An Incremental Approach to Manage Variability in Software Product Line Requirements

Samaneh Zamanifard[1*], Ramtin Khosravi[2], Hamideh Sabouri[2]

[1] Department of Computer Engineering Islamic Azad University, Kerman Branch, Joopar Road, Kerman, Iran.
[2] School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran.

* Corresponding author. Tel.: 989123485864; email: samaneh.zamanifard@gmail.com
Manuscript submitted October 18, 2014; accepted December 16, 2014.
doi: 10.17706/ijcee.2015.v7.872

Abstract: In software product line engineering, systematic variability management must be applied to all stages of the development lifecycle and every artifact produced. This is not an easy task for the artifacts describing system requirements, as they are usually less structured and formal compared to design and implementation models. In this paper, we overhead in documenting the requirements present an incremental and modular method to capture variability in system requirements described as use cases. Following the well-known technique of delta-oriented programming, our method captures commonalities among the requirements in terms of a core model and applies the optional extensions or changes to the core model expressed in terms of delta modules. The method also addresses cross-cutting changes that span several use cases. We have also developed an integrated web-based toolset to effectively manage the requirements models, facilitating product derivation at virtually no cost. We have applied the proposed approach to a real-world case study which shows an acceptable overhead in documenting the requirements and results in considerable saving in generating documents for the specific products as well as enabling traceability of features into requirements. The incremental characteristic of our method makes it also suitable for handling variability in time, by enabling the analysts to specify requirements changes in a modular way.

Key words: Software product lines, delta modeling, requirements document, variability management.

1. Introduction

Software product line engineering (SPLE) is a software development paradigm that allows companies to decrease the cost of development and increase product quality by harnessing systematic reuse via planned management of commonalities and variability among a set of related products called a product family. Unlike traditional reuse, which is mostly focused on implementation artifacts, SPLE utilizes reuse in the development of the artifacts related to all stages of the software development process, including requirements elicitation, modeling, and documentation.

A well-known approach to SPLE is to have two processes, namely, domain engineering and application engineering [1]. In the domain engineering process, after an extensive study of the problem domain, a set of reusable artifacts capturing the commonalities of among the products are developed. This includes a common architectural platform, as well as various other artifacts such as requirements and design models, reusable components, and test artifacts, forming a core reusable asset. In application engineering, during which a specific product is derived from the family, the application goes through the software development

life-cycle stages, reusing the relevant artifacts produced during domain engineering (with some configurations if needed) combined with application-specific artifacts related to the product in hand.

In this work, we focus on the requirements elicitation artifacts, aiming to provide a way to efficiently produce the set of requirements for the specific application being derived during application engineering. To this end, we must provide a way to organize the requirements of the whole family into reusable pieces, such that when a new product is to be derived, the product's requirements can be assembled from the existing pieces with a low overhead.

An important technique helping us to reach the stated goal is feature modeling, in which a product family is decomposed into a set of features organized in a containment hierarchy named feature model. Some of the features are mandatory and required in all products, whereas the others are optional which are used in some products. Feature modeling is explained in more detail in Section 2.1. Our presented method is based on the idea of delta-oriented programming [2] which is originally applied at the programming language level, and subsequently used in other stages of development such as software design [3], [4]. In the delta-oriented approach (in Section 2.2), a core module is developed which contains the parts common to all products in the family, and a number of delta modules are defined addressing the variability in the products. During domain engineering, the core and the delta modules are developed added to a repository of modules. When deriving a product, the core is combined with the relevant delta modules to produce the model/code for the specific product. When specifying a delta module, the developer determines under which conditions the delta must be included in the product. The conditions are usually expressed in terms of the feature included or excluded. In our method, the requirements are documented as use cases (in Section 2.3). The core module consists of a set of core use cases common to all products. The delta modules may add new use cases or include new parts in an existing use case scenario. Various parts of the use case descriptions are tagged so that a delta module may also change or remove parts of the use cases. We also address cross-cutting changes to several use cases using the basic ideas in the aspect-oriented development (in Section 4.3). The proposed method is supported by a tool such that the analyst can efficiently derive the requirements for a specific product just by specifying the features included in the produce (in Section 5). We have evaluated the presented method by using it in an industrial case study (in Section 6.1) in the domain of IPTV. The results of the evaluation are presented in Section 6.3, which shows our method is applicable in practice.

Several methods have been proposed for modeling requirements of an SPL, some of them are based on various UML diagrams, including use case diagrams [5], [6], sequence diagrams [7], activity diagrams [8], and even class diagrams [9]. These methods mainly use stereotypes and tagged value mechanisms in UML to specify variability in the corresponding diagrams. Although models are important parts of the analysis model, most methodologies use textual documents to capture the requirements, as they are readable by the stakeholders. Even though use cases are widely used in software industry today, but the textual use case descriptions are considered as the main artifact rather than the use case diagram (which serves as a model to give an overview of the system requirements) [10]. Moreover, the mentioned models in UML are not devised to be used in SPLE, so they do not have explicit variability management mechanisms.

Another existing approach is to annotate the variable parts of the textual requirements documents to specify the condition under which the variable parts are applied [11], [12]. This approach suffers from the fact that it is needed to explicitly describe the variants for each variation point, making the documents cluttered and unreadable, especially when dealing with a software product line with a large number of variation points. Our proposed method follows the delta-oriented approach which organizes the changes to be made to the core requirements in separate modules (in Section 4.2). Hence, the core requirements documents are kept simple and readable (in Section 4.1). As will be described, we derive the requirements

documentation for each product automatically by applying the required delta modules to the core (in Section 4.4), so there is no overhead cost during product derivation. Another advantage of our method is that its incremental nature can be easily employed to support the evolution of requirements, even in the context of single product development. In this case, a new version of a product may change various parts of the requirements. These changes can be packaged in delta modules associated with that version. The third approach is to use aspect-oriented techniques to add optional parts to specific places in the textual requirements documents [13], [14]. Some of these approaches are explained in more detail in Section 3 and compared to our work in Section 6.4.

2. Preliminaries

In this section, first we describe feature models. Then, we explain delta-oriented modeling that our approach is inspired by it. Our work is based on use case scenario which is described next.

2.1. Feature models

Commonalities and differences among various products are modeled explicitly in software product line engineering using feature models. In the well-known method of Feature Oriented Domain Analysis (FODA), proposed by Kang *et al*. [15], a feature is defined as user-visible aspect which depicts system characteristics.

A feature model represents all possible products of a software product line in terms of features and relationships among them. A basic feature model includes mandatory and optional features. The features may have or and xor relationships. It also includes required and exclusive constraints between the features:

- Mandatory feature: Feature f_1 may have a *mandatory* child f_2, meaning that the products including f_1 must include f_2 as well. The root feature is always considered as a mandatory feature.
- Optional feature: Feature f_1 may have an optional child f_2, meaning that the products including f_1 may include or exclude f_2.
- Inclusive or: Feature f may have a set of children f_1, …, f_n, with *inclusive or* relationship among them, meaning that at least one of the features f_1, …, f_n must be included in the products that include *f*.
- Exclusive or: Feature f may have a set of children f_1, …,f_n, with *exclusive or* relationship among them, meaning that exactly one of the features f_1, …, f_n must be included in the products that include *f*.
- Constraint between the features: If feature f_1 *requires* another feature f_2, all products including f_1 must include f_2 as well. Likewise, if feature f_1 *excludes* another feature f_2, there is no product including both f_1 and f_2.

Fig. 1 illustrates a subset of the feature model of our case study. Our case study is Internet Protocol Television (IPTV) back-office system of Soroush High Tech. Company [16]. IPTV allows the viewer to communicate with TV beyond simple actions such as changing the volume and channels. In IPTV, a viewer can connect to the movie database and select a movie to watch, connect to the audio database and select a track to play, shop through the TV, or play games with other people.

All IPTV back-office products must have the Content Management System (CMS) feature. A CMS includes “upload asset”, “movie asset”, and “track asset” as these features are mandatory. The “live content” feature is optional thus products do not necessarily support live content.

Product derivation is performed based on the notion of product configuration (or configuration for short). A configuration specifies the features that are included in a specific product. For example, a configuration of an IPTV product is illustrated in Fig. 2.

A configuration is valid if it respects all of the restrictions specified by the feature model. The configuration depicted in Fig. 1 is valid as it includes all mandatory features. It also contains feature Post-pay to consider the inclusive or relationship between features Post-pay and Prepay.

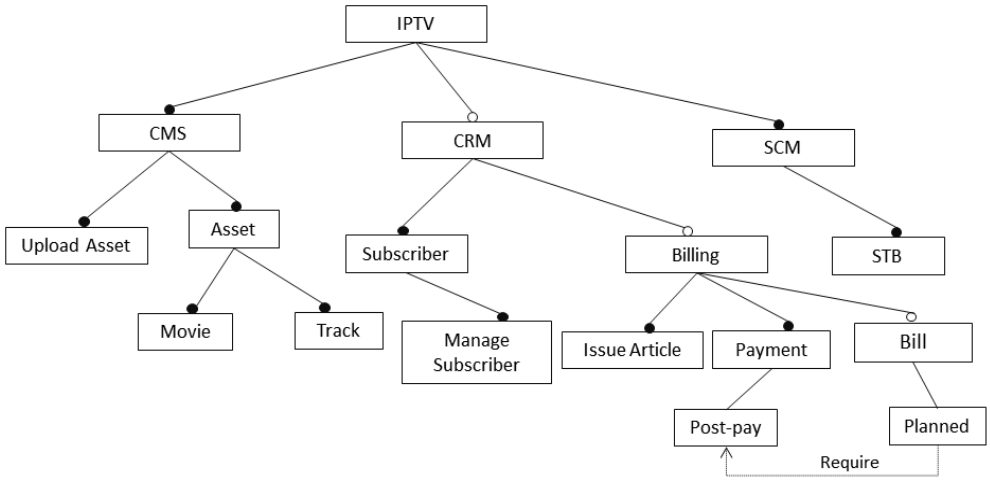

Fig. 1. A sample of IPTV configuration.

