What's in a *Feature*: A Requirements Engineering Perspective

Andreas Classen*, Patrick Heymans, and Pierre-Yves Schobbens

PReCISE Research Centre, Faculty of Computer Science, University of Namur
5000 Namur, Belgium
{acs,phe,pys}@info.fundp.ac.be

Abstract. The notion of feature is heavily used in Software Engineering, especially for software product lines. However, this notion appears to be confusing, mixing various aspects of problem and solution. In this paper, we attempt to clarify the notion of *feature* in the light of Zave and Jackson's framework for Requirements Engineering. By redefining a problem-level feature as a set of related requirements, specifications and domain assumptions—the three types of statements central to Zave and Jackson's framework—we also revisit the notion of feature interaction. This clarification work opens new perspectives on formal description and verification of software product lines. An important benefit of the approach is to enable an early identification of feature interactions taking place in the systems' environment, a notoriously challenging problem. The approach is illustrated through a proof-of-concept prototype tool and applied to a Smart Home example.

1 Introduction

Software product lines engineering (SPLE) is an emergent software engineering paradigm institutionalising reuse throughout the software lifecycle. Pohl *et al.* [1] define a software product line (SPL) as *"a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"*. In SPLE, features appear to be first class abstractions that shape the reasoning of the engineers and other stakeholders [2,1]. This shows up, for instance, in feature modelling languages [3,4,5], which are popular notations used for representing and managing the variability between the members of a product line in terms of features (see Fig. 1 and Section 2.2).

In their seminal paper, Kang *et al.* introduce FODA (Feature-oriented domain analysis), a SPL approach based on feature diagrams [3]. In this context, they define a feature as *"a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems"*. We term this definition *problem-oriented* as it considers features as being an expression of the user's requirements. Eisenecker and Czarnecki, on the other hand, define a feature in

* FNRS Research Fellow. Work originated in a research stay at the Open University.

Table 1. Definitions for the term "feature" found in the literature and their overlaps with the descriptions of the Zave and Jackson framework (excerpt of [14])

Reference	**Definition**	**R**	**W**	**S**	**D**	**other**
Kang *et al.* [3]	*"a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems"*	✓		✓		
Kang *et al.* [8]	*"distinctively identifiable functional abstractions that must be implemented, tested, delivered, and maintained"*	✓		✓	✓	
Eisenecker and Czarnecki [6]	*"anything users or client programs might want to control about a concept"*	✓	✓	✓	✓	✓
Bosch *et al.* [9]	*"A logical unit of behaviour specified by a set of functional and non-functional requirements."*	✓		✓	✓	
Chen *et al.* [10]	*"a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements"*	✓				
Batory [11]	*"an elaboration or augmentation of an entity(s) that introduces a new service, capability or relationship"*			✓	✓	✓
Batory *et al.* [12]	*"an increment in product functionality"*	✓		✓		
Apel *et al.* [13]	*"a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option."*		✓	✓	✓	

a more general way as *"anything users or client programs might want to control about a concept"* [6]. This broader definition also subsumes elements of the solution space such as communication protocols, for instance.

As shown in the first two columns of Table 1, many other definitions exist that mix to a varying degree elements of solution and problem. This leads to confusion as to what a feature generally represents, and hence to a need for clarification. Most definitions, however, make sense in their respective context. Judging them by comparing them to one another irrespective of this context would not be very sensible. In this paper, we limit our scope to requirements engineering, and complement our previous work on disambiguating feature models. In [4,5], we devised a generic formal semantics for feature diagrams. There, we clarified and compared constructs used to combine features, but did not question the notion of feature itself. In [7], we further disambiguated feature models by distinguishing features and variability that represent product management decisions from those that denote extension capabilities of the SPL's reusable software assets.

In this paper, we propose a complementary perspective that looks at features as expressions of problems to be solved by the products of the SPL. We rely on the extensive work carried out by Jackson, Zave and others during the past decade in a similar attempt to clarify the notion of requirement [15,16]. Their work has resulted in more precise definitions of the terms "requirement", "specification" and "domain assumption". Their clarifications have allowed to improve the methodological guidance given to requirements engineers in eliciting, documenting, validating and verifying software related needs. One of the most notable outcomes of their work is the identification of a fundamental *requirements concern*, i.e. a logical entailment that must hold between the various components of a requirements document. Roughly, this concern can be stated as: given a set of assumptions W on the application domain, and given a system that behaves according to its specification S, we should be able to guarantee that the requirements R are met. More formally: $S, W \vdash R$.

