Requirements Eng (2015) 20:323–343
DOI 10.1007/s00766-014-0203-1

ORIGINAL ARTICLE

A model for tracing variability from features to product-line architectures: a case study in smart grids

Jessica Díaz · Jennifer Pérez · Juan Garbajosa

Received: 16 February 2013 / Accepted: 20 January 2014 / Published online: 7 February 2014

Abstract In current software systems with highly volatile requirements, traceability plays a key role to maintain the consistency between requirements and code. Traceability between artifacts involved in the development of software product line (SPL) is still more critical because it is necessary to guarantee that the selection of variants that realize the different SPL products meet the requirements. Current SPL traceability mechanisms trace from variability in features to variations in the configuration of product-line architecture (PLA) in terms of adding and removing components. However, it is not always possible to materialize the variable features of a SPL through adding or removing components, since sometimes they are materialized inside components, i.e., in part of their functionality: a class, a service, and/or an interface. Additionally, variations that happen inside components may crosscut several components of architecture. These kinds of variations are still challenging and their traceability is not currently well supported. Therefore, it is not possible to guarantee that those SPL products with these kinds of variations meet the requirements. This paper presents a solution for tracing variability from features to PLA by taking these kinds of variations into account. This solution is based on models and traceability between models in order to automate SPL configuration by selecting the variants and realizing the product application. The FPLA modeling framework supports this solution which has been deployed in a software factory. Validation has consisted in putting the solution into practice to develop a product line of power metering management applications for smart grids.

Keywords Traceability modeling · Software product line engineering · Product-line architecture · Variability

J. Díaz (✉)
CITSEM, Technical University of Madrid (UPM) - Universidad Politécnica de Madrid, Ctra. Valencia Km. 7, 28031 Madrid, Spain
e-mail: yesica.diaz@upm.es

J. Pérez · J. Garbajosa
E.U. Informática - CITSEM, Technical University of Madrid (UPM) - Universidad Politécnica de Madrid, Ctra. Valencia Km. 7, 28031 Madrid, Spain
e-mail: jenifer.perez@eui.upm.es

J. Garbajosa
e-mail: jgs@eui.upm.es

1 Introduction

Traceability defines and maintains relationships between artifacts involved in the software life cycle [2, 20] in both forward and backward directions, e.g., from requirements to code and from code to requirements, respectively. Currently, software systems are continuously undergoing changes due to the competitiveness of the software market and their changing technologies. In software systems with highly volatile requirements, traceability has become a critical issue. Numerous researchers have put their work over past years on traceability from problem space to solution space in traditional software development and evolution [13, 43, 45, 46]. In this regard, today there are still several challenges to be dealt with. Specifically the *Center of Excellence for Software Traceability* identified eight challenges related to the purpose, cost, configuration, confidence, scalability, portability, value, and ubiquity of traceability [19]. This traceability is even more challenging in recent software development paradigms such as software product line engineering (SPLE [14, 44]). In fact, the capability of tracing variability in a family of the products is still a challenge [35], as well as important: “the traceability work that is emerging from product line engineering contexts may have wider applicability to broader traceability reuse” [19].

This paper focuses on the traceability between the artifacts resulting from the SPLE phases *domain analysis* [24] and *product-line architecting* [33]. During the domain analysis phase, *feature models* [23] are usually used to describing requirements in terms of common and variable features of the set of products that make up a SPL. Then, these features are realized and described at architectural level in product-line architecture (*PLA*) *models*. Our approach is somewhat based on the work by Ramesh and Jarke [46] and Pohl et al. [43], but particularizes the traceability definition between *requirements and architecture* in *features and PLA* by taking into account the traceability of variability. This traceability of variability is critical to configure the PLA and realize the products while ensuring that they meet the requirements, i.e., to check that the variability binding performed during the configuration of products satisfies the product requirements.

How variability is specified in feature and PLA models largely determines how variability can be traced. There has been an extensive research on supporting the representation of variability in feature models [5, 9, 23], PLA models [1, 6, 16, 58, 59, 60], as well as those approaches that propose *dedicated variability models* [7, 28, 44]. Based on this state of the art, current SPL traceability mechanisms trace the existing variability in feature models to variations in the PLA. This traceability is usually related to variations in the configuration of architectures as well as in the configuration of *composite components* [29], aka. *subsystems* [22]. These variations are realized through adding or removing components and/or connectors. This means, the configuration of architecture is customized by selecting optional, alternative, or multiple components and their respective connectors. We refer to these kinds of variations as *external variability*.

However, external variability is not enough to completely define all kinds of variations [6] and to trace them from features to PLA [60]. This happens when variations have a lower granularity than the granularity of components (e.g., classes, services, or interfaces that implement functionalities such as logging, database connections, listeners of an event-based architecture, graphical controllers), so that they are materialized inside *simple components*—or *non-composite components*. In these components, in which variability occurs inside, part of their functionality is common to the SPL and part of their functionality changes depending on the product to be realized. As a result, in order to support this internal variability, it is necessary to specify variations that are internal to components. We refer to this kind of variations as *internal variability*. In addition, this internal variability is especially relevant, but no specific, when describing variability that refers to non-functional features or quality attributes [30], since they may crosscut several components of the PLA. For example, suppose an illustrative example of a SPL for banking systems that consists of a set of core components that offer their functionality to automatic teller machines (ATM) and bank web applications (WebApp). Both ATM and WebApp aim to provide a cost-effective service to bank customers that is convenient, safe, and secure 24-h access for realizing a common set of banking transactions. A few lines of code implementing the functionality regarding quality attributes, such as *availability* or *data encryption*, are necessary. This code is scattered across the components WebApp and ATM, and it has variations in its behavior depending on the specific banking system product by selecting strict or non-strict availability or different encrypting algorithms. Therefore, this internal variability could affect many different products or there could even be conflicting quality attributes (e.g., trade-offs between availability and performance) in different products of the same family. As a result, the absence of traceability that considers internal variability implies that it is not possible to check if the SPL products with internal variability meet the requirements. Therefore, the capability of tracing internal variability is as important as the capability of tracing external variability.

This paper presents a solution to trace variability from features to both external and internal architectural variability. This solution has been constructed using the metamodeling approach, since models automate development tasks and stimulate learning and reasoning capabilities, which is essential for tracing artifacts. Therefore, our solution is constituted by a set of models for describing and tracing PLAs from features. The description of features is supported by the *feature model* [15, 23]. The description of PLAs is supported by a previous work that presents the *Flexible-PLA model* [41] as a solution for specifying both (1) external variability of the architecture configuration and composite components, and (2) internal variability of simple components. Specifically, in this paper, we present a model that supports traceability between features and PLA called *Feature-PLA traceability model*. The Feature-PLA traceability model defines the principles that govern the traceability links between the feature model and the Flexible-PLA model, i.e., the rules that must be met to create links between the two models. These rules assist software engineers in defining both coarse-grained and fine-grained links which trace external and internal variability between features and PLA. The goal of also tracing internal variability—i.e., at fine-grained level—is to reduce error-prone decisions at the time of configuring variability to derive products—from a SPL platform—according to product-specific requirements. The usage of the Feature-PLA traceability model is possible due to the FPLA modeling framework.[1]

We have put the Feature-PLA traceability model into practice in a software factory, in a project for developing a product line of power metering management applications for smart grids. Validation is performed using the case study technique following the guidelines of Runeson and Höst [48] TraceabilityLink for describing case studies. This case study allowed us to obtain evidence of that the Feature-PLA traceability model was effective and helped engineers in the development and configuration of a successful product line in an industry project.

The structure of the paper is as follows: Sect. 2 describes background in which our solution is based on. Section 3 describes the Feature-PLA traceability model. Section 4 presents the case study used to validate the Feature-PLA traceability model, i.e., its viability, effectiveness, and helpfulness in an industry project. This section also discusses about limitations of our solution. Section 5 analyzes related work. Finally, conclusions and further work are presented in Sect. 6.

2 Background

This section describes the required background to detail the contribution of this paper, i.e., the models that the Feature-PLA traceability model traces.

2.1 The feature model

Numerous methods for domain analysis can be found in the literature, although one of the most widely used is the feature-oriented domain analysis (FODA) [5, 23] in which our work is based on. The FODA method introduces the *feature modeling* technique for capturing commonality and variability of SPL in terms of features. This method defines a feature as "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" [23]. Feature modeling is graphically described through the feature diagram notation, which specifies all products of a family through a hierarchical tree-like structure. We use the extended feature metamodel definition proposed by Czarnecki et al. [15] which includes the following concepts:

- A *root feature* modularizes the model in a tree-like structure, in which there is a main root.
- *Solitary features* represent *mandatory* or *optional* characteristics of a software system which can be composed of zero or more solitary features and by zero or more *feature groups*.
- A *feature group* consists of a set of *grouped features* which in turn can be composed of zero or more solitary features and by zero or more feature groups. Feature groups can be *OR* or *XOR*. The first one forces to choose *m* grouped features (being $m \leq$ total number of grouped features). The second one forces to choose only one grouped feature.

Figure 1 exemplifies these concepts through a simple feature model of a family of e-readers. The root feature of the tree is called *e-readers family*. A set of solitary features are hooked to the root, such as *interface* and *connectivity*. The solitary feature *interface* is composed of a XOR feature group that supports customized interfaces, such as *keyboard* or *multi-touch*, whereas the solitary feature *connectivity* is composed of the solitary features *Wi-Fi* and *3G*. The solitary feature *Wi-Fi* is a mandatory feature for all products of the e-readers family, while *3G* is optional.

2.2 The Flexible-PLA model

The Flexible-PLA model [41] is a precise representation for capturing variability as part of PLAs. The main concept underlying Flexible-PLA model is the concept of *Plastic Partial Component* (PPC [41]). The concept of PPC is a solution to completely support the internal variation of architectural components. Therefore, it is a component that part of its behavior corresponds to the core of a SPL and part of its behavior is specific of a product or set of products from that SPL. The other concepts that are common to PLAs, such as components, connector, ports, are specified as it is usually done in common *architecture description languages* [34].

The variability mechanism underlying PPCs is based on the principles of invasive software composition and the combination of two approaches to define software architectures: the *component-based software development* [55] and the *aspect-oriented software development* [25]. The variability of a PPC is specified using *variability points* which hook fragments of code to the PPC known as *variants*, and *weavings* which specify where and when extending the PPCs using the variants. Weavings are defined outside from PPCs and variants so that these PPCs and variants are independent of the weaving or linking context. As a result, variants can be reused and crosscut several PPCs of the PLA. Additionally, PPCs reduce dependences and coupling between components and their variants, and enable easy and cheap (un-)weaving of variants. These advantages have been successfully applied to SPLs [17, 42, 41].

The concepts of the Flexible-PLA model are exemplified by the graphical representation of a PPC called *interface* (see Fig. 2). The PPC *interface* defines a variability

[1] It is available on: https://syst.eui.upm.es/FPLA/home.