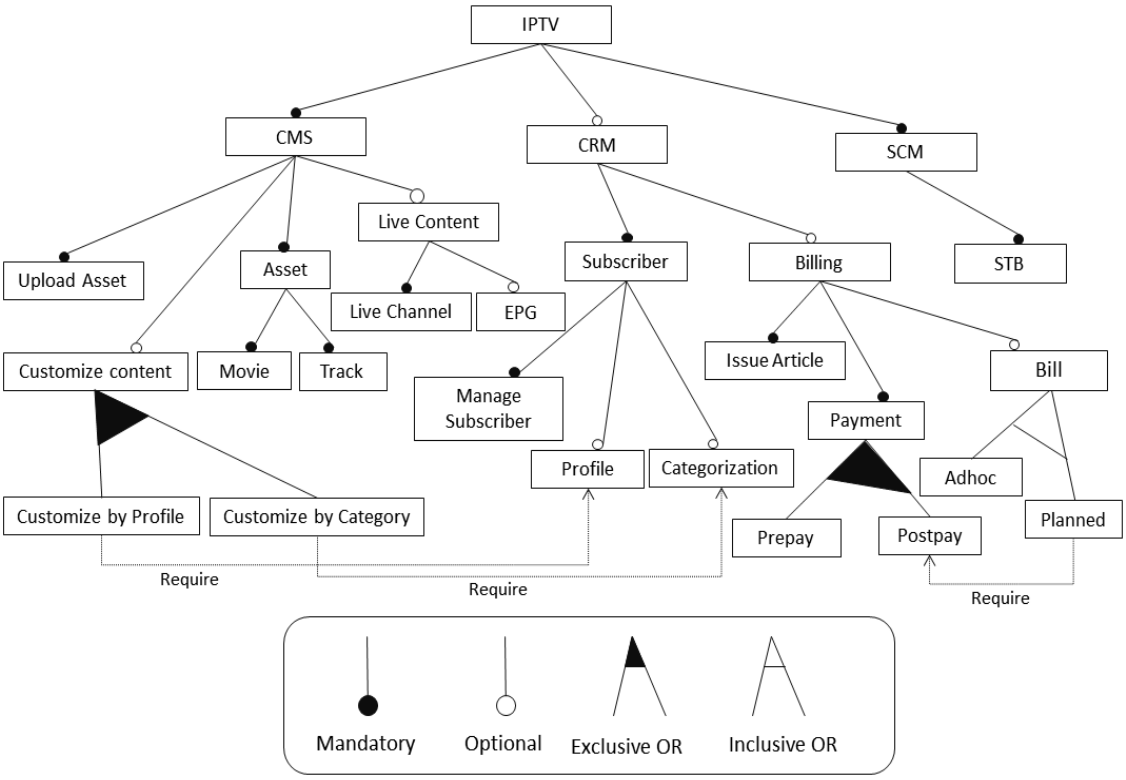

Fig. 2. A subset of IPTV feature model.

2.2. Delta-Oriented Modeling

Delta-oriented modeling is introduced by Schaefer [2]. In this approach, an SPL is represented by a core module and a number of delta modules. The core module is a valid product. In the core module, all mandatory features and a number of optional features are implemented. Delta modules represent the changes which should be applied to the core module to implement a new product. Fig. 3 shows the main idea of delta-oriented modeling.

As it is depicted in Fig. 3, features E, G, and H are mandatory features which are considered as the core module. Feature B, C, D, F, I, J, and K are implemented by delta modules. Delta modules are applied to the core module if the corresponding feature is selected in the given configuration. This model simplifies the original notion of delta modeling by assuming that each delta is associated with a single feature.

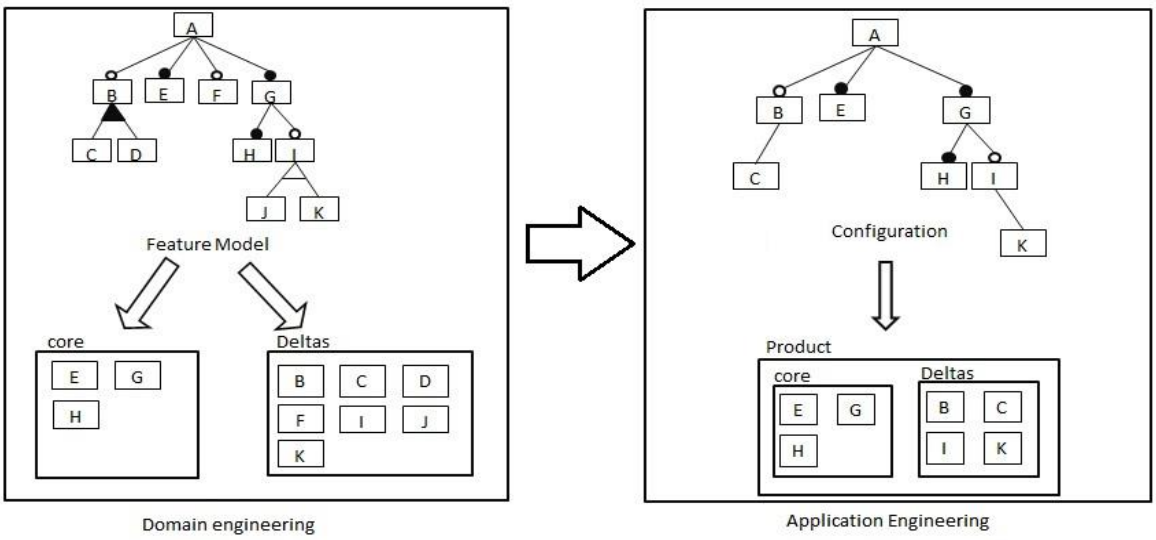

Fig. 2. Delta-oriented method.

2.3. Use Case Scenario

Use case scenario is an effective method to describe functional requirements of a system. A sequence of actions that describe system functionality is more understandable for stakeholders than an abstract model. A use cases depicts the possible interactions between the system and the user.

Use cases can be described in various ways. In [10] an extensive and efficient technique is proposed to describe a use case. This technique is based on natural language. Natural language descriptions of requirements lead to readable requirement documents for all people including non-technical people who are stockholders but they are not involved in system development. The reason is that describing the system's functionality through a sequence of actions is more understandable for stakeholders than an abstract model. Fig. 4 shows the meta-model that we use to describe use cases.

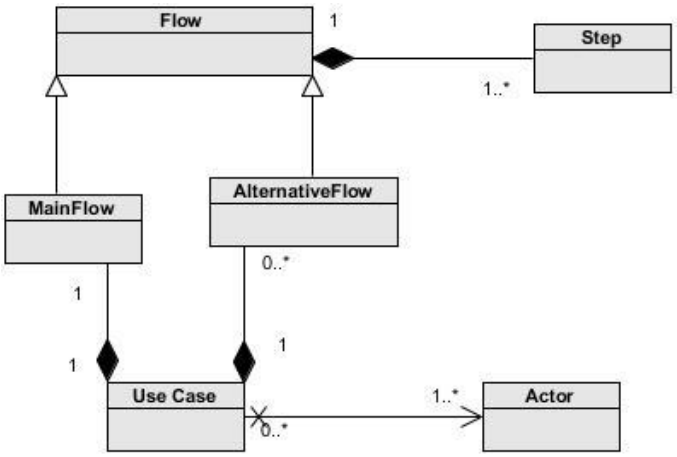

Fig. 3. The meta-model of our use case descriptions.

According to this model, use cases have an actor who triggers the system. Each use case should include one main scenario and it may contain one or more alternative scenarios. Each scenario has a flow and each flow consists of a number of steps.

The “play media request” use case of our case study is shown in Fig. 5 In the main scenario, the interactions between the system and the user are described.

Use case name: Play request

Actor: subscriber

Main scenario:

1- Subscriber requests list of media.
2- System shows list of active media.
3- Subscriber chooses relevant media and sends play request.
4- System plays the media.
5- System updates the history of played media.

Fig. 4. Play media request scenario.

3. Motivating Example

One of the main goals of SPLE is proactive reuse of software artifacts. In this paradigm, reusability is considered in all software engineering phases. Requirement engineering is the first phase. As we mentioned earlier, use case scenarios are used to describe functional requirement of a system. They include scenarios in which the steps needed to fulfill an action are listed. When describing the requirements of an SPL, some of these steps are related to mandatory features and some of them are relevant to optional features. A clear separation between variable and common parts of such documents is needed so that it would be reusable. In addition, it should be intelligible for different customers with various requirements. In this section, we explain this problem using one possible scenario of IPTV system. Fig. 3 shows the "play media request" scenario in which a media is played based on the request of a subscriber. In the IPTV system, the "manage subscriber" feature is optional as in some cases, such as using the system for a stadium, there is no need to recognize or track that who is working with IPTV system. In Fig. 5, the "play media request scenario" is described without considering this feature. Therefore, when the "manage subscriber" feature is selected, step 5 of the scenario should be changed to "System updates the history of played products for the subscriber".