In their ICSE'07 roadmap paper, Cheng and Atlee [17] acknowledge that *"reasoning about requirements involves reasoning about the combined behaviour of the proposed system and assumptions made about the environment. Taking into consideration environmental conditions significantly increases the complexity of the problem at hand"*. In this paper, we propose concepts, methods and tools to address the combined complexity of highly environment-dependent systems and systems that have to be developed in multiple (possibly many) exemplars. In such systems, a particularly difficult problem is to detect feature interactions that involve the environment [18].

In the present paper, building on Jackson *et al.*'s clarification work, we redefine the notion of feature as a subset of correlated elements from W, S and R. Doing so, we ambition to demonstrate that:

(i) a clearer definition of the notion of feature is useful to guide modellers in their abstraction process;
(ii) the relationships between feature models and other kinds of descriptions needed during SPL requirements engineering become clearer;
(iii) the requirements concern can be redefined in the context of SPLE, giving the engineers a clear target in terms of a "proof obligation";
(iv) the notion of "feature interaction" can be revisited in the light of this adapted proof obligation, further clarifying concepts and guidelines for SPL engineers.

The remainder of this paper is structured as follows. In Section 2 we recall Feature Diagrams as well as Zave and Jackson's framework for requirements engineering. We illustrate them on a Smart Home System example. The redefinition of "feature" follows in Section 3, accompanied by a discussion of its implications on the concept of "feature interaction" and a proposal for a general approach to feature interaction detection in SPLs. As a proof of concept for the definitions and the general approach, we present a particular verification approach and prototype tool in Section 4. This is followed in Section 5 by a discussion on possible improvements and future work. Related works are described in Section 6 before Section 7 concludes the paper.

2 Background

2.1 Zave and Jackson's Reference Model

The starting point of Zave, Jackson *et al.*'s work is the observation that specifying indicative and optative properties of the environment is as important as specifying the system's functionality [15,16,19].

They identify three types of descriptions: *Requirements* R describe what the purpose of the system is. They are optative descriptions, i.e. they express how the world should be once the system is deployed. For an alarm system, this could be: 'Notify the police of the presence of burglars'. *Domain assumptions* W describe the behaviour of the environment in which the system will be deployed. They are indicative, i.e. they indicate facts, as for instance 'Burglars cause movement when they break in' and 'There is a movement sensor monitoring the house, connected to the system'. The *specification* then describes how the system has to behave in order to bring about the changes in the environment as described in R, assuming the environment to be as described in W. In our example, this would be 'Alert the police if the sensor captures movement'. The central element of the reference model is the following relationship between these three descriptions:

$$S, W \vdash R, \tag{1}$$

i.e. the machine satisfies the requirements if S combined with W *entails* R. For the remainder of the paper, we will use the symbol $\mid\!\sim$ to make explicit the fact that the entailment relationship is not necessarily monotonic [20].

2.2 Modelling Variability

To reason on the variability of a SPL, Feature Diagrams (FDs) are perhaps the most popular family of notations. An example (using the FODA [3] syntax) is shown in Fig. 1 and further discussed in Section 2.3. Generally, a FD is a tree or a directed acyclic graph that serves as a compact representation of all valid combinations of features (depicted as boxes). A formal semantics of FDs can be devised by considering non-leaf boxes as Boolean formulae over their descendants [21,4]. For example, Fig. 1 uses three *or*-decompositions (depicted as dark 'pie slices'), but other operators (*and*, *xor*, cardinalities...) as well as cross-cutting constraints (*excludes*, *requires*) are also commonly used.

In our previous work [4,5], we devised a generic formalisation of the syntax and semantics of FDs, on top of which most popular FD dialects were (re)defined and compared. In this paper, we reuse this formalisation and its associated tool. More specifically, we will make use of the tool's product derivation capabilities, that is, its ability to generate valid feature combinations.

2.3 Illustrative Example

As an illustration, let us consider the case of a Smart Home System (SHS) [22]. Such a system has a high degree of integration with its environment as it mainly observes and controls properties of a house that are part of the physical world. In order to be able to build a SHS right, it is important to understand how these properties influence each other [22].

A SHS is also a typical product line [1] since the company developing and selling such systems has to be able to adapt them to each customer's home. In addition, such companies need to be able to leave configuration choices to the customer, such as the features to be included, and specificities of theses features, which allows to address different market segments. A house located in a very warm region, for instance, will probably not need a heating system. A house under surveillance by an external security company will need an adapted security system as part of the smart home package, and so on.

