Chapter 3
Variability Scope

Rafael Capilla

What you will learn in this chapter

- *The importance of binding system options*
- *The notion of variability in space*
- *Variability constraints and dependencies*
- *Automation of variability scoping techniques*

1 Introduction

A fundamental aspect of variability modelling and for software product line engineering refers to the scope of the product portfolio that is to know the number and type of the products to be produced. As software variability concerns with multiple product development and multiple product configurations, there is a need to delimit the scope of the products and determine the size of the domain in our product line. Scoping identifies what products are "in" our product line and relies on a set of allowed options described in the variability model to determine the list of feasible products that can be built. Therefore, software engineers must define which design choices and combinations of them will be valid for a given market segment.

There are many reasons (e.g. economic, business, technical) for delimiting the scope of the SPL products and thereby the scope of the variability model. Each reason must justify why a number of available choices must be out of the selection and product configuration activities, as the high number of combinations in large variability models, often belonging to industrial product lines, makes unmanageable and unfeasible the development and maintenance of a large set of software products. Consequently, delimiting the number and type of products must be driven

R. Capilla (✉)
Rey Juan Carlos University, Móstoles, Madrid, Spain
e-mail: rafael.capilla@urjc.es

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_3,

by the scope of valid options defined in the architecture. Such restrictions are often based in a set constraint and dependency rules defined for the software artefacts and used to prune the number and type of products we can develop. In this chapter we will deal with notion of variability in space and with the reasons and technical solutions used for bounding the design choices used to keep the SPL products under control.

2 Scoping Activities

The need SPLs focus on specific market segments motivates domain scoping activities. Therefore, *domain scoping* is considered one of the first SPL activities used to delimit the number and type of products that will be inside the product line. Scoping activities narrow the domain of the product portfolio for the success of the SPL from a business and economic perspective. As a result, the scope of the variable options is also delimited by rules and constraints aimed to reduce the number and type of allowed products.

As discussed in [1], product portfolio analysis results are key to evaluating and establishing the type of products we want to engineer. As the product line evolves, the product portfolio may grow or change, and the variability implemented in the architecture must be flexible enough to support new variations in a controlled manner. Scoping activities also encompass the identification of requirements that are common to all products and those ones that make the difference between SPL products. Such activities will have a great impact on commonality and variability analysis to identify the variable parts in the architecture and with reusability of components and products in mind.

Moreover, *market analysis* activities are also carried as an early step before launching the product line in order to determine the product portfolio and to encompass which assets and products will be part of the product line. Therefore, product variants are defined and modelled on the basis of scoping activities and driven by economic and business reasons that keep the product line competitive.

Otherwise, the flexibility of variability models aimed to support a broad number of products in the product line scope often relies on more technical activities and current SPL capabilities, such as extensibility of variability models to support evolution and product configuration and derivation tasks. Bosch [2] mentions three different forms of scoping:

(i) *Domain scoping* aims at defining the boundaries of the domain where artefacts and products will be used. Domain analysis techniques are often used to delimit the scope of domain products and to derive the products from domain models (see also *Design Space Models* for product line scoping [3]).
(ii) *Product scoping* defines the products that will be engineered, often under a product line approach.
(iii) *Asset scoping* focuses on the identification of those reusable assets that will be employed in the construction of the software products.

These forms of scoping are used to constraint the number and type of options of variability models in order to make them more manageable. The scoping activity is fundamental for the product line strategy and economic benefits depend on how well the scope is chosen (e.g. a large scope may waste the investment of assets while a narrow scope may lead to not supporting reuse across all relevant products) [4].

Clements [5] states the importance of product line scope as a crucial activity for bounding the limits of the product line and define what's "in" and what's "out." Proactive approaches attempt to delimit the full scope of products when a product line is launched from scratch, while reactive approaches deal more with the scope of new products as the product line evolves and when new requirements appear. Scoping is sometimes considered a fuzzy activity during variability modelling and product line start-up, but several reasons motivate its importance in a product line context.