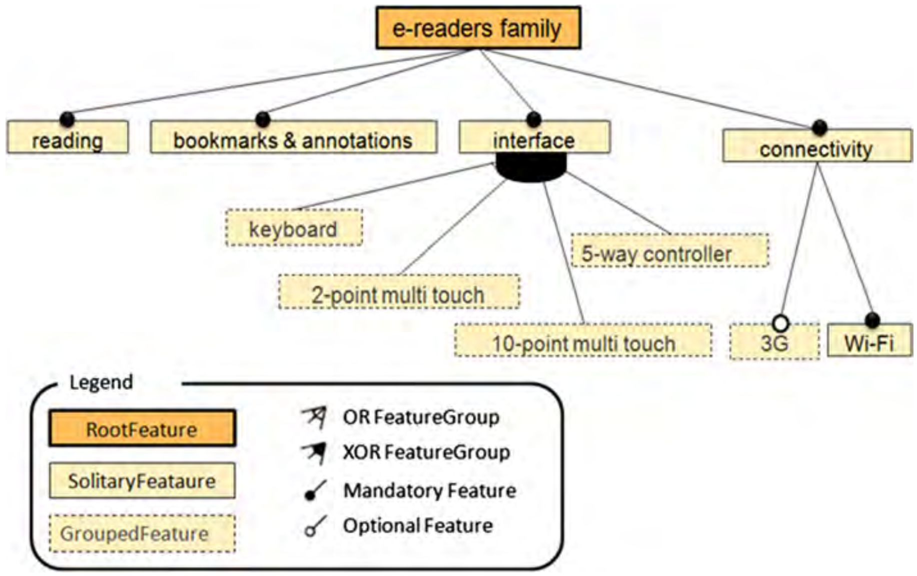


Fig. 1 SPL: feature model

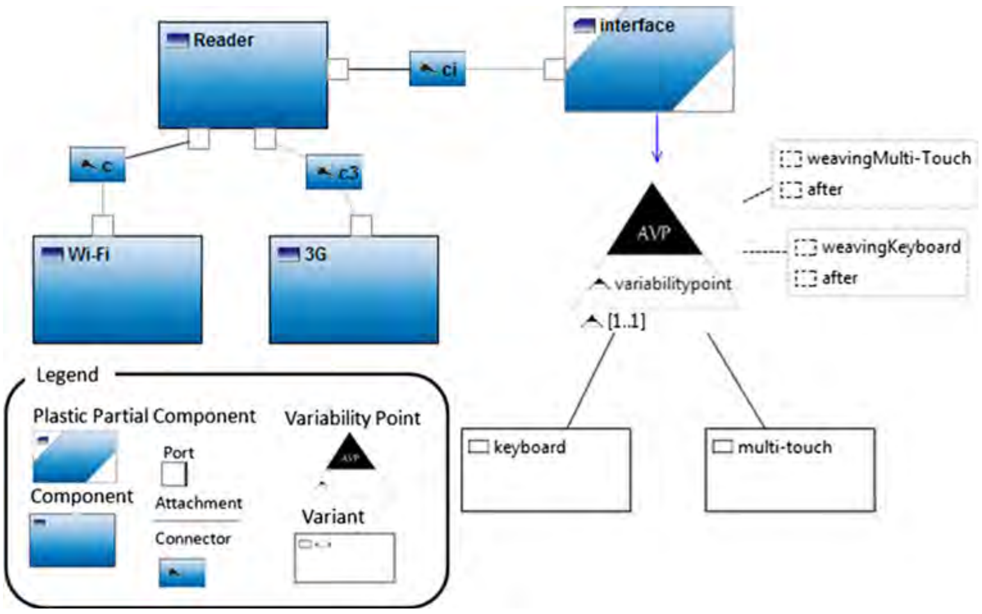


Fig. 2 SPL: Flexible-PLA model

point which hooks the code that implements the variants *keyboard* and *multi-touch*.

3 Feature-PLA traceability model

This section presents the Feature-PLA traceability model as a solution for tracing features to PLA. Tracing artifacts aims to automate development tasks, as well as to stimulate learning and reasoning. Models, traceability between models, and model transformations are the basis to automate development tasks, which is known as model-driven development (MDD [11]). Additionally, models help us to understand complex problems and their potential solutions through abstraction [52] and could stimulate learning and reasoning [53]. For this reason, the proposed solution is based on models, specifically the feature and Flexible-PLA models, as well as the Feature-PLA traceability model that defines traceability between the two first models.

The Feature-PLA traceability model provides modeling primitives to define traceability links, i.e., relationships, between elements belonging to the feature model (see Sect. 2.1) and elements belonging to the Flexible-PLA model (see Sect. 2.2). These relationships are established between the set of feature elements and a set of architecture elements that *satisfy* them (aka. *Satisfaction Links* [46]). Hence, a feature element may define some kind of constraint or goal which may be satisfied by one or more architecture elements, while an architecture element may satisfy one or more feature elements. In this regard, the Feature-PLA traceability model defines the rules that govern the creation of these relationships. These rules are called *linkage rules*.

To be able to use these modeling primitives, it is necessary to define a *domain-specific (modeling) language* (DSL [56]). The next subsections describe (1) a DSL abstract syntax through the definition of the Feature-PLA traceability metamodel, its domain concepts, relationships and rules, (2) a DSL concrete syntax by defining a graphical language representation, and (3) how putting these modeling primitives in practice.

3.1 Abstract syntax: metamodel description

Metamodels describe how models can be specified and establish the properties of models in a precise way. In addition, a metamodel is characterized because it allows the verification of those models that are constructed and conformed to it [12]. The realization of MDD principles is made around a set of OMG standards like MOF [38] which is a meta-metamodel. Specifically, our solution is based on MOF 2.0 and uses UML 2.0 to specify a metamodel which we refer to as Feature-PLA traceability metamodel.

The Feature-PLA traceability metamodel (see Fig. 3) is composed of a set of interrelated metaclasses. These metaclasses define a set of properties and services for each concept considered in the model. On the one hand, metaclasses, their properties, and their relationships describe the structure and the information that is necessary to define traceability links and their linkage rules. On the other hand, the services of metaclasses offer the primitives to develop instances by creating, destroying, adding, or removing elements which are compliant with the constructors of the metamodel.[2] Those constraints that cannot be defined through the use of relationships and their cardinality are specified by the Object Constraint Language (OCL [39]), such those described as textual information in UML notes (see Fig. 3).

The Feature-PLA traceability metamodel is created with the aim of facilitating its integration with general-purpose traceability metamodels, such as the *metamodel for requirements traceability* [27], or the *EML trace* [26]. These models define the concept of traceability link through a metaclass that supports the traceability between any two models. We have reused this metaclass, and hence, traceability links are described by the metaclass *TraceabilityLink* (see Fig. 3). It offers the primitives to instantiate traceability links. The metaclass TraceabilityLink has five attributes (see Fig. 3). Additionally, in order for the user to set the traces between the right elements, it is necessary to define a set of linkage rules that establish the constraints that govern the creation of these traces. To that end, the metaclass TraceabilityLink *defines* five linkage rules (see dashed rectangles labeled from A to E in Fig. 3). Attributes and linkage rules as described below.

The metaclass TraceabilityLink has the attributes *description*, *why*, *who*, *when*, and *satisfacing*. These properties store semantic knowledge about the traceability links. The attribute description keeps a brief description of the link. The attribute why stores the traceability link's rationale. The attributes who and when keep who creates the traceability link and when it is created, respectively. The definition of the attribute satisfacing is based on the work by Ramesh and Jarke [46] who defined a scheme for assigning qualitative degree of satisfaction to links, i.e., a measure of the extent of how long one element affects another. Hauser and Clausing [21] use four categories to relate how design affects quality requirements: *strong positive*, *medium positive*, *medium negative*, and *strong negative*. Positive values measure the degree to which features are satisfied, e.g., a recovery feature to provide response of 100 ms may be considered to be well satisfied, so that 90 and 110 ms response time may be considered to satisfy the feature with different degrees. Negative values may capture trade-offs between features, e.g., a component that satisfies an availability feature may have a strong negative impact on a performance feature. This scheme is incorporated in our traceability model as follows: an element belonging to the Flexible-PLA model may contribute toward *satisfacing* an element belonging to the feature model along these four categories. Thereby, it is possible to assign the values strong positive, medium positive, medium negative, and strong negative to the attribute satisfacing of a traceability link.

The metaclass TraceabilityLink *defines* five linkage rules (see the association relationships between the metaclass TraceabilityLink and the metaclasses LinkageRule {A–E} in Fig. 3). The linkage rules define how relationships can be established, i.e., the rules that restrict which elements belonging to the feature model can be traced to which elements belonging to the Flexible-PLA model. These linkage rules act as constraints that must take variability into account. Variability in the feature model is specified by means of optional solitary features, feature groups, and grouped features. Variability in the Flexible-PLA model is specified by means of optional components and optional connectors, which describe external variability of architecture configuration, as well as variability points and variants, which describe the internal variability of PPCs. These forms of variability constrain the traces that can be defined in such a way that the linkage rules define the following constraints:

Linkage Rule A: A mandatory solitary feature can trace to a component or a PPC.

Linkage Rule B: An optional solitary feature can trace to an optional connector. A feature group can trace to an optional connector.

[2] Most services are omitted to gain readability.

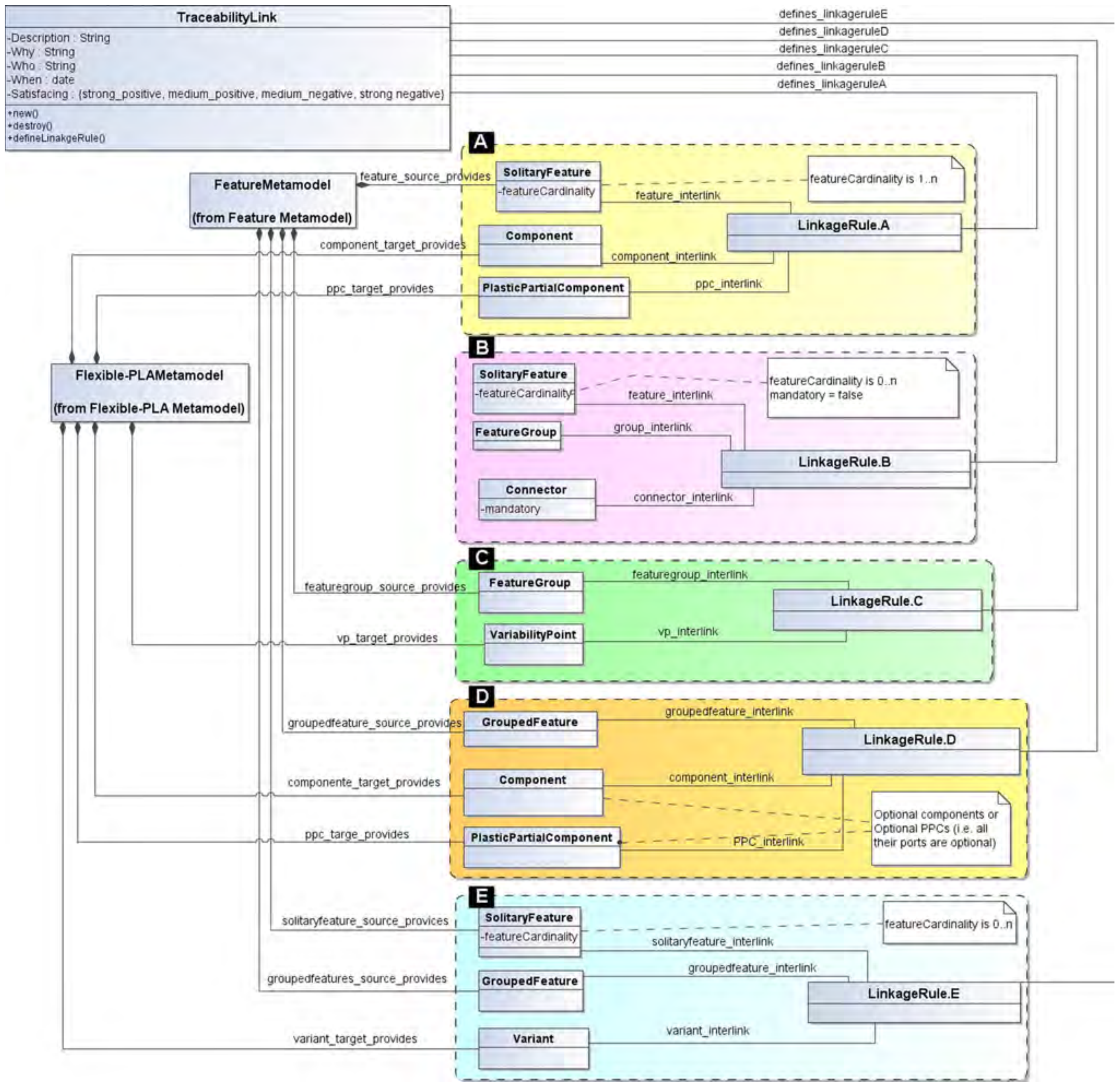


Fig. 3 Feature-PLA traceability metamodel

Linkage Rule C: A feature group can trace to a variability point.

Linkage Rule D: A grouped feature can trace to an optional component or an optional PPC.

Linkage Rule E: An optional solitary feature can trace to a variant. A grouped feature can trace with a variant.

These constraints of the traces are implemented in the Feature-PLA traceability metamodel through five metaclasses to which we refer to as *LinkageRule.{A–E}* (see Fig. 3). These metaclasses define associations with the metaclasses from feature and Flexible-PLA metamodels.[3] As a result, any link between an element from a feature model and an element from a Flexible-PLA model must be compliant with one of these linkage rules. As the linkage rules support external and internal variability, both fine-grained and coarse-grained traceability links can be defined.

