.

Table 1
Digital libraries.

ACM digital library	http://dl.acm.org/
IEEE Xplore	http://ieeexplore.ieee.org/
Scopus	http://www.scopus.com/
Engineering village	http://www.engineeringvillage.com/
Science direct	http://www.sciencedirect.com/
Springer	http://www.springer.com/

Table 2
Search string.

("feature interaction" **OR** "feature-interaction")
AND
("SPLE" **OR** "SPL" **OR** "product line" **OR** "product family" **OR** "domain engineering" **OR** "application engineering" **OR** "variant-rich" **OR** "variability" **OR** "feature-oriented" **OR** "feature modeling" **OR** "feature analysis" **OR** "core asset")
AND
("software" **OR** "system")

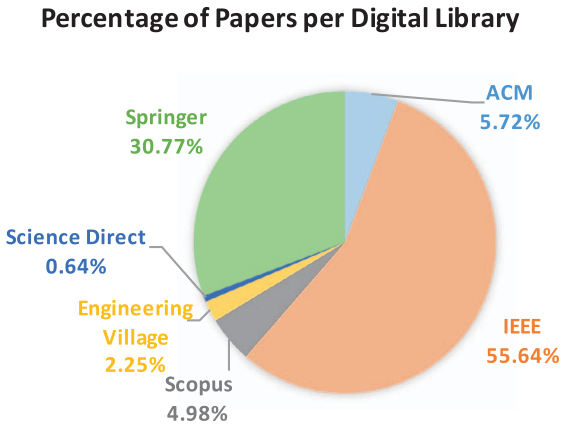

Fig. 4. Percentage of papers collected in each digital library by using our search string.

according to the method of Kitchenham et al. [38]: (i) extracting software engineering concepts and terms from the research questions; (ii) reviewing terms used in the known papers; and (iii) identifying synonyms of the key terms. Based on that, three principal keywords were chosen: *feature interaction, product line* and *software*. They served as basis for building the full string, which was created with the addition of synonyms and alternative words, and joined with *AND* and *OR* operators, as Table 2 shows.

After applying the search string in all search engines listed in Table 1, we collected a pool of 1841 studies. Fig. 4 shows the share of papers per search engine, in which the IEEE Xplore and the Springer were the ones with more papers collected. We also collected 103 duplicated studies, since more than one digital library indexed the same venues. Most of the 1841 studies were not related to the topic of interest. To better select the studies and mainly eliminate those not directly related to the mapping study's goal, we defined a set of inclusion and exclusion criteria. Then, the automatic search was performed in two stages, as Fig. 3 shows. In the first, the criteria were applied on the total of studies by reading their title, abstract and keywords. This stage excluded most of the non-related studies, resulting in a pool of 66 studies. In the second stage, we read the introduction and concluding remarks sections. From the set of 66 studies, we removed another 16 studies, based on the same set of inclusion and exclusion criteria, as follows:

- **Inclusion criteria**
  - Written in English;
  - Peer-reviewed;
  - Addressing feature interaction in SPL engineering, in which a feature interaction occurs when the behavior of one feature is influenced by the presence of another feature.
- **Exclusion criteria**

- Short papers (less than 6 pages), studies describing tutorials, workshop and poster summaries, books;
- Studies only related to single systems, i.e., when a study did not address feature interaction in SPL engineering;
- Studies addressing feature interactions in a different meaning, such as dependencies (include and exclude constraints) and interactions between features in a feature model;
- Studies that did not address any of the topics of the research questions;
- Duplicate studies. When a study has been published in more than one venue, only the most complete version of the research is considered. The remaining studies are excluded.

The manual search also consisted of two activities: manual search in conferences' proceedings and snowballing [38]. Previously, in the automated phase, we initially collected 1841 studies. Since that phase was executed in six well-recognized databases and we had already collected a significant amount of studies, the manual search was based on key conferences which commonly publish studies in the research area, which are: (i) International Workshop on Feature-Oriented Software Development (FOSD), (ii) Feature Interactions in Telecommunications and Software Systems (ICFI), (iii) International Conference on Software Engineering (ICSE), and Software Product Line Conference (SPLC). As a result, such a manual search yielded an additional set of 5 studies.

Snowballing consists of a manual search based on the reference list of relevant studies, and it is usually used to support the automated activity. This kind of reference search is also known as *backwards snowballing* [38]. For this activity, we analyzed the studies selected until this stage, i.e., 55 studies, 50 from the automated search and 5 from the manual search in conferences. The process consisted of analyzing the reference lists from those 55 studies, from which we selected 9. Thus, after merging the results from both automatic and manual search, we had a pool of 64 papers selected for the full-text reading phase.

In this last phase, we analyzed the remaining studies by carrying out a full-text reading and analysis. Then, we removed papers that: (i) presented an approach but it was not focused on feature interaction; (ii) presented a feature interaction discussion rather than a feature interaction solution; and (iii) had a more recent paper reporting the same approach. After performing a full-text reading, 35 primary studies were included in this systematic mapping study.

In order to ensure the reliability regarding the choice of the included studies, each study was evaluated by two researchers (the two first authors). Besides, they were responsible for designing the mapping protocol, searching candidate studies, reading and selecting the included studies, and also summarizing the results. The other researchers acted as reviewers. Each accepted study underwent an agreement process, and in case of uncertainty and disagreement, the authors of the candidate study were contacted to solve and give appropriate guidance.

4. Results

The mapping process consisted of three phases: planning, conducting and documenting (Fig. 1). As a result of the first and second phases, we selected 35 studies, as Table A.7 in Appendix A shows. Fig. 5 presents the distribution of the primary studies over time. The search and selection process was executed up the first quarter of 2016.

4.1. Classification scheme

In order to analyze the studies, we first extracted the title, authors, venue and publication year. Next, we attempted to extract data to answer each of the research questions guided by a classification scheme, which was designed by the first author and validated by the second. Furthermore, before proceeding with the data extraction, we performed a pilot study so that we authors could reach a mutual agreement in terms of research questions. This activity aimed at avoiding researchers misinterpreted answers.

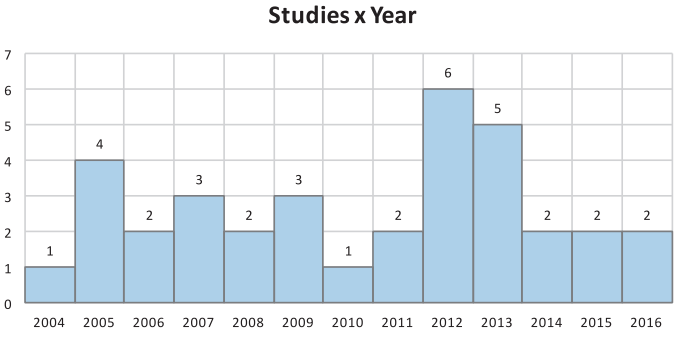

Fig. 5. Number of studies by publication year.

Table 3
Studies' venues.

Venue	#
Int. Multi-Topic Conference	[39]
Int. Conf. and Workshops on Engineering of Computer-Based Systems	[27,40]
Int. Conf. on Automated Software Engineering	[41]
Int. Conf. on Feature Interactions	[6,7,42,43]
Int. Conf. on Fundamental Approaches to Software Engineering	[44]
Int. Conf. on Generative Programming	[45]
Int. Conf. on Model-Driven Engineering and Software Development	[46]
Int. Conf. on Software Composition	[47]
Int. Conf. on Software Engineering	[29,48–50]
Int. Requirements Engineering Conference	[51]
Int. Software Product Line Conference	[52,53]
Int. Symposium on Software Reliability Engineering	[54]
Int. Symposium on the Foundations of Software Engineering	[55]
Int. Workshop on Feature-Oriented Software Development	[56,57]
Int. Workshop on Variability Modelling of Software-intensive Systems	[3,5]
Symp. on Foundations of Health Information Engineering and Sys.	[58]
Workshop on Model-Driven Engineering, Verification and Validation	[26]
FME Workshop on Formal Methods in Software Engineering	[4]
Journal of Software & Systems Modeling	[59]
Journal of Software Quality Journal	[60]
Journal of Automated Software Engineering	[61,62]
Journal of Computer Networks	[63]
Journal of IET Communications	[25]
Journal of Information and Software Technology	[28]

The scheme was based on the set of research questions designed for this study. The studies were published in journals, conferences and workshops from 2004 to 2016, and summarized according their venues in Table 3. Although we have not set a lower year-limit, the oldest paper was published in 2004. The selected studies were published in twenty-four different venues, from which two of them, namely the International Conference on Feature Interactions (ICFI) and the International Conference on Software Engineering (ICSE), stood out.