2.1 Reasons for Scoping

We can think in many reasons to enact scoping activities, but most of them may fall into the following categories:

- *Economic*: As not all the products can be built, there is a strong need to reduce the number and type of the assets and products because of economic reasons. Sometimes a product is technically feasible but difficult to sell and hence, it should not be included in the product line. For instance, an expensive product supporting a large number of configurable options that many of them will never be used. However, the case of a software product supporting only one single variation could be included in a product line if it shares a large number of assets. In other cases, a company can produce hundreds of products using a highly customizable variability model but building and maintaining such huge number of products will be highly costly (e.g. due to an excessive number of software development hours). Consequently, only those configurable assets and products that are worthy of value must be considered within the scope and a balance between the cost supporting a large number of configurable options (i.e. more products may lead to a broader scope) and a given pricing scheme must be achieved for each particular customization strategy.
- *Business*/*Strategic*/*Commercial*: Many times the variability model can support the development of a certain number of worthy configurable products, but business, strategic or commercial reasons may suggest to, for instance, delay its development. During scoping activities we do not restrict the scope of the variants for those products that will not be engineered in a certain period of time. Rather, we support such variations as part of the scope of the product line model but we decide later if certain variants (often known as *internal variability*) will be available in a new version of the product because the market demands new features (e.g. activate a new feature in the software of a mobile phone that remains hidden or unavailable in previous models of the SPL).

- *Technical*: Delimiting the scope of products or assets is necessary for maintenance reasons, as huge variability models are difficult to maintain and manage and may also increase product derivation activities. We use constraints not because the current technologies cannot support an infinite number of combinations but because of technical and other business reasons. Some systems that exhibit a large number of dependencies between their assets increase maintenance effort (e.g. the dependency network between packages in Linux kernels) and something similar may happen with variability models. Therefore, it is desirable to keep the number of dependencies and constraints under control and use tools for automating these tasks.
- *Cultural/Political*: Sometimes different configurable options are driven by cultural factors such as the language of use in different countries, which may lead to supporting a variety of languages in the GUI menu options of the product, while the functionality of the software remains the same. Delivering a software product in only a certain number of countries (e.g. due to political or military reasons) is another form to delimit the scope of the variants.

2.2 Variability in Space

Once the product line assets and products are well scoped, we can say that the pair variants and variation points defined in the variability model are ready to be used in product configuration and derivation activities in order to produce the reusable assets and the products in a given domain. The number and type of configurable products are determined by the design options defined in the architecture and implemented in the code assets. We refer to this as *variability in space*, where product line artefacts and releases are engineered and configured from the same variability model and belonging to a given domain.

Definition 3.1. Variability in space

Variability in space represents the set of products, releases and reusable assets that can be derived and configured from a concrete variability model in a given timeframe.

2.3 Notation for Binding Time

Variability in space provides the necessary ability to produce multiple products through variant selection and takes advantage against single-system development when several products and configurations must be engineered and put on time in the market. As mentioned in [6], "*feature declarations model the scope of variation in the production line*," and the adoption of software mass customization must support the complete scope of products on a predictable horizon. Also, depending on how

flexible and extensible the variability model has been designed to support evolution, new requirements should not be a problem if new design options and constraints can be easily added without changing the structure of the variability model.

In addition, the scope of the variability model is not only limited by the configurable options available but also by the constraint and dependency rules that will determine which products are allowed or within the scope. Such constraints must be described and implemented as part of the variability model, such as we explain in next sections.

3 Variability Scope

Product line scoping in its different forms have a direct impact on bounding variability. Because huge variability models applied in industrial product lines offer a large number of possible combinations, the feasibility to build only a subset of these products must rely on the limits established in the variability model to support a reduced number of allowed products.

3.1 The Graphical Limits of FODA

Variability models often use FODA trees to provide a graphical representation of the system features and how these interrelate with each other. A FODA tree describes the system features in terms of mandatory, alternative and optional variants which are also related using the notion of variation point. This hierarchy forms a tree where the root node represents the type of products we want to build.

One weak aspect of FODA trees is how constraints between features, used to delimit the variability in space, can be represented graphically. Also, representing variation points to relate variants located in different parts of the FODA tree can complicate the visualisation capabilities of the variability model, in particular in large feature models. In FODA, it is commonly accepted to draw a direct line associating two features to describe that there is a relationship between them, which can be either a constraint or a dependency rule, but constraints and dependencies are often managed separately from the graphical representation of the feature tree. With FODA, structural dependencies are modelled graphically and configuration constraints among optional and alternative features are specified separately to reduce the complexity of the graphical representation. Both of them must share the same name space. The same happens when we want to relate two or more variants and group these under a common variation point. A circle or dotted line surrounding the variants in the variation point is often used, but the logical formula describing such relationship must be written out of the FODA tree.

