11 Product Line Use Cases: Scenario-Based Specification and Testing of Requirements

A. Bertolino, A. Fantechi, S. Gnesi, and G. Lami

Abstract

Use Cases can be employed in system requirements engineering to capture requirements from an external point of view. In product line modeling, commonalities and variabilities of a family of systems have to be described. To this purpose, we have defined extensions and modifications of the Use Cases notation, called Product Line Use Cases (PLUCs). In order to guarantee the conformance of the derived product with respect to the product line we add the capability of expressing constraints over the Product Use Cases that can be derived from a PLUC. Using this notation, it is possible to express in the requirements specification of the product line not only the possible variant characteristics that can differentiate products of the same line, but also which combinations of variant characteristics are "legal" and which are not. Testing is another activity in which PLUCs show their utility. Indeed, for a product belonging to a product line, testing is a crucial and expensive part of software development. Yet the derivation of test cases for product lines has so far received little attention. We outline a simple methodology for this purpose, which relies on the early requirements specification expressed as PLUCs.

11.1 Introduction

The development of industrial software systems may often benefit from the adoption of a development cycle based on the product line engineering approach [5,16]. This approach aims at lowering production costs by sharing an overall reference architecture and concepts of the products, but at the same time allowing them to differ with respect to particular product characteristics in order to, e.g., serve different markets. The production process in product lines is therefore organized with the purpose of maximizing the commonalities of the product line and minimizing the cost of variations [14].

In the first stage of a software project, that is, requirements specification, the information and knowledge of the system under construction is acquired. Chapter 4 addresses this point. When gathering and expressing requirements on a product line two different problems have to be addressed. On one side there is the problem of capturing both requirements common to all members of the product line and requirements valid only for a subset

of products. On the other side there is the problem of specializing and instantiating the generic product line requirements into application requirements for a single product.

To deal with these problems, the relations between line and product requirements have to be handled by the adopted modeling approach, and the concepts of parameterization, specialization and generalization need to be supported by the modeling concepts. Product line requirements can be considered, in general, as composed of a constant and a variable part [1,17,25]. The constant part includes all those requirements that deal with features or functions common to all the products in the product line and, for this reason, do not need to be modified. The variable part represents those aspects that can be changed to differentiate a product from another.

Indeed, a product line can be seen as a set of products with common characteristics that link them together. While developing a product line it is possible to move from the line level (which represents those common features) to the product level (which represents the single product, with all its particular characteristics) by an instantiation process, and on the contrary from the product level to the line level by an abstraction process.

Use Cases [6] are an easy, natural way to express functional requirements of a system. Their popularity derives from the simplicity of their approach: a well structured, easy to understand document written in controlled natural language. Use Cases are widely used in modern industrial development, so it seems natural to try to find an effective way to combine them with the product line paradigm.

In this direction, we have previously proposed the notation of Product Line Use Cases (PLUC) [1,10], an extended version of Cockburn's Use Cases [7] aimed at expressing requirements of product lines. The well-known Cockburn's Use Cases allow the functional requirements of a system to be described, by imposing on requirements documents a specified structure, which separates the various cases in which the system can be used by external actors, and for each case defines scenarios of correct and incorrect usage. The PLUC notation adds variability to Cockburn's Use Cases, with the possibility of expressing variation points and optional parts.

In this chapter, we show how the PLUC notation can be exploited for two fundamental processes in product line engineering:

- The instantiation of a (legal) product from a product line at the early stage of requirements definition.
- The derivation of a scenario-based test plan for a product of a product line.

Moreover, in [10] it has been shown how PLUCs can also support the abstraction process for the definition of a product line from product instances.

The first issue is addressed by providing a PLUC with the capability to express constraints over the product-related Use Cases that can be derived from it. These constraints are expressed as Boolean conditions associated to the variation points. The information we add to PLUCs by means of such constraints provides on the one hand the ability of automatically checking whether a product-related Use Case is conformant to the product line requirements; on the other hand, the adoption of constraint-solving techniques may even allow for automatic generation of product-specific Use Cases from the line level Use Cases document.

