LOUVAIN
School of Management
RESEARCH INSTITUTE

WORKING PAPER

2017/02

Goal-oriented requirement engineering for agile software product lines: an overview

Hassan Haidar,
Manuel Kolp,
Louvain School of Management Research Institute

Yves Wautelet,
KULeuven

LOUVAIN SCHOOL OF MANAGEMENT RESEARCH INSTITUTE

Louvain School of Management Research Institute
Working Paper Series
Editor: Prof. Frank Janssen
(president-ilsm@uclouvain.be)

Goal-oriented requirement engineering for agile software product lines: an overview

Hassan Haidar, Louvain School of Management Research Institute
Manuel Kolp, Louvain School of Management Research Institute
Yves Wautelet, KULeuven

Summary

Software product lines (SPL) cover software products derived from a shared code base, ideally in a widely automated manner. Given the nature of SPL, the importance of requirements engineering (RE), a core stage in software development lifecycle, is trickier as SPLs pose more complex challenges than development of a 'single' product.

Goal models in RE represent requirements and intentions of a software system-to-be. They play an important role in the development process of SPL. Indeed, in the domain engineering phase, goal models guide the development of SPL variability by providing the rationale, while they are used for the SPL configuration in the application engineering phase. Feature Modeling can be used as a crucial technique to identify commonalities and variabilities in an SPL. However, the specific perspective shown by Feature Models (FM) is not sufficient to express all the characteristics and constraints of an SPL. Thus, using a goal-oriented approach, such as i*, to complement (and help defining) feature models would improve such models enhancing the meaning and justification of features.

Several approaches have been proposed in the literature, to generate feature models from goal models. This paper aims to choose the most useful GORE approach that help generating (mapping) feature models from goal models.

Keywords: Software product lines, Goal-oriented requirement engineering, Feature models, Agile, i*.

JEL Classification : L86

Corresponding author:
Hassan Haidar
CEMIS
Louvain School of Management Research Institute / Campus of Louvain-la-Neuve
Université catholique de Louvain (UCLouvain)
Place des Doyens 1
B-1348 Louvain-la-Neuve, BELGIUM
Email: hassan.haidar@uclouvain.be

1. Introduction

A software product line (SPL) is a family of software products derived from a shared code base, ideally in a widely automated manner [51]. Each derived product is described in terms of a valid configuration of the product line's domain model [29]. Over the recent years, software product lines have found their way into numerous industrial application domains such as; automotive, electronics, aeronautics, information and mobile systems [64].

Software product line engineering (SPLE) has become a key technology to cope with the variability of highly-configurable (software) systems, prevalent in various application domains today [18]. *Variability* in SPLE provides the required flexibility for product differentiation and diversification [21]. SPLE aims to develop a family of similar, yet well-distinguished software products based on a common core platform, where commonality and variability among the family members (product variants) are explicitly specified in terms of *features*. Each feature corresponds to (1) user-visible product characteristics in the *problem domain*, relevant for product configuration, as well as to (2) component-based artifacts for (automated) assembling of implementation variants [18]. This approach of extensive reuse and modularity of common feature artifacts among product variants facilitates a remarkable gain in efficiency compared to one-by-one variant development [25].

Feature-oriented SPL development relies, on the one hand, on the notion of features in order to identify variability and commonality between the members of an SPL. On the other hand, in literature, many approaches are presented to deal with the software product lines requirements engineering and variability management.

Dealing with features starts at the early phase of requirement elicitation. In fact, the early requirement analysis specifies stakeholder's goals through goal-oriented requirement engineering (GORE). GORE makes extensive use of goal models, i.e., models capturing user intentions for a system-to-be, and facilitates the exploration of design alternatives, described in high-level non-technical terms. Typically, the SPLE community also uses goal models in both life cycles of SPLE (Domain engineering and Application engineering). In domain engineering, they are applied for a top down development of SPLs [67] while in application engineering, they ensure the selection of features that are based on the objectives of a target application stakeholder [8].

In essence, goal models and feature models provide different variability perspectives. On the one hand, goal models represent intentional variability, which is different in the stakeholder's objectives and the way these actors may use a system-to-be to reach their targets [67]. On the other hand, feature models are commonly used to illustrate variability between various systems, which is called product line variability [46].

This paper aims to identify the most useful GORE approaches that help generating (mapping) feature models from goal models. We use the i* framework in order to realize the extended-goal models.

The paper is structured as follows: Sections 2 and 3 introduce respectively the SPL and Agile SPL foundations. Section 4 explains the domain and requirement analysis phases in which requirement engineering plays an important role. Section 5 gives core details about variability management and requirement engineering in the context of agile software product lines. In Section 6, the adopted goal-oriented approach is presented and the use of i* framework justified.

In Section 7, the mapping between extended-goal-oriented models and feature models using i* was explained. Section 8 gives a brief discussion about the paper choices. Finally, Section 9 presents a conclusion and mentions some future related works.

2. Software Product Lines

Software product lines (SPL) aim at improving software vendors to tailor software products to the requirements of individual customers [6]. In fact, SPL attempt to enhance the quality of products, decreasing the cost of development, and reducing time to market [50]. SPL is defined as "*a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way*" [25].

Software product lines engineering is prone to allow reuse in software engineering [8]. The approach provides a form of mass customization by constructing individual solutions based on a portfolio of reusable software components. An SPL is a set of software systems that share most of their *features* [78].

An essential success factor of product-line development is to set a proper focus on a particular, well-defined and well-scoped domain [6].

Definition 1: A *domain* is an area of knowledge that:
- is scoped to maximize the satisfaction of the requirements of its stakeholders,
- includes a set of concepts and terminology understood by practitioners in that area,
- includes the knowledge of how to build software systems (or parts of software systems) in that area [29].

SPLE consists of the *domain engineering* and *application engineering* life cycles [50]. Domain engineering is the process of analyzing the domain of a product line and developing reusable artifacts. Domain engineering does not result in a specific software product, but prepares artifacts to be used in multiple, if not all, products of a product line. Domain engineering targets development for reuse. In contrast, application engineering has the goal of developing a specific product for the needs of a particular customer (or other stakeholder). It corresponds to the process of single application development in traditional software engineering, but reuses artifacts from domain engineering where possible. It targets development with reuse. Application engineering is repeated for every product of the product line that is to be derived [6].

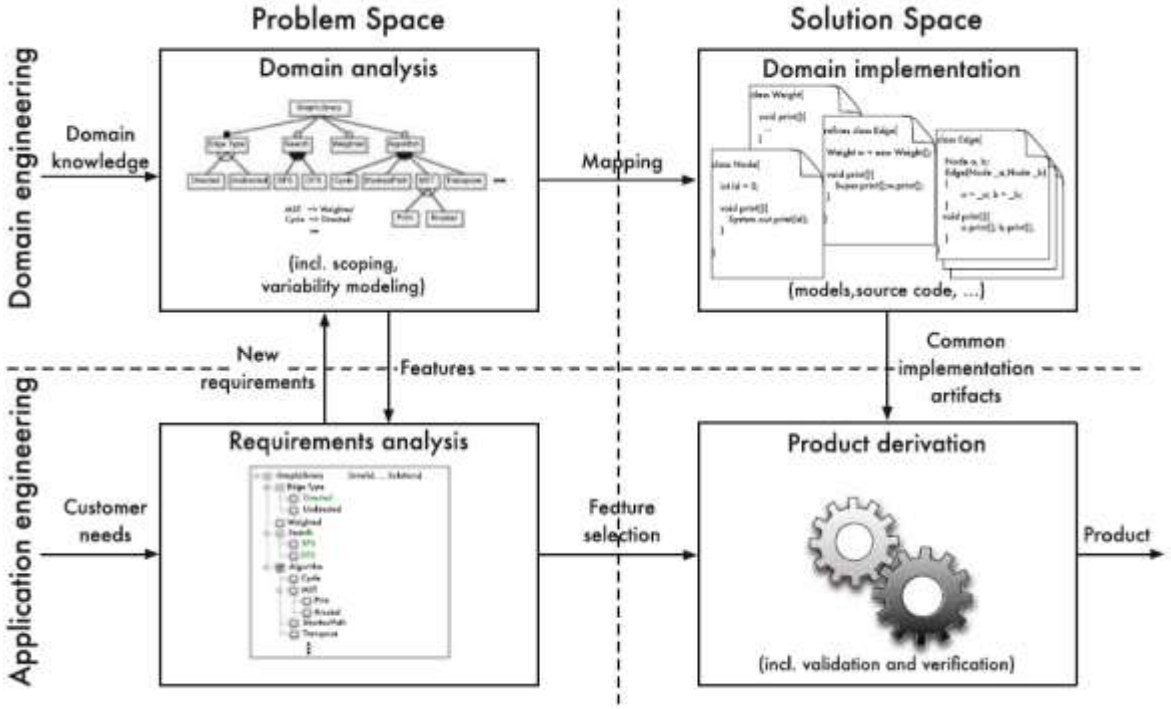

Figure 1 Overview of an engineering process for software product lines [6]

The specific characteristics of software product lines lead to a separation between domain and application engineering and between problem and solution space [6]. Figure 1 illustrates a two-dimensional structure with four clusters of tasks in product-line development and the mappings between them.

The distinction between the problem space and solution space highlights two different perspectives. The problem space *(left half of Figure 1)* takes the perspective of stakeholders and their problems, requirements, and views of the entire domain and individual products. Features are, in fact, domain abstractions that characterize the problem space. In contrast, the solution space *(right half of Figure 1)* represents the developer's and vendor's perspectives. It is characterized by the terminology of the developer, which includes names of functions, classes, and program parameters. The solution space covers the design, implementation, and validation and verification of features and their combinations in suitable ways to facilitate systematic reuse [6, 77, 78].

In addition, Figure 1 presents four clusters of tasks in SPLE process. These clusters illustrate connected stages [50, 78]:

1. *Domain analysis* is a form of requirements engineering for an entire product line. At this phase, the scope of the domain has to be determined, i.e., decide which products should be covered by the product line and, consequently, which features are relevant and should be implemented as reusable artifacts. The results of domain analysis are usually documented in a *feature model*.
2. *Requirements analysis* investigates the needs of a specific customer as part of application engineering. In the simplest case, the customer's requirements are mapped to a feature selection, based on the features identified during domain analysis. If novel

requirements are discovered, they can be fed back into the domain analysis, which may result in a modification of the feature model (and in turn the reusable domain artifacts).

3. *Domain implementation* is the process of developing reusable artifacts that correspond to the features identified in domain analysis. Depending on how variability is implemented developers may produce very different artifacts in this step, from run-time parameters and pre-processor directives to plug-ins and components, and many more.
4. *Product derivation* is the production step of application engineering, where reusable artifacts are combined according to the results of requirement analysis. Depending on the implementation approach, this process can be more or less automated, possibly, involving several development and customization tasks.

Finally, a Product-Line approach [39] has three different adoption paths. In order to start the development of an SPL, a strategy should be adopted among the following:

i. The *proactive approach* develops a product line from scratch by carefully using analysis and design methods.
ii. The *extractive approach* starts with a collection of existing products and incrementally refactors them to form a product line.
iii. The *reactive approach* begins with a small, easy to handle product line (possibly consisting only of a single product) and is extended incrementally with new features and implementation artifacts, thus extending the scope of the product line.