Figure 3.1 shows an example of a feature tree where variation points are surrounded by a dotted line and relationships between features are described

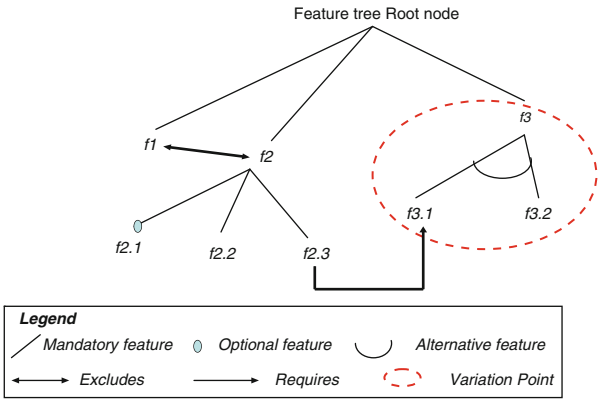

Fig. 3.1 A FODA tree example annotated with different types of relationships between features

using a solid line, but as mentioned, all this information must be described in textual form apart from the graphical representation. Therefore, FODA trees are simple and useful techniques to visualise the entire or a subset of the variability model, but the rules and constraints that define the limits of the allowed products must be defined and managed in a textual form. The existence of hundreds of features often makes hard the proper visualisation of all the potential constraints used to delimit the variability implemented in the product line products. For instance, in Fig. 3.1 we show three sample types of relationships that can be used to define the scope of the variability model, such as the following:

- Feature *f1* **excludes** feature *f2*.
- Feature *f2.3* has one **requires** relationship with feature *f3.1*. For instance, a feature cannot be activated if another feature has not been activated first. This can be seen as a special case of, the "requires" dependency.
- A **variation point** VP_x is defined to encapsulate and relate the alternative features *f3.1* and *f3.2* using, for instance an OR logical connector and having feature *f3* as parent of the relationship (e.g. $VP_{f3} = \{f3.1 \text{ OR } f3.2\}$).

Non-graphical representation techniques like matrixes can be also used to describe the dependencies and constraints of features. In addition, languages supporting rules and constraints constitute an interesting alternative as they can be processed automatically by software.

3.2 Variation Points

A variation point defines a relationship between features of a feature model and represents an area of a software system affected by variability. Variation points are used to relate two or several features located in the feature tree, and from the same parent or from different ones. Variation points encompass set of variants and other variation points that are represented by a logical formula that uses logical

connectors (e.g. OR, AND, XOR) to relate features. As not all the possible combinations are valid, the scope defined for each variation points is restricted by system constraints that limit the scope of products in space.

Variation points provide a flexible way to play with the scope of system features by grouping them as related functionality, often implemented as subsystems. When a variation point relates distinct functional parts of a system, the resultant area has a broader scope and the variability implemented in related system functional parts can be managed as a whole by means of such variation point. For instance, the variability implemented in the architecture that manages the electronics of a car can be used to describe the variations of both the Navigation subsystem while other features describe the variability implemented in the Multimedia subsystem (i.e. radio, DVD). Both subsystems can be integrated under one variation point representing an integrated multimedia system which can be also managed using a common control centre (e.g. the BMW's iDrive system consists of a button that manages all functions of the vehicle control system).

Variation points are often represented in feature trees as circles or boxes surrounding the variants included in the variation point, but because this technique may distort the representation of the feature model, variation points are better described separately in text notation or grouped in tables. The distortion of feature trees when using variation points can be reduced if we group subsystems or related functionality from the same parent, as we can avoid crosscutting lines across the feature tree. In huge variability models, it is rather difficult to avoid the existence of variation points relating distant features located in the tree or belonging to different parents, as in other case this may lead to a reorganisation of the whole variability model.

Just to give an example, Fig. 3.2 shows an example of variation points belonging to the same and to different parents. In the first case (left side of the figure), a variation point is defined and comprises three different variants. In the second case (right side of the figure), a variation point is defined to relate two distant features containing the variants defined in the feature tree but belonging to two different parents and depicted using a dotted circle line, as FODA lacks an explicit notation to describe such cases. In both situations the variation point defines the scope of certain functionality or related system features but this is managed differently. In the second case the scope seems to be broader than the first case because the functionality encompassed in the variation point shown in the right side of the figure encompasses functionality that belongs to separated or different part of the software product.