Another feature of the IPTV system is customizing the contents based on the subscriber's profile and category. If the "customize content by profile" feature is selected, step 2 of the "play media request" scenario should restrict the product list based on the subscriber profile, so it should be changed to "System shows a list of active products based on the subscriber's profile". The feature "customizes content by category" changes step two as well. If this feature is selected, the list of products is restricted based on subscriber's category. Each subscriber can belong to one or more categories such as student, golden club, and etc. Thus, when this feature is selected, step two should be changed to "System shows a list of active products based on the subscriber's categories". As it is shown in Fig. 1, these features have an OR relationship in the feature model. Thus, a product may include both features. In such product, step two should be changed to "System shows a list of active products based on subscriber's category and profile".

The "issue article" feature affects the "play media request" scenario as well. In a product with the billing feature, media have a cost. Therefore, in the scenario the system should check whether the media is bought or not. In this case, step 4 in this scenario should be changed to "System checks if the media is purchased by the subscriber, then it will be played."

To manage variability in use case scenarios, several notations have been proposed. One approach is Product Line Use Case (PLUC) [17]. The "play media request" scenario which employs PLUC notation is shown in Fig. 6 In this approach, the steps involving variability are distinguished using [V] notation. The behavior of such steps is determined in product derivation. There is no explicit relation between the feature model and product derivation.

Main scenario:

1- Subscriber requests list of media.
2- [V1]
3- Subscriber chooses relevant media and sends play request.
4- [V2]
5- [V3]

Variations

Product definition:

0. Product 1 (none of feature)
1. Product 2 (product with subscriber management feature)
2. Product 3 (product with customize content by category feature)
3. Pro duct 4 (product with customize content by profile feature)
4. Product 5 (product with billing feature)

V1: Alternative

If V1=0 then "System shows list of active media."

Else if V1=2 then "System shows list of active media based on subscriber categories"

Else if V1=3 then "System shows list of active media based on subscriber profile."

V2: Alternative

If V2=4 then "System checks if media purchased by the subscriber then it will be played."

Else if V2= 0 then "System plays selected media."

V3: Alternative

If V3=1 then "System updates the history of played media."

Else if v3=1 then "System updates the history of played media for subscriber."

Fig. 5. Play media request scenario described using PLUC approach.

Another approach to represent variation points in use cases is Product Line Use case modeling for System and Software Engineering (PLUSS) [12]. Fig. 7 shows "play media request" scenario described using PLUSS notation.

Id	User action	System response
(1)a	Subscriber requests list of media. [Not customize content]	System shows list of all active media.
(1)b	Subscriber requests list of media. [customize content by profile]	System shows list of active media based on subscriber profile.
(1)c	Subscriber requests list of media. [customize content by category]	System shows list of active media based on subscriber categories.
(2)	Subscriber chooses relevant media and sends play request.[Not billing]	System plays selected media.
(2)	Subscriber chooses relevant media and sends play request.[Billing]	System checks if media purchased by the subscriber then it will be played.
(3)	[Not manage subscriber]	System updates the history of played media.
(3)	[Manage subscriber]	System updates the history of played media for subscriber

Fig. 6. Play media request scenario described using PLUSS notation.

In this approach, the response of the system to user actions is specified based on including or excluding the related features. Thus, the behaviors of mandatory and optional features are tangled. As we mentioned earlier, some features may affect the same step of a scenario (e.g. customize contents by profile and by category features). In this case, a step "System shows a list of active products based on the subscriber's category and profile." should be considered. However in PLUSS, two separate steps are written. Therefore, it is difficult to understand the behavior of a specific product.

Another approach proposed to model variability in scenarios is Modeling Scenario Variability as Crosscutting Mechanism (MSVCM) [13]. In this approach, cross-cutting mechanism is explained. In SVCM, a mandatory scenario is described in the use case and variable steps are depicted as advices. In MSVCM, a step cannot be replaced with another step and all states should be described in advice. All of the steps in the "play media request" scenario should be defined as an advice. Main scenario does not exist for weaving

advices, so join point definition is impossible. As you could see, MSVCM approach is not used in this case.

4. Delta-Oriented Description of SPL Requirements

Software product line consists of common and variable requirements. Usually, mandatory features are associated with common requirements and optional features correspond to variable requirements. Optional features may have xor/or relationship.

We introduce the notion of variability into requirements documents by defining a core requirement document and a set of deltas. In our approach, each mandatory feature is described by a use case scenario. Consequently, the core requirements of an SPL include all use cases associated with mandatory features. Selecting an optional feature may add a new use case or change an existing use case, or both.

In the first case, the use case scenario is described similar to mandatory features. In the second case, when an optional feature modifies an existing use case, changes are described through deltas. Each delta contains a list of modifications where each modification may change an existing step of a scenario or may add a new step to it.

4.1. Description of Use Case

In our approach, use cases are either associated with a mandatory feature or an optional feature. The corresponding feature must be specified in the definition of each use case. This helps us to generate documents automatically: for a given configuration the use cases corresponding to the selected set of features form the ultimate document. In the description part of use case scenarios, we assign a tag to each step. When defining a delta, these tags are used to refer to a specific step of a scenario that should be changed.

Fig. 8 is an extended version of the meta-model in Fig. 4 and represents the use case meta-model in our approach. To model variability, use case scenarios have two main differences with use case scenarios of a single product: (1) each step of scenarios must have a tag; (2) each use case is associated with one feature.

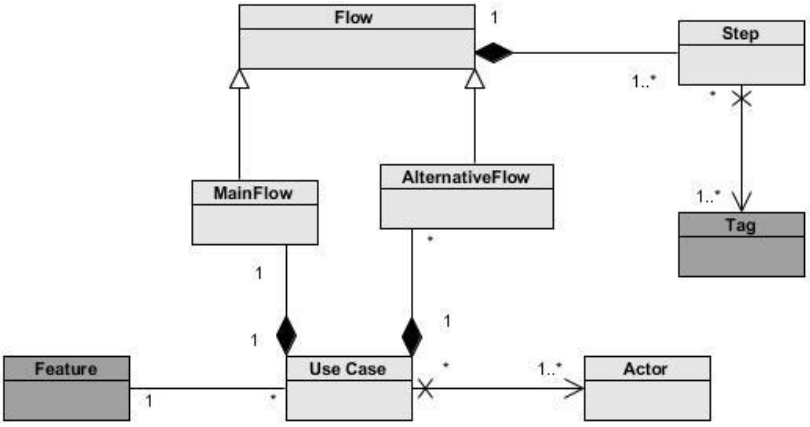

Fig. 7. Use case meta-model for delta-oriented approach.

As it is depicted in Fig. 8, each use case is related to one feature, which may be mandatory or optional. In addition, a use case scenario must have a main flow and one or more alternative flows, depending on the use case scenario. Each flow in a use case consists of a number of steps. In our proposed approach, the steps must have tag/tags.

4.2. Delta Definition

A delta represents the changes that should be applied to use cases based on inclusion or exclusion of

optional features. Each delta has a list of changes which may affect different parts of use case scenarios. The meta-model of deltas which we propose in our approach is shown in Fig. 9.

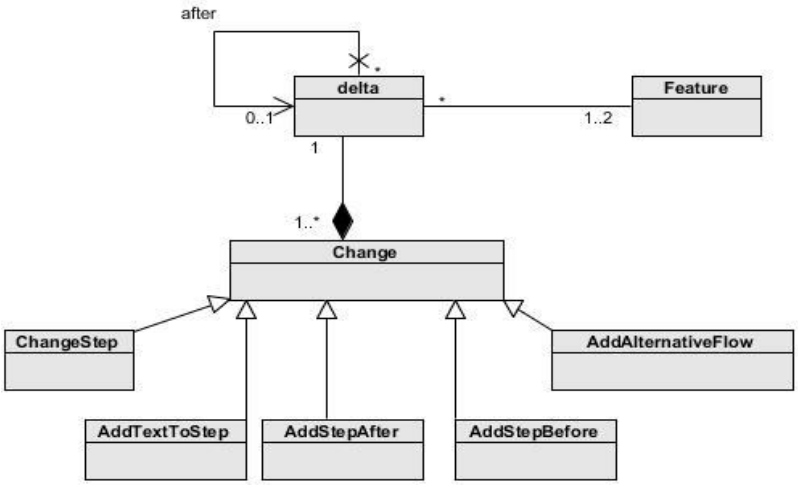

Fig. 8. Delta Meta model.

A "Change", as it is represented in the meta-model, may cause the following modifications: changing a specific step (ChangeStep), adding a step in a specific point (AddStepAfter and AddStepBefore), adding an alternative flow (AddAlternativeFlow), and adding text to the end of a specific step (AddTextTostep). All these changes can be used for the main flow or an alternative flow.