The importance of the second issue we address in this chapter comes from the observation that testing takes a predominant amount of development resources and schedule. Therefore, also reuse of test assets is a crucial issue in production processes. And, in the same manner that a product line specification and design must tackle variability, the same

need applies for testing. As evident from the discussion above, the phase in which the majority of variation points are introduced is the requirement specification phase. Accordingly, we believe that planning ahead for testing within the product line development must start from the requirements. Hence, we base the testing process of product lines back on the requirement specification, and in particular on the PLUC notation. We defined the PLUTO methodology to derive specific test cases for product lines, and to instantiate the line generic test plan into a suite of test scenarios for a specific product.

In Sect. 11.2, we present the proposed PLUC notation, with some examples of PLUC described using this notation; in Sect. 11.3 we show how to exploit the information of PLUC to support the derivation process of products conforming to the product line constraints. Section 11.4 discusses how PLUCs can be exploited to derive test cases. Section 11.5 presents related works, while Sect. 11.6 concludes the chapter.

11.2 PLUC Notation

Use cases are widely used in modern industrial development for early requirements elicitation and specification, so it seems natural to try to find an effective way to combine them with the product line paradigm.

A Use Case defines a goal-oriented set of interactions between external actors and the system under consideration. Actors are parties outside the system that interact with the system. An actor may be a class of users, roles users can play, or other systems. There are two kinds of actors: primary actors and secondary actors.

- A primary actor is one having a goal requiring the assistance of the system
- A secondary actor is one from which the system needs assistance

A Use Case is initiated by a primary actor to achieve a goal, and completes successfully when that goal is satisfied. It describes the sequence of interactions between actors and the system necessary to accomplish the task that will lead to the goal. Use Case descriptions also include possible extensions to this sequence, e.g., alternative sequences that may also satisfy the goal, as well as sequences that may lead to failure in completing the service in case of exceptional behavior, error handling, etc. The system is treated as a "black box"; thus, Use Cases capture who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internals. A complete set of Use Cases specifies all the different ways to use the system, and therefore defines the whole required behavior of the system.

Generally, Use Case steps are written in an easy-to-understand, structured narrative using the vocabulary of the domain. An instance of a Use Case is a scenario, and represents a single path through the Use Case. Thus, there exists a scenario for the main flow through the Use Case, and as many other scenarios as the possible variations of flow through the Use Case (e.g., triggered by options, error conditions, security breaches, etc.). Scenarios may also be depicted in a graphical form using UML Sequence Diagrams.

Figure 11.1 shows the template of the Cockburn's Use Case taken from [7]. In this textual notation, the main flow is expressed, in the "Description" row, by an indexed sequence of natural language sentences, describing a sequence of actions of the system. Variations

are expressed (in the "Extensions" row) as alternatives to the main flow, linked by their index to the point of the main flow from which they branch as a variation. This natural language form of Use Cases has been widely used in industrial practice to specify Use Cases, e.g., at Nokia [9].

USE CASE #	<the name is the goal as a short active verb phrase>	
Goal in Context	<a longer statement of the goal in context if needed>	
Scope & Level	<what system is being considered black box under design> <one of: Summary, Primary Task, Sub function>	
Preconditions	<what we expect is already the state of the world>	
Success End Condition	<the state of the world upon successful completion>	
Failed End Condition	<the state of the world if goal abandoned>	
Primary, Secondary Actors	<a role name or description for the primary actor>, <other systems relied upon to accomplish Use Case>	
Trigger	<the action upon the system that starts the Use Case>	
Description	**Step**	**Action**
	1	<put here the steps of the scenario from trigger to goal delivery, and any cleanup after>
	2	<...>
	3	
Extensions	**Step**	**Branching Action**
	1a	<condition causing branching> : <action or name of sub-Use Case>
Sub-Variations		**Branching Action**
	1	<list of variations>

Fig. 11.1. Use Cases template

In [1] we extended the classical Use Case definition given by Cockburn to product lines, adding variability to this formalism. The proposed extension is based on the inclusion of tags that indicate those parts of the product line requirements that need to be instantiated for a specific product in a product-specific document. For doing that, tags are included into the Use Case sections (main scenario, extensions, etc.) in order to identify and specify variations.

This extension is called PLUC, while Product-related Use Cases where all tags have been instantiated are called Product Use Cases (PUC).
The tags can be of three kinds:

– Alternative: They express the possibility to instantiate the requirement by selecting an instance among a predefined set of possible choices, each of them depending on the occurrence of a condition.