Because the scalability of the graphical representation of feature models is sometimes limited to describe and/or visualise hundreds of variation points, we need machine-processable techniques to solve this problem. However, most of FODA implementations are machine processable, as described in the Appendix of the FODA report, which includes a method of textual specification and also the extensibility of the model.

From an architecture point of view, variation points can be annotated as UML text notes and stereotypes in UML diagrams as no specific notation or neutral

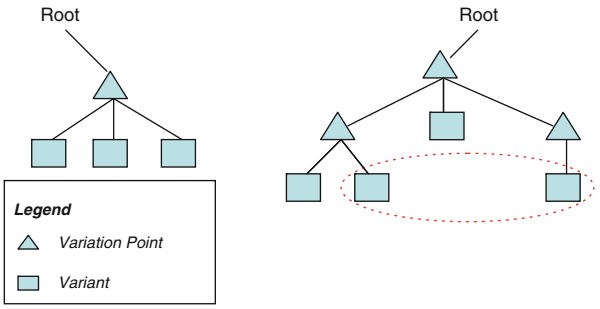

Fig. 3.2 Variation points belonging to the same and different parents

standard format exists to describe a variation point in the software architecture. This lack is common to all UML modelling tools and specific tooling has to be used to describe the variability of systems, in particular for industrial product lines where hundreds of variants and variation points need to be defined. Hence, the constraint and dependency rules used for delimiting the scope of the variability model can be hardly represented in the architecture and specific variability modelling and management tools are required.

3.3 Variability Constraints

In a feature-oriented approach, features are usually not independent each other, and the number and type of allowed products that can be technically and economically produced in a product line is often restricted using *requires* and *excludes* constraints (i.e. a kind of dependency). These variability constraints describe additional relationships between product features that can be hardly represented in the feature tree. Such dependencies can be applied either between variants and variation points in order to restrict the number of feasible product variations and thereby the number of product configurations.

- *Requires* dependency: It is used to represent that a variant V_x or a variation point VP_x needs another variant V_y or variation point VP_y. A requires dependency means that when a feature is selected the other must be present in the same product.
- *Excludes* dependency: It is used to represent that variant V_x or a variation point VP_x excludes another variant V_y or variation point VP_y. That is, an excludes dependency means that two features cannot be present in the same product.

In FODA, an arrow between two variants or variation points labelled with "requires" or "excludes" is enough to describe graphically such relationships, but the high number of such constraints in large variability models makes that all these rules must be processed automatically depending on the language used. Simple *if–then* constructs are enough to describe these dependencies, but constraint

programming constitutes another alternative to describe the dependencies between features.

Example 3.1. Requires and excludes dependencies using *if–then*

A feature fy *is required* if a feature fx is present

IF (fx) THEN fy

A feature fy *is excluded* if a feature fx is present

IF (fx) THEN NOT fy

A feature fz (e.g. a variant) *is required* if the variation point represented by features fx AND fy is present

IF (fx AND fy) THEN fz

In addition, the *requires* and *excludes* dependencies can be defined statically when product options are bounded before runtime or dynamically when such dependencies define a runtime condition during product execution or as part of a runtime reconfiguration process.

Example 3.2. Static and dynamic "requires" and "excludes" constraints

Static: During a software installation procedure, a software package requires another package before it is installed. Hence, a static requires dependency is defined and resolved.

Dynamic: During system execution, the software of an elevator checks the maximum allowed weight before the user can press the button of a given floor. In this case, a dynamic excludes dependency is realised at runtime when the maximum weight is exceeded.

Variability models delimit the solution space using *requires* and *excludes* dependencies to constraint the diversity of products, but these dependencies often complicate the variability model due to a high number of interrelated relationships between variants and variation points. As a consequence, the variability that is coded in a given subsystem or product becomes less reusable and difficult to decouple when the product options have dependencies to other system features.

3.4 Operational Dependencies

Feature dependencies have many implications in the development of product line assets and products as these are used to delimit the scope of the structural variability. However, other dependencies are possible. As mentioned in [7], *operational dependencies* represent implicit or explicit relationships between features that happen during the operation of the system. This kind of dependencies can be considered as different forms of requires and excludes dependencies but associated to runtime properties rather than to those defined statically in the feature model. Therefore, operational dependencies delimit the scope of execution features instead of the scope of the number and type of products; however, they can be used to configure products with different execution capabilities.