In the first line of delta definition, the name of the delta, its corresponding features, and the "after" clause are specified. All deltas must have a name and one corresponding feature. The second feature and the "after" clause are used to determine the features which affect other features or have priority over them. Our approach can be extended to support associating a propositional logic formula with each delta (instead of two features) and to define a priority relation among deltas. The syntax of defining a delta is as follows:

Delta **DeltaName** After **DeltaName** When **Feature1| Feature2**

In the "after" clause, the name of the delta which should be applied before this delta is specified. In addition, a delta has a list of changes. Each change determines which use case of which package should be modified. Thus, in all change definitions, the name of the use case and corresponding package must be introduced. The changes are defined as follows:

- **Modify a specific point of flow**

In this type of modification, step/steps are added before or after a specific step in flow.

Change Type: AddStepBefore/AddStepAfter

Alternative Name: Name of the Alternative Flow

Before Step/After Step: Tag of the corresponding step

Steps: List of steps with specified format ({tag} description of step)

The type of change may be set to "AddStepBefore" or "AddStepAfter". If an alternative flow should be modified, its name is specified in "Alternative Name". "Before Step/After Step", indicates the point where the changes should be applied to it, based on the type of change. Finally in "Steps", the step/steps which should be added to the flow are defined.

- **Add an alternative flow to use case scenario**

In this modification, an alternative flow is added to the use case scenario. Since a use case scenario may have more than one alternative flow, the name of the alternative flow should be specified.

Change Type: AddAlternativeFlow

Alternative Name: Name of the Alternative Flow

Steps: List of Steps with Specified format ({tag} description of step)

The type of change is set to "AddAlternativeFlow". The name of the alternative flow is specified in "Alternative Name". Finally, the steps of the added alternative flow are defined in the "Steps" part.

- **Modify a specific point of flow**

In this type of modification, a specific step in a flow is replaced with a number of steps.

Change Type: ChangeStep

Alternative Name: Name of Alternative Flow

Relevant Step: Tag of the corresponding step

Steps: List of Steps with Specified format ({tag} description of step)

The type of change is set to "ChangeStep". When a specific step should be replaced by a number of steps in an alternative flow, the name of the flow should be specified in the "Alternative Name". The step being replaced is indicated in "RelevantStep". Finally in the "Steps" part, the substitute steps are defined.

- **Add text to the end of a specific step in a flow**

In this type of modification, a text is added to the end of a specific step in a flow.

Change Type: AddTextToStep

Alternative Name: Name of Alternative Flow

Step: Name of Relevant Step

Tag: Tag Name

Text: Text

The type of change is set to "AddTextToStep". In the "Step" part, we indicate the step that the text should be added to it. If the added text requires updating the tag, the new tag should be specified in "Tag". The added text is specified in the Text.

4.3. Cross-Cutting Feature Management

The effects of some features are scattered among many use cases. For example, adding the method of payment (postpaid or prepaid) feature affects all financially related scenarios in a similar way. To manage these kinds of features in delta definition, we may indicate the affected use cases by specifying a string that is included in their names, or even using star mark (*) to select all use cases. For the selected set of use cases, we use tags to specify where the changes should by applied. In this way, we can modify multiple use cases using a single delta. This facility may be used with all types of modifications. Fig. 10 shows how a delta may change multiple use cases. The delta in this figure is associated with the "Issue Article" feature. Consequently, if a configuration contains the "Issue Article" feature, this delta should be applied to the relevant use cases. This feature also adds a use case where the "media purchase" scenario is considered. After each purchase, an article should be issued for subscriber. The related delta contains the following changes:

- In all asset definition scenarios, a step must be added for the price of media.
- In all asset search scenarios, price should be added to the steps in which the search fields are defined.
- In all asset search scenarios, price should be added to the steps in which the result is defined.

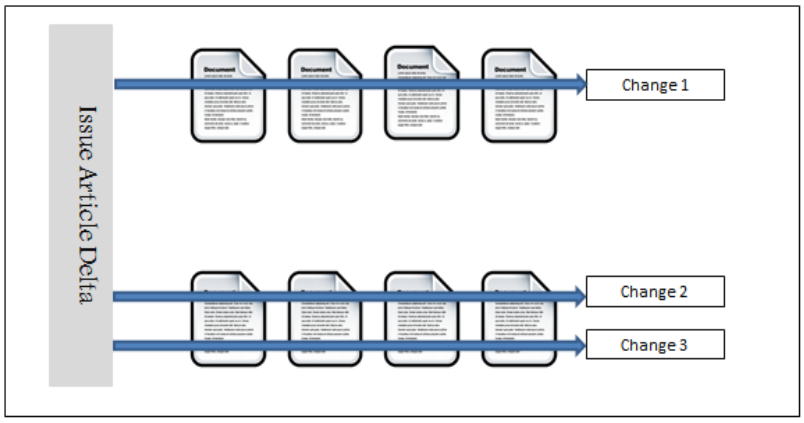

Fig. 9. Changing multiple use cases by applying a single delta.

As mentioned before, each delta has a list of modifications. Each change may modify a group of use cases. Type of change 1 in Fig. 10 is "AddStepBefore". It specifies that a step must be added before the save step in all asset definition scenarios for media price definition. Types of change 2 and 3 are "AddTextToStep". These steps add the price to the search fields and results respectively, in all asset search scenarios.

4.4. Product Derivation

In product derivation, features are chosen from the feature model based on customer needs. Constrains and relations between features are checked at this level. If no feature is chosen, only the mandatory features are considered and documents are generated based on the core requirements.

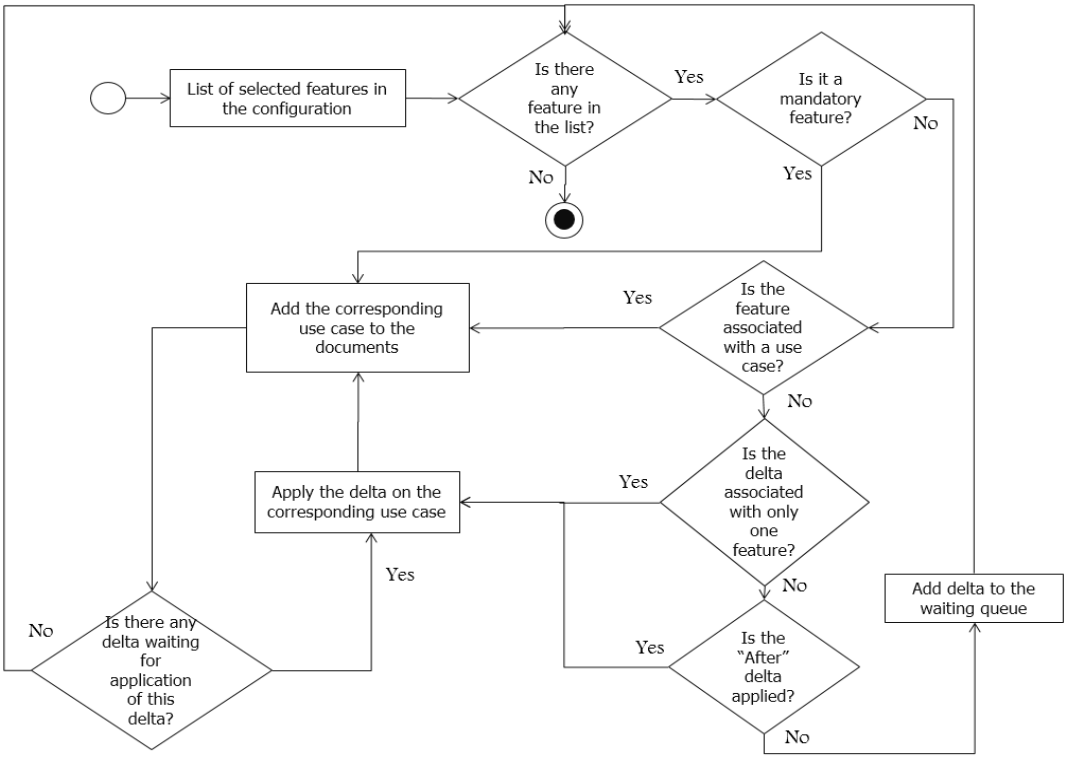

Fig. 10. Flowchart of creating documentation for a particular product.

To generate the documents of a specific product, first we consider all use cases that correspond to the mandatory features. Then, we consider optional features and the use cases associated with them. The flowchart for generating the documents of a specific product is presented in Fig. 11.

To generate documents, first we apply all deltas that associated with only one feature as these deltas do not have conflict with other deltas. All Deltas are marked after they are applied. Then, we consider deltas that are associated with two features. These deltas are related to features which affect other features or have priority over them. If a delta should be applied after the application of a specific delta and the specific delta has not been applied yet, it is added to a waiting queue. After applying each delta, the waiting queue is checked and if some deltas are waiting on this delta, they will be applied. When the waiting queue gets empty and all deltas are applied, the process is finished and the document for specific product is ready.

5. Implementation

We developed a tool to automatically generate documents for products. Moreover we can track the effects of each feature in all software development phases using this tool. A web-based tool is also implemented to access requirements every time, everywhere.

To use our tool, features, constraint, and relations between them should be defined first. In next step, use cases should be defined. In this step, the relations among use cases and features are specified. Each step in the use case scenario must have one or more tags to indicate what is happening in this step. Addressing steps in deltas is also done by tags. Features that affect use cases are described with deltas. The use cases that are affected by adding a feature are determined in delta definition. To generate documents for a specific product, the feature tree is shown to the user and the user chooses a set of features based on the customer needs. Implemented tool generates documents with the HTML format.