– Parametric: Their instantiation is connected to the actual value of a parameter in the requirements for the specific product.
– Optional: Their instantiation can be done by selecting indifferently among a set of values, which are optional features for a derived product.

The instantiation of these types of variabilities will lead to a set of different product-related Use Cases. Although mostly significant in scenario descriptions, tags can be inserted in each field of a Use Case, thus leading to variability of actors, preconditions, etc.

Two examples of a PLUC are provided in Figs. 11.2 and 11.3. These PLUCs apply to different mobile phones belonging to a same PL. We assume that the products differ at least for the set of games made available to the user and for the provision or not of WAP connectivity.

The example in Fig. 11.2 describes the behavior of the phones belonging to the product line when a game is played by the user, while the example in Fig. 11.3 describes the function of answering an incoming call.

PL USE CASE GamePlay

Goal: Play a game on a **[GP0]** Mobile Phone and record score
Scope: The **[GP0]** Mobile Phone
Level: Summary
Precondition: The **[GP0]** Mobile Phone is on
Trigger: Function GAMES has been selected from the main menu
Primary actor: The Mobile Phone user
Secondary actors: The {**[GP0]** Mobile Phone} (the system)
The Mobile Phone Company
Main success scenario
1. The system displays the list of the {**[GP1]** available} games
2. The user selects a game
3. The user selects the difficulty level
4. The user starts the game and plays it until completion
5. The user records the score achieved {and **[GP2]** sends the score to Club XXX via WAP}

Extensions
1a. No game is available:
1a1. return to main menu
3a. The user starts the game and plays it until an incoming call arrives. **See CallAnswer**.

Variations
GP0: Alternative:
0. Model 0
1. Model 1
2. Model 2

GP1: Parametric
if **GP0**=0 then display msg "No game available"

else if **GP0**=1 then Snake II or Space Impact

else if **GP0**=2 then Snake II or Space Impact or Bumper.

GP2: Optional
when **GP0**=2

Fig. 11.2. Example of a Use Case in the PLUC notation

As shown in the examples, the variation points within the Use Case are enclosed within curly brackets, and the tags are identified by proper labels ([GP*i*] for GamePlay PLUC in Fig. 11.2 and [CA*i*] for CallAnswer PLUC in Fig. 11.3). Moreover, the possible instantiations of the variable parts and the type of the variations are defined within an ad hoc Variations section within the PLUC.

A product line definition is given by a set of PLUCs describing the various (generic) requirements for all the derivable products.

When considering the repository of all Use Cases specified for a product line, it can happen that some scenarios in a PLUC depend on other scenarios in another PLUC. In other words, some functional requirements may span across several Use Cases, bypassing the modeling capabilities of the simple formalism of PLUCs seen so far. We refer to these requirements as *cross-cutting* features. We handle cross-cutting in a simple way: When a scenario in a PLUC interacts with a scenario in another PLUC, we introduce a textual note like "see PLUC name." This is for instance the meaning of the note "See CallAnswer" within the GamePlay PLUC of Fig. 11.2, i.e., if an incoming call arrives as the user is playing a game, the related steps to be undertaken can be found in the CallAnswer PLUC.

11.2.1 Specification of a PLUC

The specification of the tags into a PLUC is a critical step for making the PLUC approach effective in practice. The examples we have shown in Figs. 11.2 and 11.3 just refer to single use cases, each of which is intended to give all the possibilities foreseen within the product line for the particular function described by the use cases. The derivation of a product will amount to the instantiation to a given value of all the tags of all the PLUCs of the product line: However, not all the combinations of values will be feasible, or "legal," products. Some more information is needed at the level of the PLUC definition in order to set some constraints on the variability of the tag values. This requires a method to formalize the three kinds of tags described in Sect. 11.2 (Alternative, Optional, and Parametric), as a necessary preliminary step for the verification of the compliance of a PUC to the product line constraints. In fact, the constraints that characterize the products belonging to a product line can be expressed in terms of the relations among the different tags indicating the variation points, both belonging to a single PLUC, and belonging to several PLUCs (thus addressing cross-cutting features).