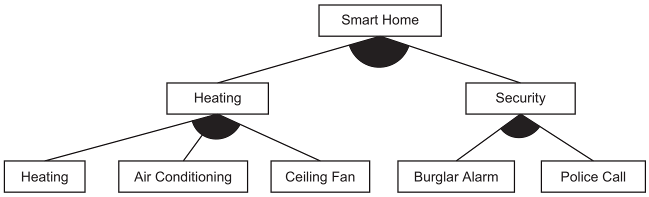

Fig. 1. Feature diagram for the SHS

The simplified system we use as an illustration consists of two features. Each feature has several sub-features as depicted on the FD in Fig. 1. All features are decomposed with an *or*-operator. This means that a product might only consist of the security feature, for instance. A short description of the two main features is given below.

Heating. Its objective is to assure that the temperature of the room lies within a range of user-defined values. In order to achieve this, the system can use the heating, the air conditioning and a ceiling fan.

Security. The security feature has to make sure that the police and the neighbourhood are alerted of any burglars and intruders. The system is equipped with a movement sensor, a telephone connection to alert the police, and an alarm to alert the neighbours.

A problem diagram [19] for a particular product consisting of the ceiling fan and the police call features is depicted in Fig. 2. It shows two requirements (dashed ovals) on the right-hand side, one for each feature, and the SHS on the left-hand side (rectangle with two stripes). In between there are real world domains. When a domain can directly control or observe properties of another domain, those domains are said to share phenomena. This is depicted by a line between the domain boxes. Dashed lines denote the phenomena in terms of which the requirements are defined. The arrow heads point to the domains of which the phenomena are constrained by the requirement. The problem diagram basically shows the problem's topology, it is schematic and is to be completed by precise descriptions of R, S and W.

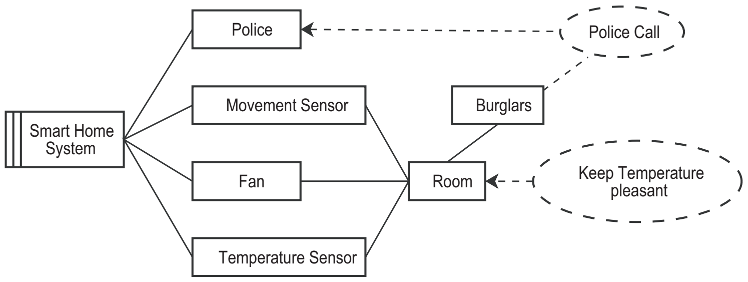

Fig. 2. Composite problem diagram for the simplified SHS

As shown in Section 2.1 and in Fig. 2, the description of the security feature indeed covers all of the above constituents. We also showed that, intuitively, its $S, W \mid\!\sim R$ relation holds. Yet, it is still easy to imagine that other features can differ, overlap or conflict with this feature in terms of their S, W and R constituents. Conflicts are typically called feature interactions. As we will see in the next section, a new and precise definition of "feature interaction" will follow from our redefinition of "feature".

3 Towards a General Definition of "Feature" in Requirements Engineering

Table 1 shows how, in the context of requirements engineering, current definitions of "feature" mix or ignore the various elements (R, W, S) and sometime also subsume design (D) or other concerns. The irregular distribution of checkmarks, although subjective and debatable, still suggests that there is room for clarification.

3.1 "Feature" Revisited

Based on the previous observations, and following to some extent an idea suggested by Chen *et al.* [10], we redefine a feature as a set of related requirements, domain properties and specifications:

Definition 1 (Feature). *A feature is a triplet, $f = (R, W, S)$, where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification.*

By adopting this definition, we emphasise three essential constituents of features. We can then relate them using the central proof obligation of the reference model, which acts as a consistency criterion for a feature. A feature $f_1 = (R_1, W_1, S_1)$ is said to be consistent if we have $S_1, W_1 \mid\!\sim R_1$.

Just as in the Zave-Jackson framework, we are not prescriptive about the formalisms to use for the descriptions of S_1, W_1 and R_1. Hence, they can be chosen freely. This also means that the form this proof should take is not fixed either. In this, we follow Hall *et al.* in [23]: if judged sufficiently dependable, the proof can be provided by any mean, including non-exhaustive tests, or even arguments such as *"Dijkstra programmed it"*, as long as the choice is justified with respect to the level of confidence expected by the developers. Being general, the definition thus allows all kinds of formal or informal proofs. If we want to be able to automate the approach, however, we have to restrict ourselves to automatable modes of proof. This path will be further explored in Section 4, where we present a prototype tool that automates this proof.