3. Agile Software Product Lines

Nowadays, software industry has to rapidly produce quality software and respond to changes in a flexible and quick manner. This has driven the definition of new techniques, processes, tools, and notations [60].

Generally, SPLE involves considerable upfront planning and design with heavyweight software process to achieve the organizational goals in the same way as structured (software) project management. This heavyweight process contradicts some of the goals of an SPL such as quicker time-to-market, just-in-time or pull flow [30]. On the other hand, the agile paradigm –hugely adopted nowadays in software development life-cycle – achieves organizational goals through practices, principles, and values such as focusing on people and interactions, working software, customer collaboration, responding to change, and continuous improvement [14, 40].

Software product lines (SPLs) and Agile are approaches that share similar goals in the same way that IT project management and Agile do. The main difference is the way in which these goals are met (difference in principles and practices) [60]. Unlike SPL engineering, Agile involves a lightweight process and low upfront planning and design. Nevertheless, both approaches, SPL and Agile, can be combined to address business and user goals [30].

SPL targets to structure several related systems in a stable domain, defining a common architecture, managing the variabilities among the products, and providing systematic reuse. However, domains are often unstable, the company may not have sufficient resources for the upfront SPL investment required, and requirements and technology volatility can occur in the projects. As Agile addresses many of these issues through incremental, iterative and

evolutionary development, an Agile SPL approach can handle scenarios hostile to systematic reuse in a more appropriate way [30].

As said in Section 2, there exist several ways to start the development of an SPL. Since the reactive approach offers a fertile soil for agile principals, this approach should be adopted.

The *reactive approach* [7, 39, 78] is an instance of Boehm's spiral model [17], an agile method to adopt a product-line approach. Developers start with a software product line SPL_0 that realizes an initial version of the envisioned software product line. In incremental steps from SPL_i to SPL_{i+1}, the product line progressively grows toward its ideal (Figure 2), covering the full variation spectrum, as defined during domain analysis which can also be incremental [6].

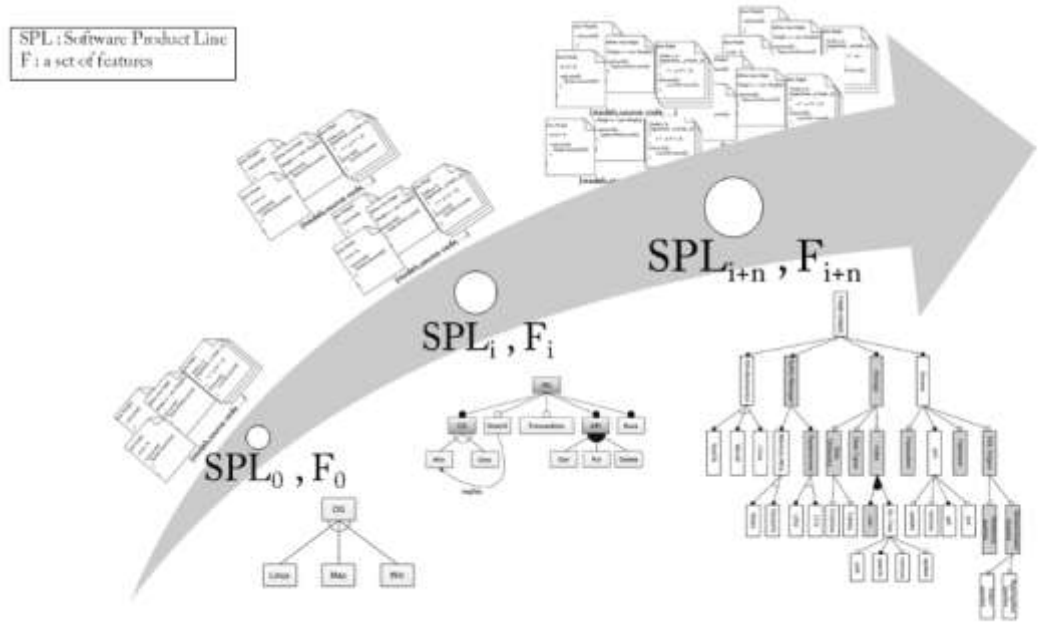

Figure 2 reactive approach: the SPL_i grows progressively as well as number of features

The reactive approach has three typical steps, which are:

1. Exploration and characterization of the requirements leading to a new product currently not covered by the product line;
2. Describing the delta (∂) leading to the improved product;
3. Implementing the delta (∂) in a suitable way.

Moreover, this approach presents typical patterns for maintaining and evolving a product line during its lifetime. Conceptually, the reactive approach [7] is positioned between the proactive and the extractive ones [39]. It requires less upfront planning than the proactive approach. However, including a feature may require invasive and expensive changes to the product line, because it has not been designed with that feature in mind. At the same time, the reactive approach is typically considered to be more structured than the extractive one, because each iteration follows clear planning steps. Overall, the reactive process aligns well with Agile methods of software construction [6].

4. Domain analysis and requirement analysis

4.1. Domain analysis

For SPLE community, *Domain Analysis (DA)* is a form of requirement analysis for the entire product line [77, 78]. As illustrated in Figure 1, DA is a part of the problem space and the first phase in the engineering process of the SPL approach. This phase encompasses all activities for eliciting and documenting the common and variable requirements of the product line [50].

The input for this phase consists of the product roadmap [50, 78]. The output comprises reusable, textual and model-based requirements and, in particular, the variability model of the product line itself. In addition, this phase anticipates prospective changes in requirements such as laws, standards, technology changes, stakeholder's goals and market needs for future applications. Hence, the output does not include the requirement specification of a particular application, but the common and variable requirements for all foreseeable applications of the product line.

Domain analysis contains two primary tasks: *domain scoping and domain modeling* [6].

Domain scoping is the process of deciding on the extent or range of a product line. During this step, domain engineers decides which of all possible requirements arising in a domain should be considered. The scope describes desired features or specific products that should be supported. Thus, at this step, domain experts collect information about the target domain, for example, by analyzing handbooks, existing systems, interviews with domain experts, potential customers, and so on [76]. It has to be highlighted that scoping decisions are design decisions depending on the goals of the company developing a software product line. As such, they are typically subjective and based on previous experience.

Domain modeling captures and documents the commonalities and variabilities of the scoped domain since the scope of a product line should be recorded. As a first approximation, stakeholders could give examples for possible products, as well as counter examples documenting which products are and which are not in the scope of the product line. Typically, commonalities and differences between desired products are identified and documented in terms of features and their mutual dependencies. The domain modeling is expressed in term of feature models [78].

4.2. Requirement analysis

Requirement analysis concerns especially the *"application engineering"* part (see Figure 1). This phase encompasses all activities for developing the application requirement specification [50]. One of the main issues at this stage is the detection of deltas (∂) between application requirements and the available capabilities of the platform.

Requirement analysis in product-line engineering is similar to requirement analysis in traditional software engineering. Requirement analysts solicit the customer's requirements, typically, using well-known requirement-analysis techniques, such as interviews and document analysis [25, 50]. However, in product-line engineering, one can build on the knowledge gathered during domain analysis.

Domain analysis has already identified possible (business) requirements arising in the domain, so requirement analysts try to map the customer's requirements to those identified earlier during domain analysis. Ideally, requirement analysis can be reduced to the selection of existing features, so that a product can be assembled using reusable implementations artifacts associated with these features. If a customer's requirement cannot be mapped to one or more existing features, several strategies are possible [6]:

1. Domain engineers can decide that the requirement is out of the scope of the product line, so they simply cannot provide a corresponding feature or product.
2. Domain engineers can assemble the next best product without this feature and manually extend the resulting product with custom extensions. This way, they invest additional implementation effort during application engineering, which is not integrated back into the product line.
3. Finally, domain engineers can decide to change the scope of their product line and include the additional requirement in the form of a new feature or changes to existing features, including domain artifacts. That is, they go back to domain engineering and implement a new feature or modify existing ones. Subsequently, they can map the customer's requirement to these features, from which other customers can also benefit.

Here, domain engineers face the emergent question: which path to take? This is a business decision that must be weighed. Additional development in application engineering to patch up a product is certainly cheaper in the short run than developing a new feature available for all the products (which involves again domain scoping and modeling steps), but other products of the product line cannot benefit from that development [76, 78].

Since this paper adopts a reactive approach, the third choice would be adopted. In fact, this strategy fits the Agile software product-line approach.

5. Variability management and requirements elicitation for ASPL

The broader the domain of a product line is, the larger is the number of possible *stakeholders' requirements* that can be covered in the form of individually tailored products. However, the broader the domain, the smaller is the set of similarities among products. Therefore, two issues play a crucial role in the ASPL approach: the explicit *handling of variability* and the systematic *reuse* of implementation artifacts [7, 78].

On the one hand, feature-oriented SPL development relies on the notion of features in order to identify variability and commonality between the members of an SPL. On the other hand, in literature, many approaches are presented to deal with the software product line requirements engineering and variability management [77].

Requirement engineering (RE) is at the core of ASPL. Similar to requirements engineering for a single system [22, 48] the success of a software product line highly depends on the correct understanding of the context in which it will be used, the understanding of the domain requirements, and also the proper modeling, and analysis of the requirements [3]. Additionally, due to the development principles (i.e., variability management and extensive reuse [7]) in software product lines, requirement engineering is more challenging and critical than

requirement engineering for a single system. As already pointed out, requirement engineering in product lines can be divided into domain requirement engineering and application requirement engineering lifecycles. In the domain requirement engineering, requirements are developed with the purpose of incorporating reusability and variability into requirements artifacts. On the other hand, in the application requirement engineering phase, target requirements are developed by reusing reference requirements [9].

5.1. Variability management

One of the main concepts in SPLs is *variability* which provides the required flexibility for product differentiation and diversification [21]. Variability is defined as the ability of SPLs to be exchanged, customized, configured or extended to be of use in a specific context. This is achieved through alternative definitions of reusable devices, included in the family of software products that give rise to different products [56] in a way similar to polymorphism in software engineering.

For [59] and [7], to efficiently implement software product lines, the underlying code has to be variable. As variability is very general concept, the paper adopts the following definition of variability:

Definition 2: Variability is the ability to derive different products from a common set of artifacts.

Variability management is an aspect that distinguishes SPLs from other software development approaches in the sense that it involves extremely complex and challenging tasks that must be supported by appropriate methods, techniques, and tools [21]. Several representations of variability, together with mechanisms that facilitate its systematic exploitation, have been reported in the literature, and some efforts have been made to unify the concepts and relations of the field of Software Product Lines [11].

In order to model such variability, methods can be broadly categorized into two groups: the feature-based group which comprises methods that model the common and variable features of a product family (an analysis-oriented perspective), and the architecture-based group, which includes methods that describe variability in terms of components, interfaces, connectors and so on (a more design-oriented perspective) [10]. At the requirement engineering (RE) stage the focus is on the first group, that is, languages based on the modeling of features [54]. In fact, in domain engineering, the "*feature modeling*" approach is used to understand the target domain and develop reusable artifacts [29].