[3] The feature metamodel is described in [15], while the Flexible-PLA metamodel is described in [41].

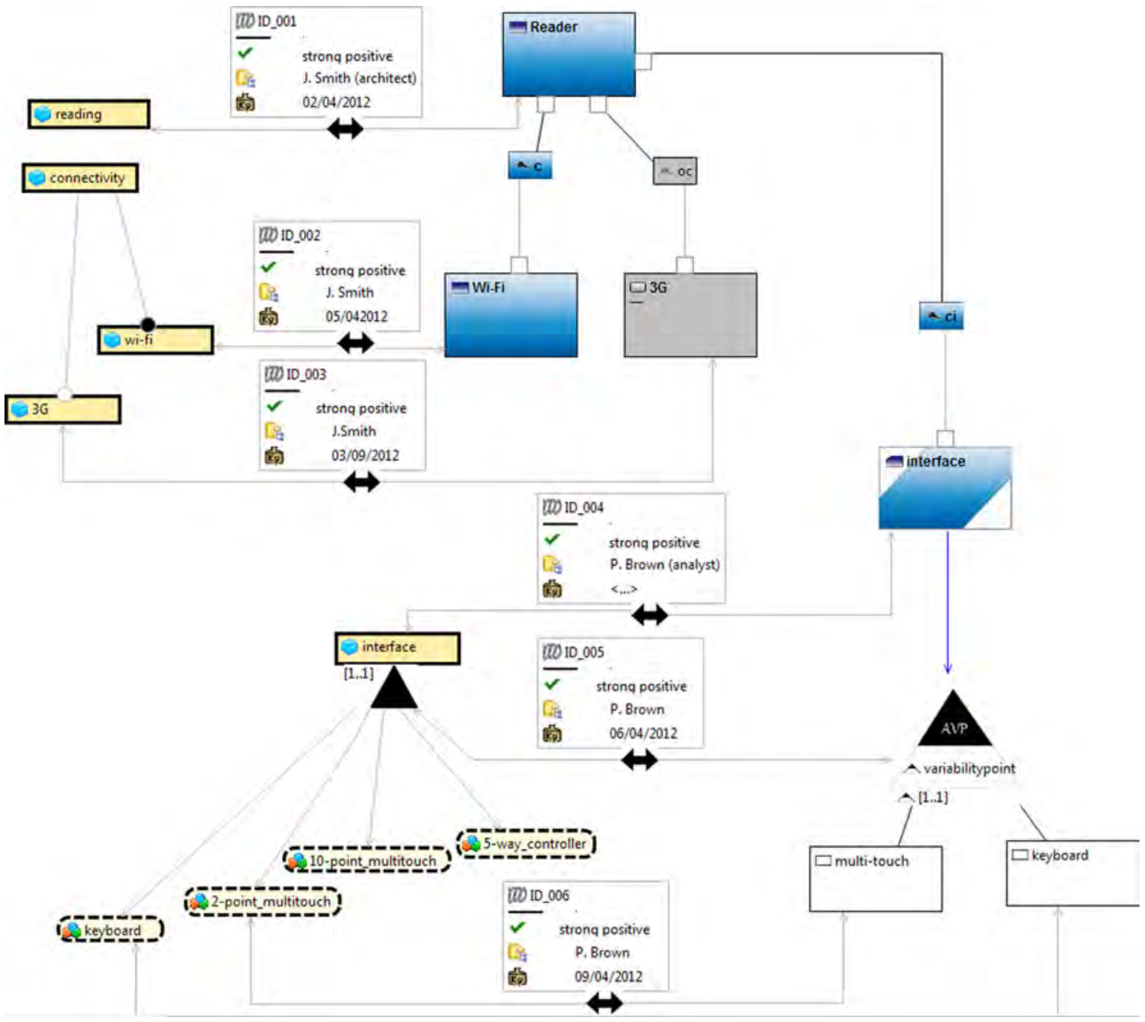
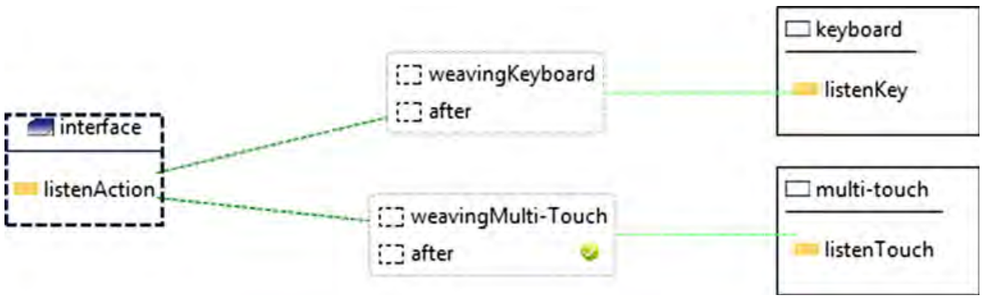


Fig. 4 SPL: Feature-PLA traceability model

Fig. 5 SPL: weaving definition



In this regard, it is necessary to highlight that we decided not to add a new attribute (with the types of linkage rules and define the corresponding OCL constraints) into the metaclass TraceabilityLink in order to preserve the metaclass TraceabilityLink of general-purpose traceability models. In order to realize this typing, we defined five linkage rules through five metaclasses. In this way, we guarantee that the Feature-PLA traceability model can be reused by and integrated in other traceability models.

Finally, it is necessary to highlight that this metamodel conforms to a general-purpose traceability model (see [18]) which is located in an upper layer of the MOF architecture [38] (meta-metamodel layer). In this way, the metaclasses *TraceabilityLink* and *LinkageRule.{A–E}* conform to two meta-metaclasses (*TraceLink* and *LinkageRule*) of this meta-metamodel. The definition of the metaclasses *LinkageRule.{A–E}* allowed us to extend the rationale of the linkage rules.

3.2 Concrete syntax: graphical language description

A graphical modeling language has been defined as this kind of languages is usually more intuitive.

Figure 4 illustrates the Feature-PLA traceability graphical language through an example of a SPL of e-readers. Figure 4 shows six traceability links and their properties—satisfacing, who, and when (see ID_001 to ID_006). The traceability link ID_003 defines a relationship between the optional solitary feature *3G* and the optional component that implements it (e.g., see the properties: strong positive, J.Smith and 03/09/2012). This link traces a variation that is materialized by adding or removing the component *3G* to/from the configuration of the PLA. Therefore, this link traces external variability. The traceability link ID_005 defines a relationship between a point of variability related to the types of interfaces—a feature group—and the variability point that implements it. Finally, the traceability link ID_006 defines a relationship between the grouped feature *2-point multitouch* and the variant that implements it. This link traces a variation that is not materialized by adding or removing a component because of its small size—it is a service called *listenTouch* (see Fig. 5). This variation is materialized by weaving or unweaving the variant *multi-touch* to/from the PPC *interface*, i.e., by injecting (or not) the service *listenTouch* instead, before, or after the execution of the service *listenAction* of the PPC *interface*. Therefore, this link traces internal variability.

3.3 Feature-PLA traceability model in practice

The solution presented in this paper for tracing variability from features to PLA is supported by the FPLA modeling framework. FPLA is an open-source graphical tool that is available for the community as an Eclipse plug-in.[4] The use of the FPLA modeling framework to put this solution into practice is described through a set of activities as follows.

1. SPL domain analysts model the problem space, i.e., specify common and variable features through a feature model.
2. SPL architects model the solution space (PLA), i.e., specify the PLA structural configuration through a Flexible-PLA model.
3. Both domain analysts and architects define the traceability links between a feature model and a Flexible-PLA model, i.e., establish the relationships between elements from these two models through a Feature-PLA traceability model.
4. SPL developers implement and test the components and services of the SPL. The resulting source code is linked to the components specified in the Flexible-PLA model. To do this, Flexible-PLA models provide links to external sources.
5. Product engineers configure specific products through the binding of the variability according to the product needs—product-specific requirements. The FPLA modeling framework allows product engineers to specify this binding (see the *\checkmark mark* in Fig. 5 that selects the multi-touch feature).
6. Product engineers examine the Feature-PLA traceability model to ensure that the variability binding was correctly performed. This means, to check that the binding performed in the PLA meets and satisfies the product-specific requirements.
7. Finally, the FPLA modeling framework automatically binds the variability from PLA to code in order to configure components and generate code for specific products. This means, FPLA automatically generates code skeletons from Flexible-PLA models and composes the code from external sources by model-to-text transformations.

4 Case study

This section aims to provide empirical evidence that validates that the use of the Feature-PLA traceability model is viable in an industry project, as well as effective and helpful for developing and configuring software product lines. Since the goals to be validated are qualitative, we use the case study technique. Case study research is a technique that consists of the investigation of contemporary phenomena in their natural context [61] to search for evidence, gain understanding, or test theories by primarily using qualitative analysis [47].

The case study was conducted in an experimental i-smart software factory (iSSF [31]) which is deployed in the Technical University of Madrid (UPM[5]) and Indra Software Labs.[6] Specifically, the case study was performed within an industrial project on *smart grids* [32] to develop a SPL of a family of *power metering management systems*. The authors of this paper have been involved since 2011 with this particular investigation.

The iSSF is a software engineering research and education setting in close cooperation with the top industrial and research collaborators in Europe. It is a global and distributed software development initiative set up at the

[4] https://syst.eui.upm.es/FPLA/home.

[5] http://www.upm.es/internacional.

[6] http://www.indracompany.com/en.

end of 2011. Indra Software Labs leads this initiative at the corporate level in Spain, in conjunction with UPM, although it is framed into a broader scope that includes other software factories such as that located at the University of Helsinki, University of Eastern Finland, University of Bolzano, and companies such as Tieto and Indra in Spain. This initiative aims to put in practice models and tools that will contribute both toward the implementation of the new processes and methodologies, and the monitoring and tracking of the results.

The iSSF in which the case study has been run, comprises laboratories in two different geographical locations in Madrid (UPM and Indra's factories), equipped with sophisticated computer and monitoring equipment. This equipment facilitates tracking of the project's progress using real-time data from development tools. The iSSF facility continuously runs projects in 16-week cycles. Therefore, it is a suitable setting to deploy, track, and evaluate the applicability of the Feature-PLA traceability model.

Next, the case study is reported according to the guidelines for conducting and reporting case study research in software engineering by Runeson and Höst [48]. The goal of reporting a case study is twofold: to communicate the findings of a study, and to work as a source of information for judging the quality of the study. With this twofold goal, the reporting of the case study is described as follows.

4.1 Case study design

This section describes the case study, the research objective and questions, the data collection procedure, analysis and validation procedures, and the subjects participating in the case study.

4.1.1 Research objective and questions

Evidence of the viability of the Feature-PLA traceability model can be obtained by putting the model into practice in a real-life setting. Therefore, the research objective focuses on evaluating the effectiveness of the Feature-PLA traceability model as well as the helpfulness that it could provide SPL engineers. The criterion to validate the achievement of the objective is defined as the capabilities to (1) trace both coarse-grained and fine-grained variability in order to satisfy the traceability of most common kinds of variations, and (2) provide knowledge to help SPL engineers at the time of configuring the different products that make up a SPL, i.e., when variability has to be bound according to the product-specific requirements. Hence, the research questions to be answered through the case study analysis can be formulated as follows:

RQ_1: Are Feature-PLA traceability modeling primitives effective in providing SPL engineers the means for specifying traceability for most common kinds of variations that they define on their product family?

RQ_2: Do Feature-PLA traceability models assist and guide SPL engineers at the time of configuring products of their product family?

Research question RQ_1 aims to find out if the Feature-PLA traceability model effectively provides SPL engineers with mechanisms to trace all types of variations that they commonly define, which includes the capability of tracing those features that are realized through external variations of the PLA configuration (i.e., traceability of external variability) as well as the traceability of those features that are realized through internal variations of the components (i.e., traceability of internal variability). The *level of effectiveness* is a dependent variable, i.e., a variable of interest for being analyzed and evaluated. It is measured in terms of the percentage of variations existing in the domain of the SPL under study (i.e., variations in the feature model) that can be traced by the modeling primitives provided by the Feature-PLA traceability model. The potential independent variables that might have an influence on the dependent variable are the *project size*, the *SPL domain*, *the complexity of Feature and PLA models*, and the total *number of variations* identified in the product family.