Checking each feature on its own, however, is not sufficient. A SPL can contain hundreds of features, the combination of which define the products. Most of the complexity of variability management resides in the interoperation of features: some can depend on each other, and some, conversely, can disrupt each other. This means that a system cannot be proven correct by proving each feature separately. We now address this issue.

3.2 "Feature Interaction" Revisited

Feature interactions have long been a research topic in the telecommunications domain. The particularity of an interaction is that it is a property that two or more features only exhibit when put together, and not when run individually [24].

Based on the previous definition of feature, and based on the Zave-Jackson framework, we propose the following definition.

Definition 2 (Feature interaction). *Given a set of features $p = f_1..f_n$, expressed as R_i, W_i, S_i for $i = 1..n$ and $n \geq 2$, features $f_1..f_n$ are said to interact if*

(i) they satisfy their individual requirements in isolation,
(ii) they do not satisfy the conjunction of these requirements when put together,
(iii) and removing any feature from p results in a set of features that do not interact.

i.e. if:

$$
\begin{aligned}
& \forall f_i \in p \ . \ S_i, W_i \mid\!\sim R_i \\
\wedge\ & \bigwedge_{i=1}^{n} S_i, \bigwedge_{i=1}^{n} W_i \not\mid\!\sim \bigwedge_{i=1}^{n} R_i \\
\wedge\ & \forall f_k \in p \ . \bigwedge_{i \in \{1..k-1,k+1..n\}} S_i, \bigwedge_{i \in \{1..k-1,k+1..n\}} W_i \mid\!\sim \bigwedge_{i \in \{1..k-1,k+1..n\}} R_i
\end{aligned}
$$

A feature interaction in a system $s = \{f_1..f_q\}$ is then any set $p \subseteq s$ such that its features interact.

Points (i) and (ii) of Definition 2 express the fact that an interaction only occurs when features are put together. The objective of point (iii) is to make sure that a feature interaction is always minimal, i.e. all features that are part of an interaction have to be present for the interaction to occur. If a feature could be taken out of a set of interacting features without affecting the interaction, it would not be part of the interaction anyway. Ignoring (iii) would also mean that for each interaction between i features ($i < |allfeatures|$), a new interaction of $i+1$ features could be found simply by adding any of the remaining features.

3.3 Two Kinds of Interactions

Since a feature can add elements on both sides of the $\mid\!\sim$ relation, non-satisfaction of the proof obligation is not necessarily a monotonic relation. This is, however, assumed in the previous paragraph, i.e. we assumed that:

$$\bigwedge_{i=1}^{k} S_i, \bigwedge_{i=1}^{k} W_i \not\mid\!\sim \bigwedge_{i=1}^{k} R_i \;\wedge\; S_{k+1}, W_{k+1} \mid\!\sim R_{k+1} \quad \Rightarrow \quad \bigwedge_{i=1}^{k+1} S_i, \bigwedge_{i=1}^{k+1} W_i \not\mid\!\sim \bigwedge_{i=1}^{k+1} R_i$$

As a counterexample consider a situation in which two features need a third one to work properly: features f_1 and f_2, for instance, both do print logging, which means that both require access to some dot-matrix printer connected to the system in order to print their logs. These features interact because each works fine in isolation, but once both are put together the first one that gains access to the printer would exclude the other from doing so, thereby preventing it from printing its logs, hence violating the global requirement. If we consider now a third feature f_3 that is a wrapper for the printer API and allows simultaneous access from multiple processes, then it is easy to imagine that f_3 would prevent f_1 and f_2 from interacting. Assuming that $f_i = (S_i, W_i, R_i)$ we would have the following relations:

$$S_1, W_1 \mid\!\sim R_1 \qquad S_2, W_2 \mid\!\sim R_2 \qquad S_3, W_3 \mid\!\sim R_3$$
$$S_1, S_2, W_1, W_2 \not\mid\!\sim R_1, R_2 \qquad S_1, S_2, S_3, W_1, W_2, W_3 \mid\!\sim R_1, R_2, R_3$$

The above example shows that adding a feature to a set of interacting features could as well solve the interaction. This observation does not invalidate the preceding definition of a feature interaction. It merely points out a second type of interaction, which we do not consider. What we define as an interaction is basically the fact that *simultaneous* presence of several features causes malfunctions, and that these features *cannot be present* in the system at the same time. A second type of interaction would be just the opposite, i.e. the fact that a number of features *have to be* present in the system at the same time, because *individual* presence would lead to malfunctions. While interactions of the first type are harmful and have to be prevented, interactions of the second one are desired and have to be assured. For the FD, this generally results in adding *excludes*-constraints between features concerned by the first case and *requires*-constraints between features concerned by the second case.