Tracking features' effects in documents is one of main challenges in SPL. In our proposed approach, the relation between use cases and deltas are clarified. An xml file is considered in the implemented tool, to prevent reviewing all use cases and deltas to track the effects of features. In this file, for each feature we specify the related use cases and deltas. After defining use cases and deltas, this file is automatically updated. In our tool, user can select any feature and pursue its related use cases and deltas.

5.1. Defining Use Cases

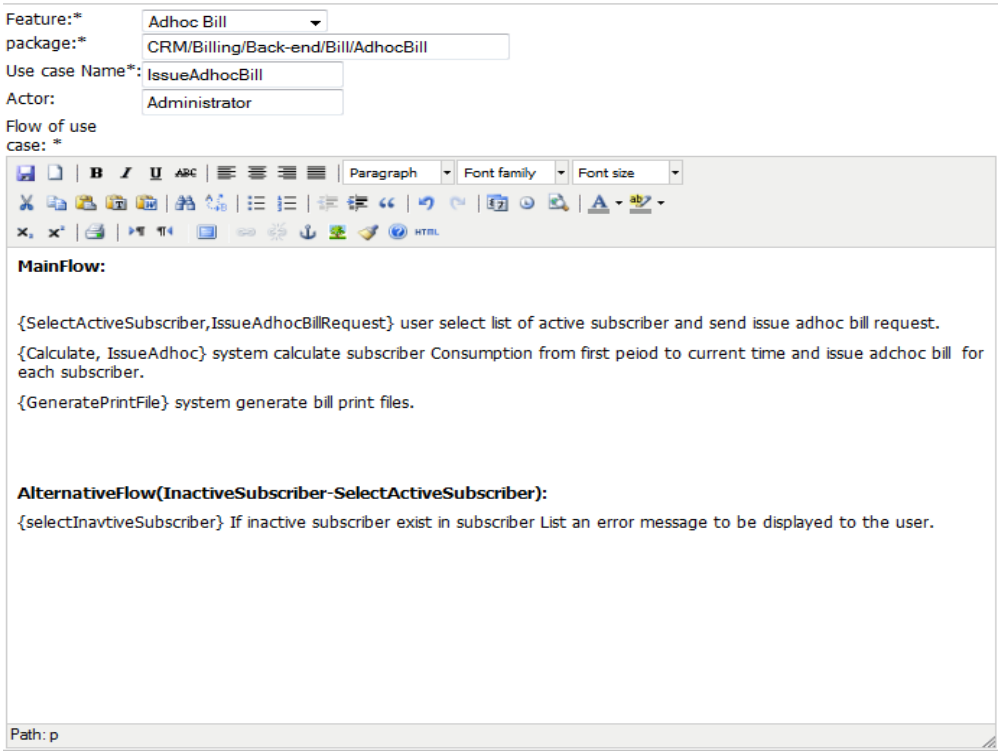

Fig. 11. Adhoc bill use case.

Use cases are defined using our tool along with their relations with features. Each step in the use case scenario has one or more tags. The tags define the tasks which are done in each step. A sample use case definition using our tool is shown in Fig. 12.

5.2. Defining Deltas

Delta definition is similar to the approach proposed in [2]. To prevent engaging user with writing deltas, our tool have the facility of defining deltas through a form. Delta definition form is shown in Fig. 13.

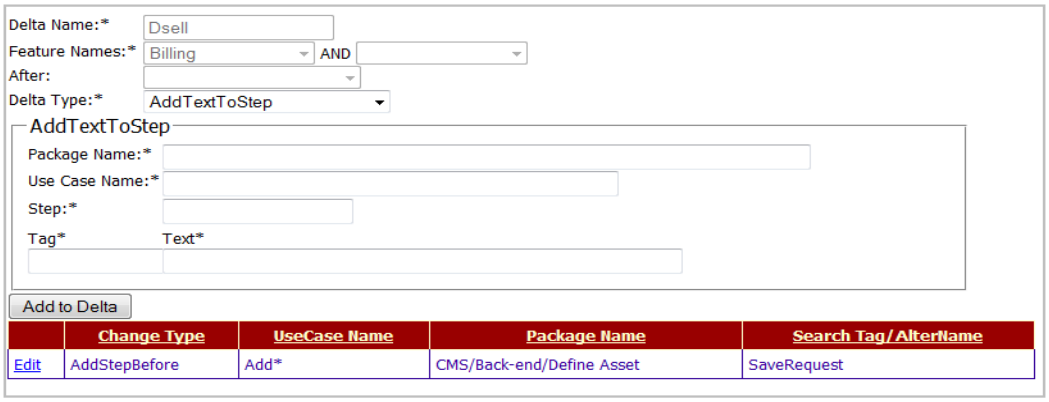

Fig. 12. Delta definition in the tool.

As shown in Fig. 13, the name, the feature names, and the "after" fields should be specified for delta definition. All defined deltas must have a name and a corresponding feature. The second feature and the "after" fields should be specified for features which have an effect or priority to other features. Each delta has a list of changes. For a feature that its effects are scatter in many use cases, one change which affects many use case is defines in delta then, the tool will weave changes on the indicated steps.

5.3. Product Derivation

When configuring a specific product, the list of existing features is displayed and some features are selected based on customer needs. The tool generates documents in HTML format, based on the selected features. Fig. 14 shows how a document is generated using our tool.

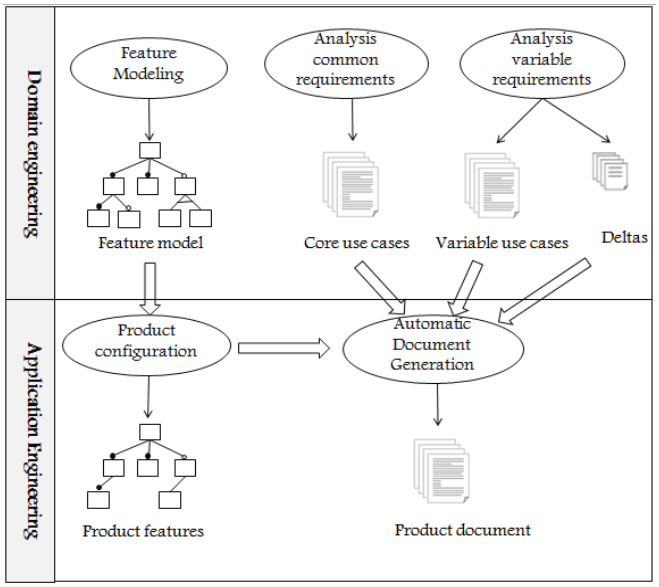

Fig. 13. Delta-oriented description of requirement and configuration.

In the domain engineering phase, common and variable requirements of software product line are identified. Feature model is modeled based on identified requirements. In next step, feature effects are determined. Mandatory features are associated with use cases. Optional features may modify existing use cases or create new ones. Features that change existing use cases are defined using deltas.

In application engineering, a set of features are selected based on customer needs. Product documents are generated according to the selected features and their corresponding use case documents and deltas.

6. Evaluation

To evaluate our proposed approach, we applied it to an industrial case study. In this section, we present the case study and the method that the company uses to manage requirements. The traditional method that is currently used by the company to manage requirements is compared with the delta-oriented approach. Finally, our approach is compared with relevant approaches based on a number of criteria.

6.1. Case Study

Since IPTV can be used in different organizations such as Telco, hotel, educational facilities, hospitals, conference halls, sport stadiums, and so forth, it is a good example of SPL. IPTV back-office system has many features which are selected based on customer needs.

The feature model of IPTV consists of 120 features. We have selected a subset of these features such that it covers different types of features and feature relationships (Mandatory, optional, XOR, and OR). Fig. 1 shows the selected subset of IPTV feature model.

As shown in Fig. 1, the “Content Management System (CMS)”, “upload asset”, “assets” features are mandatory and every IPTV system should support these feature. On the other hand, the “live content” feature is optional. For example, the “live content” feature is not selected for a conference hall’s IPTV solution. The “customize by profile” and “customize by category” features which are sub-features of customize content, are optional as well. These features customize TV content based on subscriber category and profile. They require the “profile” and “category” features of the customer relationship management (CRM) system. CRM subsystem is related to management of subscribers. In some products such as hotel and Telco, subscribers are managed and in some of them such as sport stadium it is not important to manage the subscribers, so this subsystem is also optional. CRM system has two main features (“Billing” and “Subscriber”) and many sub features (“Manage subscriber”, “Profile”, “Categorization”, “Payment”, “Bill”, and “Issue Article”). In the “System Configuration Management (SCM)” the basic setting of IPTV is done therefore this subsystem is mandatory.

6.2. Current Approach of the Company

In the current method of the company, requirements are managed using use case descriptions. Use case descriptions are written in Microsoft Word. If a feature causes adding a new use case, a new use case scenario is written. Features which affect other use case scenarios are applied to relevant documents. Features footprints in the use case scenario are shown by a footnote. The tracks of features are maintained in excel files. The relations between use cases and features are manually manipulated in an excel file.

Fig. 15 shows the "play media request" scenario described using the traditional method. This scenario is affected by four features: “subscriber management”, “customize content by profile”, “customize content by category”, and “billing”.