Based on a previous work [7], we describe the following six operational dependencies[1]:

- *Usage dependency*: It represents a feature that depends on other features for the correct system functioning. For instance, the location of certain services in a mobile phone depends on the correct functioning of the GPS system feature.
- *Modification dependency*: The behaviour of a feature might be modified by another feature while it is in activation. For instance, the feature that activates the Anti-lock braking system (ABS) in the car depends on the features controlling the sensors of the wheel, and the ABS feature works differently based on the information received from the sensors.
- *Activation dependency*: The activation of a feature depends of another feature, and it can be classified into the following four categories:
  - *Exclusive-activation dependency*: This dependency refers to features that cannot be active at the same time.
  - *Subordinate-activation dependency*: It represents a feature that can be active while another feature is also active.
  - *Concurrent-activation dependency*: Two or more features that are subordinated to an active parent feature must be also active at the same time (i.e. concurrently).
  - *Sequential-activation dependency*: Some subordinators of a parent feature must be active in sequence, and the parent feature will be active after the completion of the sequence.

The complexity of modern software systems may lead to many expected and unexpected situations where the status and operation mode of a system may change and more operational dependencies may arise to deal with new situations when the environment changes.

4 Automating Variability Scoping Checking

In large variability models, where hundreds of features are required, the number of constraints and dependencies may become unmanageable and hence, automatic mechanisms are necessary (1) to check that the right products will be produced, and (2) to ensure the compatibility between hundreds of constraints and dependency rules.

The automatic analysis of feature models can be used to check the scope of the product line products and their different configurations based on the provided variability in order to ensure the compatibility of hundreds of product constraints.

[1] In this chapter we will consider operational dependencies as part of a previous work of one of the co-authors of this book rather than a mere reference to the related work.

Table 3.1 Mapping features to propositional logic and CSP notations

Feature relationship	Propositional logic	CSP

OR
P = (X OR Y)
	$P \leftrightarrow (X \wedge Y)$	If (P > 0)
		Sum (X, Y)
		Else
		X = 0, Y = 0

Excludes X *excludes* Y	$\neg (X \wedge Y)$	If (X > 0) Y = 0
Requires X *requires* Y	$A \rightarrow B$	If (X > 0) Y > 0

As nicely described in [8], there are different techniques that can be used to check the consistency of dependencies in feature model. In this chapter we summarise two representative techniques used to automate the analysis of feature models.

- *Propositional logic*: It uses a propositional formula consisting of a set of primitive variables related by logical connectors aimed to constraint the values of the variables. A feature model can be mapped as a propositional formula and then use SAT solvers[2] to determine the satisfiability of the formula expressed using first-order logic. The formula can be specified in Conjunctive Normal Form (CNF) and uses three logical symbols, as connectors (i.e. $\neg$, $\wedge$, $\vee$) that are used by most SAT solvers. Features are mapped to variables in the propositional formula and the relationships between features are described using several formulas and including constraints.
- *Constraint programming*: Is a programming paradigm where relations between variables are stated in the form of constraints. These constraints can be described using Constraint Satisfaction Problems (CSPs) (e.g. A or B is true) where the values for the variables are found and all constraints are satisfied. Conversely to propositional formulas, a CSP solver can deal with numerical values in addition to Boolean ones. Feature models can be mapped as CSP variables with values TRUE or FALSE, while the relationships between features are defined as constraints. A description of the usage of CSP solvers in the automated analysis of feature models can be found in [9].

Table 3.1 shows an example on how constraints and dependencies of a feature model can be expressed in propositional formulas and CSP.

5 Areas of Practice

Product line scoping is a key activity for the success of the product line. In the early stages of the SPL phases, domain scoping is sometimes perceived a fuzzy task and difficult to carry out. Hence, one first area of practice is to define clearly the scoping

[2] A SAT solver is a software that takes as input a propositional formula and determines if the formula is satisfiable, that is there is a variable assignment that evaluates the formula to true. Input formulas are often specified in Conjunctive Normal Form (CNF) notation [8].

activities in the SPL approach used, not only at the process level but also in variability modelling tasks. Some well-known SPL approaches like PuLSE [10] have a domain scoping phase (i.e. PuLSE-Eco) used to identify the scope of the product line and determine the product line members. Other approaches (Software Engineering Institute's Framework for Software Product Line Practice 5.0[3]) combine economic and business reasons to establish SPL scoping activities with more technical activities focused on production constraints.