The classification scheme was developed iteratively and revised every time a new study was added. Each question classifies the study according to its own parameter, for example, RQ1 is related to solutions, RQ2 encompasses feature interactions types and RQ3 identifies the phase in the software lifecycle. Thus, as new papers were being analyzed, new categories within each question were created when needed. After reading the last selected study, the classification scheme was then established.

4.2. Feature interaction solutions

Different solutions have been proposed to handle feature interactions in different software development phases. In this study, we identified 3 main categories: detection, resolution and analysis.

Table 4
Primary studies.

Category	Primary studies
Early detection (De)	[4,25–27,39,41,42,44,46,49,54,56,58,60,61]
Source code detection (Dsc)	[3,28,29,50,53,59,63]
Early resolution (Re)	[5,40,46,54,55,60,62]
Source code resolution (Rsc)	[43,47,48,52,57,59]
Early analysis (Ae)	[6,7,51]
Source code analysis (Asc)	[45]

The objective of a *detection* approach is to use or create strategies to identify cases of feature interaction. A *resolution* approach often considers that the interactions have already been identified, and focuses on solving the interaction problem to deliver the desired products. Finally, we named *analysis* approaches the rest, i.e., any approach that aims to model, specify, manage and discuss feature interaction. These categories are not mutually exclusive, they can be combined to cover many software development lifecycle phases.

We classified the identified approaches according to the lifecycle phase when feature interaction is studied. We defined two main stages: *early phases* and *source code*. In the former, the studies deal with interactions without the need to work with source code; conversely, the latter considers studies that handle interactions using the code of the system.

Table 4 shows the amount of studies by approach, as follows: (i) **early phases detection (De)**, when it takes place before the implementation, during feature modeling and specification activities; (ii) **source code detection (Dsc)**, approaches that created/used a strategy to detect feature interactions in source code; (iii) **early resolution (Re)**, solving interactions through specifications and models before the system implementation; (iv) **source code resolution (Rsc)**, approaches that add, adapt or remove code to solve the interaction; (v) **early analysis (Ae)**, approaches that analyze feature interactions based on requirements, or other early artifacts; and finally (vi) **source code analysis (Asc)**, which analyzes approaches (including interaction prevention) based on source code. We next discuss the categories of the identified approaches.

4.2.1. Early detection (De)

Early detection approaches concern about identifying feature interactions during the first stages of the software development lifecycle. Studies in this category aim to predict feature interactions based on models and [25,27,46,49,56,58,60] and specifications. For example, Atlee et al. [4] simulated violations among features based on *bisimilarity* in transition systems, and Hu et al. [25] proposed a Semantic Web-based policy interaction detection (SPIDER) method to automatically detect interactions from ontologies and *SWRL (Semantic Web Rule Language) rules*.

Aspect-oriented feature analysis techniques are also used to specify how features relate to each other, by providing separation of concerns at feature level [39,46,60]. These techniques predict interactions based on the identification of *dependencies* in crosscutting concerns aided by divergences either between dependency diagrams and other UML models [39] supported by critical pair analysis [46], or among goals described in a Goal-Oriented Requirement Language (GRL) [42]. In addition, Mussbacher et al. [60] used GRL combined with the Use Case Maps (UCM) notation to describe scenarios and architectures. GRL and UCM together constitute the User Requirements Notation (URN), which is extended with aspect-oriented concepts (AoURN). They aimed at identify interactions based on *traceability*, from feature model to goal model.

To support the identification and analysis of potential safety-critical feature interactions, Liu et al. [27] also proposed a traceability strategy, but in this case from requirements to fault models. Similarly, Bessling and Huhn [58] presented the System-Theoretic Process Analysis (STPA) to identify unexpected feature behavior and interactions of components through the specification of safety requirements, fault models and fault injection.

Another class of studies uses *model checking* to detect interactions [41,44,49,61]. The main idea is to generate appropriate interfaces on each feature to preserve its properties, and check for interactions by combining interface information. Usually, these approaches develop proof-of-concept prototypes based on propositional calculation [41], event calculus, SAT solver [44], and SAT-based symbolic model checking algorithms (IMC and IC3) [49], as a way to find violating products. The analysis of SPL requirements models allows early detection and correction of errors in features specification [49].

De approaches also exploit the *Alloy modeling language* to support the feature-oriented design. On the one hand, features are defined as Alloy modules and their correctness properties are specified in the post-conditions of the features actions, to be translated as Alloy assertions for consistency checking [54]. On the other hand, Dietrich et al. [26] first specify interactions with the feature-oriented requirements modeling language (FORML), then translating the FORML model into an Alloy model. In both approaches, the Alloy Analyzer is the responsible for the automatic feature interaction detection.

4.2.2. Source code detection (Dsc)

This category is related to approaches that detect feature interactions during or after the implementation phase. One strategy is to first annotate the source code with features specifications and then to use a *model checker* that automatically detect conflicts based on those specifications [50,53,63]. To accomplish the feature specification task, the following techniques have been used: design by contract [53,63], aspectJ and FeatureAlloy [63], and assertions [50].

For example, Scholz et al. [53] used design by contract based on Java Modeling Language (JML) to formally specify the behavior of methods and classes, and a model checker to identify conflicts. Furthermore, Apel et al. [50] developed the SPLVerifier, a model-checking tool for C-based and Java-based SPL.

Other studies [3,28,29,59] concern about the *variability aspects* of an SPL and how it can influence the feature interaction analysis. A common technique is the use of *ASTs (Abstract Syntax Trees)* to parse the source code to support the identification of feature interactions. Different parsers have been used, for example, TypeChef (C language) [28], Java Compiler Tree API [59], and Fuji tool (feature-oriented compiler for Java) [3].

Rodrigues et al. [28] analyzed the source code with TypeChef and counted, for example, the occurrence of *ifdef* statements and functions, as a way to compute dependencies among features. Linsbauer et al. [59] presented a variability extraction approach, based on ASTs, to identify traces and dependencies from features and feature interactions to their implementation artifacts. ASTs also supported the identification of feature-oriented design patterns, which enforce feature interactions by class refinements and caller–callee relations [3].

Lastly, features' combination may lead to surprising *non-functional properties* results, i.e., when a particular combination of features has an unexpected influence on a non-functional property. Siegmund et al. [29] carried out a performance prediction that determines the influence of an interaction on performance based on a small set of measurements on code.

4.2.3. Early resolution (Re)

The Early resolution category consists in solving a problem caused by an interaction between two or more features through adaptations in models. Three studies that performed early detection (De) also presented mechanisms to resolve interactions [46,54,60]. Apel et al. [54] allow the developer to resolve interactions using Alloy derivatives, which *disable properties* of one of the involved features. In addition to disable properties, another strategy is to *change the order* in which the selected features are added in the product [46,60].