Note that step 2 is changed completely due to selection of the “customize by category” and “customize by profile” features. Consequently, it is confusing to understand the behavior when these features are not supported. Using the traditional approach makes it difficult to understand the behavior of a specific product using this scenario. In this method, common and variable requirements are tangled and customizing

document based on customer needs is time consuming and error prone. Moreover, managing cross-cutting requirements is an exhausting task. Many documents should be edited for adding cross-cutting features. This defect exists in PLUC and PLUSS but SVCM and the delta-orient approach do not have this problem because they manage cross-cutting features as one task that automatically affects many documents. Another problem with the traditional method, PLUC, and PLUSS is forgetting to edit the document. This causes problems in tracking the effects of features later. It may also lead to inconsistencies. In the traditional method, it is hard to understand the behavior when both of the features that affect the same step of a use case cause are not supported. It causes misunderstanding for customer and other people that use document in next phases.

Use case name: Play request

Actor: subscriber

Main scenario:

1- Subscriber requests list of media.
2- System shows list of active media based on subscriber categories[1] and profile[2].
3- Subscriber chooses relevant media and sends play request.
4- System checks if media purchased[3] by the subscriber then it will be played.
5- System updates the history of played media for subscriber[4].

[1]Related to customize content based on category
[2] Related to customize content based on profile
[3] Related to billing feature
[4] Related to subscriber management feature

Fig. 14. The "play media request" scenario which is described using the traditional method.

6.3. Comparison between Delta Approach and Current Company Method

To evaluate our approach, we constitute a team of four people, consisting two analysts and two programmers. One of the analysts and programmers had experience with the traditional method and the others were just familiar with IPTV domain. The traditional method and the delta-oriented approach for requirement documentation were described for the team. We made sure that they understand both approaches correctly by testing them using a simple example. Thirty use cases which were related to different types of features were chosen. These use cases were related to twenty eight features. We compare the efficiency, required time and documents clarity of the two approaches in the following phases: requirements documentation, understanding requirements by programmer, and product derivation.

6.3.1. Requirements documentation

We asked two analysts to document thirty use cases using two approaches. The time spent by each analyst on writing scenarios and applying changes to use cases was recorded using both approaches. Applying the delta-oriented approach on the case study has the overhead of adding a tag to each step. Nevertheless, it is time-saving when managing cross-cutting concerns. In Fig. 16, the time spent for documentation using each approach is shown. Mandatory features result in creating new use cases and optional features may lead to adding a new use case or changing an existing one.

As shown in Fig. 16, documenting mandatory features using the traditional method is faster than the delta-oriented approach because of the overhead of defining tags for each step. On the other hand, when documenting cross-cutting features that affect many use cases, delta-oriented approach is more efficient. For example, when adding the "Customize by Category" feature, category type should be defined in all asset type definitions. Thus, all use cases which are relevant to asset definition should be edited. In the traditional

method, all of the relative use case documents should be opened and the new step should be added in them. However, in the delta-oriented approach all these tasks are done by defining and applying one delta. Another significant point in Fig. 16 is the time that is spent to document the prepay feature. In this case, the delta-oriented approach is more time-consuming than current method because in former all of the combinations of this feature with other features are considered. "Prepay" and "Post-pay" features have an OR relationship therefore both of them may be selected in a configuration. Thus, in the definition of delta, the combination of these two features should be also considered. Applying the delta-oriented approach imposes little overhead in some cases but brings benefits such as legible and comprehensible documentation in software development cycle.

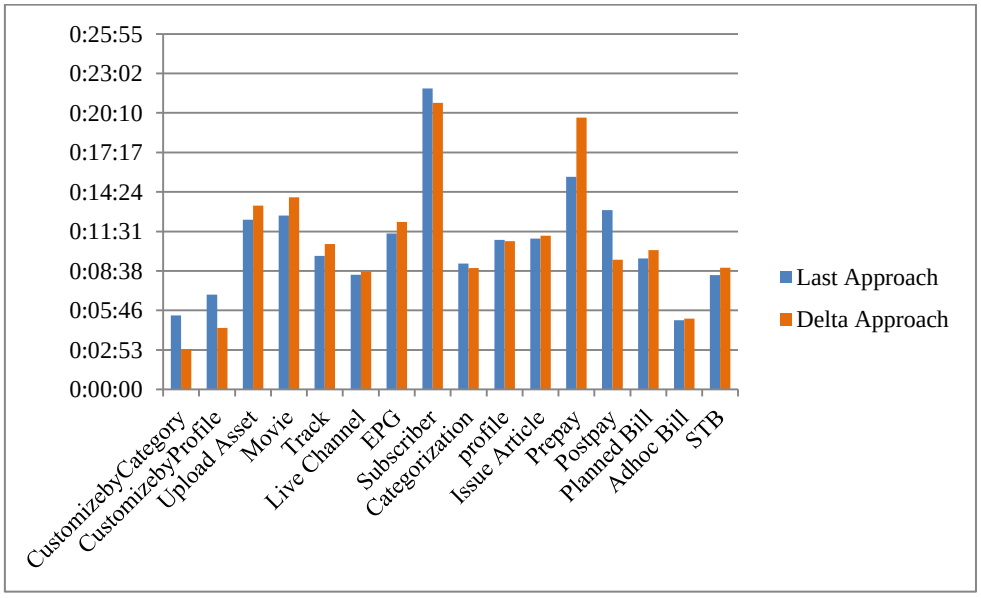

Fig. 16. Comparing the time taken for the analyst in the delta method and the current company method to document requirements.

6.3.2. Understanding the requirement by programmer

We gave the documents generated using the traditional method and the delta-oriented approach to the programmers. To read each document just once and evaluate how much the programmer understood scenarios in each approach, we divide use cases into two groups A and B. Then, we gave the first and second programmers the use case sets A and B accordingly. These use cases were developed using the traditional approach. In the next step, the use cases were developed using the delta oriented approach. Then, we gave the first and second programmers the use case sets B and A accordingly to avoid their previous knowledge bias their understanding of the new use cases. A test is taken from programmers to ensure that they understood scenarios. The results show that most of the problems occur when using the traditional method to change the main steps of a scenario. Understanding a scenario which was modeled using deltas was acceptable. The amount of time that programmers spend to understand relative scenario in two approaches is measured. The results are shown in Fig. 17. As shown in Fig. 17, for a scenario that is not affected by any feature, the programmer studies the relevant document. Thus, in this case, the time spent to understand documents generated by two approaches differs slightly. The time that is spent using the delta-oriented approach is less than the time spent on the traditional method because in latter the programmer should track the feature effects in the excel file, and then find the corresponding use case description in the package. In the delta-oriented approach, the programmer studies the relevant use cases by selecting a feature in the feature model. For the scenarios that are affected by many features, the programmer should

spend more time to understand documents generated using the traditional method because common and variable features are tangled in document. To understand the main scenario, the programmer should overlook the variable parts. In the case of cross-cutting features that affect many scenarios, such as "customize by category" and "customize by profile" features, considerable amount of time is spent to study the documents generated using the two approaches. In the delta-oriented approach, the programmer tracks feature effects by studying the corresponding deltas. However, in the current method, the programmer should track the feature effects using the excel file and following changes on use cases considering use case scenarios.

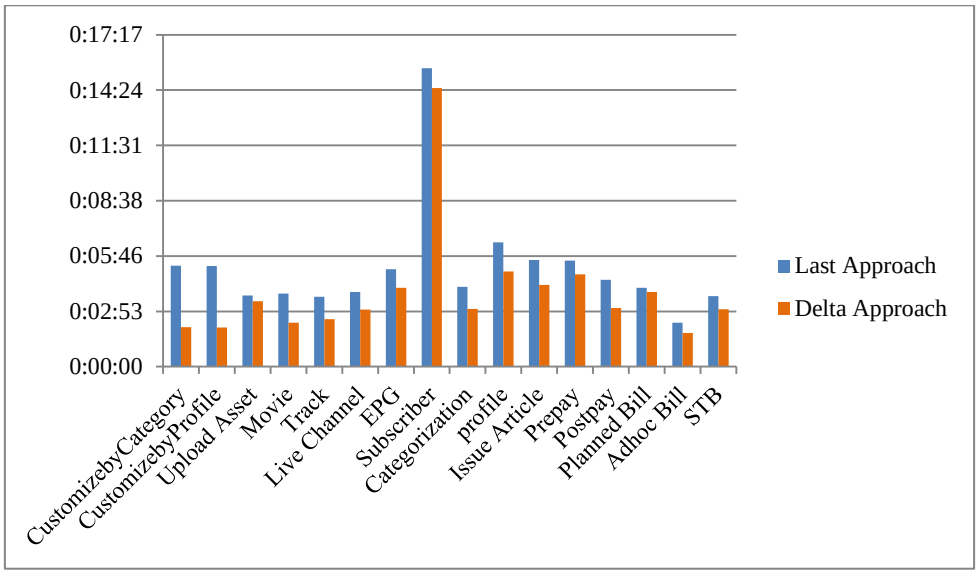

Fig. 17. Comparing the spent time for the programmer to understand a scenario using current method and delta-oriented approach.