Closer to variability management techniques, staged configuration of feature models are used to iteratively select features in order to reduce the variability in the feature model [11]. This technical activity can be seen as a way to reduce the scope of the final products during product derivation. In other cases, new features can be added to enhance the functionality of a given product (e.g. the calculator product line incrementally adds new functionality by adding features). Using this approach, errors can be detected easily on each stage.

Another area of practice concerns with the evolution of the current asset and product features. If variability models become too rigid to expand the scope for new product line members, a reorganisation of the structural variability is needed, maybe because the variability model is unable to support runtime changes. In more flexible approaches, where runtime variability is supported, existing features can be modified or new ones added affecting the scope of the product line.

6 Summary

Product line scoping is an important and challenging area to determine the allowed product configurations that will belong to the product line. As discussed in the chapter, there are several reasons (e.g. economic, business, technical, etc.) that justify the need for product line scoping activities at various levels of abstraction such as domain or product scoping.

We have described the notion of variability in space to refer to the number and type of products to be produced from a given variability model and limit the scope of the products in the product line. FODA and their successors (i.e. extended notations of the original FODA) or constraint programming techniques are of common use to describe variability models and to determine the valid product configurations.

Finally, other forms of dependencies between features highlight these relationships from a runtime perspective rather than from the structural point of view, as many systems that use context information requires additional capabilities to adapt themselves to a new environment, and variability that is managed at execution time play an important role. Hence, these new dependencies must be

[3] http://www.sei.cmu.edu/productlines/frame_report/index.html.

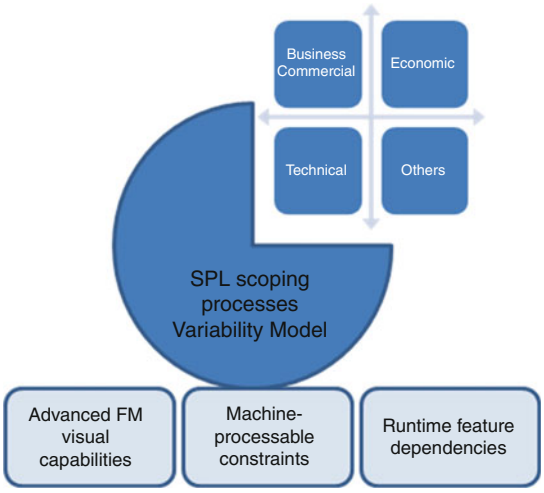

Fig. 3.3 Drivers and techniques for SPL variability scoping activities

able to address those runtime concerns among features that exploit runtime conditions.

Figure 3.3 summarises the main reasons for SPL scoping activities and the related techniques used to delimit the scope of variability models.

7 Outlook

Well-defined product line and variability scoping techniques are still needed. However, feature models are widely used to describe the variability of software systems, but other representation forms are required to accomplish the interrelationships between hundreds of features. Hence, new ways to represent large variability models and filtering techniques to describe a subset or a subsystem containing variability are welcome. Moreover, the scalability of feature models must be managed efficiently to expand or reduce the scope of the product line variants and hence facilitate the evolution of the product line. New trends and techniques in runtime variability models will help to support the dynamicity of systems and ecosystems and represent dynamic relationships between features that the structural variability cannot describe.

References

1. Pohl, K., Böckle, G., Van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)

2. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach. Addison-Wesley, Reading, MA (2000)
3. Tekinerdogan, B., Aksit, M.: Managing variability in product line scoping using design space models. In: Software Variability Management Workshop, Ankara, Turkey, pp. 5–12. University of Twentepp (2003)
4. Schmid, K.: A comprehensive product line scoping approach and its validation. In: ICSE'02, pp. 593–610. ACM DL (2002)
5. Clements, P.: On the importance of product line scope. In: Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain. LNCS, vol. 2290, pp. 70–78. Springer (2001)
6. Krueger, C.: Easing the transition to software mass customization. In: Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain. LNCS, vol. 2290, pp. 282–293. Springer (2001)
7. Lee, K., Kang, K.C.: Feature dependency analysis for product line component design. In: ICSR 2004. LNCS, vol. 3107, pp. 69–85. Springer (2004)
8. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Inf. Syst. **35**(6), 615–636 (2010)
9. Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A.: Using Java CSP Solvers in the automated analysis of feature models. In: GTTSE 2005. LNCS, vol. 4143, pp. 399–408. Springer (2005)
10. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.-M.: PuLSE: a methodology to develop software product lines. In: SSR 1999, pp. 122–131 (1999)
11. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer (2004)