To express the variability tags of the PLUCs in a formal way we have to take into account all the possible situations that can arise during the writing of a PLUC, paying particular attention to the variable tags of the PLUC itself.

First of all, we have to define the formalism to be used for expressing those relationships:

1. A tag is a variable which can assume any value inside a domain (often it is a finite, explicitly enumerated domain). As already shown in the examples, for readability we denote tags with the abbreviation of the PLUC name and a number (e.g., CA0).

PL USE CASE CallAnswer
Goal: Answer an incoming call on a **[CA0]** Mobile Phone
Scope: The **[CA0]** Mobile Phone
Precondition: Signal is available; Mobile Phone is switched on
Trigger: Incoming call
Primary actor: The user
Secondary actors: The {**[CA0]** Mobile Phone} (the system)
The Mobile Phone Company
Main success scenario
1.The user accepts the call by pressing the Accept button
2. The system establishes the connection by following the {**[CA1]** appropriate} procedure.
Extensions
1a. The call is not accepted:
1a.1. the user presses the Reject button
1a.2. scenario terminates
PL Variability Features
CA0: Alternative:
0. Model 0
1. Model 1 **[CA2]**
2. Model 2 **[CA2]**

CA1: Parametric:
case **CA0** of
0: Procedure A:
2.1 Connect Caller and callee
1 or 2: if **CA2=** a then Procedure B
2.1 Interrupt the game
2.2 Connect Caller and callee
else if **CA2=** b then Procedure C:
2.1 Save current game status
2.2 Interrupt the game
2.3 Connect Caller and callee

CA2: Alternative:
a. games available, but if interrupted status is not saved
b. games available, and if interrupted status is saved

Fig. 11.3. Another PLUC example

2. A tag predicate is a Boolean proposition asserting the value of a tag, such as (CA0 == 1), or an expression connecting such propositions using classical propositional connectives. We use the symbols "||" (the logical OR operator), "&&" (the logical AND operator) , "==" (the "equal to" logical operator), "=>" (the logical implication operator) and "~" (the logical NOT operator). We denote tag predicates with a name such as CA0_tag.
3. A tag predicate for a tag may include propositions about other tags, so to define relationships between the values of the tags. Moreover, other expressions can set constraints over the tag's values; such constraints can span over more than one PLUC.

Using this formalism we can describe the essential types of tags by a logical expression able to capture their meaning:

– *Alternative tag* indicates mutual exclusion, which means that during the instantiation process one and only one from a set of different values can be assigned to the tag. This type of relationship can be expressed with a logical Exclusive or.

- *Optional tag* represents a subset of a PLUC steps that can or cannot be present in an instantiated PUC, depending of the value of some other instantiated tag (i.e., if a mobile phone type contains game C, the PUC called "starting a game" will have a step "print GAME C on screen," otherwise this step will not be present in the PUC). The propositional connective that models this type of relationship is Implication.
- *Parametric tag* indicates that some subsets of PLUCs steps can be chosen so that at least one of them will be chosen for a specific PUC, but more than one is allowed to be chosen (i.e., there can be more than a way to start a game in a mobile phone interface, and at least one must be present). This relationship is modeled with a Logical or.

The two examples of PLUCs shown in Figs. 11.2 and 11.3 can be used to show the process to be followed to represent the tags indicating variability in a formal way using the formalism described above. For each of the variability tags in the two PLUCs we derive a logical expression:

GP0_tag (alternative):	(GP0 == 1 XOR GP0 == 2 XOR GP0 == 0);
GP1_tag (parametric):	((GP0 == 0 && GP1 == "display msg "No game available"") || (GP0 == 1 && GP1 == "Snake II or Space Impact") ||(GP0 == 2 && GP1 == "Snake II or Space Impact or Bumper"));
GP2_tag (optional):	(GP0 == 2 => GP0 == "and sends the score to Club XXX via WAP ") || ((GP0 == 1 || GP0 == 0) => (VGP2 == null));
CA0_tag (alternative):	(CA0 == 1 XOR CA0 == 2 XOR CA0 == 0);
CA1_tag (parametric):	(CA0 == 0 && CA1 == "procedure A") || ((CA0 == 1 || CA0 == 2) && CA2 == a && CA1 == "procedure B") || ((CA0 == 1 || CA0 == 2) && CA2 == b && CA1 == "procedure A");
CA2_tag (alternative):	(CA2 == a XOR CA2 == b)
GP-CA-constraint:	CA0_tag == GP0_tag

The last expression is actually a constraint that relates two PLUCs: In this case this constraint simply states that the first tag is actually common to the two PLUCs.