5.2. The feature concept

Features are a fundamental notion in modern software engineering [6, 7, 27]. In fact, features have already profoundly affected how software is engineered. They can substantially improve the communication among all stakeholders and will likely lead to new, more effective ways to modularize and develop software [27].

Features are of primary interest in product-line engineering since they can be used to distinguish the products of a product line. The notion of a feature is inherently hard to define precisely as it captures, on the one hand, intentions of the stakeholders of a product line, including end users and, on the other hand, design and implementation-level concepts used to structure, vary, and reuse software artifacts [6]. Therefore, the notion of feature has many definitions. [24] has ordered some of these definitions from abstract to technical as shown in Table 1 classification:

Table 1 Abstract and technical definitions of "feature notion"

1	[37]	*"A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"*
2	[38]	*"A distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained"*
3	[29]	*"A distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept"*
4	[16]	*"A logical unit of behavior specified by a set of functional and non-functional requirements"*
5	[20]	*"A product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements"*
6	[12]	*"a product characteristic that is used in distinguishing programs within a family of related programs"*
7	[24]	*"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"*
8	[70]	*"An optional or incremental unit of functionality"*
9	[13]	*"An increment of program functionality"*
10	[7]	*"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"*

According to [6, 7, 77], the first seven definitions treat features mainly as a means to communicate between the different stakeholders of a product line (end users, managers, programmers, and so forth), in order to distinguish software products. The last three definitions treat features as design decisions and implementation-level concepts that are part of the software construction phase. These different views on features stem from the different use of features in the different phases of product-line engineering. Based on the essence and commonalities of prior usage, we endorse [6] to define features as follows:

Definition 3: A *feature* is a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all the phases of the software life cycle [6].

The product portfolio of a product line is defined by its features and their relations. A specific product is identified by a subset of features, called a *feature selection*. Not all feature selections are valid and specify meaningful products. A constraint on the feature selection is called a

feature dependency. Feature dependencies are modeled explicitly in product lines as part of feature modeling.

Definition 4: A product of a product line is specified by a valid feature selection (a subset of the features of the product line). A feature selection is valid if and only if it fulfills all feature dependencies [6].

Features, feature selections, feature constraints, and products emerge in all sorts of product lines, and especially in software product lines (SPL). In the rest of Section 5, we discuss the role of the feature concept in the product-line engineering process. We introduce feature models as a formalism to describe features and their constraints. Finally, a formalization of the feature model in propositional logic opens the door for formal methods of analyzing product-line variability.

5.3. Feature modeling

Modeling variability is a crucial step in SPLE approach. A common approach of variability modeling is to express it in terms of common and optional features, a process called *feature modeling*. Feature modeling presenting *feature models* and their graphical representation as *feature diagrams*, are currently the most popular form of variability models [53, 50, 28].

Feature modeling takes place in domain analysis, but its output plays a central role in other phases of product-line development, e.g., for requirements analysis and product derivation [6, 7]. In fact, the feature modeling specifies the set of valid products. Especially it helps analysts to introduce the graphical language of feature diagrams. Moreover, it allows them to connect the graphical representation to a formal representation that is the basis of engineering tools.

5.3.1. Feature Model (FM)

By definition, a feature model (FM) documents a product line variability, more precisely, the features of a product line and their relationships [7]. Since FM describes the relationship between features, it leads to define which feature selections are valid.

In its simplest form, a feature model comprises a list of features and an enumeration of all valid feature combinations [6, 76, 77]. However, such enumeration quickly becomes too large to be practical; therefore, other modeling approaches are required.

In this sense, [6] lists some potential information that domain analysts could collect on features such as proposed in Table 2:

Table 2 some potential information to collect on features [6]

1	Description of a feature and its corresponding (set of) requirements
2	Relationship to other features, especially hierarchy, order, and grouping
3	External dependencies, such as required hardware resources
4	Interested stakeholders
5	Estimated or measured cost of realizing a feature
6	Estimated or measured cost of realizing a feature
7	Potentially interested customers and estimated revenue
8	Configuration knowledge, such as 'activated by default'
9	Configuration questions asked during the requirements analysis step
10	Constraints, such as "requires feature X and excludes feature Y"
11	All kinds of behavioral specifications, including invariants and pre- and post-conditions
12	Known effects on non-functional properties, such as "improves performance and increases energy consumption"
13	Rationale for including a feature in the scope of the product line
14	Additional attributes, such as numbers and textual parameters, used for further customization during product generation
15	Potential feature interactions

In feature-oriented design and implementation, feature diagrams are considered as standard visual representation, whose semantics is specified by a translation into propositional logic. Thus, feature diagrams define a feature model as a hierarchy of features and constraints among them.

5.3.2. Feature Diagram

Classically, as in object-oriented modeling languages such as UML [79], BPMN [80], models are visually represented in a schematic form by diagrams in addition to formal specifications such e.g., OCL [81] in the case of UML. This has been motivated by practitioners and experiences on the field since it is more convenient and less tedious to model with diagram than formal specifications. Hence, a *feature diagram* is a graphical notation to specify a feature model. It is a tree whose nodes are labeled with feature names. Different notations convey various `parent-child` relationships between features and their constraints [6, 7].

If a feature `f` is a child of another feature `p`, `f` can be selected only when `p` is also selected. Typically, a feature diagram includes mutual relations between features. For example, the parent feature denotes a more general concept and the child a specialization [6, 76].

`Mandatory` and `optional` features are distinguished by a small circle on the child node. In fact, a filled bullet denotes a mandatory feature, whereas an empty bullet denotes an optional feature (see *Figure 5*). The parent node is labeled with `p`, the child node with `f` [78, 6, 7].

In Figure 4, the edges between a parent feature and a group of child features f_i are connected via an empty arc. This graphical element denotes a choice of exactly one feature out of a feature group (that is, choose one from $\{f_1 \ldots f_n\}$). Mathematically, it denotes *exclusive* disjunctions of features [6].

Figure 3 shows child features connected via a filled arc. This graphical element denotes an unrestricted choice of one or more features out of a feature group. It is chosen if, at least, one feature of the collection has to be selected, but there are no other restrictions. Mathematically, it denotes an *inclusive* disjunction of features [78, 6].

Definition 5: [6] defines a *feature diagram* as a graphical representation of a feature model as a tree over the feature *set* **F** and feature relations in terms of *parent*(p)–*child*(f) and integrity constraints. Each edge in the tree is defined by exactly one feature constraint, that is, by a declaration of one of the feature constraint types ***mandatory***, ***optional***, ***alternative***, or ***or***.

$$\mathtt{root(f)} \equiv \mathtt{f} \tag{1}$$

$$\mathtt{mandatory(p,f)} \equiv \mathtt{f} \Leftrightarrow \mathtt{p} \tag{2}$$

$$\mathtt{optional(p,f)} \equiv \mathtt{f} \Rightarrow \mathtt{p} \tag{3}$$

$$\mathtt{alternative(p,\{f_1,\ldots,f_n\})} \equiv \mathtt{((f_1} \vee \ldots \vee \mathtt{f_n)} \Leftrightarrow \mathtt{p)} \wedge \bigwedge_{i<j} \neg\mathtt{(f_i} \wedge \mathtt{f_j)} \tag{4}$$

$$\mathtt{or(p,\{f_1,\ldots,f_n\})} \equiv \mathtt{(f_1} \vee \ldots \vee \mathtt{f_n)} \Leftrightarrow \mathtt{p} \tag{5}$$

Additionally, a set of *cross-tree* constraints may be defined. The corresponding propositional formula of the feature constraints and the cross-tree constraints are conjoined resulting in one propositional formula that represents the semantics of the whole feature diagram [7]. See Figures 3, 4 and 5.

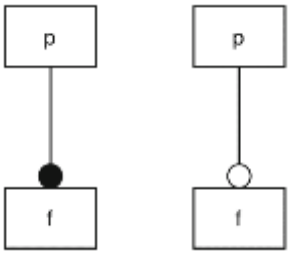

Figure 4 Optional and mandatory features. A filled bullet denotes a mandatory feature, and an empty bullet denotes an optional feature [7]

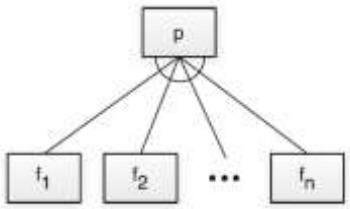

Figure 3 One-out-of many choice. This choice corresponds to a generalized **xor** operator [7]

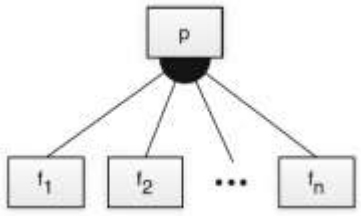

Figure 5 Some-out-of-many choice. This choice corresponds to the logical **or** operator [7]

Furthermore, Figure 6 illustrates the *metamodel* of feature. It summarizes the whole foundations seen above on feature management approach.

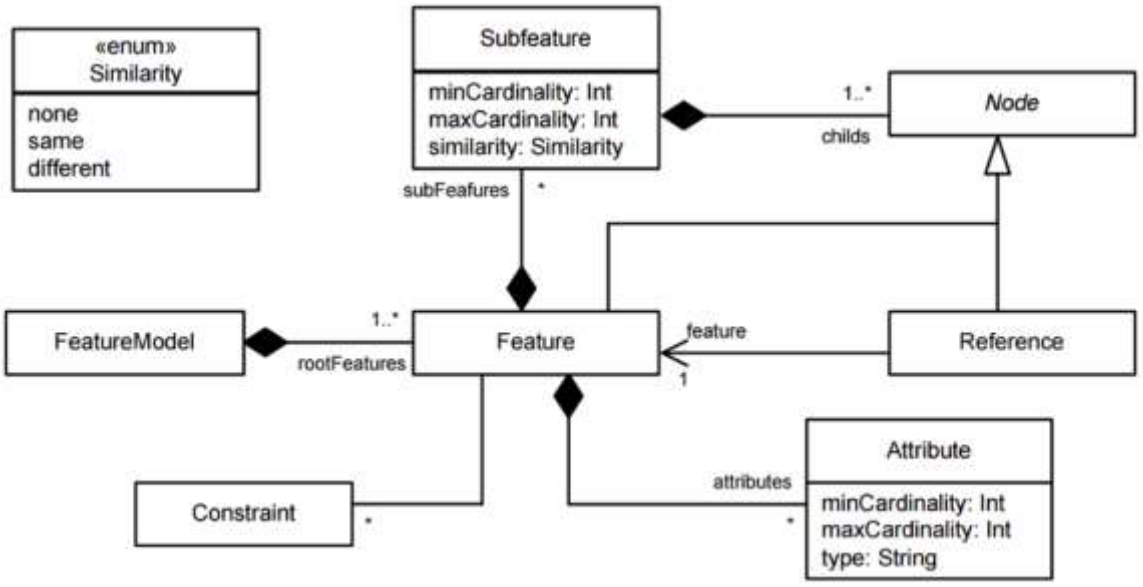

Figure 6 Feature Metamodel

6. Elicitation of requirements

As stated in Section 4.2 requirements analysis in product-line engineering is similar to traditional software engineering. In fact, in the application-engineering phase, analysts identify the requirements of a target application, design and realize the application by reusing the existing domain concepts and developed artifacts [9].