3.4 Systematic Consistency Verification

Building on the preceding definitions and assuming that descriptions S, W and R are provided for each feature of a SPL, we now propose a set of consistency rules that need to be satisfied by these descriptions. We then present four algorithms as a general approach to feature interaction detection based on these consistency rules. A proof-of-concept instance of this approach is presented in Section 4.

The starting point is again the $S, W \mid\!\sim R$ relation which has to be proven correct for all products of the SPL. Unfortunately, proving this relation alone is not sufficient, as it would be trivially satisfied if we had $S, W \mid\!\sim false$. Similarly, if $R \mid\!\sim false$, or $R, W \mid\!\sim false$, the requirement would be too restrictive. As these cases need to be excluded, we have to perform several preliminary proofs. In consequence, we identify a total of six proofs:

$$\begin{gathered} S \not\mid\!\sim false \qquad W \not\mid\!\sim false \qquad R \not\mid\!\sim false \\ S, W \not\mid\!\sim false \qquad W, R \not\mid\!\sim false \\ S, W \mid\!\sim R \end{gathered} \tag{2}$$

These proofs have to be verified for each single feature, as well as for each product of the SPL. The whole process can be described by the following four algorithms.[1]

A1 *Feature consistency check:* verify the proofs (2) on all features of the SPL. This algorithm has to be run first, because we have to make sure that all features are consistent before the SPL is verified. Its algorithmic complexity is $O(n\gamma)$ where γ is the complexity of verifying one relation of the form $S_i, W_i \mid\!\sim R_i$ an n the number of features in the SPL.

A2 *Product consistency check:* verify the proofs (2) for a given product that is part of the SPL. If the complexity of verifying a relation of the form $\bigwedge_{i=1}^{n} S_i, \bigwedge_{i=1}^{n} W_i \mid\!\sim \bigwedge_{i=1}^{n} R_i$ is assumed to be $\Gamma(n)$, then this algorithm is of complexity $O(\Gamma(n))$. It is invoked by algorithm A3.

A3 *Product line consistency check:* verify the consistency of the whole SPL. Given the feature diagram d, generate all products that are part of the SPL and invoke the preceding algorithm (A2) for each one. The complexity in this case is $O(2^n + |[\![d]\!]|\Gamma(n))$.

A4 *Find interactions:* identify the actual feature interaction in the case an inconsistency has been detected by algorithm A3. This algorithm will be invoked by A3 as needed. The complexity here is $O\big(2^n\Gamma(n)\big)$.

We believe that this approach is sufficiently general to act as an umbrella for a large number of feature interaction detection techniques, depending on the form the $S, W \mid\!\sim R$ proof takes. The next section provides one proof-of-concept instance.

4 A Proof-of-Concept Instance

In order to experiment with the general approach introduced in the preceding section we created an instance of this approach by choosing an automatable formalism (viz. the Event Calculus), based on which we developed a proof-of-concept prototype that automates all composition and validation tasks.

[1] Due to space constraints, the descriptions of these algorithms as well as their complexity results are shortened and simplified. For a detailed account please refer to [14].

4.1 Prototype Tool

If all descriptions are expressed in a formalism that allows for automated reasoning, the algorithms presented in Section 3.4 can be largely automated. We chose the Event Calculus (EC) [25], because it is intuitive and well suited for "commonsense" descriptions such as those found in many domain properties. Among the available EC implementations, we chose the *discrete event calculus reasoner (Decreasoner)*, an EC implementation by Mueller [25]. Decreasoner does model-checks on a set of EC formulae by transforming them into a SAT problem to be solved by a third-party SAT solver. After running the SAT-solver, Decreasoner analyses the model it found and presents it as a narrative of time points, events and fluents.

Using the EC and the Decreasoner implementation, we developed a prototype reasoning tool, called *FIFramework*,[2] as a plugin for the Eclipse platform. The tool offers a dedicated EC editor, which simplifies the editing of formulae through syntax highlighting and code assistance and enforces feature descriptions to be conform to Definition 1. It also adds a number of new buttons to the toolbar, which allow the user to launch the verifications introduced in Section 3.4 and thus provides the capability of proving the absence of interactions as defined in Definition 2 behind a push-button interface. Each time a specific verification is launched, the tool automatically gathers all formulae needed for the particular verification. These formulae are then written to a Decreasoner compatible input file, and Decreasoner is invoked to process the file. Once this is done, the result is analysed, a report generated and the final result presented to the user.