Research question RQ_2 aims to find out if the knowledge stored in Feature-PLA traceability models is really helpful for SPL engineers at the time of configuring the products of their product family. In this regard, helpfulness is defined in this paper as the facilities provided for engineers to enable product configuration (i.e., selection of variants and construction of product applications). As a dependent variable, the *level of helpfulness to configure products* is qualitatively estimated by analyzing questions asked to the SPL engineers involved in the cases study through a set of interviews. These questions asked the SPL engineers about specific situations in which the assistance of Feature-PLA traceability models to configure products was analyzed. Hence, the SPL engineers were asked if Feature-PLA traceability models helped them when trying to bind variability to configure specific products from the product family, while ensuring the product requirements compliance. The potential independent variables which might have an influence on the dependent variable are the *engineers experience*, the *project size*, the *PLA complexity*, the *misinterpretation of interview questions*, and the total *number of variations* identified in the product family.

It is necessary to mention that it is in the nature of case studies that independent variables cannot be controlled [47]. This and other potential threats to validity are discussed in Sect. 4.2.3.

4.1.2 Data collection procedure

In the case study, we have gathered both quantitative and qualitative data. The collection methods which have been used are the following:

- Observation. Two observers attended project meetings and visited the team twice a week. They took notes from these meetings and, thanks to the iSSF technologies, meetings were video recorded, transcribed, and analyzed using the *constant comparison method* as described in [57].
- Questionnaire and interview. Stakeholders were interviewed following a questionnaire[7] open to the discussion. These interviews were video recorded, transcribed, and analyzed using the constant comparison method.
- Archival data. In addition to the storage of the video recordings, the information about the project was collected in Redmine[8].
- Analysis of work artifacts. Feature-PLA traceability models generated with the FPLA modeling framework were gathered.

4.1.3 Analysis and validity procedure

In this case study, both quantitative and qualitative analysis were used to examine the data gathered. For quantitative data, this case study uses analysis of descriptive statistics. For qualitative data, the procedure to explore the *chain of evidence* [47] from collected data is described as follows: Interviews and meetings are recorded, transcribed, grouped by quotes and coded. Coding means that parts of the text are given a code representing a certain topic of interest—one code is usually assigned to many pieces of text, and one piece of text can be assigned more than one code and codes can form a hierarchy of codes and sub-codes [47]. The coded material is enriched with comments and reflections (i.e., *memos*). From this material, it is possible to identify evidence that answers the research questions.

As data gathered in case studies is mainly qualitative [47], and it is typically less precise than quantitative data, it is important to use *triangulation* to increase the precision of the study. There are several types of triangulation [54]: (1) *methodological triangulation*, i.e., the use of different methods to measure the same concern; (2) *data source triangulation*, i.e., the use of multiple data sources at potentially different occasions; and (3) *observer triangulation*, i.e., the use of more than one observer in the case study [57]. In order to increase the precision of the qualitative analysis and its obtained results, the three types of triangulation were used in this case study. Methodological triangulation was performed through interviews, observations, and the analysis of archival data. Data source triangulation was performed by interviewing the SPL engineers both separately and together. Finally, observer triangulation was applied by replicating specific data collection sessions by two different observers.

4.1.4 Case study description

The case study consists of a project to model, design, and implement a "power quality monitoring and a remote control and smart metering" platform. It is part of two

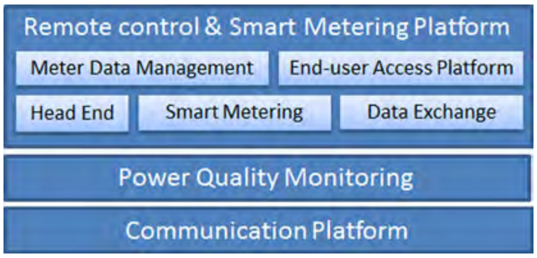
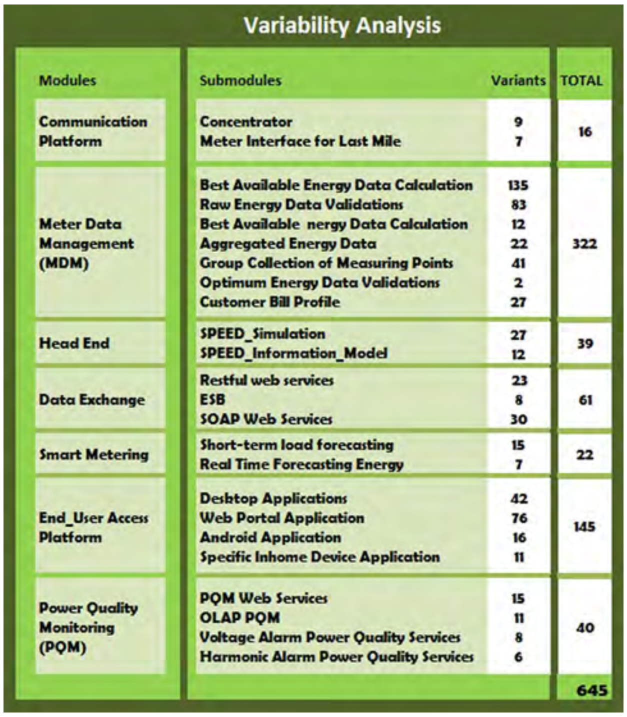


Fig. 6 Modules of the power quality monitoring and the remote control and smart metering platform



Variability Analysis

Modules	Submodules	Variants	TOTAL
Communication Platform	Concentrator	9	16
	Meter Interface for Last Mile	7	
Meter Data Management (MDM)	Best Available Energy Data Calculation	135	322
	Raw Energy Data Validations	83	
	Best Available nergy Data Calculation	12	
	Aggregated Energy Data	22	
	Group Collection of Measuring Points	41	
	Optimum Energy Data Validations	2	
	Customer Bill Profile	27	
Head End	SPEED_Simulation	27	39
	SPEED_Information_Model	12	
Data Exchange	Restful web services	23	61
	ESB	8	
	SOAP Web Services	30	
Smart Metering	Short-term load forecasting	15	22
	Real Time Forecasting Energy	7	
End_User Access Platform	Desktop Applications	42	145
	Web Portal Application	76	
	Android Application	16	
	Specific Inhome Device Application	11	
Power Quality Monitoring (PQM)	PQM Web Services	15	40
	OLAP PQM	11	
	Voltage Alarm Power Quality Services	8	
	Harmonic Alarm Power Quality Services	6	
			645

Fig. 7 Power quality monitoring and the remote control and smart metering platform variability analysis

[7] The script of the interviews is available on https://www.surveymonkey.com/s/TSYCCN6.

[8] Redmine is web-based project management and bug-tracking tool http://www.redmine.org/.

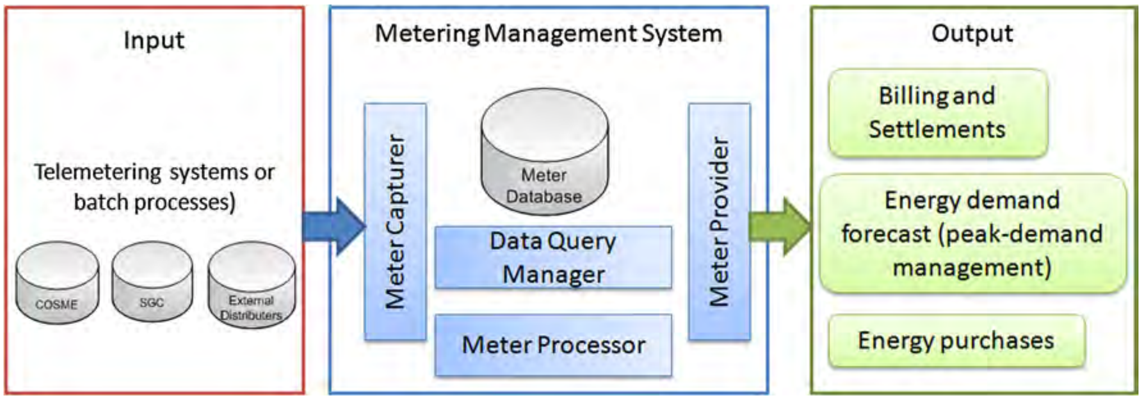


Fig. 8 Metering management system—an overview and interfaces with external systems

larger ITEA2 projects called IMPONET[9] (127 man years) and NEMO&CODED[10] (112 man years), and a third national project called ENERGOS[11] (24.3 million Euros). These three projects focused on supporting complex and advanced requirements in energy management, specifically in electric power networks that are conceptualized as smart grids [32]. Smart grids are composed of an aggregation of a broad range of energy resources, from large generating systems (traditional sources, e.g., nuclear power plants, hydro power plants) to smaller generating systems (called microsources, e.g., small solar farms, distributed wind generators), operating as a single system providing both power and heat [32]. Smart grids promote the integration of renewable energy resources and their distributed, open, and self-controlled nature.

The power quality monitoring and the remote control and smart metering platform is a software intensive software composed of a set of coarse-grained modules: communication platform, power quality monitoring, meter data management, end-user access platform, head end, smart metering, and data exchange (see Fig. 6). At this coarse-grained level, the platform presents variability related to the optionality of the module *power quality monitoring* which depends on the grid, i.e., if the grid requires guarantee only the power supply or also power quality. Each one of these modules has multiple levels of decomposition and variability with different levels of detail that are briefly described below. This is why this project was envisioned as a SPL that allows configuring the platform depending on the smart grid requirements.

A representative example of the multiple levels of variability is the *end-user access platform*. This module is configurable by considering the following variants: type of GUI, end-user, and data. Regarding the type of GUI, the *end-user access platform* was designed to support Web application, desktop application, android application, as well as specific in-home device's application. Regarding the end-user, the functionalities and the information that are provided by the access platform to the end-users vary in the case of a distributor, a retailer, or a customer. This means that the information provided by the *end-user access platform* is variable depending on the end-user and the end-user requirements. Finally, the information that is shown in the GUI and the technologies used to display that information are variable depending on whether the data are provided in real time or using historical data.

Other examples of multiple levels of variability are the *smart metering* and the *power quality monitoring* modules. The first one implements a set of forecasting algorithms that vary depending on the energy data used, for the next 24 or 48 h, or whether it is calculated using real-time energy data, historical energy data from the database, or both of them. The second one implements a set of power quality algorithms that can be configured in order to provide a variety of information, such as events, disturbances, alarms control. Finally, the *communication platform* implements a data distribution service (DDS [37]) based on the publication-subscription paradigm. This module is in turn a source of internal variability that crosscuts the other modules. Hence, DDS defines domains, partitions, and topics in order to specify different data space and organize the flow of data. The subscription to the topics is variable depending on, for example, the events or alarms to be controlled.

In order to illustrate the complexity of the system, and in particular the level of variability, the platform has more than 600 variants (see Fig. 7). In this paper, we specifically

[9] Intelligent Monitoring of Power NETworks http://www.itea2.org/project/index/view?project=10032.

[10] NEtworked MOnitoring & COntrol, Diagnostic for Electrical Distribution http://www.itea2.org/project/index/view?project=1131.

[11] Technologies for automated and intelligent management of power distribution networks of the future http://www.indracompany.com/sostenibilidad-e-innovacion/proyectos-innovacion/energos-technologies-for-automated-and-intelligent-.