In the case of single information system (IS[1]) development, many requirement-engineering approaches are present and used within the software engineering methodologies. In fact, IS are more and more huge and complex. In order to deliver the expected IS on time, considering all the constraints linked to the development ecosystem, the concerned community has to lead an optimal requirements elicitation. The requirement elicitation aims to specify the IS completeness and correctness, besides guaranteeing the quality, validation and acceptance [45].

In order to achieve the objectives of this paper, the focus is on the goal-oriented requirement engineering models. This paper adopts the i* framework due to its wide adoption in software industry and research.

[1] Information System (IS) means the target software together with its environment made of human agents, devices, legacy software, etc.

6.1. Goal oriented requirement engineering

The goal-oriented approach of requirement engineering (GORE) focuses on the organizational goals of stakeholders to elicit requirements and intentions of a system. In fact, GORE refers to the use of goals – i.e., stakeholders' goals – are used to elicit, elaborate, structure, specify, analyze, negotiate, document, evaluate, and modify requirements [45] [61].

Definition 6: A goal is a prescriptive statement of intent that the system should satisfy through the cooperation of its agents [61].

Definition 7: An agent is an active system component playing a specific role in goal satisfaction [61].

According to the definition 6, the word "system" may refer to the *system-as-is*[2] or the *system-to-be*[3]. Goals are perspective[4] statements like requirements and unlike domain properties, as the latter, are descriptive[5]. Thus, goals are declarative, unlike operational procedures for achieving them. Therefore, the goal must be formulated in terms of phenomena shared among agents; such phenomena are controlled by some agents and monitored by others [32, 61, 83, 84].

As stated by [33, 61], goal identification is not necessarily an easy task. Sometimes goals are stated explicitly as system objectives during elicitation sessions. More often, they are kept implicit and analysts need to "mine" them from preliminary documents and from elicited material such as scenarios, workflow, descriptions and other operational details provided by stakeholders. Goals provide a basic abstraction for addressing the *WHY and WHO* dimensions of requirement engineering. In fact, when goals are explicitly stated as system objectives, analysts may ask *HOW* questions during elicitation to find out contributing sub-goals. When they are kept implicit in the scenarios and operational descriptions being elicited, analysts may ask *WHY* questions to make them explicit.

Goals can be stated at the different levels of abstraction:

- At higher levels, there are coarser grained goals stating strategic objectives related to the organization.
- At lower levels, there are finer grained goals stating technical objectives related to system design options.

This difference in goal level and granularity suggests a specification-structuring mechanism based on contribution links among goals. A coarser-grained goal may be refined into finer-

[2] The system-as-is, the system as it exists before the machine is built into it.
[3] The system-to-be, the system as it should be when the machine will be built and operated in it.
[4] Perspective statements state desirable properties about the system that may hold or not depending on how the system behaves. Such statements need to be enforced by system components. They are in the optative mood [61]
[5] Descriptive statements state properties about the system that hold regardless of how the system behaves. Such properties hold typically because of some natural law or physical constraint. Descriptive statements are in the indicative mood [33].

grained goals contributing to it or, conversely, finer-grained-goals may be abstracted towards coarser-grained goals to which they contribute [32].

There are different types of goals. A goal is either a ***behavioral goal*** or a ***soft goal***. A behavioral goal declaratively prescribes intended system behaviors. It can be established in a clear-cut sense; a system behavior satisfies it or not. A behavioral goal is an ***Achieve goal*** or a ***Maintain goal*** according to the pattern of temporal behavior that it prescribes. Achieve goals prescribe behaviors where some target condition must eventually become true. Maintain goals prescribe behaviors where some 'good' condition must be maintained or some 'bad' condition must be avoided (*Avoid goal*). Behavioral goals are used for building operational specifications of the system. A soft goal prescribes preferences among alternative system behaviors. It is more fulfilled along some alternatives and less along others. Soft goals cannot be established in a clear-cut sense. They are used as criteria for selecting system options among multiple alternatives [33, 61].

Different categories of goals may overlap. A ***functional goal*** states the intent underpinning a system service. Satisfaction goals and information goals are common sub-categories of functional goals. ***A non-functional goal*** states a quality or constraint on service provision or on system development. Accuracy, safety and security goals deserve special attention in mission-critical systems [82, 83, 84].

Goal refinement provides a natural mechanism for structuring complex specifications at different levels of concern. A goal refinement graph shows the refinement and contribution links among goals. Requirements and expectations appear as leaf nodes in this graph [61].

Goals play a crucial role in the RE process. They yield a precise criterion for requirements completeness and pertinence. Goal contribution links yield chains of satisfaction arguments that can be used for showing the alignment of the system-to-be with the organization's strategic objectives. Goals also provide anchors for risk analysis, conflict management and comparison of alternative options. Goal refinement structures provide useful information for evolution support and traceability management [32, 33].

Figure 7 Goal types

Definition 8: a *goal model* is a triple $GM = \{\mathcal{G}, \mathcal{C}, \mathcal{D}\}$. $\mathcal{G}$ is a set of goals (also called intentions or intentional elements). Intentions are (hard) goals ($\mathcal{G}_g$), tasks ($\mathcal{G}_t$) and soft goals ($\mathcal{G}_s$). $\mathcal{C}$ and $\mathcal{D}$ describe intentional relations on $\mathcal{G}$, $\mathcal{C}$ denotes positive and negative contributions ($\mathcal{G} \times \{Make, Help, Some+, Unknown, Some-, Hurt, Break\} \times \mathcal{G}$). $\mathcal{D}$ are decomposition relations of intentional elements $\mathcal{G} \times \{IOR, XOR, AND\} \times \mathcal{P}(\mathcal{G})$. [9]

Briefly, the use of GORE elicitation is based on a multi-view model showing how goals, objects, agents, scenarios, operations, and domain properties are inter-related in the system-as-is and the system-to-be [52]. Several models based on GORE such as i* [69, 32], KAOS [61], Lightswitch [51], NFR [23], GRL [34], and URN [34] are widely studied by academics and practitioners.

All these GORE languages are dedicated to the problem understanding and goal elicitation. They differ on the point of view they adopt to analyze the problem:

- *KAOS* focusses on the hierarchical decomposition of goals and their operational statements.
- *Lightswitch* addresses the goal-directed behavior of enterprises based on the maintenance of success in a changing environment. It relies on General Systems Thinking (GST) and Cybernetics principles.
- The *NFR* approach maps non-functional requirements to softgoals, decomposes them into a hierarchy of sub-softgoals and finally looks for impeding softgoals in other branches of the tree.
- *GRL* was standardized by the ITU-T, from the telecommunication field. It is also based on i* but generalizes the i* concepts and it has an explicit meta-model.
- *TROPOS* is based on i* but differs from it as it covers more than the early requirements phase and it has seen its syntax and semantics evolve apart. The main idea behind *Tropos* is to allow goals and softgoals up to the architectural design level [69, 82, 83, 84].

Among these goal modeling approaches this paper adopts the i* framework due to its wide adoption in software industry and research. Moreover, i* as a visual language, supports communication between stakeholders. In fact, i* deal well with the early requirements phase which is communication intensive.

6.2. i* Goal modeling approach

Scenarios based on GORE approaches, such as i*, model the early phase of the requirements analysis, and consider organizational contexts, intentions and rationales for stakeholders demands [15] [22]. The early requirements phase is thus concerned with the understanding of the problem and its social or organizational context. It investigates the problem of domain by eliciting the *WHO* (i.e., the stakeholders) and the *WHY* (i.e., the goals and their rationale) that support the development of the system under study. As already pointed out, early requirements engineering is a decisive phase of the IS development because overlooked or misunderstood requirements may lead to expensive errors during later stages [47].

6.2.1. About i*

The i* modeling language [69, 66, 65] was introduced to fill the gap in the spectrum of conceptual modeling languages, focusing on the intentional (why?), social (who?), and strategic (how? how else?) dimensions [33]. i* is a visual language where every construct of the language is depicted by a symbol and the whole information is represented on diagrams [31]. It is composed of the strategic dependency model (SD) and the strategic rationale model (SR). These models share a set of i* constructs which are linked by several relationship types. The following section defines briefly each of these terms and shows their visual aspects (see Figure 8).

6.2.2. i* construct types

- An ***actor*** is an active entity that undertakes actions to achieve goals by using its know-how. This construct is specialized in the notions of agent, role and position.
- An ***agent*** is a physical or tangible actor (i.e., a human being or a hardware operator (along with some software)) whose characteristics (e.g., skills, knowledge, and limitations) are not easily portable to another actor.
- A ***role*** is an abstract characterization of an actor's behavior according to a specialized context or a certain activity domain. These characteristics are easily transferable to other actors.
- A ***position*** is a set of roles taken on by an agent.
- An ***actor boundary*** sets the intentional boundary of an actor. All the elements inside its boundary are explicitly desired by the actor.
- A ***goal*** denotes an intentional desire of an actor. The goal does not give details about how it can be fulfilled.
- A ***softgoal*** is a goal whose satisfying criteria are not clearly established. On the contrary, an actor subjectively assesses the satisfaction of a softgoal. While a goal is said to be satisfied, a softgoal may be *satisfied*.
- A ***task*** is the manner that an actor considers to fulfil a goal or a softgoal.
- A ***resource*** denotes an informational or physical entity desired by an actor.
- A ***belief*** is an assumption about the world that the actor thinks is true. A belief differs from a goal, as it is true independently from any desire of the actor.

6.2.3. i* relationship types

- A ***means-end link*** indicates a relationship between an end (i.e., an objective to be reached) and a means to reach the objective.
- A ***decomposition link*** connects a task to the sub-element(s) which it is composed of. These sub-elements can be goals, softgoals, tasks or resources.
- A ***strategic dependency link*** indicates that an actor depends on another actor to obtain an intentional element. An intentional element is basically a thing that a user wants (i.e., a goal, a softgoal, a task, a resource, a belief).

- A ***contribution link*** is used to signify that an intentional element (i.e., a goal, a softgoal, a task, a resource, a belief) influences positively or negatively the satisfaction or the fulfilment of the softgoal. There are nine levels of contribution. The level of contribution is textually added on the link.

    - ***Make*** indicates that the contribution is positive enough to make the softgoal satisfied.
    - ***Help*** indicates a partial positive contribution but not sufficient on its own to satisfy the softgoal.
    - ***some+*** indicates a positive contribution without providing its strength (i.e., it is either a make or a help contribution)
    - ***Unknown*** denotes a contribution but its impact is unknown: it influences the softgoal positively or negatively.
    - ***some-*** indicates a negative contribution without providing its strength (i.e., it is either a hurt or a break contribution)
    - ***Hurt*** indicates a partial negative contribution but not sufficient on its own to deny the softgoal.
    - ***Break*** indicates a negative contribution sufficient enough to deny a softgoal.
    - ***AND*** means that the softgoal is satisfied if every sub-element is achieved.
    - ***OR*** means that the softgoal is satisfied if one of the sub-elements is achieved.