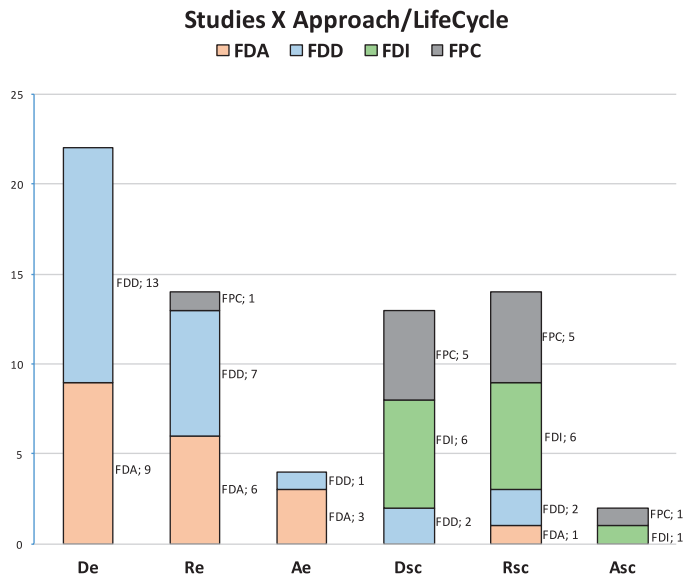

Fig. 6. Lifecyle phases per approach. DA: domain analysis; DD: domain design and specification; DI: domain implementation; and PC: product configuration and generation.

Another set of *Re* approaches considers that feature interactions have been previously detected and then presents solutions to resolve interactions using modeling strategies [5,40,55,62]. *Feature exclusion, mutual exclusion, dependencies, precedences and priorities* are examples of techniques to solve unintended interactions at specification time [55,62]. Besides that, *ignoring interaction and feature adaptation* are also other examples [5]. For instance, Padmanabhan and Lutz [62] developed the DECIMAL tool and a decision model to investigate whether those techniques could be represented as constraints in DECIMAL. Moreover, Mosser et al. [5] considered the way an interaction is resolved as a variation point in the configuration process.

Feature model-based approaches [40] also represent a way to resolve interactions. For example, either a feature can be integrated to others or a new feature can be added in the feature model to resolve an interaction. In addition, Sochos et al. [40] proposed a new feature model relation, *interacts relation*. Unlike the previous ones, Bocovich and Atlee [55] presented a *fine grained method* to resolve interactions at requirements stage. For each feature, variables, actions and outputs are specified to respond system inputs and environmental conditions. Resolution strategies include features priority and to assign an output variable to either the average, minimum, maximum or the sum of the values described by the features' actions.

4.2.4. Source code resolution (Rsc)

Providing implementation alternatives is the main characteristic of source code-based feature interaction resolution approaches. Typically, those approaches explore solutions to the *optional feature problem* in the SPL development [47,48,52,59], i.e., when optional features are apparently independent at their specifications, but are not in their implementation, indeed.

Hence, different resolution strategies have been proposed to deal with feature interactions, such as: *refactoring derivatives* [47,48,52,59], *preprocessors* or similar tools for *conditional compilation* [43,52,57], *changing feature behavior, moving code* from one feature to another, and *multiple implementations* per features. Kästner et al. [52] discussed all the aforementioned strategies to eliminate implementation dependencies and solve interactions related to the optional feature problem. However, those strategies may impair code quality, require additional effort, and decrease performance.

In this way, Takeyama and Chiba [47] proposed a design principle to reduce the effort in developing derivatives. They used a feature oriented programming language, the FeatureGluonJ (based on JAVA), to implement derivatives in a reusable manner for every combination of sub-features. Liu et al. [48] and Linsbauer et al. [59] also discussed derivatives as the main strategy to resolve interactions. Basically, the code that causes the interaction between features is removed from them and used to create a new module, separately.

Another common strategy is to use preprocessors by either annotating or coloring the features' source code. For example, to support the specification of features through algebra concepts, Batory et al. [57] painted each fragment of code in a program by at least one color. Silva et al. [43] presented a different approach, which consisted in annotating the source code with the Java annotations API, based on dependency models in both design and implementation stages.

4.2.5. Early analysis (Ae) and source code analysis (Asc)

Finally, the two last categories are related to approaches that do not discuss detections or resolutions of feature interactions, but rather propose ways to analyze them through either models and specifications based on early-phases artifacts (Ae) or source code (Asc).

Three studies were identified as early analysis (Ae) approaches [6,7,51]. Bredereke [6] and Gibson et al. [7] analyzed interactions based on the *requirements specification*. The former extended the formalism Z to specify a family of requirements into requirements modules, and used a type checker tool to point out contradictions in the family. The latter discussed possible interactions at the requirements stage during the analysis of an e-voting SPL system. Furthermore, a *modeling approach* was proposed by Shaker et al. [51] to explicitly model intended interactions among state-machine of features.

Unlike Ae approaches, Kim et al. [45] presented a source code-based analysis. This approach models interactions as a derivative tree and extended the CIDE tool, an Eclipse plugin for *coloring the source code*, to map different nestings of colors to that tree.

4.3. Software lifecycle

Fig. 6 shows how the primary studies are spread over the FOSD phases: domain analysis (FDA), domain design & specification (FDD), domain implementation (FDI), and product configuration & generation (FPC). There are two main groups, early-stage approaches (De, Re and Ae) and source code approaches (Dsc, Rsc, Asc). The pattern presented in Fig. 6 shows that FDA and FDD are more common in the former group, while FDI and FPC in the latter, as we discuss next.

Approaches focused on managing feature interactions at early stages, i.e., *early detection (De), early resolution (Re)* and *early analysis (Ae)*, are mostly concentrated in working at FDA and FDD, as Fig. 6

shows. Usually, FDD and FDA phases of early-stage approaches deal with: (i) (automatic) detection based on requirements modeling, features and software artifacts [4,26,39,40,46,55,56,62]; (ii) automatic reasoning [25,46,54]; (iii) formal specification based on model checking [41,54,61], SAT solvers [44,49]; and also (iv) software safety analysis [27] with hazard definitions and failure modeling [58].

Meanwhile, as expected, only source code approaches (De, Re and Ae) involved domain implementation (FDI) and product configuration & generation (FPC) phases. In general, they usually performed feature implementation and configuration activities through: (i) feature modules, derivatives, feature selection, and composition [47,48,52,53,57,59,63]; (ii) aspect specification [47,50,52,63]; (iii) refactorings [3,48]; (iii) source code annotations [43,45,53,57]; (iv) preprocessor directives [28]; and (vi) the influence of performance on a product configuration [29].

Although FDD and especially FDA are not the focus of source code approaches, two Dsc [53,63] and two Rsc [43,57] approaches presented activities in those phases, which consisted of strategies for either model checking approaches [53,63], feature composition [57], or dependency models analysis in both design and implementation [43]. Furthermore, a group of those studies carried out partial automatic source code detection and resolution. To accomplish this task, they used different tools, such as SPL Conqueror [29], SPL Verifier [63], TypeChef [28] and Fuji tool [3].

4.3.1. Artifacts

Early and source code approaches adopted many software artifacts to support their activities. For example, source code approaches used classes, methods, fields, statements and variation points [3,28,57,59]. Additionally, Table 5 shows various modeling techniques used to support feature interaction approaches.

Regarding the variability model, most approaches used the feature model as main model [3,29,39,40,47,51,56,57,59,60], but often with slight differences and similar names, such as feature diagram [5,44,58], product model [58], feature machine [55], AoURN FM [60], feature structure tree [53], generic feature model [47] and variability model [42].

Other models are very specific to their approach, as Table 5 shows, such as: (i) Goal Oriented Requirement Language (GRL) model [42]; (ii) aspect feature model and the point-cut feature model [39]; error model and fault injection model [58]; ontologies [25,51]; dependency models [43,45,46,59] and graphs [28,29,45,50,59,60].

4.4. Feature interactions types

Unintended feature interactions are usually considered as either external or internal to the system [1]. Externally-visible interactions include functional and non-functional behaviors, and internal interactions involve structural and operational interactions. Fig. 7 shows the amount of papers in each category.

Table 5
Software artifacts.