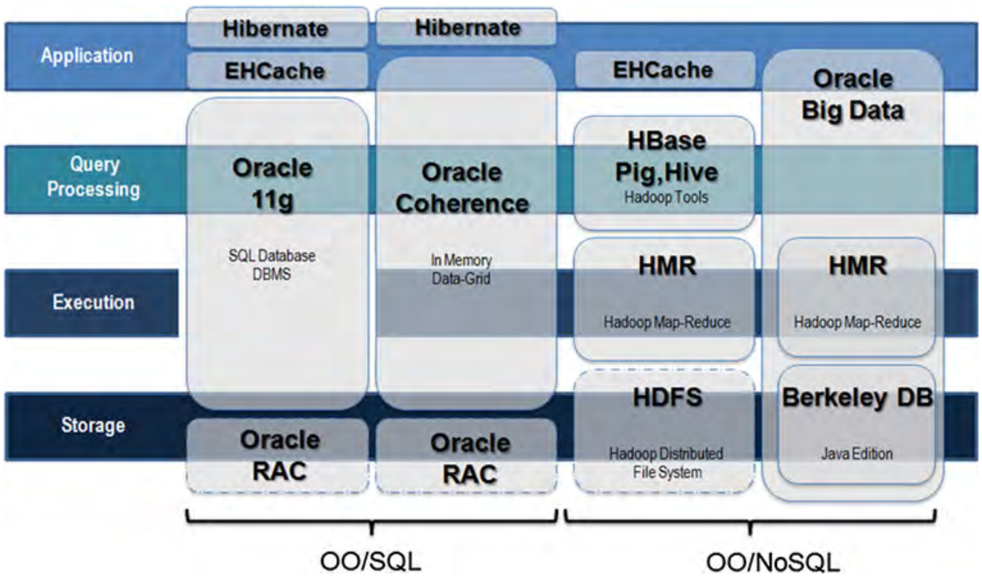


Fig. 9 OPTIMETER SPL—evaluation of large data storing technologies for metering management systems

report the part of the SPL that develops a family of power metering management systems for smart grids, i.e., the *meter data management* module. This is due to space and understandability reasons. We refer to this part of the SPL as *OPTIMETER SPL*.

OPTIMETER SPL focuses on the development of a family of power metering management systems for smart grids (see the central box of Fig. 8). A power metering management system captures and manages meter data from a large number of distributed energy resources. It validates, stores, and processes these data, and provides them to external systems. Figure 8 shows an overview of a metering management system and its interaction with external systems to capture and provide meter data. The overview of the system functionality is as follows:

1. Meter capturing. This involves integrating all meter capturing processes (see *meter capturer* in Fig. 8) which are currently being supported by telemetering systems and batch processes that collect measurements at substations (see box *Input* in Fig. 8). The purpose is to have a single database with the energy metering data.
2. Meter processing. This includes three operations: the validation of meter data according to an established validation formula, the calculation of the optimal vector for a measuring point for a type and period of energy data, and the estimation of energy data according to a established estimation formula (see *meter processor* in Fig. 8).
3. Meter providing. This involves defining the interface (see *meter provider* in Fig. 8) with client information systems, such as billing and settlements, energy demand forecast, and energy purchases, to exchange data with them (see box *output* in Fig. 8).

Data processing should be done in real time. To do this, it is necessary to account for performance when loading the large amounts of energy data coming from the meter capturing processes as well as performance when querying these data. The OPTIMETER SPL aims to provide a family of systems, each of which is intended to support the different data storing technologies shown in Fig. 9. The objective is to carry out various *proof of concept* of large data storing technologies to evaluate their performance. Therefore, the data storing technology is a variability point.

Meter providing should be available 24/7. Metering management systems should guarantee availability 24 h 7 days per week of their core functionality to the external systems. Several applications require to have strict 24/7 availability, while others permit a weaker, non-strict availability. Strict availability must provide recovery and repair in milliseconds, whereas non-strict availability is less available and cheaper. Therefore, the strictness of availability is another variability point.

The OPTIMETER SPL is being iteratively and incrementally developed in the iSSF in Scrum subprojects [50] of 8 iterations, aka. sprints (1sprint = 2 weeks). This case study focuses on two of these Scrum subprojects which we refer to as *Optimeter I* and *Optimeter II*. Optimeter I consisted of the development of the OPTIMETER SPL platform from which a set of metering management system applications can be efficiently developed and produced (*domain engineering* [44]). Optimeter II consisted of the

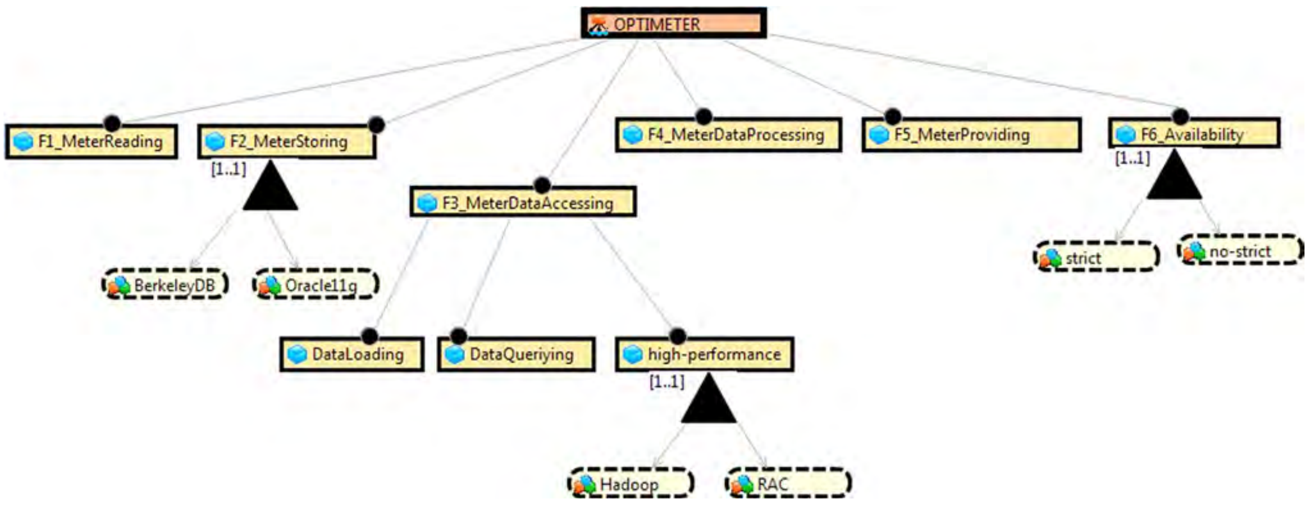


Fig. 10 Optimeter I—feature model

development of two of these product applications (*application engineering* [44]): a metering management application running over the *Berkeley* database[12] and *Hadoop* clustering[13] with strict availability, and a metering management application running over the *Oracle 11g* database[14] and *Oracle Real Application Clusters (RAC)*[15] with non-strict availability (see Fig. 9).

4.1.5 *Subject description*

In total, 10 people participated in Optimeter I and II: four analysts/developers, two product owners, one scrum master (who performs both the tasks of the Scrum master and of a part-time architect), and one full-time architect. During the domain engineering—i.e., Optimeter I—the people involved in the project are referred in this case study as *SPL engineers*, while during the application engineering—i.e., Optimeter II—the peopled are referred as *product engineers*. It is necessary to highlight that the engineers involved in Optimeter I and II are not the same. Finally, two observers had access to all project information and collaborated directly with product owners and fellow team members.

[12] Oracle Berkeley DB is a high-performance embeddable database providing Java Object and Key/Value storage (NoSQL). http://www.oracle.com/technetwork/products/berkeleydb/.

[13] Apache Hadoop is a framework for running applications on large cluster built of commodity hardware. http://hadoop.apache.org/.

[14] Object-relational database management system. http://www.oracle.com/technetwork/database/.

[15] Software for clustering and high availability in Oracle db environments. http://www.oracle.com/technetwork/products/clustering/.

4.2 Results

This section describes the execution, analysis, and interpretation of the results from the case study execution, as well as the evaluation of its validity.

4.2.1 *Case study execution*

This section describes the execution of Optimeter I first, and then the execution of Optimeter II. These executions has been performed following the activities presented in Sect. 3.3. The models resulting from these activities have been captured through snapshots from the FPLA modeling framework.

The first activity was feature domain analysis. Figure 10 shows the feature model that represents the features that OPTIMETER SPL must meet. The feature model has three points of variability—feature groups—that implement different data storing technologies (database and clustering) and different degrees of availability. The feature model is described in detail as follows:

- F1_Meter Reading (see Fig. 10) consists of reading metering data associated with different energy resources, periods (quarterly, hourly, daily, and monthly), and intervals.
- F2_Meter Storing (see Fig. 10) consists of a large data store. There are two mutually exclusive alternative variations: one variant is Berkeley DB and the other variant is Oracle 11g (see the grouped features *BerkelyDB* and *Oracle11g* in Fig. 10).
- F3_Meter Data Accessing (see Fig. 10) consists of initial data loading of historical metering data of 1 month and querying of these data. Both loading and querying require to leverage high performance through