As depicted on the workflow shown in Fig. 3, the tool builds on and implements the ideas and definitions of Section 3. The starting point is a SPL which has its variability documented in a FD. The features of this diagram are modelled using problem diagrams, and formalised using the EC. Processing the feature diagrams delivers a list of products (sets of features) [4], and the associated problem diagrams are represented by a set of EC files. Given this input, our tool automates all EC formulae composition and verification tasks. Through interactive usage, the algorithms of Section 3.4 (except algorithm A4) can then be effectively automated.

4.2 Results

An in-depth illustration of the approach as well as of FIFramework can be found in [14], where the author models and analyses a SHS product line consisting of 16 different features (actually an extended version of the example used in Section 2.3). The illustration starts with a feature analysis and a problem analysis which identifies the different domains and requirements. Based on this analysis, a product consisting of 11 features is specified formally with the EC. This results in a total of 30 EC formulae expressing domain assumptions, requirements and specifications which are then introduced into FIFramework.

[2] Available online at www.classen.be/work/mscthesis

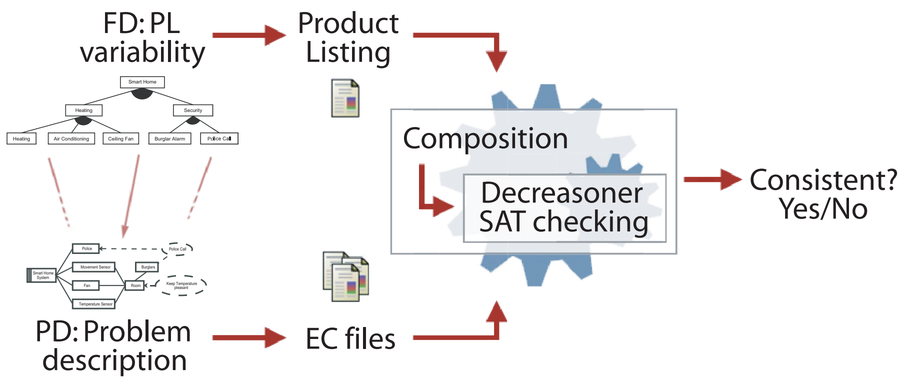

Fig. 3. FIFramework workflow

Using the feature interaction detection algorithms of Section 3.4, we were able to identify two interactions, one between the away-from-home and the energy control feature (two features not included in this paper) and one between the police call and the heating feature (those of the Section 2.3). The two features work just fine in isolation. Once deployed in the same room, however, they interact. This is due to the fact that the movement sensor of the police call feature will not only capture burglars, but also the movement of the ceiling fan, leading to false alarms.

We reiterate that our example and tool are not realistic, but only intend to demonstrate the feasability of the general approach. Before trying a real-world case study, we need to improve the scalability of our tool.

5 Discussion

During SPL requirements engineering, if one looks at variability only from the lens of FDs, one will be limited by the fuzzy notion of feature. The only formal notion of product validity that one will be able to use is that of a product (set of features) satisfying the constraints in the FD. Since features are very coarse-grained abstractions, and in the absence of more detailed verification, the safety of such approaches can seem questionable. On the contrary, further distinction and formalisation of the features' constituents (S, W and R), as proposed in this paper, allows to uncover hidden interactions while still remaining at the problem definition level. The approach relies on a stronger notion of product satisfiability relying on the satisfaction of its first proof obligation, i.e. absence of feature interactions and a guarantee of the overall requirement being met. The results of this formal analysis, however, should be in turn reflected in the FD, updating it with *excludes* and *xor* constraints so that their satisfaction implies satisfaction of the first proof obligation. Furthermore, it is conceivable to use the approach to identify unnecessary constraints in the FD that should be relaxed because they prevent useful combinations that have no harmful interaction. This bi-directional model co-evolution process, further discussed in [14], is a topic of on-going work.

As we have seen, the first instance of our general feature interaction detection framework is a running prototype that relies on the EC and its Decreasoner implementation. Although the EC and Decreasoner are powerful tools to express and reason on many "common sense" descriptions, branching time properties and behaviour are out of its scope. Furthermore, the Decreasoner implementation only uses finite-time discrete EC, putting the burden of defining valid time intervals on the analyst. We are already considering moving to temporal logic, because of the abundance of literature and powerful model checkers.