Category	Artifacts
UML diagrams	class, sequence, collaboration diagrams, state machine, use cases, activity models, business process
Goals	goal model, GRL model, environment model
Dependency	dependency model, dependency graph
Feature and product	feature model, feature diagram, product model, feature machine, AoURN FM, feature structure tree, generic feature model, variability model
Aspects	aspect feature model, point-cut feature model
Error/Fault	error model, fault injection model
Ontology	ontologies, world model
Graph	implication graph, call graph, AoURNs concern interaction graph, GRL graph, dependency graph

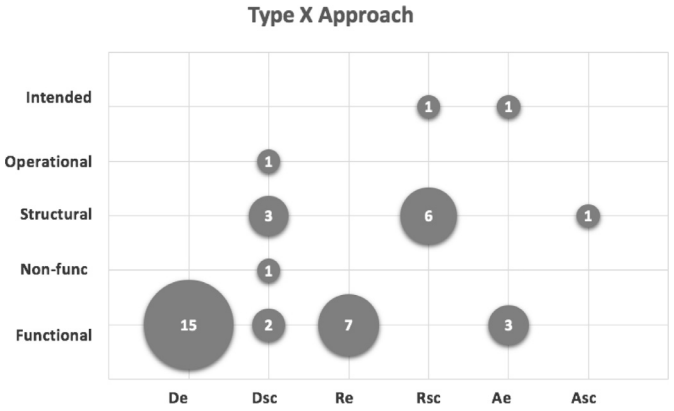

Fig. 7. Number of approaches per feature interaction type. De: early detection; Dsc: source code detection; Re: early resolution; Rsc: source code resolution; Ae: early analysis: Asc: source code analysis.

Functional feature interactions are related to interactions that violate the functional specification of a configurable system. Once most approaches presented feature interactions solutions based on models and specifications, such as approaches not based on source code, they deal with interactions at the functional level, as Fig. 7 shows.

However, functional interactions are also present in *Dsc* approaches [50,63]. Apel et al. [63] analyzed whether feature-based specifications can be used to detect feature interactions in combination with formal specifications and verification techniques. Moreover, Apel et al. [50] provided a model-checking tool to check whether a feature composition satisfies the specifications of the involved features. Conversely, non-functional feature interactions are less common than functional ones. For example, Siegmund et al. [29] proposed to predict system performance based on selected features. They aimed at detecting the system performance by analyzing its influence on the involved features.

Although some source code approaches presented external interactions, most *Dsc, Rsc* and *Asc* approaches came up with solutions more related to internal feature interactions, i.e., structural and operational interactions. Structural interactions were more frequent than operational ones, since the approaches usually work directly in the code through modules, directives and assertions.

Besides unintended interactions (functional, non-functional, structural and operational), Fig. 7 shows two approaches that work strictly with intended interactions in SPL engineering [47,51]. Takeyama and Chiba [47] discussed the difficulty to maintain a large number of derivatives, and Shaker et al. [51] proposed to model intended interactions as state-machine of features.

4.5. Domains

The 35 primary studies concern many application domains, such as smart homes, automotive, e-mail systems, database management systems, medical devices, phone systems, network systems and antivirus. Each primary study considers a different number of domains, from one up to forty program families with different purposes and sizes.

Fig. 8 gives an overview of the distribution of the studies based on their application domain and feature interaction approach. The Figure only shows the domains that appeared more than once. For example, six different studies worked with *database systems* and four approached *automotive systems*.

Regarding the feature interaction approaches, we observed that some domains were more common in some approaches than in others. For instance, *Phone and telecommunication systems* were concentrated in *early approaches*, while *database systems, file compressor* and *games* were the opposite, they appeared in *source code approaches*, as Fig. 8 shows. The predominance of De and Dsc in Fig. 8, reflects the behavior of the set of primary studies, which has mostly detection approaches.

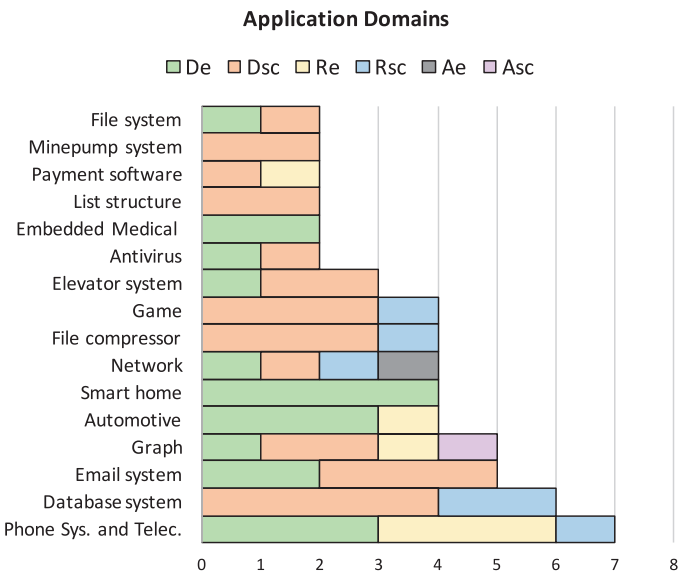

Fig. 8. Number of studies by application domain. File system: [54,63]; Minepump: [50,63]; Payment: [60,63]; List structure: [53,63]; Embedded medical device: [27,58]; Antivirus: [39,59]; Elevator system: [49,50,63]; Games: [3,4,59]; File compressor: [4,50,59]; Network: [6,54,59,63]; Smart home: [25,42,44,56]; Automotive: [49,51,53,55]; Graph: [3,45,50,54,63]; Email: [41,50,59,61,63]; Database: [3,29,48,52,59,63]; Phone system and telecommunications: [40,47,49,51,54,57,62].

In addition to the domains showed in Fig. 8, a large number of systems were also found. Among the types of systems that did not appear repeatedly in the set of primary studies are: text editor, web server, web browser, operating system, e-voting, bank system, microwave oven product line, graphical model editor, family of adaptation protocols, product-line that produce tools to manipulate Jak files, an event service SPL, and others.

4.6. Empirical assessment methods

From the set of included primary studies, 22 out of 35 employed *case study* as their empirical assessment method. However, only 4 presented a study detailed enough for an empirical research method, that is, employed a well-structured case study with basic information, such as hypothesis, research questions, methodology, results, discussion and threats to validity.

In the remainder subset, the central focus of their "case study" was to demonstrate the proposal in practice. The main used terms were: "explore the potential benefits and drawbacks of our approach", "demonstrate the utility", "demonstrate the approach", "illustrate our approach", "a demonstration of practicality and generality of our approach", and "case study as proof of concept". Other empirical assessment methods were also considered in the set of included primary studies, as Fig. 9 shows.

Among the 35 studies, 6 presented an evaluation method with many details about the evaluation practices and process [3,28,29,50,59,63], as shown in Fig. 9. For example, Rodrigues et al. [28] presented an *empirical study* to assess to what extent feature dependencies exist in actual software families following a well-designed process with goals, questions, metrics, subject selection, instrumentation, operation, results, discussion and threats to validity. They evaluated the source code of 40 software families from different domains.

In addition, Apel et al. [63] presented an *exploratory study* with 10 feature-oriented systems, and the *case studies* in [3,29,50,59] evaluated their approaches with 6, 7, 6, 3 different systems, respectively. Apel et al. [50] evaluated its approach with 3 systems, but they implemented 2 versions (in C and Java) for each system. In addition, real-world case studies were performed in [3,29,59].

5. Discussion

This section discusses the key findings of our study, as well as outlines directions for future research. In this systematic mapping study, we identified three main categories of approaches that handle detection, resolution and analysis of feature interaction. Each one was further classified according to the moment an interaction was managed, i.e., early phases studies or source code studies. Appendix A shows a summary of the 35 primary studies. Along this section, we discuss about the different feature interaction solutions, tools, domains and directions for further research.