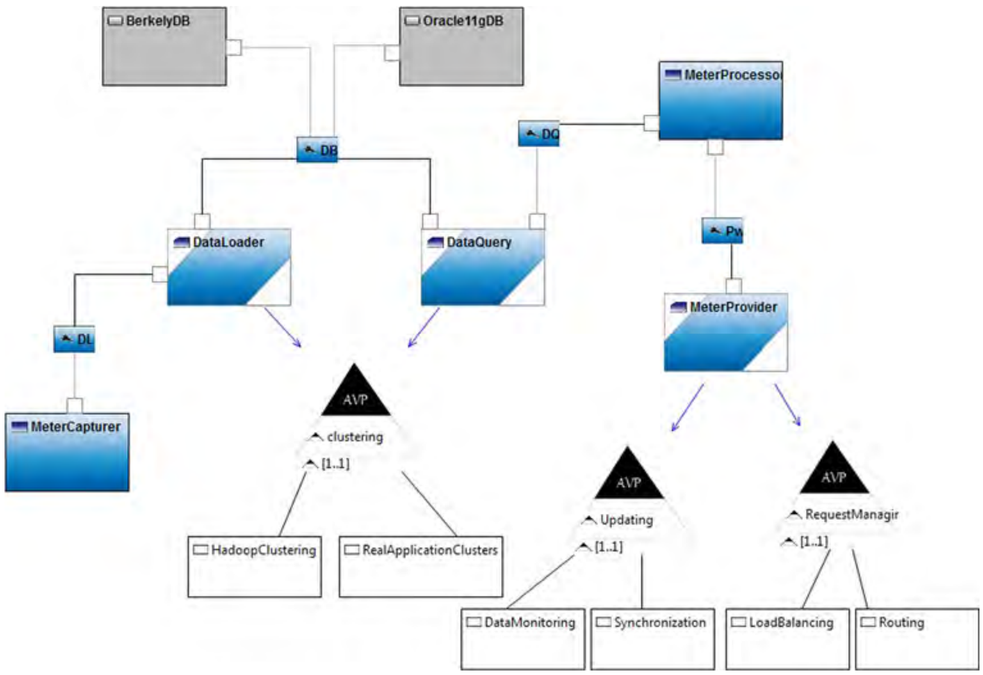


Fig. 11 Optimeter I—Flexible-PLA model

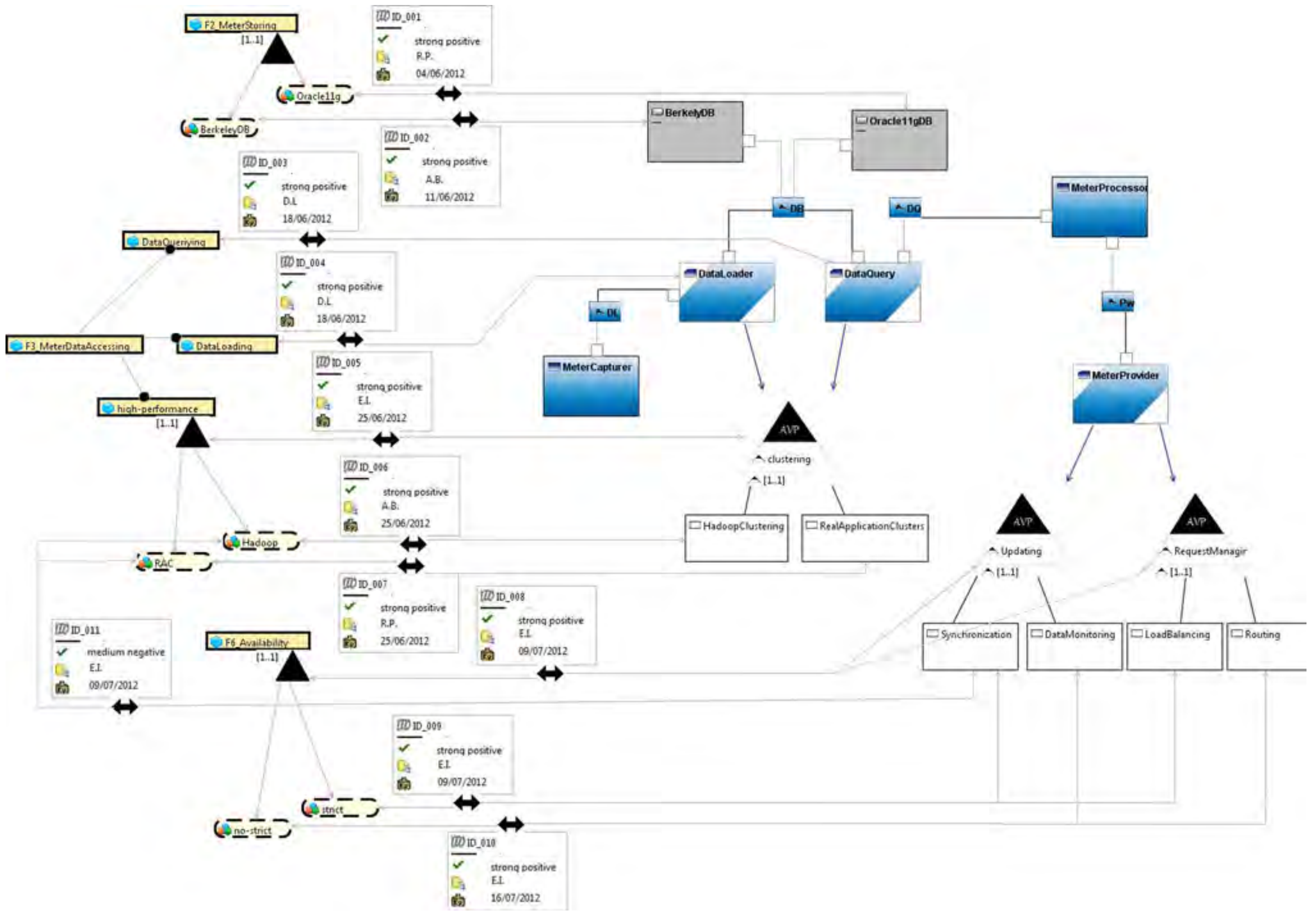
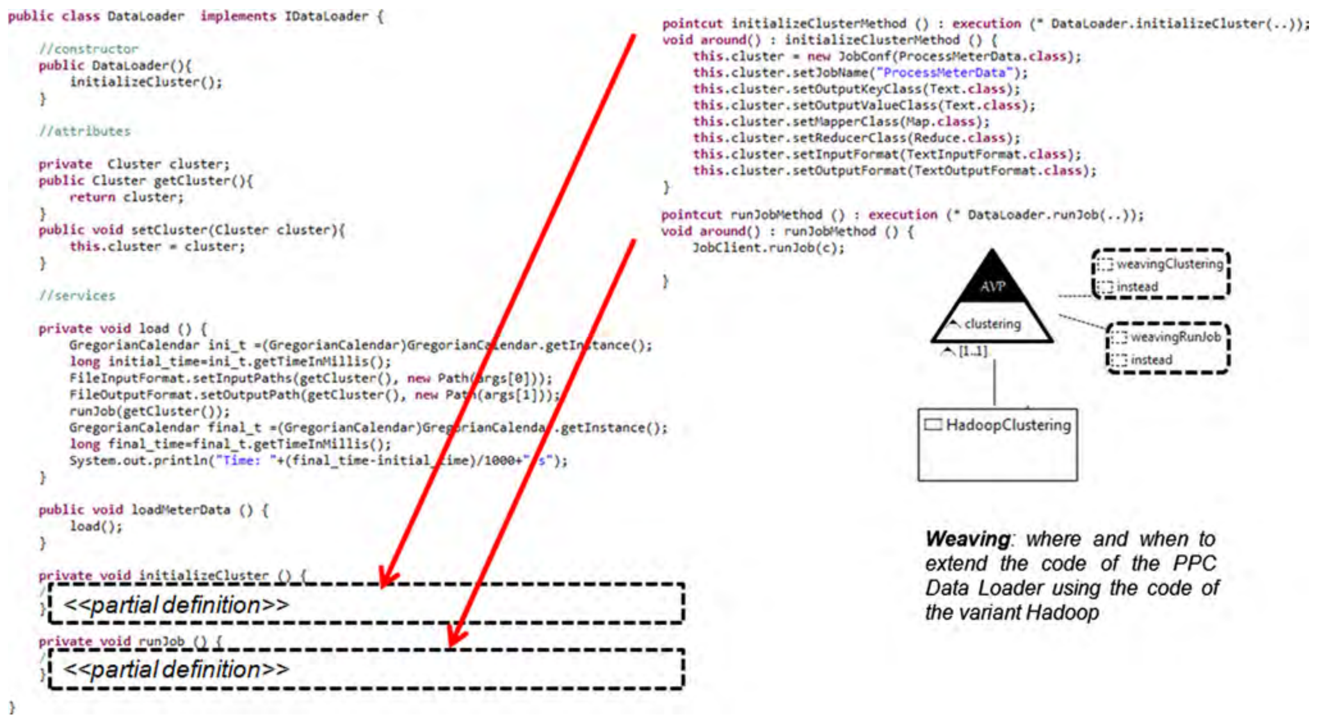


Fig. 12 Optimeter I—Feature-PLA traceability model



```java
public class DataLoader  implements IDataLoader {

    //constructor
    public DataLoader(){
        initializeCluster();
    }

    //attributes

    private  Cluster cluster;
    public Cluster getCluster(){
        return cluster;
    }
    public void setCluster(Cluster cluster){
        this.cluster = cluster;
    }

    //services

    private void load () {
        GregorianCalendar ini_t =(GregorianCalendar)GregorianCalendar.getInstance();
        long initial_time=ini_t.getTimeInMillis();
        FileInputFormat.setInputPaths(getCluster(), new Path(args[0]));
        FileOutputFormat.setOutputPath(getCluster(), new Path(args[1]));
        runJob(getCluster());
        GregorianCalendar final_t =(GregorianCalendar)GregorianCalendar.getInstance();
        long final_time=final_t.getTimeInMillis();
        System.out.println("Time: "+(final_time-initial_time)/1000+" s");
    }

    public void loadMeterData () {
        load();
    }

    private void initializeCluster () {
    <<partial definition>>
    }
    private void runJob () {
    <<partial definition>>
    }
}
```

```java
pointcut initializeClusterMethod () : execution (* DataLoader.initializeCluster(..));
void around() : initializeClusterMethod () {
    this.cluster = new JobConf(ProcessMeterData.class);
    this.cluster.setJobName("ProcessMeterData");
    this.cluster.setOutputKeyClass(Text.class);
    this.cluster.setOutputValueClass(Text.class);
    this.cluster.setMapperClass(Map.class);
    this.cluster.setReducerClass(Reduce.class);
    this.cluster.setInputFormat(TextInputFormat.class);
    this.cluster.setOutputFormat(TextOutputFormat.class);
}

pointcut runJobMethod () : execution (* DataLoader.runJob(..));
void around() : runJobMethod () {
    JobClient.runJob(c);

}
```

***Weaving**: where and when to extend the code of the PPC Data Loader using the code of the variant Hadoop*

Fig. 13 Optimeter I—PPC DataLoader

the use of clustering technologies. There are two mutually exclusive alternative variations: one variant is Hadoop clustering over Berkeley DB and the other variant is RAC over Oracle 11g (see the grouped features *Hadoop* and *RAC* in Fig. 10).

- F4_Meter Data Processing (see Fig. 10) consists of the algorithms for validating raw and optimal data, as well as calculating the optimal vector (integrated processing) of raw and optimal data, i.e., the energy data for a specific origin, period, and date is retrieved, and the system adds data to obtain the energy data of the next period.
- F5_Meter Data Providing (see Fig. 10) consists of an interface that provides metering data query to external systems.
- F6_Availability (see Fig. 10). It ensures availability of metering data 24 h 7 days per week. There are two mutually exclusive alternative variations: One variant implements strict availability and the other variant implements non-strict availability (see the grouped features *strict* and *non-strict* in Fig. 10).

The second activity was product-line architecting. Regarding availability, various architectural tactics are proposed in the literature [8, 51]. The SPL engineers selected *active redundancy* and *passive redundancy* tactics to implement strict and non-strict availability, respectively. These tactics are briefly described as follows.

- The tactic active redundancy is based on a “configuration wherein all of the nodes (active or redundant spare) in a protection group receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s)” [51]. Therefore, from the architectural view, this tactic requires (1) a *load balancer* for all nodes—active and redundant nodes—to process identical inputs, and (2) a *synchronizer* in order for the active and redundant nodes to maintain an identical state. If there is a failure, the repair occurs on time as the redundant spare has an identical state to the active node. The cost of this tactic is high due to the cost of synchronization between redundant spare and active node(s).
- The tactic passive redundancy is based on a “configuration wherein only the active members of the protection group process input traffic, with the redundant spare(s) receiving periodic state updates” [51]. Therefore, from the architectural point of view, this tactic requires (1) a *router* to ensure that only the active node process all the inputs, as well as to change the route to the redundant node(s) when there is a failure, and (2) a *periodic data controller* in order for active and redundant node(s) to maintain periodic state updates. If there is a failure, the router selects a redundant spare after checking the state update. This

tactic achieves a balance between the more highly available but more complex active redundancy tactic and the less available but significantly less complex spare tactic.

The PLA resulting is shown in Fig. 11 and described as follows. The feature F1 is implemented by the component *MeterCapturer*, which reads text files of metering data associated to different energy resources, periods (quarterly, hourly, daily, and monthly), and intervals, and processes the previously read data to form key/value pairs. The variability of the feature F2 is implemented by the optional components *BerkeleyDB* and *Oracle11g*. The feature F3, and specifically the subfeatures DataLoading and DataQuerying are implemented by the PPCs *DataLoader* and *DataQuery*, respectively. The architects took advantage of the PPC's variability mechanism to specify the variability of the feature *high performance* as internal variability. This variability is internal to the PPCs *DataLoader* and *DataQuery*, i.e., the variability crosscut these two PPCs. Hence, both PPCs implement the variability of performance through the variability point *clustering* and the variants *HadoopClustering* and *RealApplicationClusters* (see Fig. 11). These variants implement the operations for clustering and distributing work around a cluster to improve the data accessing performance (for loading and querying). Figure 13 shows an extract of the PPC *DataLoader* code and how internal variability works. Specifically, the figure shows how the code of the variant *HadoopClustering* is linked to the code of the PPC *DataLoader* through the weavings. The feature F4 is implemented by the component *MeterProcessor*, which implements the algorithms for validating metering data and calculating optimal vectors. Finally, the feature F5 is implemented by the PPC *MeterProvider*. Again, the architects took advantage of the PPC's variability mechanism to specify the variability of the feature F6 as internal variability to the PPC *MeterProvider*. This PPC implements the variability of availability through the variability points *Updating* and *RequestManaging*, and the variants *DataMonitoring*, *Synchronization*, *LoadBalancing*, and *Routing*.

The third activity of the case study execution was the definition of traceability links between the Optimeter feature model (see Fig. 10) and the Optimeter Flexible-PLA model (see Fig. 11). The resulting Feature-PLA traceability model is described as follows (see Fig. 12): The links *ID_001* and *ID_002* trace the grouped features *BerkelyDB* and *Oracle11g* to the optional component *BerkeleyDB* and *Oracle11gDB*, respectively. The links *ID_001* and *ID_002* trace the features *DataQueriying* and *DataLoading* to the PPCs *DataLoader* and *DataQuery*, respectively. The feature group that implements the variability of performance is traced to the variability point *clustering* through the link *ID_005*. The links *ID_006* and *ID_007* trace the grouped features *Hadoop* and *RAC* to the variants *HadoopClustering* and *RealApplicationClusters*, respectively. The feature group that implements the variability of availability is traced to the variability points *Updating* and *RequestManaging* through the link *ID_008*. The link *ID_009* traces the grouped feature *strict* to the variants *Synchronization* and *LoadBalancing*. The link *ID_010* traces the grouped feature *non-strict* to the variants *DataMonitoring* and *Routing*. All these traceability links store semantic knowledge. To gain readability, Fig. 12 only shows the attributes *satisfacing*, *who*, and *when*. The value of the attribute satisfacing from all of these links is *strong positive*. This means that the architectural elements—components, PPCs, variants—involved in the links fully satisfy the expected functionality of the features also involved in the links. Finally, the link *ID_011* traces the variant *Synchronization* to the feature *high performance*. This link shows the value *medium negative* for the attribute satisfacing, which means that the synchronization may negatively affect to the performance.