6.3.3. Product derivation

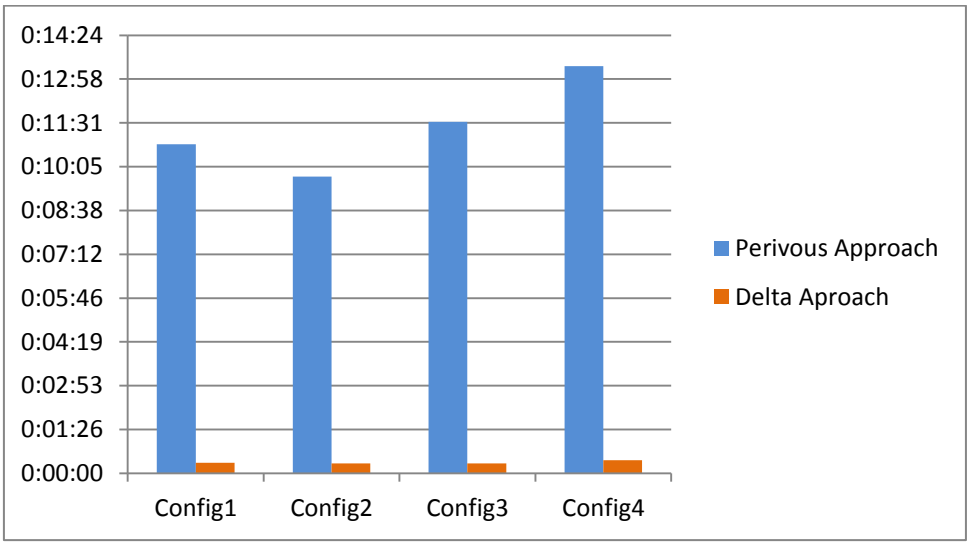

Fig. 18. The time that is spent to generate documents for four configurations using the two approaches.

Our approach has a great performance in automatic generation of documentation. We consider four different configurations, which cover all types of features and they are mostly selected in IPTV derivation, to

evaluate the time spent on generating documents in the two approaches. The results show that the time spent for document generation using the traditional method is significantly higher than our approach. The time spent for document generation by applying the two approaches is shown in Fig. 18. As it is shown in Fig. 18, the amount of time spent to generate documents for different configurations using Delta-oriented approach are nearly the same. In this approach, the documents are generated automatically in a short time. Using the traditional method, the relevant use cases are identified in the excel file based on the selected features. Then, a copy of the corresponding documents is provided and the irrelevant changes are omitted from these documents. This explains the considerable amount of time that is required to generate documents using this method.

6.4. Criteria-Based Evaluation

Table 1. Criterion-Based Assessment of Existing Approaches

criteria	Delta	SVCM	PLUSS	PLUC
Readability	□	□	■	□
Simplicity and expressiveness	■	■	□	□
Type of distinction	◘	◘	■	■
Documentation	□	□	□	□
Dependency	◘	◘	◘	□
Evolution	■	◘	◘	◘
Adaptability	◘	◘	◘	◘
Scalability	■	■	□	□
Support	■	□	■	□
Unification	■	□	□	□
Standardizeability	■	□	□	□

In [18] a comparative study on feature-based notations is provided to model variability in the requirements. In this survey, eleven criteria are proposed to evaluate a feature-based model. These criteria are as follows:

1) **Readability:** The notation which specifies the common and variable parts of SPL, should graphically visualize in a readable form.
2) **Simplicity and expressiveness:** the notation must be simple and expressive. It should be easily understandable by users without additional explanation and contain minimal number of objects.
3) **Type distinction:** Types of variability should be distinguished using a notation. The notation should cover all different types of variability.
4) **Documentation:** The notation must have solution to specify variation point properties. (Binding time, justification of variability, etc.)
5) **Dependencies:** Dependencies between the variable parts of the product line should be represented by the notation.
6) **Evolution:** The evolution of product line must be supported. This means adding new requirements, integrating changes, and updating the existing requirements should be readily applicable.
7) **Adaptability:** The notation must be flexible and fit each company specific needs.
8) **Scalability:** The notation should cover modeling large-scale systems.
9) **Support:** The notation should have tool support and be integrated into existing tools.

10) **Unification**: The notation should be unified into the whole product line development cycle.
11) **Standardizeability:** The notation should be standardizeable: interoperability among users and tools improves when sharing a unique language.

Table 1 shows the results of evaluating approaches discussed in this paper based on the above criteria. The notation that is used in the table is based on [18]. A black square shows full support, a white square means no support, and a black square containing a white circle represents support with limitations.

Readability: PLUC, SVCM, and our approach do not introduce notation to graphically visualize different part of product line. PLUSS provides a way to display feature model which is readable.

Simplicity and expressiveness: Delta-oriented and SVCM methods generate documents automatically based on customer needs which are specified by selecting a set of features. In these approaches, documents are simple and they do not involve any complex notation. PLUSS approach has a complicated notation that understanding it needs additional effort. PLUC does not have a complicated notation. However, the model of using a document for a specific product needs additional explanation.

Type of distinction: PLUSS and PLUC introduce notation which distinguish between types of variability. Our approach and SVCM rely on feature model to distinguish between types of variability.

Documentation: none of the approaches manage variation point properties.

Dependency: PLUC does not mention how to manage dependencies. PLUSS, SVCM, and our approach rely on feature model to manage feature dependencies.

Evolution: in our approach, adding, updating, and changing a feature is managed easily using deltas. In SVCM approach, if a feature affects a step of main use case, the step must be omitted form the use case and be written in advice. In this situation, knowledge configuration table must be updated. If a feature has effects on other features, a row of knowledge configuration table must be changed based on the feature priority. Thus, evolution in the SVCM approach leads to some complexities. In PLUC and PLUSS, adding and changing a feature is possible but applying a cross-cutting feature which affects many use cases is difficult. In these approaches, keeping the documents consistent and compatible requires a lot of attention and activities.

Adaptability: Delta-oriented approach separates core requirements form variable requirements. The effects of a new feature are described using deltas. This approach can be used in every organization by adapting the meta-model which is shown in Fig. 8. As mentioned before, SVCM approach does not support deletion or modification of steps in a scenario so it is not adaptive for all kinds of scenarios and modifications. PLUC becomes very complicated for a large case study with many potential configurations. Such complexity reduces the readability and clarity of the document in such a way that it may not be much useful. In PLUSS, common and variable requirements are tangled thus understanding the behavior of a specific product is difficult, especially in large systems with many possible features.

Scalability: Delta-oriented and SVCM approaches separate common and variable requirements so they are scalable and applicable to large case-studies. In PLUC, variability is specified based on different configuration. Thus, large case studies with a large number of configurations cannot be managed using this approach. Additionally, manipulating documents with this scale of complexity is very difficult. PLUSS approach manages variable and common requirement in a single document. Therefore, with system growth, the readability of the document decreases and document manipulation becomes difficult.

Support: PLUC do not introduce a tool. PLUSS add functions to use rational DOORS. In SVCM a tool is implemented but they do not point how to integrate with tools like kits. Our tool is based on XML which is standard language and can be integrated readily.

Unification: PLUC approach does not support unification as variability is specified based on configurations and tracking the effects of features is almost impossible. In the PLUSS approach, it is very

hard to track the effects of features because variable and common requirements are modeled in a single document. Thus, tracking one feature involves investigating the whole document. In SVCM, the effects of a feature are kept in a configuration knowledge table. This table contains the feature's name and the tasks that should be done for this feature. Use cases on which advice is applied are not traceable from this table and one should review all documents to find joint point that advice weave. In our implemented tool for delta-oriented approach, the effects of a feature are displayed when the feature is selected.

Standardizeability: In PLUC the relation between feature effects in variable part and step scenario is not considered when a feature has different effects on two or more steps, and it lead to conflict. PLUSS uses notation in steps and each step has a specific notation based-on feature types. If two features affect one step or one feature affects more than one step, this approach has also ambiguities such as PLUC. SVCM and our approach separate the main scenarios and features effect. Documents are generated based on customer need so they resolve ambiguities.

7. Related Work

One of the main challenges in software product line engineering is modeling variability. In the past few years a number of approaches have been proposed for this purpose. We have categorized the proposed approaches into three groups: modeling variability in the UML diagrams, use notations to model variability in use case scenarios, and aspect-oriented approaches.

A number of approaches are presented to model variability in UML diagrams. Gomaa [5], de Oliveira Junior, E.A., *et al*. [19], Bragança, A. and Ricardo J. Machado.[20], and Junior, E.A.O., Itana MS Gimenes, and José C. Maldonado [6] use stereotypes to model variability in use case diagram. To model variability in use case diagram, Azevedo, S., *et al*. [21] extend the existing relations and many stereotypes. Braganca, A., and Ricardo Jorge Machado [22] extend the UML use case meta-model to remove all ambiguities. In the proposed approach, they add notes to "extend" and "include" relationships for an automatic mapping between use case diagram and feature model.

Clauß [9] proposes a generic model to manage variability in the class diagram. This approach differentiates between the location of variation points, respective variants, and relation between them. Variation points and variants are shown with <<variation point>> and <<variant>> stereotypes respectively. Ziadi[7] proposes <<optional>>, <<variation>> and <<variant>> stereotypes to model variability in class diagrams. This approach uses OCL (Object Constraint Language) to manage constraints among features.