On a more conceptual note, we found in various experiments that several descriptions, mainly among the domain assumptions, are shared by all features. We thus intend to extend Definition 2 (and consequently our consistency checking procedure) so that it accounts for shared descriptions explicitly. It is, for instance, conceivable to assume that there exists a *base configuration*, i.e. some R_b, W_b and S_b that hold for all features, and to include it in each proof.

Another crucial point for the scalability of our approach is the modularity of the manipulated models. Although there exist guidelines for structuring FDs [8,26], such models can become very large, and feature decomposition criteria can be quite subjective, making it hard to navigate through models. On the other hand, over the years, the Zave-Jackson framework has evolved into the Problem Frames (PF) approach [19]. PFs facilitate the description of complex problems by decomposing them into "basic problems" that match patterns (frames) from a repertoire of recurrent simple situations. This decomposition approach provides much clearer criteria and better modularity than feature-based decomposition. However, it sometimes appears that very large problems are hard to decompose, and a prior feature decomposition allows for a high-level exploration of the problem space. Also, PFs lack a way to represent variability explicitly. The complementarity of the two approaches and the co-evolution of their respective diagrams (FDs and problem diagrams) was already investigated by the authors in [27], but still has to be linked to our feature interaction detection approach.

6 Related work

The Feature-Oriented Reuse Method (FORM) [8] has its own typology of features. It extends the basic FD notation with four classes of features organised in four respective layers: capability features, operating environment features, domain technology features and implementation technique features. This classification is similar to the S, W, R classification of the reference framework, but is less formal. In particular, no proof obligation is proposed. The main purpose is to structure the FD.

Several authors suggest other classifications for variability in general. Pohl *et al.* [1] distinguish *internal* and *external* variability. External variability is what is relevant to customers, while internal variability is technical. No formal relation between them is defined though. Similar, yet different, is the distinction of *product line variability* and *software variability* by Metzger *et al.* [7], who separated business-related variability *decisions* from platform variability *data*. However, all these distinctions consider features as black boxes. They do not distinguish or restrict the content of the features for each variability class.

Silva [28] introduces a method for detecting and solving discrepancies between different viewpoints. The suggested approach is similar to what we presented here, in that Silva also uses the same RE framework and compares different descriptions against each other. It differs from our case, in that Silva considers descriptions of the same feature, that stem from different analysts, while we consider descriptions of different features.

Laney *et al.* [29] also work with problem diagrams and EC to detect and solve run-time behavioural inconsistencies. In case of a conflict, the feature with the highest priority (expressed using composition operators) prevails. Their approach is focused on run-time resolution using composition operators whereas ours focuses on design-time resolution in the SPLE context.

We also note that there are efforts underway in the feature interaction community to detect and solve inconsistencies in SHS [22]. Although they allow to discover interactions in the physical world, their approach remains domain-specific whereas our framework is domain-independent.

7 Conclusion

In this paper, we laid down the foundations for a general approach to automated feature interaction detection supporting the early stages of software product line engineering. Central to this approach are novel definitions of two fundamental concepts: "feature" and "feature interaction". These definitions are themselves grounded in the Zave-Jackson framework for requirements engineering and allow to link it to the popular notation of feature diagrams. The most important benefit of the approach is to allow for a formal, fine-grained analysis of feature interactions, which is one of the most challenging problems in software product lines. More and more widespread, but particularly difficult to detect, are interactions that involve the environment. Our framework provides a general means to tackle them as early as possible in the development lifecycle, when the corrective actions are orders-of-magnitude cheaper than in subsequent stages.

We also reported on the instance of the general approach into a proof-of-concept prototype that uses the Event Calculus as a concrete specification language, and an off-the-shelf SAT solver. The tool could be tried out on a SHS, exemplifying our concepts and allowing to uncover non-trivial feature interactions occurring in the system's environment.

Our future work will target scalability mainly by (i) adopting temporal logic and its associated industrial-strength model-checkers, (ii) improving the modularity of the models by integrating our approach with problem frames, (iii) investigating possibilities to do compositional verification, and (iv) integrating the tool into a toolchain that we are currently developing for formal specification and analysis of software product lines. Cooperation is also underway with industry to apply our techniques to a real SHS.

Acknowledgements

This work was partially funded by the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy, by the Belgian National Bank and the FNRS.