Once the features, the PLA, and the traceability links were described and modeled by the SPL engineers, the following activities were the implementation and the testing (see the fourth activity in Sect. 3.3). The resulting source code (such as the code shown in Fig. 13) is also linked to the components specified in the Feature-PLA traceability model. All these activities comprise a typical domain engineering process in which the commonality and the variability of a SPL is defined and realized [44]. The result is a common structure—the OPTIMETER SPL platform—from which a set of derivative products—metering management system applications—can be efficiently developed and produced.

Next, Optimeter II started. Each one of the two product owners involved in the case study selected to implement two different products:

- A metering management system running over Berkeley DB and Hadoop, which has to be strictly available 24/7 (*product A*).
- A metering management system running over Oracle 11g DB and RAC, which has to be available 24/7 but it is possible to relax this restriction (*product B*).

At this time, the product engineers configured specific products according to the products specifications that the owners expected to get (see the fifth activity in Sect. 3.3). This means that the product engineers bound the variability. To do this, the product engineers examined the Feature-PLA traceability model to ensure that the binding was correctly performed according to the products specifications. Hence, the product engineers, by means of the link *ID_009* in Fig. 12, checked that the configuration of the product A requires the binding of the variants *Synchronization* and *LoadBalancing* in order to meet strict availability. They also checked, by means of the link *ID_011* in Fig. 12 that the variant *Synchronization* could affect the required high performance. Finally, the product engineers checked, by means of the link *ID_010* in Fig. 12, that the configuration of the product B should bind the variants *DataMonitoring* and *Routing* which implement a variation less available but that does not jeopardize performance.

After selecting the specific variants for the products A and B, the last activity (see the sixth activity in Sect. 3.3) was performed as follows. This activity consisted of the generation of the code for the products A and B, i.e., the binding of the variability at the code-level. To do this, the product engineers used the FPLA modeling framework to automatically generate the code for each one of these two products. Hence, for the product A, the weavings that insert the code of the variant *HadoopClustering* into the PPC *DataLoader* were automatically generated (see Fig. 13). Similarly, for the product B, the weavings that insert the code of the variant *RealApplicationClusters* into the PPC *DataLoader* were automatically generated. In this way, the PPC *DataLoader* can be easily configured to support Hadoop clustering as shown in Fig. 13, or to support Real Application Clusters.

The development of these two projects provided the necessary data to conduct the case study analysis and interpretation.

4.2.2 Analysis and interpretation

Quantitative and qualitative analysis were used to examine the data gathered during the case study. The data collected consisted of the models resulting from the projects (see Figs. 10, 11, 12), archival data from Redmine, as well as the questionnaires and interviews performed with the SPL and product engineers. The analysis of these data has permitted to find evidence to answer each one of the research questions:

> RQ_1: Are Feature-PLA traceability modeling primitives effective in providing SPL engineers the means for specifying traceability for most common kinds of variations that they define on their product family?

The evidence to answer RQ_1 is explored through descriptive statistics that measures the number of variations of interest for the SPL engineers that they were able to trace by the modeling primitives provided by the Feature-PLA traceability model. The number of points of variability is three—data storage, clustering, and availability—with a total of six variants—BerkeleyDB and Oracle11g for data storing, Hadoop and RAC for clustering, and finally strict and non-strict availability.

The traceability of the variability for data storing was well supported through links between grouped features and optional components (see the links ID_001 and ID_002 in Fig. 12). As the architects took advantage of the PPC's variability mechanism to specify internal variability of components—specifically to specify the variability of clustering and availability—they required the capability of tracing this variability which is internal to one or more components. Hence, the SPL engineers were able to trace the variants Hadoop and RAC to the architectural elements that implement these two different clustering technologies through links between grouped features and variants (see the links ID_006 and ID_007 in Fig. 12). The SPL engineers were also able to trace the variants strict and non-strict availability to the architectural elements that implement two different availability tactics with different repair time—active and passive redundancy—through links between grouped features and variants (see the links ID_006 and ID_007 in Fig. 12).

Therefore, as it can be verified in Fig. 12, the SPL engineers were able to effectively trace all kinds of variations they required.

> RQ_2: Do Feature-PLA traceability models assist and guide SPL engineer at the time of configuring the products of their product family?

The evidence to answer RQ_2 is assessed by analyzing the interviews given to the SPL and product engineers. From these interviews, the following excerpts can be highlighted:

> It could have been very difficult for us—the product engineers—to be able to determine a valid configuration for a metering management system application requiring strict or non-strict availability without the use of the Feature-PLA traceability model (see Fig. 12).

This means that the use of the Feature-PLA traceability model of Fig. 12 was particularly useful for the product engineers to understand the system as they hadn't been developed the OPTIMETER SPL platform.

> To configure the products A and B, we needed knowledge that helped us to perform the binding according to their respective requirements. Without the knowledge provided by the Feature-PLA traceability model (see Fig. 12), it may had been difficult to know (1) if a metering management system application requiring strict availability had to implement the services for synchronization and load balancing, or (2) if a metering management system application

requiring non-strict availability had to implement the services for routing and data monitoring. This means, without the traceability model, we hadn't feel confident about whether the variants we bound implemented all the services to satisfy the requirements of the products A and B. So the traces between (1) the feature strict availability to the variants *Synchronization* and *LoadBalancing*, and (2) the feature non-strict availability to the variants *Routing* and *DataMonitoring* were really useful to ensure that the ginding of variability was realized correctly.

Feature-PLA traceability models may be useful to identify where a feature is implemented in the PLA. As a result, it may also be useful to identify, given a change in a feature, where the change impacts the PLA. From the Feature-PLA traceability model of Fig. 12, it is easy to observe that a change in the tactic to implement strict availability may impact the variants *Synchronization* and *LoadBalancing*. Perhaps, this is not easy to locate in the code, but by making it available at the architecture-level, Feature-PLA traceability models facilitate this task. This impact knowledge may help us to correctly implement a change while maintaining the integrity of the architecture.

These excerpts from the SPL and product engineers put in evidence that our solution for tracing variability assisted and helped them at the time of configuring the two metering management systems (products A and B) from the OPTIMETER PLA.

4.2.3 Evaluation of validity

Case studies are qualitative in nature. For this reason, collected data from case studies are usually very difficult to be objectively judged [61]. To improve the internal validity of the results presented, the independent variables that could influence this case study have been identified as follows: The engineer's experience has a great influence. Its influence has been reduced as the expertise of the engineers who participated in the case study were very different (1 vs. 7 years). However, the influence of project's size and architecture's complexity cannot be reduced due to the inherent nature of case studies, which normally focus on one project. Also to improve the internal validity of the results, triangulation of source data has been used to increase the reliability of the results. In this regard, interviews were individually conducted with the engineers, although several questions were asked in a group setting to encourage discussion.

Construct validity is concerned with the procedure to collect data and with obtaining the right measures for the concept being studies. It addresses among others misinterpretation of interview questions which was mitigated by discussing the interpretations of interviews with the interviewees to validate them.

However, the major limitation in case study research concerns external validity, i.e., "the generality of the results with respect to a specific population" [57], as only one case is studied. In return, case studies allow one to evaluate a phenomenon, a model, or a process in a real setting. This is something important in software engineering in which a multitude of external factor may affect to the validation results, and that other techniques such as formal experiments, although they permit replication and generalization, do not consider as they are conducting under controlled settings.

Reliability is concerned with replication, in case studies with the fact that the same results would be found if redoing the analysis. This is why interviews were recorded and interpretations were reviewed by other participants in the study in order to avoid researcher bias.

4.3 Case study conclusions

We obtained evidence of the viability of the Feature-PLA traceability model through the execution of a case study performed in an experimental laboratory called i-smart software factory. It combines both academic and industrial efforts in R&D, with remarkable facilities for tracking the projects' progress. The case study puts the proposed traceability solution into practice within the development of a SPL of power metering management systems for smart grids. The results show evidence of that (1) the Feature-PLA traceability modeling primitives were effective in providing the capabilities for tracing most common kinds of variations that the SPL engineers required define, and (2) the Feature-PLA traceability provided knowledge that helped the product engineers to make better decisions at the time of configuring the products A and B during Optimeter II as they did not know the OPTIMETER SPL platform because they had not participated on its construction during Optimeter I. These promising results did not interfere with other practices and did not incur a big cost, making traceability possible. However, the use of the Feature-PLA traceability model requires to know and understand the modeling concepts on which they are based on, as well as to learn the usage of the FPLA modeling framework. The learning curve of these concepts as well as the usage of FPLA could slow down the process of putting traceability into practice. In fact, the SPL engineers expressed reluctance at the time of putting traceability into practice, although later, the product engineers found this traceability essential to do their work during the configuration of variability to derive the products A and B.

5 Related work

Recently, there has been a growing recognition of the importance of traceability in SPLE, which has resulted in more and more research in this area. Hence, Moon et al. [36] defined a variability trace metamodel that connects two metamodels: a metamodel for requirements and a metamodel for architecture. Ajila et al. [3] presented an evolution model that defines a dependency relationship structure of various SPL artifacts. Satyananda et al. [49] presented a framework for formally identifying traceability between feature and architecture models using *formal concept analysis*, functional decomposition, and a set of mapping analysis rules. Finally, Berg et al. [10] also defined a conceptual variability model that captures variability information across the various artifacts involved in the SPLE development. All these approaches[16] offer support for tracing SPL, including traceability of variability. The granularity of traceability links relies largely on the granularity of elements to be traced, whether requirements, architectural elements, or classes. The approaches before mentioned support architectural variability by adding or removing components or connections. However, these approaches do not have the capabilities for tracing the variability that is internal to components, i.e., variations that have fine granularity and cannot be designed as components. In this sense, our traceability model takes an step forward due to the fact that it is based on the Flexible-PLA model which allows SPL engineers to specify both external and internal variability thanks to the PPC's variability mechanism. The fact that internal variability can crosscut several components, and that is modularized and reused by PPCs (i.e., this variability is not scattered through these components), makes it easier its traceability. Therefore, our approach makes both coarse-grained and fine-grained traceability possible.

Additionally, Satyananda et al. [49] defined a set of mapping analysis rules similar to the linkage rules we propose. These rules are textually described while the linkage rules we propose are formally stated by the Feature-PLA traceability model. Models are completely subject to automation, which (1) makes it easier to define traceability links while their correctness is guaranteed by model-conformation, (2) promotes learning and reasoning over the knowledge they contain, and (3) provides the capabilities to (semi-)automatically generate other artifacts, such as code, through model transformations.

Finally, it is important to mention the work of Anquetil et al. [4] that defined a common traceability framework across the various activities of SPL development and specified a metamodel for a repository of traceability links. This framework provides a big picture of traceability for SPL by offering modeling primitives for tracing any artifact involved in the SPL development. This complete framework does not embed all these artifacts but embed references to them in order to make manageable the high number of artifacts that a complete SPL construction requires to trace. As a result, sources and targets of traceability links are paths where the artifacts are stored or can be found (documents, diagrams, or classes). The fact that these artifacts are external to the traceability model makes it difficult to guarantee that a change in an artifact is also updated in the traceability model. Additionally, this *artifacts outsourcing* makes it difficult to understand the traceability models and their usage as a guidance during the configuration of the products of a SPL while ensuring that the variability binding meets the product requirements. This is due to the fact that the relationships inside artifacts (e.g., a feature has a XOR feature group) are not included in the traceability framework and traceability links do not contain rationale and information about the traceability-making process. The Feature-PLA traceability model reduces its scope by focusing on the traceability between feature and PLA models and prioritizes the knowledge and guidance that traceability models can provide during SPL product configuration to ensure the requirements compliance. This is supported by including the source and target artifacts—the feature and PLA models—into the traceability model, as well as their relationships, and enriching traceability links with rationale and information about the traceability-making process.

[16] Although other papers propose other traceability approaches [40, 43,46], we did not include them here as they do not consider SPLE.

6 Conclusions and further work

SPLE is facing new challenges, being one of the most important the traceability of variability. To deal with this challenge, this paper presents a solution for tracing feature and PLA models called Feature-PLA traceability model, as well as the modeling framework that support it. The Feature-PLA traceability model defines a set of linkage rules to trace variable features to both the coarse-grained variability of complex components—external variability—and the fine-grained variability of simple components—internal variability.