5.1. Feature interaction solutions

When analyzing the selected studies, we could observe most of them consider either predicting interactions based on models or specifications, or solving and detecting interactions based on how the program code was implemented. Although some source code-based approaches also presented feature specifications [57,59,63] and modeling

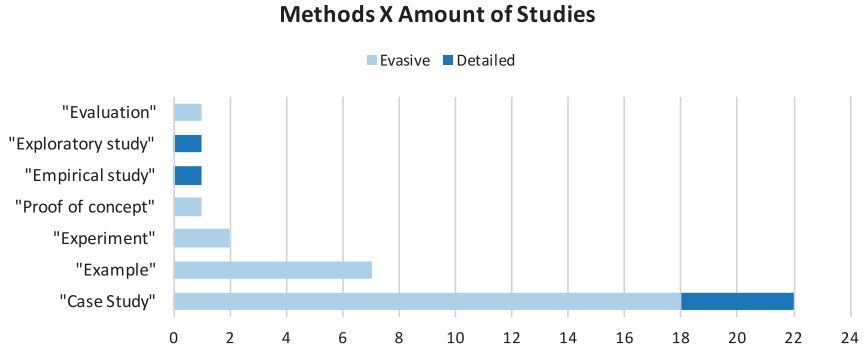

Fig. 9. Empirical methods as named by the studies.

[5,43,53], they were focused on discussing strategies [53,63] to identify interactions after implementing the software. For these approaches, source code represents the main software artifact. We next present the main concerns about detections, resolutions and analysis of feature interactions.

Detection approaches. Early detection (De) is the most represented category with 15 papers (Table 4). They predict feature interactions in SPL based on feature specification and modeling. A prior interaction detection may be used to support developers when implementing the features.

In this way, *De* approaches focused on identifying whether predefined constraints still remain after feature combinations. We identified seven different early detection strategies: (i) traceability from specifications to goal and fault models [27,60]; (ii) dependencies in aspects cross-cutting concerns [39,46]; (iii) dependencies in UML and goal models [42,56]; (iv) verification of Alloy assertions [26,54]; (v) model checking or propositional calculation of feature properties and models [41,44,49,58,61]; (vi) verification of SWRL inferences [64]; and (vii) bisimilarity in transition systems [4].

Conversely, source code approaches (Dsc) do not have as main focus to define models and specifications of features and software artifacts. Although they also aim to identify whether feature properties still remain after feature combinations, those properties commonly arise from source code analysis. The *Dsc* approaches used three strategies to detected interactions: (i) model checker [50,53,63]; (ii) AST-based parser [3,28,59]; and (iii) non-functional properties measurements [29].

Both, *De* and *Dsc*, have verification techniques based on model checking, the difference between them is that in the latter feature-based specifications are defined using both models and source code analysis. Typically, model checking strategies are combined with modeling (e.g.,design by contract) and implementation methods (e.g., code annotations, aspects). Another source code detection strategy is to use AST (Abstract Syntax Tree) to parse the code and support the identification of dependencies and design patterns.

De approaches usually do not depend on software programming languages and were mainly used to estimate feature interaction costs in software systems. Otherwise, *Dsc* approaches were concentrated on indicating either the absence or presence of feature interactions by means of specific languages: C, C++, and Java.

Resolution approaches. Early resolutions (Re) usually deal with model adaptations, especially regarding the arrangement of features. Five strategies were identified in SPL approaches: (i) disabling properties inside a feature [54]; (ii) change feature order [46,60]; (iii) feature exclusion, precedence and adaptation [5,62]; (iv) addition of features and relations, feature composition [40]; and (v) feature prioritization [5,55]. These strategies are usually supported with alloy specifications, goal, aspects, decision, UML, and dependency models.

Source code resolutions (Rsc), as the name says, propose fixing interactions problems on code. We also identified five strategies, as follows: (i) feature behavior changes [52]; (ii) moving code between features [52]; (iii) multiple feature implementations [52]; (iv) conditional compilation [52], coloring code [57], annotations [43]; and (v) refactoring derivatives [47,48,52,59].

In addition, four out of thirteen resolution approaches also presented how to detect interactions. The remainder assumed that the interactions had already been identified and did not provide details of the identification process. Regarding the programming language, similarly to *Dsc*, the *Rsc* approaches worked with C, C++, and Java. The latter was the most common programming language used in both *Dsc* and *Rsc* approaches.

Analysis approaches. Finally, early analysis (Ae) and source code analysis (Asc) represented the category of studies that neither detected nor resolved interactions, but presented ways to improve modeling [45,51], requirements specification [6,7] and source code derivatives [45]. The latter discussed derivatives with AHEAD and AST.

Table 6
Papers and tools. C: category; A: assessment method.

Paper	Tool	C	A
[46]	MATA tool	De/Re	
[54]	FeatureAlloy	De/Re	
[60]	jUCMNav	De/Re	
[27]	DECIMAL, PLFaultCAT	De	
[44]	FIFramework	De	
[39]	Aspect miner tool	De	
[26]	FORML2Alloy	De	
[58]	SCADE, VIATRA	De	
[49]	ABC toolset	De	
[40]	DOME tool, IBMs Rat. Req. Pro, RPM Package Manager	Re	
[62]	DECIMAL tool	Re	
[6]	type checker tool, CADiZ	Ae	
[53]	SpeK, FeatureHouse, ESC/Java2 and Simplify	Dsc	
[29]	SPL Conqueror	Dsc	✓
[63]	FeatureHouse	Dsc	✓
[50]	SPLVerifier, FeatureHouse, CPAChecker, JAVA PathFinder	Dsc	✓
[28]	TypeChef	Dsc	✓
[48]	AHEAD Tool Suite	Rsc	
[45]	modified CIDE, FMCA	Asc	

5.2. Tools and validation

Most approaches predict interactions based on the behavior of features and products expressed in a set of states, transitions [4,26,27,46,49,58] and model checking [41,44,54,61]. Many of them are supported by tools that treat features as formal expressions, such as model checker-based tools [26,54] and SAT solvers [44,49]; critical pair analysis tool [46], or even specific requirements design, analysis tools [27,58] and aspect-based tools [39,60].

In addition, *Dsc* and *Rsc* approaches are mainly dependent on implementation alternatives to find solutions for feature interaction problems, such as design patterns [3], feature modules and derivatives [47,48,52,57,63], and code annotations [50,57,59]. Five out of seven *Dsc* approaches and one *Rsc* were assisted by tools to support the implementation process, as for example: FeatureHouse,[1] AHEAD Tool Suite,[2] CIDE[3] and SPL Conqueror.[4]

Table 6 shows the primary studies, corresponding tools, and feature interaction solution category. In general, 60% of early-stage approaches (*De, Re* and *Ae*) were supported by tools [6,26,27,39,40,44,46,49,54,58,60,62] and 50% of source code papers (*Dsc, Rsc* and *Asc*) presented tools to assist their approach [28,29,45,48,50,53,63].

Although some approaches were supported by tools, only a few of them presented detailed empirical assessment methods [28,29,50,63], as Table 6 shows. Actually, detailed methods were only presented in source code approaches, usually through a mix of real and toy SPL. For example, Apel et al. [63] conducted an exploratory study on the basis of ten small JAVA systems with existing specifications. Conversely, Rodrigues et al. [28] analyzed 40 industrial families written in C.

Early-stage approaches (De, Re, and Ae), even those ones supported by tools, presented evasive experimental results. Most of them aimed to exemplify the approach or present a proof of concept. In general, only 17% of the total studies conducted a proper and detailed validation. It is noticed that performing empirical studies is still challenging in the software systems field.

[1] http://www.infosun.fim.uni-passau.de/spl/apel/fh/.
[2] https://www.cs.utexas.edu/users/schwartz/ATS.html.
[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/.
[4] http://www.infosun.fim.uni-passau.de/se/projects/splconqueror/.

Fig. 10. Domains over time.

5.3. Domains

The feature interaction problem has been discussed in the telecommunications domain for years. During the 80's, the concerns of the telecommunications community revolved around the future. The community was looking for a fast introduction of new features in the systems as well as a smooth separation of applications from the system. This is currently referred to as modularization, components development, or even feature-oriented development.