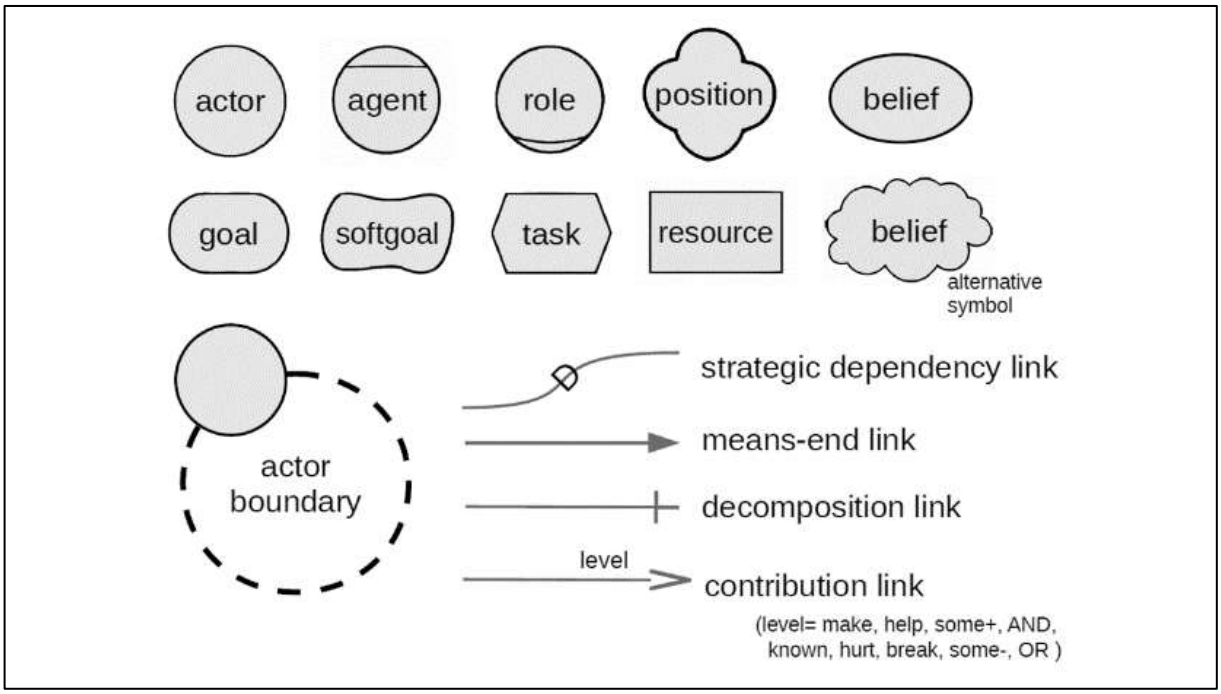

Figure 8 i* standard notations [31]

6.2.4. Strategic Dependency model (SD)

The strategic dependency model expresses the intentional relationships among actors. Visually speaking, it is a graph composed of a set of nodes and links. Each node denotes an actor and can be linked to one or several actors to indicate that this actor depends on the other for something in order to attain some goal. The "something" is visually represented by a node and is called the ***dependum*** [31]. A relationship between two actors is thus graphically represented by two links:

- a link from the ***depender*** actor to the ***dependum*** node
- and another link originating from the ***dependum*** to the ***dependee*** actor.

6.2.5. Strategic Rationale model (SR)

A strategic rationale model is a network that shows the rationale behind the actors' dependencies. SR models are, somehow, SD models that have been "opened up" to show the specific intentions of some of the actors of the SD model. The dependums that appear in the SD model are detailed: the rationale that supports each dependum is elicited through a hierarchy of goals, softgoals, tasks, resources and beliefs. This hierarchy details why the depender wants this dependum. Moreover, the SR model also develops another hierarchy having for root that dependum: it explains in terms of goals, softgoals, tasks, resources and beliefs how the dependum is obtained/satisfied [31].

7. Goal-oriented RE for Agile SPLs

From the previous discussions, Feature Modeling appears as a crucial technique to identify commonalities and variabilities in an SPL. However, Feature Models show only a specific perspective, which is not sufficient to express all the characteristics and constraints of an SPL [5]. Using a goal-oriented approach, such as i*, to complement (and help define) feature models would improve such models enhancing meaning and justification to features. Goal-oriented modeling provides a way to identify variabilities at an early phase of requirements, allowing alternative options to satisfy stakeholder's goals.

7.1. Goal-oriented models (GM) and Feature Models (FM)

This section presents the distinctions between goal-oriented models (goal models) and feature models in Table 3. In addition, a comparison of variability types in both models are outlined.

Tab. 3 Comparison of GM and FM

GM	***FM***
Is an artifact representing stakeholders' objectives and strategies. It describes the intentional space of a domain stakeholders [69]	Describes the configuration space of a product line. Accordingly, it represents characteristics of software products that belong to a product line [69]
Different terms are defined for goals: achieve, maintain, avoid, and cease [68]	Features can only be selected or deselected [68]
An intentional decomposition (except for soft goals) in a goal model is an entailment, i.e., an *AND-decomposition* of source intentional elements is one possible combination (among others) of source intentional elements that implies the target intentional element [41]	An *AND-decomposition* in a feature model is a *PartOf* relation that implies that a parent feature contains its child features [41].
Behavioral variability represents the various behaviors of a single system that may be used by its user [43]	*Product line variability* refers to differences between products in a product line, which may exist among their requirements, design models and implementation models [64]
OR decompositions (and similarly XOR-decompositions) represent intentional/behavioral variability [9]	*OR decompositions (and similarly XOR-decompositions)* represent product line variability [9]

7.2. OR / XOR decompositions

In goal models, OR and XOR relations represent variability in the stakeholders' goals (i.e., intentional variability), and the ways these goals can be achieved (i.e., behavioral variability) [9]. When developing a reference design and implementation models from the family goal model, these variability relations can lead to either *intentional/behavioral variability or product line variability* in the reference models.

First case: *OR decompositions (and similarly XOR-decompositions) represent intentional/behavioral variability*

OR decompositions (and similarly *XOR*-decompositions) represent intentional/behavioral variability, if all the products having a target intentional element involved in the OR relation (XOR relations), contain all source intentional elements of the OR-decomposition (XOR-decomposition) in their goal models.

Second case: *OR decompositions (and similarly XOR-decompositions) represent product line variability*

OR decompositions (and similarly *XOR*-decompositions) represent product line variability if source intentional elements involved in OR-decompositions (XOR-decompositions) vary between different products that contain the target intentional element.

At this stage, it should be noticed that feature models not only show variabilities in requirements but also encapsulate product line variabilities in designing and implementing models [37]. Therefore, a feature model may contains several variability relations that do not exist in the goal model.

Consequently, investigation of the notion of variability in the goal and feature models disclose the differences in their semantics of variability. *Variability relations* in goal models can be either *product line* variability or *intentional/behavioral* variability while feature models only represent the *product line* variability.

7.3. Extending the standard goal model

According to [9], the existing goal model notations do not discriminate between product line variability and intentional/behavioral variability. Thus, this paper presents an extension to the standard goal model notation in order to distinguish the types of variability in the family goal model.

For *OR-decompositions (XOR-decompositions)*, the product line variability and intentional/behavioral variability are distinguished using the ***variation point (VP)*** notation. OR-decompositions, showing product line variability, are labeled as ***VP***. In fact, VP defines alternative ways for fulfilling the goal [68].

In standard goal modeling, AND-decomposing a target intentional element (goals or tasks) into source intentional elements implies that the satisfaction of the target intentional element depends on the satisfaction of all of its sources [69] [61] [34]. However, in the extended-goal models presented in this section the fulfillment of some source of intentional elements in AND-decomposition may be necessary for the target intentional element of a particular product. However, their non-fulfillment does not make the target intentional element unsatisfiable in other products. To enable extended-goal models to present and describe these situations, [9] adds the notion of *optional goals*, resembling optional features in feature model. Hence, intentional elements in an AND-decomposition can be optional if their non-fulfillment does not lead to non-fulfillment of the target intentional element.

Moreover, for the extended-goal model, [9] uses soft goal as means for resolving product line variability in the intentional space. Therefore, *contribution links* can propagate the desired satisfaction level of soft goals into goals and tasks and help in the selection of a proper variant of product lines-based intentional variability.

> ***Definition 9:*** an *extended-goal model* $EGM = \{\mathcal{G}, \mathcal{C}, \mathcal{D}^F\}$ extends a goal model $GM = \{\mathcal{G}, \mathcal{C}, \mathcal{D}\}$ as follows: the decomposition relation $\mathcal{D}$ is extended by decompositions that cover product line variability $\mathcal{D}^F \subseteq \mathcal{G} \times \{IOR, XOR, AND, AND\text{-}O, IOR\text{-}VP, XOR\text{-}VP\} \times \mathcal{P}(\mathcal{G})$. [9]

7.4. Relating intentional elements to features (mapping)

In the domain engineering process, goal models capture intentional variability [67] [42] and describe the intentions behind existing features in the software product line. Hence, using the goal model, engineers can ensure that existing features and variability relations in feature models are aligned with intentional variability in the goal models. They can also trace back differences in products to differences in the intentions of the stakeholders [9].

According to the model proposed by [9], in order to represent an explicit mapping (i.e., a mapping indicated by domain engineers), a mapping relation for each mapped task is developed (See Figure 9). In addition, if a *feature* is mapped to more than one *goal or/and task*, then the corresponding feature appears in the mapping relations of all those goals or/and tasks. After explicit mapping between tasks in an *extended-goal model* and features in a *feature model*, engineers can drive implicit mappings between intermediate tasks and goals and features through existing relations in goal models and feature models. Based on [9] proposition, this paper adopts the following definition of ***mapping:***

Definition 10: Let $EGM = \{\mathcal{G}, \mathcal{C}, \mathcal{D}^F\}$ be an extended-goal model where $\mathcal{G} = (\mathcal{G}_g \cup \mathcal{G}_t \cup \mathcal{G}_s)$ and $FM = \{\mathcal{F}, \mathcal{F}_M, \mathcal{F}_O, \mathcal{F}_{IOR}, \mathcal{F}_{XOR}, \mathcal{F}_{incl}, \mathcal{F}_{excl}\}$ a feature model, $\Phi_i (\mathcal{G}_i, \mathcal{F}_i)$ is a mapping relation between an intentional element $\mathcal{G}_i \in (\mathcal{G}_g \cup \mathcal{G}_t)$ and a set of features $\mathcal{F}_i \subset \mathcal{F}$, and Φ denotes the set of all mappings between EGM and FM. [9]