The description and the traceability of the variability that is internal to one or many components is as important as the description and the traceability of the external variability. It is essential to cope with most kinds of variation that SPL engineers could define on their product families. Supporting both coarse-grained and fine-grained traceability of variability helps product engineers at the time of configuring this variability to derive products. This means that product engineers can examine Feature-PLA traceability models to ensure that variability bindings satisfy the product requirements.

As future work, the knowledge stored in Feature-PLA traceability models could be used to analyze the impact of changing requirements, i.e., to analyze how a change in features may affect the architecture by traversing the traces that link them. This was suggested by the engineers involved in the case study during the interviews. Additionally, the knowledge currently stored could be extended to capture more types of knowledge, such as domain knowledge, design decisions, assumptions.

The Feature-PLA traceability model and its usage still have several limitations that should be addressed in the near future. The main one is scalability, such as a scalable visualization. However, this limitation is more related to the algorithms to leverage and visualize the traceability knowledge than the expressiveness of the traceability model.

Acknowledgments The work reported in here has been partially sponsored by the Spanish fund: INNOSEP (TIN2009-13849), IMPONET (ITEA 2 09030, TSI-02400-2010-103), i-SSF (IPT-430000-2010-038), NEMO&CODED (ITEA2 08022, IDI-20110864), and ENERGOS (CEN-20091048). Finally, it is also funded by the UPM (Technical University of Madrid) under their researcher training program.

References

1. Adachi Barbosa E, Batista T, Garcia A, Silva E (2011) Pl-aspectualacme: an aspect-oriented architectural description language for software product lines. In: Crnkovic I, Gruhn V, Book M (eds) Software architecture, lecture notes in computer science, vol 6903, Springer, Berlin, pp 139–146
2. Aizenbud-Reshef N, Nolan BT, Rubin J, Shaham-Gafni Y (2006) Model traceability. IBM Syst J 45(3):515–526. doi:10.1147/sj.453.0515
3. Ajila S, Kaba A (2004) Using traceability mechanisms to support software product line evolution. In: Information reuse and integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on, pp 157–162. doi:10.1109/IRI.2004.1431453
4. Anquetil N, Kulesza U, Mitschke R, Moreira A, Royer JC, Rummler A, Sousa A (2009) A model-driven traceability framework for software product lines. Software and systems modeling, p 25. doi:10.1007/s10270-009-0120-9. URL: http://www.springerlink.com/content/wvm4hv8r78117785
5. Antkiewicz M, Czarnecki K (2004) Featureplugin: feature modeling plug-in for eclipse. In: eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange. ACM, New York, NY, USA, pp 67–72. doi:10.1145/1066129.1066143
6. Bachmann F, Bass L (2001) Managing variability in software architectures. In: SSR '01: Proceedings of the 2001 symposium on Software reusability. ACM, New York, NY, USA, pp 126–132. doi:10.1145/375212.375274
7. Bachmann F, Goedicke M, Leite J, Nord R, Pohl K, Ramesh B, Vilbig A (2004) A meta-model for representing variability in product family development. In: Linden F (eds) Software product-family engineering, lecture notes in computer science, vol 3014, Springer, Berlin, pp 66–80
8. Bass L, Clements P, Kazman R (2003) Software architecture in practice. Addison-Wesley Pearson Education, Boston, MA, USA
9. Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later: a literature review. Inf Syst 35(6):615–636
10. Berg K, Bishop J, Muthig D (2005) Tracing software product line variability: from problem to solution space. In: SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries. South African Institute for Computer Scientists and Information Technologists, Republic of South Africa, pp 182–191
11. Beydeda S, Book M, Gruhn V (2005) Model-driven software development. Springer, Berlin
12. Bezivin J (2005) On the unification power of models. Softw Syst Model 4(2):171–188
13. Cleland-Huang J, Gotel O, Zisman A (2012) The grand challenge of traceability (v1.0). Springer, London
14. Clements P, Northrop L (2002) Software product lines: practices and patterns. Addison-Wesley, Boston, MA, USA
15. Czarnecki K (2005) Mapping features to models: a template approach based on superimposed variants. In: GPCE 2005—generative programming and component engineering. 4th international conference. Springer, pp 422–437
16. Dashofy EM, Hoek AVD (2002) Representing product family architectures in an extensible architecture description language. In: PFE '01: revised papers from the 4th international workshop on software product-family engineering. Springer, pp 330–341
17. Díaz J, Pérez J, Garbajosa J, Yagüe A (2013) Change-impact driven agile architecting. In: Proceedings of the 46th Hawaii international conference on system sciences (HICSS '13), Hawaii, USA, 7–10 Jan 2013, IEEE Computer Society Press, pp 4780–4789
18. Espinoza A, Garbajosa J (2008) A proposal for defining a set of basic items for project-specific traceability methodologies. In: Software engineering workshop, 2008. SEW '08. 32nd Annual IEEE, pp 175–184
19. Gotel O et al (2012) The grand challenge of traceability (v10). In: Cleland-Huang J, Gotel O, Zisman A (eds) Software and systems traceability, Springer, London, pp 343–409
20. Gotel O, Finkelstein C (1994) An analysis of the requirements traceability problem. In: Proceedings of the first international conference on requirements engineering, pp 94–101. doi:10.1109/ICRE.1994.292398
21. Hauser JR, Clausing D (1988) The house of quality. Harv Bus Rev 66(3):63–73
22. Jacobson I, Griss M, Jonsson P (1997) Software reuse. architecture, process and organization for business success. Addison-Wesley, Boston
23. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University, Pittsburgh, PA, USA, CMU/SEI-90-TR-21 ESD-90-TR-222
24. Khurum M, Gorschek T (2009) A systematic review of domain analysis solutions for product lines. J Syst Softw 82(12):1982–2003
25. Kizcales G, Lamping J, Mendhekar A, Maeda C (1997) Aspect-oriented programming. In: Proceedings of the 11th European conference on object-oriented programming (ECOOP), lecture notes in computer science, vol 1241. Springer
26. Kolovos DS, Paige RF, Polack FAC (2006) On-demand merging of traceability links with models. In: In Proceedings of 3 rd ECMDA traceability workshop

27. Letelier P (2002) A framework for requirements traceability in uml-based projects. In: In Proceedings of 1st International. Workshop on traceability in emerging forms of software engineering, pp 32–41
28. Loughran N, Sánchez P, Garcia A, Fuentes L (2008) Language support for managing variability in architectural models. In: SC'08: Proceedings of software composition, 7th international symposium, lecture notes in computer science, vol 4954, Springer, pp 36–51
29. Magee J, Kramer J (1996) Dynamic structure in software architectures. In: Proceedings of the 4th ACM SIGSOFT symposium on foundations of software engineering, SIGSOFT '96, ACM, New York, NY, USA, pp 3–14
30. Mahdavi-Hezavehi S, Galster M, Avgeriou P (2013) Variability in quality attributes of service-based software systems: a systematic literature review. Inf Softw Technol 55(2):320–343
31. Martin JL, Yague A, Gonzalez E, Garbajosa J (2010) Making software factory truly global: the smart software factory project. In: Fagerholm F (ed) Software factory magazine. http://www.softwarefactory.cc/magazine, p 19
32. Massoud Amin S, Wollenberg B (2005) Toward a smart grid: power delivery for the 21st century. Power Energy Mag IEEE 3(5):34–41. doi:10.1109/MPAE.2005.1507024
33. Matinlassi M (2004) Comparison of software product line architecture design methods: COPA, FAST, FORM, KOBRA and QADA. In: ICSE '04: Proceedings of the 26th international conference on software engineering. IEEE Computer Society, Washington, DC, USA, pp 127–136
34. Medvidovic N, Taylor RN (2000) A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng 26(1):70–93. doi:10.1109/32.825767
35. Mens T (2010) Future research challenges in software evolution and maintenance—report from EC expert meeting. ERCIM News 81
36. Moon M, Chae HS, Nam T, Yeom K (2007) A metamodeling approach to tracing variability between requirements and architecture in software product lines. In: CIT '07: Proceedings of the 7th IEEE international conference on computer and information technology, IEEE Computer Society, Washington, DC, USA, pp 927–933
37. Object Management Group (2006) Data distribution service for real-time systems, v1.2
38. Object Management Group (2006) Meta-object facility (MOF) specification 2.0 TR formal-06-01-01. http://www.omg.org/spec/MOF/2.0/PDF/
39. Object Management Group (2011) OCL specification version 2.2. http://www.omg.org/spec/OCL/2.2/
40. Olsen G, Oldevik J (2007) Scenarios of traceability in model to text transformations. In: Akehurst D, Vogel R, Paige R (eds) Model driven architecture: foundations and applications, lecture notes in computer science, vol 4530, Springer, Berlin, pp 144–156
41. Pérez J, Díaz J, Soria CC, Garbajosa J (2009) Plastic partial components: a solution to support variability in architectural components. In: Proceedings of joint working IEEE/IFIP conference on software architecture 2009 and European conference on software architecture 2009, WICSA/ECSA 2009, Cambridge, UK, 14–17 Sept 2009. IEEE, pp 221–230
42. Pérez J, Díaz J, Garbajosa J, Alarcón PP (2010) Flexible working architectures: agile architecting using ppcs. In: Proceedings of the 4th European conference on software architecture (ECSA 2010), LNCS, Springer, Berlin, pp 102–117
43. Pohl K, Brandenburg M, Gülich A (2001) Integrating requirement and architecture information: a scenario and meta-model approach. In: REFSQ'01: Proceedings of The seventh international workshop on requirements engineering: foundation for software quality, pp 68–84
44. Pohl K, Böckle G, Linden F (2005) Software product line engineering: foundations, principles and techniques. Springer, Germany
45. Poshyvanyk D, Di Penta M, Kagdi H (2011) Sixth international workshop on traceability in emerging forms of software engineering: (tefse 2011). In: 33rd international conference on software engineering (ICSE 2011), pp 1214–1215. doi:10.1145/1985793.1986052
46. Ramesh B, Jarke M (2001) Toward reference models for requirements traceability. IEEE Trans Softw Eng 27(1):58–93. doi:10.1109/32.895989
47. Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng 14:131–164
48. Runeson P, Höst M, Rainer A, Regnell B (2012) Case study research in software engineering: guidelines and examples. Wiley, Hoboken
49. Satyananda TK, Lee D, Kang S, Hashmi SI (2007) Identifying traceability between feature model and software architecture in software product line using formal concept analysis. In: Proceedings of the international conference computational science and its applications. IEEE Computer Society, Washington, DC, USA, pp 380–388
50. Schwaber K, Beedle M (2002) Agile software development with scrum. Prentice-Hall, Englewood Cliffs
51. Scott J, Kazman R (2009) Realizing and refining architectural tactics: Availability. Tech. rep., CMU/SEI-2009-TR-006 ESC-TR-2009-006, Pittsburgh, USA
52. Selic B (2003) The pragmatics of model-driven development. IEEE Softw 20(5):19–25. doi:10.1109/MS.2003.1231146
53. Staab S, Walter T, Grner G, Parreiras F (2010) Model driven engineering with ontology technologies. In: Amann U, Bartho A, Wende C (eds) Reasoning web semantic technologies for software engineering, Lecture Notes in Computer Science, vol. 6325, Springer, Berlin, pp 62–98
54. Stake RE (1995) The art of case study research. Sage, London
55. Szyperski C (2002) Component software: beyond object-oriented programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA
56. Taha WM (2009) Domain-specific languages IFIP TC 2 working conference, DSL, lecture notes in computer science, vol 5658. Springer, Berlin
57. van Heesch U, Avgeriou P, Hilliard R (2012) A documentation framework for architecture decisions. J Syst Softw 85(4):795–820. doi:10.1016/j.jss.2011.10.017
58. van der Hoek A, Heimbigner D, Wolf AL (1999) Capturing architectural configurability: variants, options, and evolution. Tech. rep., Technical Report CU-CS-895-99, Department of Computer Science, University of Colorado, Boulder, Colorado
59. van Ommering R, van der Linden F, Kramer J, Magee J (2000) The koala component model for consumer electronics software. Computer 33(3):78–85. doi:10.1109/2.825699
60. Weiler T (2003) Modelling architectural variability for software product lines. In: SVM'03: Proceedings of the software variability management workshop, pp 53–61
61. Yin R (2008) Case study research. Design and methods. 4th edn. Sage, London