Alférez, M., *et al*. [23] specify functional requirements using use cases. They model variability using feature models. In this approach, the relations between use cases and features are stored in a table of tracing link. The detailed behavior of a use case is captured through activity diagram. They suggest defining composition rules to show how a variant use case changes the normal execution of a mandatory use case.

In [8], activity diagrams are used to depict the behavior of use cases. The stereotypes <<extension_point>>, <<Inclusion_poin>>, <<region_point>>, <<before>> and <<after>> are considered to improve the reorganization of node activity. In [7], in addition to expanding the class diagram, the sequence diagram is extended as well. To manage variability in sequence diagrams, this approach considers three states: optionality, variation and virtual. Optionality has two aspects: optionality for objects interaction and optionality in interaction that is managed by <<optionalLifeline>> and <<optionalInteraction>>stereotypes. Variation is managed by <<variation>> and <<variant>> stereotypes. In this approach, virtual is depicted with <<virtual>> stereotype and means that this part of the diagram could redefine for each product by another sequence diagram.

Cross-cutting features and features affecting some parts of a use case scenario cannot be shown in diagrams. There are a number of approaches that use notations when describing use case scenario to model

variability. Pohl [1] has proposed an orthogonal method to model variability in textual requirements descriptions. The authors mention that XML and XSL can be used to describe variability in textual requirements clearly. XML models variability in documents and XSL processes XML to generate documents. Bertolino [11] proposes an approach which is based on adding tags to the scenario to model variability. John [24] variability. Eriksson [12] uses "black box Flow of event" to describe use case scenarios and proposes a notation to describe variability in use case specifications. In this category, common and variable parts of a scenario are tangled and it is difficult to understand the behavior of a specific product. Moreover, managing cross-cutting features which affect many scenarios is an exhausting and tedious task which is time consuming.

There are a number of aspect-oriented approaches to manage cross-cutting features. Anthonysamy and Somé [14] propose a method to model use cases using aspect-oriented approach. This approach, introduced a <<variability>> relation and uses Petri net and the UCEd tool. This tool gets a set of use cases in natural language and generates state chart automatically. Bonifácio and Borba [13] propose an approach named MSVCM (Modeling Scenario Variability As Cross-cutting Mechanism). In this approach, a use case model consists of use cases and advices. Advices are weaved in a specific point of the main scenario. The condition causes adding advice in the main scenario is manipulated in configuration knowledge. Weaving process gets SPL use case model, feature model, product configuration, and configuration knowledge. Then, it generates the use case model of a specific product.

8. Conclusion and Future Work

In this paper, we introduced an incremental approach to describe SPL requirements. In our method, a core model consisting of use case documents is built and is associated with the mandatory features in the SPL. Optional features may add new use cases or change existing ones. When an optional feature causes adding a new use case, the use case scenarios is written as mandatory features. To provide a way to specify changes to a use case document, we tag the steps in the use case scenarios. For all products which are derived from product family, mandatory use cases are considered as the core documents. Use cases are added and modified by applying deltas based on the selected features in the given configuration. Our method is supported by a web-based tool which enables automatic derivation of requirements documents for specific products. The implemented tool generates use case description documents with the HTML format based-on the selected features in the configuration. We have successfully applied the proposed approach on a real-world case study and compared it with existing methods of managing requirement. The results show that our approach has advantages such as generating use case scenarios per product automatically, decreasing the cost of documentation, creating readable documents for stockholders, and tracing feature effects. On the other hand, there is an acceptable overhead in the documenting a use case.

A feature of our method is to specify the changes required by the optional features in separate modules which make it scalable in terms of the number of features. Consequently, our method can be used in large projects without cluttering the requirements with lots of variations. As a downside, the variations of a use case cannot be identified from the use case document itself, and the feature model and the delta modules must be considered to find out the variations. To make this easier, the implemented tool provides tracing facility, by which the modeler can trace a feature to its related use cases and features. Thus, the developer can find out which parts of the system are affected by a given feature, system testers can update their test scenarios easily, and the project manager may observe the effects of a feature in the system and estimate the amount of work needed to be done properly.

The modular and incremental nature of our approach enables it to be used to effectively manage different versions of the requirements in a single product or SPL too (known as variability in time). This requires

minor modifications in the method, such as annotating each feature and the corresponding scenarios with version numbers. These version numbers can be used to derive the requirements related to a specific version of the product automatically.

References

[1] Pohl, K. G. B., & Linden, F. V. D. (2005). *Software Product Line Engineering* (pp. 22–34). Springer.

[2] Schaefer, I., Bettini, L., Bono, V., Damiani, F., & Tanzarella, N. (2010). Delta-oriented programming of software product lines. *Software Product Lines: Going Beyond* (pp. 77-91).

[3] Sabouri, H., & Khosravi, R. (2013). Delta modeling and model checking of product families. *Fundamentals of Software Engineering* (pp. 51-65).

[4] Khedri, N., & Khosravi, R. (2013). Handling database schema variability in software product lines. *Proceedings of 2013 20th Asia-Pacific Software Engineering Conference*.

[5] Gomaa, H. (2006). Designing software product lines with uml 2.0: From use cases to pattern-based software architectures. *Reuse of Off-the-Shelf Components* (pp. 440-440).

[6] Oliveira Jr., E. A., Gimenes, I. M., and Maldonado, J. C., (2010). Systematic management of variability in UML-based software product lines. *Journal of Universal Computer Science*, *16(17)*, 2374-2393.

[7] Ziadi, T., Hélouët, L., & Jézéquel, J.-M. (2004). Towards a UML profile for software product lines. *Software Product-Family Engineering* (pp. 129-139).

[8] Braganca, A, & Machado, R. J. (2006). Extending UML 2.0 Metamodel for complementary usages of the «extend» relationship within use case variability specification. *SPLC*.

[9] Clauß, M. (2001). Generic modeling using UML extensions for variability. *Proceedings of the Workshop on Domain Specific Visual Languages at OOPSLA*.

[10] Cockburn, A. (2000). *Writing Effective Use Cases (The Crystal Collection for Software Professionals)*. Addison-Wesley Professional Reading.

[11] Bertolino, A., Fantechi, A., Gnesi, S., Lami, G., & Maccari, A. (September 2002). Use case description of requirements for product lines. *Proceedings of the International Workshop on Requirements Engineering for Product Lines* (pp. 12-18).

[12] Eriksson, M., Börstler, J., & Borg, K. (2005). The PLUSS approach–domain modeling with features, use cases and use case realizations. *Software Product Lines* (pp. 33-44), Springer.

[13] Bonifácio, R., & Borba, P. (2009). Modeling scenario variability as crosscutting mechanisms. *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development* (pp. 125-136).

[14] Varela, P., Araújo, J., Brito, I., & Moreira, A. (March 2011). Aspect-oriented analysis for software product lines requirements engineering. *Proceedings of the 2011 ACM Symposium on Applied Computing*.

[15] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. *DTIC Document*.

[16] Soroush HighTech. Retrieved from http://soroush.net/En/.

[17] Bertolino, A., & Gnesi, S. (2003). Use case-based testing of product lines. *ACM SIGSOFT Software Engineering Notes*, *28(5)*, 355-358.

[18] Djebbi, O., & Salinesi, C. (2006). Criteria for comparing requirements variability modeling notations for product lines. *Proceedings of IEEE Fourth International Workshop on Comparative Evaluation in Requirements Engineering* (pp. 20-35).

[19] de Oliveira Jr., E. A., Gimenes, I., Huzita, E. H. M., & Maldonado, J. C. (October 2005). A variability management process for software product lines. *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research* (pp. 225-241).

[20] Bragança, A., & Machado, R. J. (2005). Deriving software product line's architectural requirements from

use cases: an experimental approach. *Proceedings of 2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, Rennes, France.

[21] Azevedo, S., Machado, R. J., Bragança, A., & Ribeiro, H. (2010). The UML «extend» relationship as support for software variability. *Software Product Lines: Going Beyond* (pp. 471-475), Springer.

[22] Braganca, A., & Machado, R. J. (2007). Automating mappings between use case diagrams and feature models for software product lines. *Proceedings of 11th International Software Product Line Conference*.

[23] Alférez, M., Kulesza, U., Sousa, A., Santos, J. P., Moreira, A., Araújo, J., & Amaral, V. (July 2008). A model-driven approach for software product lines requirements engineering. *SEKE*.

[24] John, I., & Muthig, D. (2005). Tailoring use cases for product line modeling. *Proceedings of the International Workshop on Requirements Engineering for Product Lines*.

Samaneh Zamanifard was born in Tehran, in 1983. She received her B.Sc. degree in software engineering in 2006 from Islamic Azad University South Tehran Branch and now she is a MSc. student in software engineering in Islamic Azad University Kerman Branch. Her research interests include modeling and analysis of software product lines. She also has been working as a system analyst of enterprise systems for about 9 years in various domains, including insurance systems, core banking, and IPTV.

Ramtin Khosravi is an assistant professor at School of ECE, University of Tehran. He received his Ph.D. degree in 2005 from Sharif University of Technology. He has been active in software development industry for 15 years in several domains such as automotive industry, eLearning, and financial domains. His research interests are mainly in modeling and analysis of software product lines and also formal methods in distributed computing.

Hamideh Sabouri was born in Tehran in 1983. She received her Ph.D. degree in 2013 from the University of Tehran. Her research interests are mainly in modeling and analysis of software product lines, formal methods, and static analysis techniques.