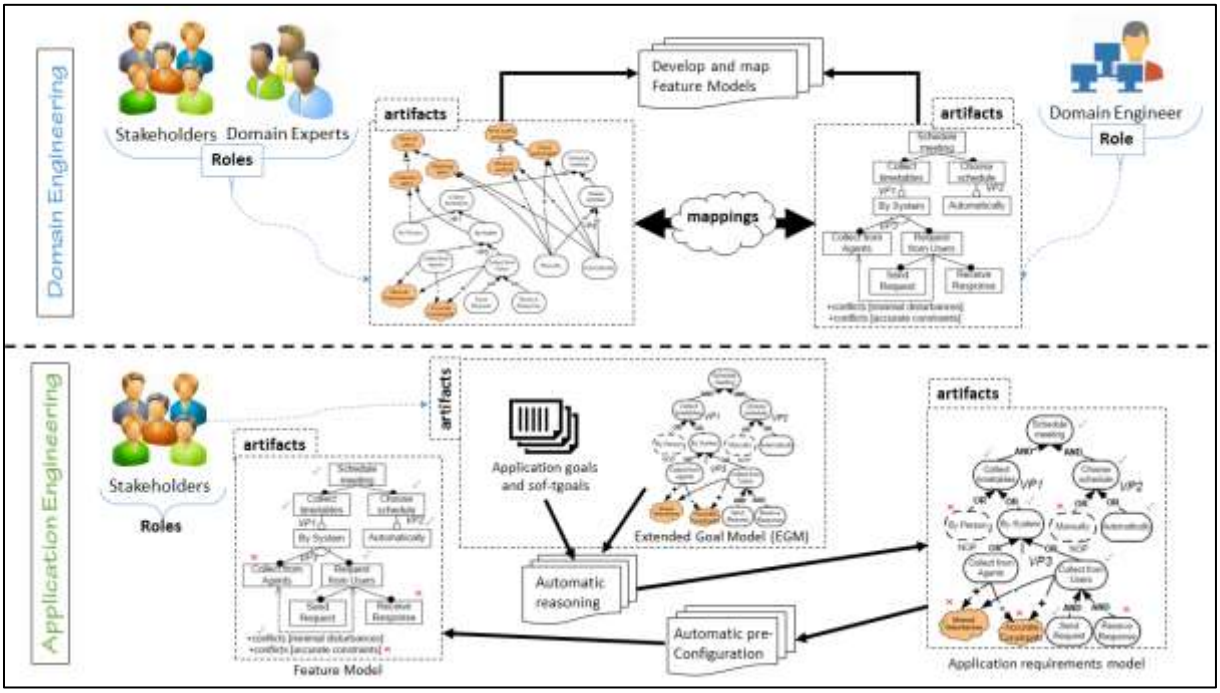

Figure 9 GORE approach for Agile Software Product line process – (Phase 1: Domain engineering) – (Phase 2: Application engineering)

7.5. Mapping between EGM and FM using i*

As stated above, GMs represent how the system-to-be and its environment can achieve root-level goals together. FMs, on the other hand, are only used to represent variability within the system-to-be. Therefore, in order to generate FMs, we need to identify a subset of goals in the extended-goal model (EGM) that are intended to be achieved by the system-to-be [67].

According to [67] and based on what [9] presents, in the first place, domain engineers need to identify two types of leaf-level goals. They have to identify which goals are assigned to the system-to-be and which are assigned to actors in its environment. The *leaf-level goals* to be achieved by the system environment are replaced with *NOP (no operation)* goals, they can identify parts of the goal model that are not assigned to the system and must not be mapped into features. They replace a non-leaf goal with an NOP goal to indicate that it is not the responsibility of the system if all of its subgoals are NOP goals.

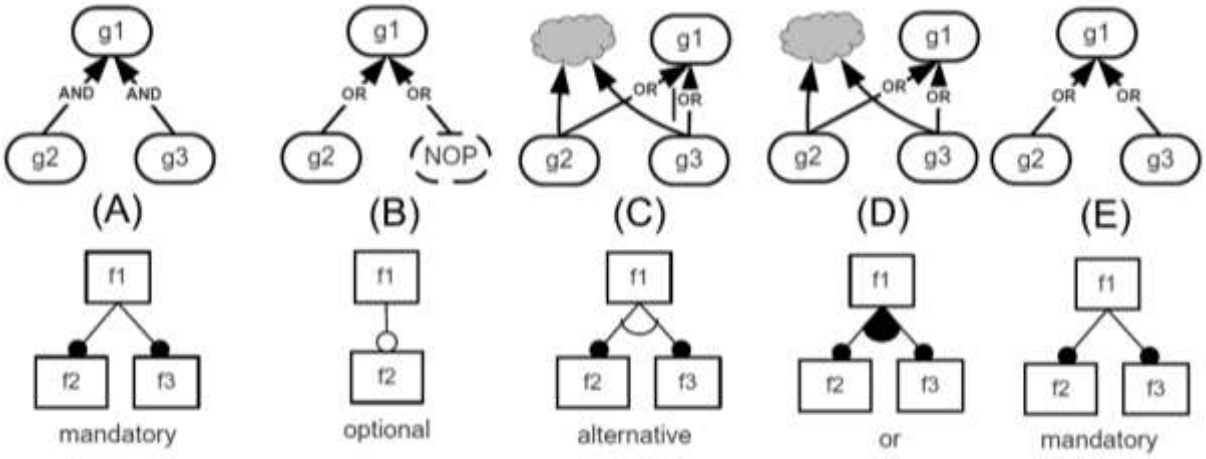

Figure 10 Semantics mapping between goals and features [67]

Next, every remaining goal node is mapped into a feature with the same name. It is now easier to see that AND/OR-decompositions of goals (Figure 10 A/D), if mapped into features, produce sets of Mandatory and OR-features respectively. However, Alternative and Optional feature sets do not have counterparts in the AND/OR-goal models. Thus, in order to generate these types of features engineers need to annotate the goal models. First, they analyze whether some of the OR-decompositions are, in fact, XOR-decompositions (where exactly one subgoal must be achieved) and then annotate these decompositions with the symbol **"|"** or *"XOR"* (figure 10 C). The annotated OR-decomposition corresponds to a feature refined into a set of alternative features. Similarly, to produce optional features, they identify patterns where a goal is OR-decomposed into a number of subgoals with at least one subgoal (NOP) being delegated to an agent in the environment of the system-to-be (figure 10 B). Then, the non-NOP sibling subgoals will be mapped into optional features. The generated user-oriented feature models reflect the fact that decompositions in goal models are more restrictive than in feature models. Thus, they produce feature models where features must have subfeatures of a single type and cannot have more than one set of Alternative or OR-features. One can further group them into mixed-type feature decompositions if appropriate [67] [9].

Moreover, for [67], *constraints* can be used in feature diagrams to represent relationships among variable features that cannot be captured by feature decompositions. These constraints include, for example, *mutual exclusion* and *mutual dependency*. Goal models allow the analysis of *alternative goal decompositions* with respect to their contributions to certain quality criteria. However, feature models provide no such facility and therefore the selection of features for a member of a product line family is not explicitly guided by non-functional requirements. To alleviate this issue, softgoal contributions present in goal models can be used to generate feature model constraints that relate features with corresponding goals that *help (+)* or *hurt (−)* the same softgoal. For instance, if two system-delegated goals help (respectively, hurt) the softgoal **S**, then both their corresponding features will most likely have to be included in (respectively excluded from) the system, provided that the softgoal is of importance for that system variant in the goal model. Thus, domain engineers generate a mutual dependency constraint between the two features. The constraint label includes the strength of the softgoal contribution and its name to document the source of the constraint (e.g., +depends[S], if both goals help S). Similarly, if two system-delegated goals have opposite contributions to a softgoal, then selecting both corresponding features in a system that tries to satisfy the softgoal will be counterproductive. This will result in a mutual exclusion constraint between the two features. Thus, the constraints help in the feature selection process by accounting for stakeholders' quality concerns.

Feature constraints are parameterized by a *softgoal **S*** to indicate that they are significant only when S is important to the stakeholders. As well, the strength of the softgoal contributions determines the strength of the constraints (as shown by $++|+|-|--$). Trade-off can be made when multiple feature constraints are parameterized by multiple softgoals. The preferences and priorities in the softgoals give rise to the ranking of the importance of the feature constraints in the feature model. The process is also extended to support constraints among feature sets [67].

In order to illustrate the steps stated in this section, the example of "Schedule Meeting"[6] [62] is used. To "Schedule Meetings" (a stakeholder goal), one needs to "Collect Timetables" and to "Choose Schedules". Each of the subgoals of "Schedule Meetings" has two alternative solutions, done either manually "By Person" or automatically "By System". A system can collect a timetable "From Agents" or directly "From Users", which can be done by "Sending Requests" and "Receiving Responses". In addition, the "Minimal Effort" softgoal can be achieved by minimizing "Collection Effort" and minimizing "Matching Effort". Similarly, "Good Quality Schedule" is guaranteed by having "Minimal Conflicts" and "Good Participation". Furthermore, "(Collecting Timetable) By Person" is a tedious task for a meeting scheduler, thus it hurts the criteria of "(minimal) Collection Effort". Figure 11 shows a goal model of the example of "Schedule Meeting".

[6] The example of "Schedule Meeting" is a simplified version of the *The Meeting Scheduler Problem: Preliminary Definition* provided in [69].

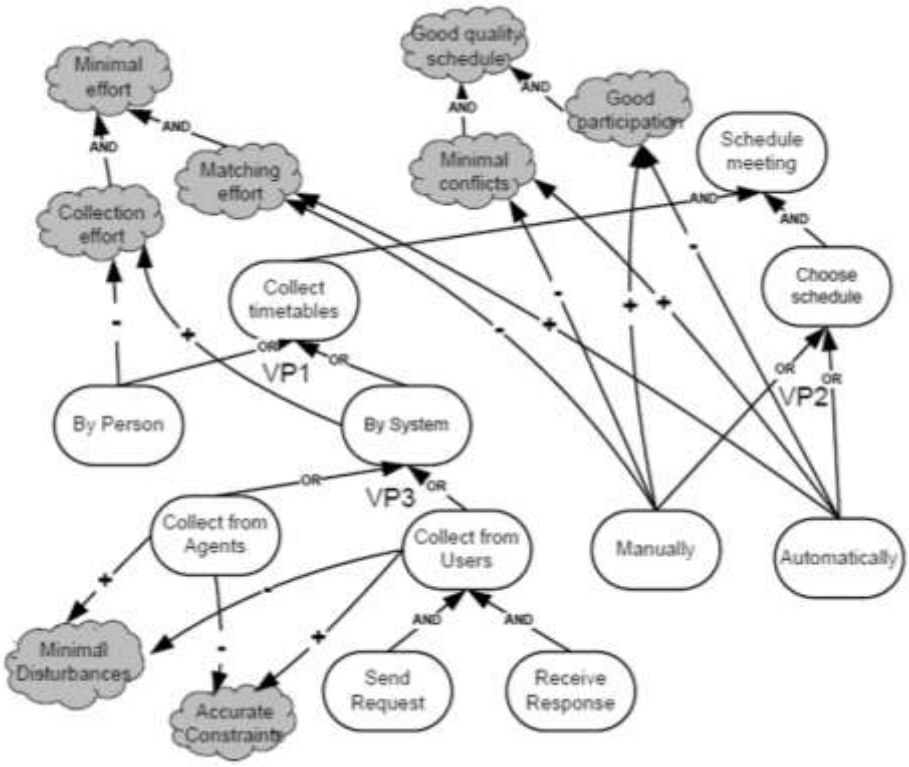

Figure 11 A goal model of the "Schedule Meeting" [67]

Figure 9 is simplified into a system-only goal model (Figure 12a), then four types of features are created, and two conflicting constraints are generated based on the two pairs of conflicting contributions to softgoals (Figure 12b). In the figure, one can see the correspondence between variation points (VP) in the two models.