Fig. 10 shows the four most common domains regarding the paper's publication year. The study of feature interaction in *phone systems and telecommunications* is still present in recent years, as well as other domains, such as databases, email, and graph systems. Furthermore, Fig. 11 presents the most common domains addressed by research, from three perspectives: (i) approaches, domains, and amount of studies; (ii) approaches, FOSD phases, and amount of studies; and (iii) approaches, domains, and FOSD phases. Domains spread over both early and source code approaches have broader coverage in relation to software lifecycle phases, compared to approaches concentrated on only either early or source code stage.

On the one hand, telecommunication and phone [40,47,49,51,54,57,62], email [41,50,59,61,63], graph [3,45,50,54,63], and network [6,54,59,63] systems involved activities in all areas of the lifecycle, such as: (i) FDA: requirements analysis, feature modeling, automatic reasoning; (ii) FDD: formal specification, model checking, Alloy; (iii) FDI: implementation techniques, modules, aspects, design patterns; (iv) FPC: composition, feature selection. On the other hand, automotive systems did not involve FDI activities, as well as smart homes, which did not consider neither FDI nor FPC activities. Therefore, detection and resolution analysis based on implementation and product configuration are still missing for these two domains.

Despite early detection (De) has over twice as many studies compared to source code detection (Dsc), both categories are the ones with the most variety of domains. Although *Dsc* has fewer papers, they deal with a greater number of different domains in their approaches. While 12 out of 15 *De* papers discussed their approaches in 1 domain, almost all *Dsc* papers (except 1) worked with at least 6 domains. For example, Rodrigues et al. [28] performed an empirical study covering 37 domains with different industrial software families.

Conversely, *De* papers are usually tied to their own field of study. For example, Hu et al. [25] proposed an ontology-based approach that is specific to the smart home domain. A discussion on whether it is applicable to a different domain is missing. The same occurs for [7,27,42,56,58,60]. Accordingly, the modeling and specification approaches of the *De* category are usually more specific and may be less generalizable than the *Dsc* approaches.

Even when the authors claimed their approaches could be generalized to other areas, the studies do not discuss how relevant the results would be for other contexts; there is no discussion about the implication of the results achieved for other application domains, nor any discussion about possible and potential limitations of using the proposed approaches, which limits any inference in this sense.

5.4. Directions for further research

- Most detection approaches may not scale for large-sized, as an expressive number of specifications, variant combinations and dependencies must be defined manually. They commonly need a costly formal specification of either features, products or dependencies. An interesting strategy could be dynamically analyze systems to collect specifications and interactions at control and data flows. Tools could be used to assist the developer when implementing features by informing at runtime which (i) types of interactions they have; (ii) the specific conditions they are triggered, such as specific inputs and variable ranges; and (iii) ways the developer could use to solve that interaction.
- As a way to cover different aspects of an SPL, some strategies could be used together or even applied in different domains to maximize the use of the same approach before starting to define a new one. We found few studies that deal with effectiveness or suitability of combining strategies. For instance, Liu et al. [27] proposed a tool-based approach to support safe evolution of SPL requirements using a model-based approach. However, some discussions are still

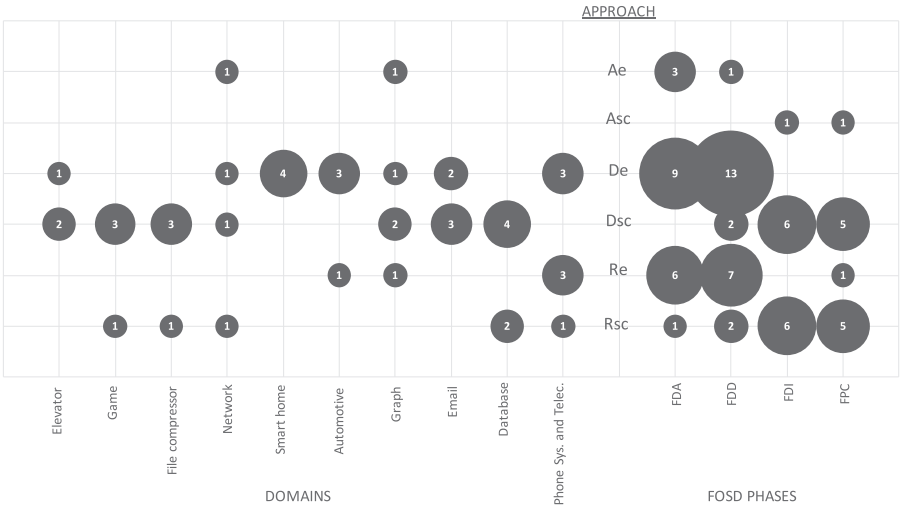

Fig. 11. Approaches, domains, and FOSD phases.

missing. For example, early detections could benefit from structural (source code) and operational (data and control flows) analysis to evaluate the efficiency of previously detected interactions. The opposite could also be interesting, traceability from source code to models.
- With the arrival of smartphones and applications, many other interactions problems may have emerged and could be further investigated, such as: (i) interactions among apps from the same supplier but different systems families; (ii) interactions among apps from a different supplier; (iii) interactions between an app and the mobile operating system; and also (iv) internal interactions to a single app, which is the most common interaction in the development of systems in general, and has been the focus of the research community that investigates feature interaction issues.

6. Threats to validity

There are some threats to the validity of our study. They are discussed next:

- Vocabulary: The literature search was in part guided by keywords. This is not ideal for our topic, since the words used for describing it are used in other fields with a very different meaning. This is reflected by the large number of irrelevant responses to our queries, that has been carefully sorted manually. Conversely, it is possible that some studies of a related topic in a different field use a different vocabulary, and would thus have been missed. We used snowballing to recover some references that could have been missed by the keyword approach.
- Research questions: Our research questions are relatively general, since the goal of this study is to give an overview of the field. There may be a set of more specialized research questions to explore in further research.
- Publication bias: Our study is limited to approaches published in the scientific literature. There may be another set of approaches, as important as those analyzed in this investigation, which has not been published as regular research papers, in particular when developed inside companies that cannot be published for strategic reasons. Another bias is the restriction to publications in English. Since the field uses mainly English, the introduced bias is minimal.
- Subjectivity bias: The papers have been classified by the first author mainly, with the other authors verifying the adequacy. This procedure is a compromise in terms of effort and objectivity.

7. Concluding remarks

This article presented a systematic mapping study to investigate the state-of-the-art of feature interactions in SPL engineering. The 35 included primary studies were classified according to their proposed solutions (RQ1), feature interactions types (RQ2), software lifecycle (RQ3), software domains (RQ4) and empirical assessment methods (RQ5).

More than 43% of the studies discussed how to identify interactions at early phases of the SPL development, mainly based on traceability, dependencies, verification of Alloy assertions, feature exclusion, precedence, and adaptation. Another 40% comprised approaches focused on source code to detect and resolve interactions. For example, they were based on: model checkers, non-functional properties measurements, AST-based parser multiple implementations, conditional compilation, and derivatives. The remaining studies provided an initial discussion about feature interaction management, such as models, specification and ways to prevent interactions.

Often, strategies are partial and only address specific points of a feature interaction investigation. On the one hand, when interactions are detected, it is not possible to identify the trace of violations caused by them. On the other hand, if they are resolved with a specialized module or implementation changes, the previous step on how they came up with those interactions is not explained. In addition, the approaches are generally academic and provide limited experimental results, rarely perform effective and detailed case studies.

As future work, we intend to combine the evidence identified in this mapping study with evidence from controlled experiments to identify hypotheses and theories that may be used to design and combine methods and tools for feature interactions in highly configurable systems. Moreover, we plan to collect the available systems and to categorize their discovered feature interactions to a Web repository. It may serve as a testbed for new approaches or even evolutions and adaptations of already developed approaches.