References

1. Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
2. Batory, D.S.: Feature-oriented programming and the ahead tool suite. In: 26th International Conference on Software Engineering (ICSE 2004), Edinburgh, United Kingdom, May 23-28, 2004, pp. 702–703 (2004)
3. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (November 1990)
4. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Feature Diagrams: A Survey and A Formal Semantics. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE 2006), Minneapolis, Minnesota, USA, September 2006, pp. 139–148 (2006)
5. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. In: Computer Networks (2006). special issue on feature interactions in emerging application domains, vol. 38 (2006), (doi:10.1016/j.comnet.2006.08.008)
6. Eisenecker, U.W., Czarnecki, K.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
7. Metzger, A., Heymans, P., Pohl, K., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: Proceedings of the 15th IEEE International Requirements Engineering Conference (RE 2007), New Delhi, India, October 2007, pp. 243–253 (2007)
8. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: Form: A feature-oriented reuse method with domain-specific reference architectures. Annales of Software Engineering 5, 143–168 (1998)
9. Bosch, J.: Design and use of software architectures: adopting and evolving a product-line approach. ACM Press/Addison-Wesley, New York (2000)
10. Chen, K., Zhang, W., Zhao, H., Mei, H.: An approach to constructing feature models based on requirements clustering. In: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE 2005), pp. 31–40 (2005)
11. Batory, D.: Feature modularity for product-lines. In: OOPSLA 2006 Generative Programming and Component Engineering (GPCE) (tutorial) (October 2006)
12. Batory, D., Benavides, D., Ruiz-Cortes, A.: Automated analysis of feature models: Challenges ahead. Communications of the ACM (December 2006)
13. Apel, S., Lengauer, C., Batory, D., Möller, B.: Kästner, C.: An algebra for feature-oriented software development. Technical report, Fakultät für Informatik und Mathematik, Universität Passau (2007)

14. Classen, A.: Problem-oriented modelling and verification of software product lines. Master's thesis, Computer Science Department, University of Namur, Belgium (June 2007)
15. Zave, P., Jackson, M.A.: Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology 6(1), 1–30 (1997)
16. Gunter, C.A., Gunter, E.L., Jackson, M., Zave, P.: A reference model for requirements and specifications. IEEE Software 17(3), 37–43 (2000)
17. Cheng, B.H., Atlee, J.M.: Research directions in requirements engineering. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), May 20-26 (2007)
18. Metzger, A., Bühne, S., Lauenroth, K., Pohl, K.: Considering Feature Interactions in Product Lines: Towards the Automatic Derivation of Dependencies between Product Variants. In: Feature Interactions in Telecommunications and Software Systems VIII (ICFI 2005), June 2005, pp. 198–216. IOS Press, Leicester (2005)
19. Jackson, M.A.: Problem frames: analyzing and structuring software development problems. Addison-Wesley, Boston (2001)
20. Makinson, D.: General Patterns in Nonmonotonic Reasoning. In: Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 2, pp. 35–110. Oxford University Press, Oxford (1994)
21. Batory, D.S.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
22. Wilson, M., Kolberg, M., Magill, E.H.: Considering side effects in service interactions in home automation - an online approach. In: Proceedings of the 9th International Conference on Feature Interactions in Software and Communication Systems (ICFI 2007), Grenoble, France, September 2007, pp. 187–202 (2007)
23. Hall, J.G., Rapanotti, L., Jackson, M.: Problem frame semantics for software development. Software and System Modeling 4(2), 189–198 (2005)
24. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. Computer Networks 41(1), 115–141 (2003)
25. Mueller, E.T.: Commonsense Reasoning. Morgan Kaufmann, San Francisco (2006)
26. Lee, K., Kang, K.C., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering. In: Gacek, C. (ed.) ICSR 2002. LNCS, vol. 2319, pp. 62–77. Springer, Heidelberg (2002)
27. Classen, A., Heymans, P., Laney, R., Nuseibeh, B., Tun, T.T.: On the structure of problem variability: From feature diagrams to problem frames. In: Proceedings of the First International Workshop on Variability Modelling of Software-intensive Systems, Limerick, Ireland, LERO, January 2007, pp. 109–117 (2007)
28. Silva, A.: Requirements, domain and specifications: a viewpoint-based approach to requirements engineering. In: ICSE 2002: Proceedings of the 24th Int. Conference on Software Engineering, pp. 94–104. ACM Press, New York (2002)
29. Laney, R., Tun, T.T., Jackson, M., Nuseibeh, B.: Composing features by managing inconsistent requirements. In: Proceedings of the 9th International Conference on Feature Interactions in Software and Communication Systems (ICFI 2007), Grenoble, France, September 2007, pp. 141–156 (2007)