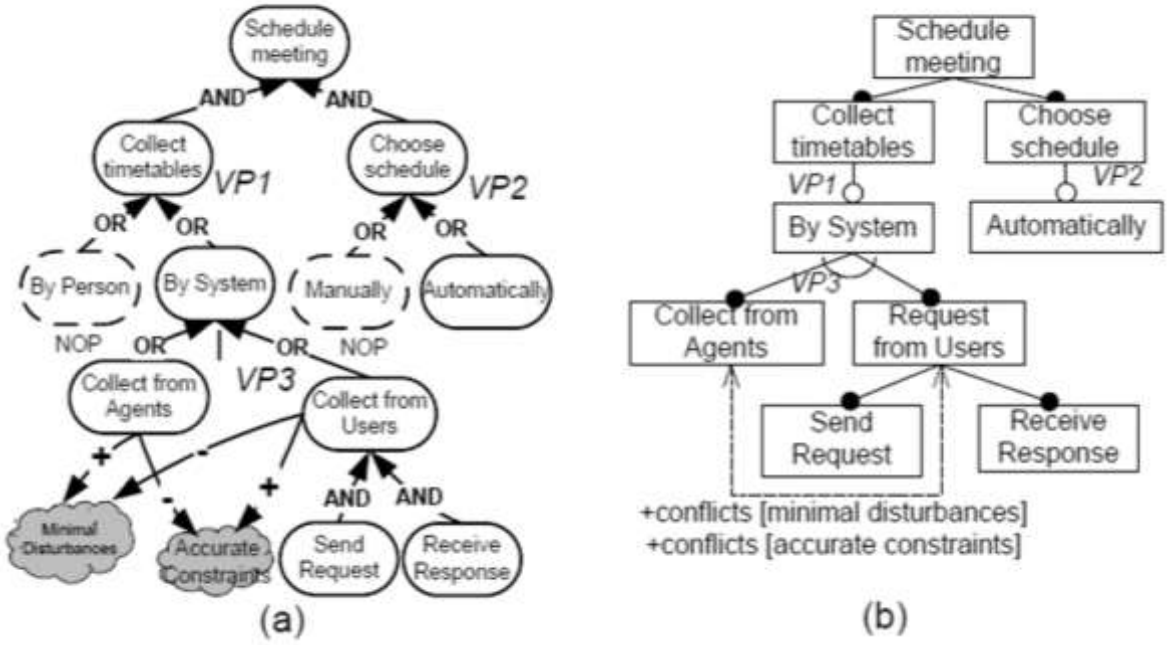

Fig. 12 a feature model (FM) generated from the goal model of "Schedule Meeting" in fig. 9 – (a) is a system-only goal model of the running example – (b) is the generated FM. [67]

8. Discussion

In this section, we present our arguments for choosing to adopt the explained goal-modeling approach. In fact, several publications recommend using GORE in the context of SPLE.

For example, Yu et al. [67, 68] developed an approach to (1) generate design models from goal models and (2) use the goal models for configuration of the generated design models. They introduce in [67] a set of annotations for generating feature models from goal models, which they extend in [68] to design models such as state-charts and components.

In addition, Silva et al. [55] employ aspectual i* to support variability in software product lines. Mandatory features are considered internal elements of i*, while optional, alternative and OR features are treated as aspectual elements in their approach. Composition rules are defined for all variabilities in the models. Introducing aspects in i* models and describing composition rules for all variabilities increased the complexity of models, which leads to scalability problems. Finally, this work is based on the assumption that each optional and alternative feature is mapped into one aspect.

Furthermore, goal and feature model relations are investigated by Antonio et al. [5]. They propose an approach to derive feature models from i* goal models. A feature is defined as a relevant characteristic of the system, while a system characteristic allows achieving a certain goal. They integrated cardinalities into i* models to represent (optional and alternative cases) variability in the models. Furthermore, a means-end relation may be transferred into OR-relation or alternative relation according to the cardinality of means involved in the relation. In addition to the extension on i*models, they introduced a set of heuristics to produce a feature model from i* models.

Many other examples were found in literature that deal with the issue of the use of goal-modeling technique in SPLE. However, this paper adopt the approach presented by Asadi et al [9]. Here after, it is explained why the approach of Asadi et al. was adopted.

The main target of paper's authors is to develop a good framework that could be used in the context of agile software product lines (ASPL). Requirement engineering it is a corner stone since it is needed at different stages of the ASPL process. It is thus important to choose the adequate RE approaches that present a useful requirement elicitation for the ASPL.

The work introduced by Asadi et al [9], is found adequate for Agile SPL. In fact, their approach mainly differs from other approaches in providing a comprehensive validation approach that takes potential changes in feature models. Those changes can appear during the design of SPLs and validates whether variability relations in the feature model are aligned with the intentional relations in the family goal model. Additionally, they discriminate between product line variability and behavioral variability in goal models, which is neglected in most of the time. Consequently, their contribution advances the state-of-the art in GORE for SPLs with a formal and correct validation approach for family requirement models. Their proposed validation assures that intentional variability of an ASPL captured in goal models, does not violate constraints of technical variability captured in feature models. The proposed validation approach compares the influence of intentional relations given by a goal model with feature relations from the feature model. That is why their approach was adopted.

9. Conclusion and future work

The aim of this paper was to figure out an adequate requirement engineering approach that fit agile software product lines (ASPL). After investigating the literature, it was found that goal oriented requirement engineering (GORE) approaches are good candidates to complement (Agile)SPL. In fact, GORE approaches could offer useful technique to use in "Problem Space" of the ASPL process.

Both goal models and feature models have been used successfully in software engineering research and practice to capture stakeholders' needs and intentions [32] [2] [4] and manage product line variability [38] [44]. Their combinations advance requirements engineering and configuration in the ASPL context. Applying goals in software product line not only facilitates identifying features in domain engineering life cycle, but also ease the selections of features based on stakeholders' intentions and needs in application engineering life cycle. Several publications and tooling support demonstrate the practical applicability of the goal-oriented requirement engineering in the context of SPLE. Among these references, the approach presented by Asadi et al [9] was adopted for its great usefulness in agile contexts.

In addition, based on [9] approach and ASPL foundations, this paper shows how feature models could be generated from i* model. The example of "Schedule Meeting" was used to illustrate the different steps.

To conclude, the main goal of this paper was to show how a feature model could be generated from a goal-oriented model, and also to justify the choice of the approach used to achieve this target.

In our future work, we will investigate the use of user stories (US) within the ASPL process and we will develop a framework that could be used for ASPL.

10. References

1. Agra C., Sousa A., Melo J., Lucena M., and Alencar F.: Specifying guidelines to transform i* Model into User Stories: an overview. In Proceedings of the 8th International i* Workshop (iStar 2015). CEUR Vol-978. (2015).

2. Ali R., Dalpiaz F., Giorgini P.: A goal-based framework for contextual requirements modeling and analysis. Requir. Eng. 15(4), 439–458 (2010)

3. Alves V., Niu N., Alves C., and Valença G.: "Requirements engineering for software product lines: A systematic literature review," Information and Software Technology, vol. 52, p. 806–820, (2010).

4. Amyot D., Ghanavati S., Horkoff J., Mussbacher G., Peyton L., Yu E.: Evaluating goal models within the goal-oriented requirement language. Int. J. Intell. Syst. 25, 841–877, (2010).

5. António S., Araújo J.A., Silva C.: Adapting the i* framework for software product lines. In: Proceedings of the ER 2009Workshops on Advances in Conceptual Modeling—Challenging Perspectives, pp. 286–295. Springer, Berlin (2009)

6. Apel S., Lengauer C., Möller B., Kästner C.: An algebraic foundation for automatic feature based program synthesis. Sci Comput Program 75(11):1022–1047. (2010).

7. Apel S., Batory D., Kästner C., and Saake G.: Feature-oriented Software Product Lines: Concepts and Implementation. Springer-Verlag, Berlin Heidelberg (2013).

8. Asadi M., Bagheri E., Mohabbati B., and Gašević D.: Requirements Engineering In Feature Oriented Software Product Lines: An Initial Analytical Study. In Proceedings of the 16th International Software Product Line Conference, SPLC '12, vol. 2, pp. 36-44. (2012).

9. Asadi M., Gröner G., Mohabbati B., and Gasevic D.: Goal-oriented modeling and verification of feature-oriented product liens. Softw Syst Model 15: 257 (2016).

10. Asikainen T., Männistö T., and Soininen Kumbang T.: a domain ontology for modelling variability in software product families. Adv. Eng. Inform. 21 (1), pp. 23–40. (2007).

11. Bachmann F., Goedicke M., Leite J., Nord R., Pohl K., Ramesh B., Vilbig A.: A meta-model for representing variability in product family development. Software Product-Family Engineering, Springer, pp. 66–80. (2004).

12. Batory D., Sarvela JN., Rauschmayer A.: Scaling step-wise refinement. IEEE Trans Software Eng (TSE) 30(6):355–371. (2004).

13. Batory D., O'Malley S.: The design and implementation of hierarchical software systems with reusable components. ACM Trans Software Eng Methodol (TOSEM) 1(4):355–398. (2005).

14. Beck K., et al.: Manifesto for Agile Software Development. http://agilemanifesto.org/ (accessed October 2016), (2001).

15. Bhuiyan M. M. R., Zahidul Islam M. M., Krishna A., and Ghose A.: Co-evolution of Agent Oriented Conceptual Models and Use Case Diagrams. In Proceedings of the 6th international Conference on Quality Software, QSIC 2006, (2006).

16. Bosch J.: Design and use of software architectures: Adopting and evolving a product-line approach. ACM Press/Addison-Wesley, (2000).

17. Boehm B. W.: A Spiral Model of Software Development and Enhancement. Computer 21(5):61–72. http://dx.doi.org/10.1109/2.59. (1988).

18. Bürdek J., Lochau M., Bauregger S., Holzer A., von Rhein A., Apel S., and Beyer D.: On facilitating reuse in multi-goal test-suite generation for software product lines. In: Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, (2015).

19. Cao L. and Ramesh B.: Requirements Engineering Practices: An Empirical Study, IEEE Software, Volume: 25, Issue: 1. page(s): 60-67, (2008).

20. Chen K., Zhang W., Zhao H., Mei H.: An approach to constructing feature models based on requirements clustering. In Proceedings of the International Conference Requirements Engineering (RE). IEEE Computer Society, pp 31–40, (2005).

21. Chen L., Ali Babar M.: A systematic review of evaluation of variability management approaches in software product lines Inf. Softw. Technol., 53 (4), pp. 344–362, (2011).

22. Cheng B. H. C., and Atlee J. M.,: Research Directions in Requirements Engineering, in 2007 Future of Software Engineering, Washington, DC, USA, p. 285–303, (2007).