Appendix A. Summary of studies

Table A1
Primary studies.

#	Study	Category	Main strategy	Lifecycle	FI type	Tool
1	Atlee et al. [4]	De	bisimilarity	DD	functional	
2	Dietrich et al. [26]	De	Alloy assertions	DD	functional	✓
3	Liu et al. [27]	De	traceability	DA/DD	functional	✓
4	Razzaq et al. [39]	De	dependencies	DA/DD	functional	✓
5	Blundell et al. [41]	De	model checker	DD	functional	
6	Metzger et al. [42]	De	dependencies	DA	functional	
7	Classen et al. [44]	De	model checker	DA/DD	functional	✓
8	Ben-David et al. [49]	De	model checker	DA/DD	functional	✓
9	Alférez et al. [56]	De	dependencies	DA/DD	functional	
10	Li et al. [61]	De	model checker	DD	functional	
11	Bessling et al. [58]	De	model checker	DA/DD	functional	✓
12	Hu et al. [64]	De	SWRL inferences	DA/DD	functional	
13	Jayaraman et al. [46]	De/Re	dependencies/order	DD	functional	✓
14	Apel et al. [54]	De/Re	Alloy/disabling	DA/DD	functional	✓
15	Mussbacher et al. [60]	De/Re	traceability/order	DA/DD	functional	✓
16	Padmanabhan et al. [62]	Re	precedence	DA/DD	functional	✓
17	Sochos et al. [40]	Re	add features & relations	DA/DD	functional	✓
18	Bocovich et al. [55]	Re	priority	DA/DD	functional	
19	Mosser et al. [5]	Re	priority & adaptation	DA/DD/PC	functional	
20	Scholz et al. [53]	Dsc	model checker	DD/DI/PC	structural	✓
21	Siegmund et al. [29]	Dsc	non-func. measurement	PC	non-func	✓

(continued on next page)

Table A1 (*continued*)

#	Study	Category	Main strategy	Lifecycle	FI type	Tool
22	Apel et al. [63]	Dsc	model checker	DD/DI/PC	functional	✓
23	Apel et al. [50]	Dsc	model checker	DI/PC	functional	✓
24	Rodrigues et al. [28]	Dsc	AST-based parser	DI	operational	✓
25	Schuster et al. [3]	Dsc	AST-based parser	DI	structural	
26	Linsbauer et al. [59]	Dsc/Rsc	AST-based parser/derivatives	DI/PC	structural	
27	Batory et al. [57]	Rsc	coloring code	DD/DI/PC	structural	
28	Takeyama et al. [47]	Rsc	derivatives	DI/PC	structural/ intended	
29	Liu et al. [48]	Rsc	derivatives	DI/PC	structural	✓
30	Silva Filho et al. [43]	Rsc	annotations	DA/DD/DI	structural	
31	Kästner et al. [52]	Rsc	derivatives	DI/PC	structural	
32	Bredereke [6]	Ae	requirements specification	DA	functional	✓
33	Gibson et al. [7]	Ae	requirements specification	DD	functional	
34	Shaker et al. [51]	Ae	improve modeling	DA/DD	functional/ intended	
35	Kim et al. [45]	Asc	derivatives	DI/PC	structural	✓

References

[1] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, B. Garvin, Exploring feature interactions in the wild: the new feature-interaction challenge, Proceedings of the 5th International Workshop on Feature-Oriented Software Development, FOSD '13, ACM, New York, NY, USA, 2013, pp. 1–8.

[2] T.F. Bowen, F.S. Dworack, C.H. Chow, N. Griffeth, G.E. Herman, Y.J. Lin, The feature interaction problem in telecommunications systems, Software Engineering for Telecommunication Switching Systems, 1989. SETSS 89., Seventh International Conference on, (1989), pp. 59–62.

[3] S. Schuster, S. Schulze, I. Schaefer, Structural feature interaction patterns: case studies and guidelines, Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '14, ACM, New York, NY, USA, 2013, pp. 14:1–14:8.

[4] J.M. Atlee, U. Fahrenberg, A. Legay, Measuring behaviour interactions between product-line features, Proceedings of the Third FME Workshop on Formal Methods in Software Engineering, Formalise '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 20–25.

[5] S. Mosser, C. Parra, L. Duchien, M. Blay-Fornarino, Using domain features to handle feature interactions, Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '12, ACM, New York, NY, USA, 2012, pp. 101–110.

[6] J. Bredereke, Configuring members of a family of requirements using features, Feature Interactions in Telecommunications and Software Systems VIII, ICFI'05, 28–30 June 2005, Leicester, UK, IOS Press, 2005, pp. 96–113.

[7] J.P. Gibson, E. Lallet, J.-L. Raffy, Feature interactions in a software product line for e-voting. in: M. Nakamura, S. Reiff-Marganiec (Eds.), ICFI, IOS Press, 2009, pp. 91–106.

[8] P.A. da Mota Silveira Neto, I. do Carmo Machado, J.D. McGregor, E.S. de Almeida, S.R. de Lemos Meira, A systematic mapping study of software product lines testing, Inf. Softw. Technol. 53 (5) (2011) 407–423.

[9] J.F. Bastos, P.A. da Mota Silveira Neto, E.S. de Almeida, S.R. de Lemos Meira, Adopting software product lines: a systematic mapping study, Evaluation Assessment in Software Engineering (EASE 2011), 15th Annual Conference on, (2011), pp. 11–20.

[10] I.F. da Silva, P.A. da Mota Silveira Neto, P. O'Leary, E.S. de Almeida, S.R. de Lemos Meira, Agile software product lines: a systematic mapping study, Softw. Pract. Exp. 41 (8) (2011) 899–920.

[11] L.R. Soares, P. Potena, I. do C. Machado, I. Crnkovic, E.S. de Almeida, Analysis of non-functional properties in software product lines: a systematic review, 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, (2014), pp. 328–335.

[12] T. Vale, E.S. de Almeida, V. Alves, U. Kulesza, N. Niu, R. de Lima, Software product lines traceability: a systematic mapping study, Inf. Softw. Technol. 84 (2017) 1–18.

[13] S. Apel, C. Kästner, An overview of feature-oriented software development, J. Object Technol. 8 (5) (2009) 49–84.

[14] L.R. Soares, I. do Carmo Machado, E.S. de Almeida, Non-functional properties in software product lines: a reuse approach, Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '15, ACM, New York, NY, USA, 2015, pp. 67:67–67:74.

[15] K. Czarnecki, U.W. Eisenecker, Generative Programming: Methods, Tools, and Applications, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[16] C.W. Krueger, The biglever software gears unified software product line engineering framework, 2008 12th International Software Product Line Conference, (2008). 353–353.

[17] D. Beuche, Modeling and building software product lines with pure:: variants, Proceedings of the 16th International Software Product Line Conference-Volume 2, ACM, 2012. 255–255.

[18] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, T. Leich, FeatureIDE: an extensible framework for feature-oriented software development, Sci. Comput. Program. 79 (2014) 70–85.

[19] D. Batory, P. Höfner, J. Kim, Feature interactions, products, and composition, Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering, GPCE '11, ACM, New York, NY, USA, 2011, pp. 13–22.

[20] G. Bruns, Foundations for features, Feature Interactions in Telecommunications and Software Systems VIII, ICFI'05, 28–30 June 2005, Leicester, UK, (2005), pp. 3–11.

[21] FeatureOpt: taming and optimizing feature interaction in software-intensive automotive systems, (https://iktderzukunft.at/en/projects/feature-opt.php#contactAddress). Accessed: 2017-07-20.

[22] A.L.J. Dominguez, Feature interaction detection in the automotive domain, 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, (2008), pp. 521–524.

[23] P. Zave, E. Cheung, S. Yarosh, Toward user-centric feature composition for the internet of things, arXiv:1510.06714 (2015).

[24] M. Calder, M. Kolberg, E.H. Magill, S. Reiff-Marganiec, Feature interaction: a critical review and considered forecast, Comput. Netw. 41 (1) (2003) 115–141.

[25] H. Hu, D. Yang, L. Fu, H. Xiang, C. Fu, J. Sang, C. Ye, R. Li, Semantic web-based policy interaction detection method with rules in smart home for detecting interactions among user policies, IET Commun. 5 (17) (2011) 2451–2460.

[26] D. Dietrich, P. Shaker, J.M. Atlee, D. Rayside, J. Gorzny, Feature interaction analysis of the feature-oriented requirements-modelling language using Alloy, Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa '12, ACM, New York, NY, USA, 2012, pp. 17–22.

[27] J. Liu, J. Dehlinger, H. Sun, R. Lutz, State-based modeling to support the evolution and maintenance of safety-critical software product lines, 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), (2007), pp. 596–608.

[28] I. Rodrigues, M. Ribeiro, F. Medeiros, P. Borba, B. Fonseca, R. Gheyi, Assessing fine-grained feature dependencies, Inf. Softw. Technol. 78 (C) (2016) 27–52.

[29] N. Siegmund, S.S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, G. Saake, Predicting performance via automated feature-interaction detection, Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 167–177.

[30] I. Abal, C. Brabrand, A. Wasowski, 42 variability bugs in the Linux kernel: aqualitative analysis, Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, ACM, New York, NY, USA, 2014, pp. 421–432.

[31] R.J. Hall, Feature combination and interaction detection via foreground/background models, Comput. Netw. 32 (4) (2000) 449–469.

[32] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Y. Bontemps, Generic semantics of feature diagrams, Comput. Netw. 51 (2) (2007) 456–479.

[33] T. Ohta, Y. Harada, Classification, detection and resolution of service interactions in telecommunication services, Feature Interact. Telecommun. Syst. (1994) 60.

[34] M. Plath, M. Ryan, Feature integration using a feature construct, Sci. Comput. Program. 41 (1) (2001) 53–84.

[35] C. Prehofer, Feature-oriented programming: a fresh look at objects, Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), 1241 Springer, 1997, pp. 419–443.

[36] J. Liu, Feature interactions and software derivatives. J. Object Technol. 4 (3) (2004) 13–19.

[37] K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson, Systematic mapping studies in software engineering, Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, EASE'08, British Computer Society, Swinton, UK, 2008, pp. 68–77.

[38] B.A. Kitchenham, D. Budgen, P. Brereton (Eds.), Evidence-Based Software Engineering and Systematic Reviews, Chapman and Hall/CRC, Boca Raton, FL, USA, 2015.

[39] A. Razzaq, R. Abbasi, Automated separation of crosscutting concerns: earlier automated identification and modularization of cross-cutting features at analysis

phase, Multitopic Conference (INMIC), 2012 15th International, IEEE, 2012, pp. 471–478.
[40] P. Sochos, M. Riebisch, I. Philippow, The feature-architecture mapping (farm) method for feature-oriented development of software product lines, 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems - ECBS, IEEE, 2006, p. 9.
[41] C. Blundell, K. Fisler, S. Krishnamurthi, P.V. Hentenrvck, Parameterized interfaces for open system verification of product lines, Proceedings. 19th International Conference on Automated Software Engineering, 2004. (2004), pp. 258–267.
[42] A. Metzger, S. Bühne, K. Lauenroth, K. Pohl, Considering feature interactions in product lines: towards the automatic derivation of dependencies between product variants. FIW, (2005), pp. 198–216.
[43] R.S. Silva Filho, D.F. Redmiles, Managing feature interaction by documenting and enforcing dependencies in software product lines, Feature Interactions in Software and Communication Systems IX, 33 (2008).
[44] A. Classen, P. Heymans, P.-Y. Schobbens, What's in a feature: a requirements engineering perspective, International Conference on Fundamental Approaches to Software Engineering, Springer, 2008, pp. 16–30.
[45] C.H.P. Kim, C. Kästner, D. Batory, On the modularity of feature interactions, Proceedings of the 7th international conference on Generative programming and component engineering, ACM, 2008, pp. 23–34.
[46] P. Jayaraman, J. Whittle, A.M. Elkhodary, H. Gomaa, Model composition in product lines and feature interaction detection using critical pair analysis, International Conference on Model Driven Engineering Languages and Systems, Springer, 2007, pp. 151–165.
[47] F. Takeyama, S. Chiba, Implementing feature interactions with generic feature modules, International Conference on Software Composition, Springer, 2013, pp. 81–96.
[48] J. Liu, D. Batory, C. Lengauer, Feature oriented refactoring of legacy applications, Proceedings of the 28th international conference on Software engineering, ACM, 2006, pp. 112–121.
[49] S. Ben-David, B. Sterin, J.M. Atlee, S. Beidu, Symbolic model checking of product-line requirements using sat-based methods, Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, 1 IEEE, 2015, pp. 189–199.
[50] S. Apel, A.V. Rhein, P. Wendler, A. Größlinger, D. Beyer, Strategies for product-line verification: case studies and experiments, Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013, pp. 482–491.
[51] P. Shaker, J.M. Atlee, S. Wang, A feature-oriented requirements modelling language, Requirements Engineering Conference (RE), 2012 20th IEEE International, IEEE, 2012, pp. 151–160.
[52] C. Kästner, S. Apel, M. Rosenmüller, D. Batory, G. Saake, et al., On the impact of the optional feature problem: analysis and case studies, Proceedings of the 13th International Software Product Line Conference, Carnegie Mellon University, 2009, pp. 181–190.
[53] W. Scholz, T. Thüm, S. Apel, C. Lengauer, Automatic detection of feature interactions using the java modeling language: an experience report, Proceedings of the 15th International Software Product Line Conference, Volume 2, ACM, 2011, p. 7.
[54] S. Apel, W. Scholz, C. Lengauer, C. Kästner, Detecting dependences and interactions in feature-oriented design, Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on, IEEE, 2010, pp. 161–170.
[55] C. Bocovich, J.M. Atlee, Variable-specific resolutions for feature interactions, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 553–563.
[56] M. Alférez, A. Moreira, U. Kulesza, J. Araújo, R. Mateus, V. Amaral, Detecting feature interactions in SPL requirements analysis models, Proceedings of the First International Workshop on Feature-Oriented Software Development, ACM, 2009, pp. 117–123.
[57] D. Batory, P. Höfner, B. Möller, A. Zelend, Features, modularity, and variation points, Proceedings of the 5th International Workshop on Feature-Oriented Software Development, ACM, 2013, pp. 9–16.
[58] S. Bessling, M. Huhn, Towards formal safety analysis in feature-oriented product line development, International Symposium on Foundations of Health Informatics Engineering and Systems, Springer, 2013, pp. 217–235.
[59] L. Linsbauer, R.E. Lopez-Herrejon, A. Egyed, Variability extraction and modeling for product variants, Softw. Syst. Model. (2016) 1–21.
[60] G. Mussbacher, J. Araújo, A. Moreira, D. Amyot, AoURN-based modeling and analysis of software product lines, Softw. Qual. J. 20 (3–4) (2012) 645–687.
[61] H.C. Li, S. Krishnamurthi, K. Fisler, Modular verification of open features using three-valued model checking, Autom. Softw. Eng. 12 (3) (2005) 349–382.
[62] P. Padmanabhan, R.R. Lutz, Tool-supported verification of product line requirements, Autom. Softw. Eng. 12 (4) (2005) 447–465.
[63] S. Apel, A. Von Rhein, T. Thüm, C. Kästner, Feature-interaction detection based on feature-based specifications, Comput. Netw. 57 (12) (2013) 2399–2409.
[64] H. Hu, D. Yang, L. Fu, H. Xiang, C. Fu, J. Sang, C. Ye, R. Li, Semantic web-based policy interaction detection method with rules in smart home for detecting interactions among user policies, IET Commun. 5 (17) (2011) 2451–2460.