23. Chung L., Nixon B., Mylopoulos J., and Yu E.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, (2000).

24. Classen A., Heymans P., Schobbens P.: What's in a feature: A requirements engineering perspective. In Proceedings of the International Conference Fundamental Approaches to Software Engineering (FASE) Lecture notes in computer science, vol 4961. Springer, pp 16–30, (2008).

25. Clements P. and Northrop L.: Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, (2001).

26. Cockburn A.: Agile Software Development: The Cooperative Game. Boston, Pearson, (2007)

27. Czarnecki K.: Forward in: Feature-oriented Software Product Lines: Concepts and Implementation. Springer-Verlag, Berlin Heidelberg, (2013).

28. Czarnecki K., Grünbacher P., Rabiser R., Schmid K., Wąsowski A.: Cool features and tough decisions: A comparison of variability modeling approaches. In Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS). ACM Press, pp 173–182, (2012).

29. Czarnecki K, Eisenecker U., Generative programming: Methods, tools, and applications. ACM Press/Addison-Wesley, p. 34, (2000).

30. Da Silva I.F., Da Mota Silveira Neto P.A., O'Leary P., De Almeida E.S., and De Lemos Meira S.R. Using a multi-method approach to understand agile software product lines Information and Software Technology, 57 (1), pp. 527-542, (2015).

31. Genon N. Unlocking Diagram Understanding: Empowering End-Users for Semantically Transparent Visual Symbols, PhD thesis, Université de Namur, Press universitaire de Namur, Namur, (2016).

32. Giorgini P., Mylopoulos J., Sebastiani R.: Goal-oriented requirements analysis and reasoning in the TROPOS methodology. Eng. Appl. Artif. Intell. 18, 159–171, (2005).

33. i* wiki webpages (istarwiki.org), http://istar.rwth-aachen.de (accessed on November 2016), (2016).

34. ITU-T: Recommendation Z.151 (10/12): User Requirements Notation (URN) Language Definition, Geneva, (2008).

35. Jaqueira A., Lucena M., Alencar F., Agra C., and Aranha E. Integrating User Stories and i* Models: A Systematic Approach. In Proceedings of the 17th International Conference on Enterprise Information Systems (ICEIS-2015), pages 428-438, (2015).

36. Jaqueira A., Lucena M., Aranha E., Alencar F., and Castro J., Using i* Models to Enrich User Stories. In Proceedings of the 6th International i* Workshop (iStar 2013), (2013).

37. Kang K., Cohen SG., Hess JA., Novak WE., Peterson AS. Feature-oriented domain analysis (FODA) feasibility study. Tech Rep CMU/SEI-90-TR-21, SEI, (1990).

38. Kang K., Kim S., Lee J., Kim K., Kim G., Shin E.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Ann Softw Eng 5(1), pp.143–168, (1998).

39. Krueger C.W.: Easing the transition to software mass customization. In Proceeding of the International Workshop on Software Product-Family Engineering (PFE) Lecture Notes in Computer Science, vol 2290. Springer, pp 282–293, (2002).

40. Larman C., and Vodde B.: Scaling Lean and Agile Development. Thinking and Organizational Tools for Large-Scale Scrum. Addison Wesley. Westford, Massachuestts, USA, (2008).

41. Liaskos S., Lapouchnian A., Wang Y., Yu Y., Easterbrook S.: Configuring common personal software: a requirements-driven approach. In: 13th IEEE International Conference Requirements Engineering, pp. 9–18, (2005).

42. Liaskos S., Lapouchnian A., Yu Y., Yu E., Mylopoulos J.: On goal-based variability acquisition and analysis. In: Requirements Engineering, 14th IEEE International Conference, pp. 79–88, (2006).

43. Liaskos S., Litoiu M., Jungblut M. D., Mylopoulos J.: Goal-based behavioral customization of information systems. In Proceedings of the 23rd International Conference on Advanced Information Systems Engineering. CAiSE'11, pp. 77–92. Springer, Berlin, (2011).

44. McGregor J., Muthig D., Yoshimura K., Jensen P.: Successful software product line practices. Softw. IEEE 27(3), 16–21, (2010).

45. Mesquita R., Jaqueira A., Agra C., Lucena M. and Alencar F. US2StarTool: Generating i* Models from User Stories. In Proceedings of the 8th International i* Workshop (iStar 2015), CEUR Vol-978, (2015).

46. 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: 15th IEEE International Requirements Engineering Conference, pp. 243–253. RE, IEEE, (2007).

47. Mylopoulos J. and Easterbrook S. Information Systems Analysis and Design. In Lecture notes - University of Toronto, (2003).

48. Nuseibeh B., and Easterbrook S., "Requirements engineering: a roadmap," in Proceedings of the Conference on the Future of Software Engineering, p. 35–46, (2000).

49. O'hEocha C., & Conboy K.: The role of the user story agile practice in innovation. In P. Abrahamsson & N. Oza (Eds.), Lean enterprise software and systems (pp. 20-30). New York: Springer, (2010).

50. Pohl K., Böckle G., van der Linden F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Berlin, (2005).

51. Regev G.: A Systemic Paradigm for Early IT System Requirements Based on Regulation Principles: The Lightswitch Approach. PhD thesis, Swiss Federal Institute of Technology (EPFL), (2003).

52. Robert A., Elliott Sr., and Edward B. Allen.: A methodology for creating an IEEE standard 830-1998 software requirements specification document". In Journal of Computing Sciences in Colleges, pp. 123-131, vol. 29 issue 2, December 2013, (2013).

53. Schmid K., Rabiser R., Grünbacher P.: A comparison of decision modeling approaches in product lines. In Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS). ACM Press, pp 119–126, (2011).

54. Sepúlveda S., Cravero A., and Cachero C.: Requirements modeling languages for software product lines: A systematic literature review, Information and Software Technology, vol. 69, pp. 16–36. (2016).

55. Silva C.T.L.L., Alencar F.M.R., Araújo J., Moreira A., de Castro J.B.: Tailoring an aspectual goal-oriented approach to model features. In: SEKE, pp. 472–477, (2008)

56. Sinnema M., Deelstra S.: Industrial validation of COVAMOF, J. Syst. Softw., 81 (4), pp. 584–600, (2008).

57. She S., Lotufo R., Berger T., Wa˛sowski A., Czarnecki K.: Reverse engineering feature models. In Proceedings of the International Conference Software Engineering (ICSE). ACM Press, pp 461–470, (2011).

58. Sobernig S., Apel, S., Kolesnikov S., and Siegmund N.: Quantifying Structural Attributes of System Decompositions in 28 Feature-oriented Software Product Lines: An Exploratory Study. Technical Reports / Institute for Information Systems and New Media, 2014/01. WU Vienna University of Economics and Business, Vienna, (2014).

59. Svahnberg M., van Gurp J., Bosch J.: A taxonomy of variability realization techniques. SoftwPract Exp 35(8):705–754, (2005).

60. Tian K., and Cooper K.: Agile and Software Product Line Methods: Are They So Different? http://www.cs.upc.edu/events/aple/TianCooper.pdf (accessed on October 2016), (2014).

61. Van Lamsweerde A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, New York (2009).

62. Van Lamsweerde A., Darimont R., and Massonet Ph.: The Meeting Scheduler Problem: Preliminary Definition. Université Catholique de Louvain, Unité d'Informatique, Belgium, (1995).

63. Weiss, D.M., Lai, C.T.R.: Software Product-line Engineering: A Family-based Software Development Process. Addison-Wesley Longman, (1999)

64. Weiss D.M.: The Product Line Hall of Fame. In: SPLC, p. 395. IEEE, (2008)

65. Yu E.: Modelling Strategic Relationships for Process Reengineering. PhD thesis, Department of Computer Science, University of Toronto, (1995).

66. Yu E.: Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97), pages 226–235,Washington, DC, USA. IEEE Computer Society, (1997).

67. Yu Y., do Prado Leite J.C.S., Lapouchnian A., Mylopoulos J.: Configuring features with stakeholder goals. In Proceedings of the 2008ACM Symposium on Applied Computing, pp. 645–649. SAC '08, ACM. New York, NY, USA, (2008).

68. Yu Y., Lapouchnian, A., Liaskos S., Mylopoulos, J., Leite J.: From goals to high-variability software design. In: Foundations of Intelligent Systems. Volume 4994 of LCNS, pp. 1–16. Springer, Berlin, (2008).

69. Yu E., Giorgini P., Maiden N., Mylopoulos J. (eds.): Social Modeling for Requirements Engineering. MIT, Cambridge, MA, (2011).

70. Zave P.: An experiment in feature engineering. In: Programming Methodology. Springer, pp 353–377, (2003).

71. Kolp M., Giorgini P. and Mylopoulos J., Organizational multi-agent architectures: a mobile robot example, In Proc. of the 1st Int. Joint Conference on Autonomous Agents Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, pp. 94-95, (2002).

72. Mouratidis H., Kolp M., Faulkner S. and Giorgini P. A secure architectural description language for agent systems, In Proc. of the 4th Int. Joint Conference on Autonomous

Agents Multiagent Systems, AAMAS 2005, July 25-29, Utrecht, The Netherlands, pp. 578-585, (2005).

73. Mylopoulos J., Kolp M. and Giorgini P. Agent-Oriented Software Development, In Methods and Applications of Artificial Intelligence, Proc. of 2nd Hellenic Conference on AI, SETN 2002, Thessaloniki, Greece, April 11-12, pp. 3-17, (2002).

74. Faulkner S. and Kolp M., Towards an agent architectural description language for information systems, In Proc. of the 5th Int. Conference on Enterprise Information Systems, ICEIS 2003, Angers, France, April 23-26, pp. 59-66, (2003).

75. Giorgini P., Kolp M., Mylopoulos J., Multi-agent and software architectures: A comparative case study, In Agent-Oriented Software Engineering III, 3rd Int. Workshop, AOSE 2002, Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions, pp. 101-112, (2002).

76. Kang K. C., Sugumaran V., and Park S. Applied Software Product Line Engineering. 1st ed. Boston, MA: Auerbach Publications, (2010).

77. Clements P. and Northrop L. Software Product Lines: Practices and Patterns. 8th ed. Boston, MA: Addison-Wesley Longman Publishing Co. (2012).

78. van der Linden F., Schmid K., and Rommes E. Software Product lines in Action: The Best Industrial Practice in Product Line Engineering. Berlin: Springer-Verlag, (2007).

79. OMG. OMG Unified Modeling Language (UML™) Specifications. Version 2.5, http://www.omg.org/spec/UML/2.5/ (2015).

80. OMG. OMG Business Process Model and Notation (BPMN™). Version 2.0, http://www.omg.org/spec/BPMN/2.0/ (2011).

81. OMG and IBM. Object Constraint Language (OCL™). Version 2.4, http://www.omg.org/spec/OCL/2.4 (2014).

82. Hoang H. Quality-Aware Agent-Oriented Information-System Development. PhD thesis, Université Catholique de Louvain (UCLouvain), Louvain Research Institute of Management (ILSM), (2010).

83. TROPOS Project, http://www.troposproject.org/ (accessed 06/01/2017).

84. Kolp M., and Mylopoulos J. Software architectures as organizational structures, In Proc. ASERC Workshop on "The Role of Software Architectures in the Construction, Evolution, and Reuse of Software Systems", Edmonton, Canada, (2001).