Due to the expressive power of propositional calculus, it is possible to define some more complex and structured relationships, which can be used to more easily describe some common situations we can find when we read through a PLUC. We have just considered those kinds of expressions that define the three types of tags we have identified. A deeper analysis of the needs of actual applications of PLUCs may enlighten the need for other types of tags that should be analogously formalized.

The constraints that define the borders and the characteristics of a product line and that must drive the specification of a PUC are expressed by means of the formalization of the tags as seen above. These tags may be considered as the way to represent the conditions to be satisfied in order to make a variability solution not contradictory with the product line characteristics.

In summary, a PLUC describes the general behavior which all products should yield during the accomplishment of a specific task: It acts like a template from which it is possible to derive single PUCs by the instantiation process of its tags, which can be of many different types.

11.3 PUC Derivation from PLUC

In this section we describe our approach to effectively verify the compliance of a PUC to the product line constraints. Our approach is in fact inherently conceived to handle closed product lines, where it is intended that application engineering does not change the requirement model. On the other hand, the verification of conformance during tag instantiation, following the principles described below, provides the application engineers with a means to detect those cases in which this could happen, and to identify the requirement parts that should be changed to allow for the design of the application outside the product line.

The process of instantiating tags consists of assigning an actual value to each variable appearing in the tag expressions of PLUCs we are interested in. The instantiation of the tags expressing the variabilities of the product line corresponds to the definition of the compulsory characteristics of the PUC we are deriving. In other words, the instantiation of the tags defines the requirements of a particular product belonging to the product line.

A possible instantiation of the tags of the two PLUCs in Figs. 11.2 and 11.3 is:

```
CA0 == GP0 == 1
CA1 == "procedure A"
CA2 == b
GP1 == "display msg "No game available""
GP2 == null
```

This instantiation produces two PUCs derived by the two given PLUCs. A PUC is compliant to the product line if, evaluating the tags expressions defining the constraints in the product line with the instantiation of variables given for that PLUC, all the tags are evaluated true. Otherwise, the PUC cannot be accepted as belonging to the product line: an inconsistent PUC has been identified. The expressions having value false indicates the points of the instantiation determining the non-compliance. Then it is simple to identify those instantiation to be modified to achieve the compliance to the product line constraints.

In the example the value of tag expressions of the PLUC with the actual values of the variables for the considered instantiation are:

```
CA0_tag: true
CA1_tag: true
CA2_tag: true
GP0_tag: true
GP1_tag: false
GP2_tag: true
```

This means that a PUC with the variabilities solved with the above values does not describe any valid product of the product line. In this case the lack of compliance is easily identified as the erroneous instantiation of GP1.

One of the main merits of the methodology we have described is the ease of inserting changes in product line requirements expressed by means of PLUCs. In fact, if a tag is modified, because of the parametric nature of the approach, the effects of the modification affect only its definition and not its individual occurrences over the PLUCs. Moreover, if some new tags have to be added, the effort for doing that is mainly concentrated on the corresponding formal definition, and, once the new tag formula has been defined, the updating of the product line requirements simply consists in the inclusion of the tag at the appropriate place of the affected PLUCs.

We note that our approach when used for the instantiation process (from product line to product) allows a designer to enforce closed PLs, i.e., it prevents the insertion of requirements which are not allowed. Then in this sense it is conceived for closed product lines. On the contrary, it is interesting to note how the described methodology can also be used for supporting the impact analysis of possible new variabilities on the existing (or planned) products belonging to the product line. When a new variable feature is to be added in the product line, it is of interest to evaluate its impact on the whole set of the products of the product line. In particular, for evaluating if the new variability will determine incompatibility with some of the existing or planned products of the product line, a preliminary verification can be made adopting the verification procedure shown above.

This approach is promising due to its simplicity and effectiveness for being implemented in an automatic way. In fact, it gives the advantage of an explicit identification of the variability points in a product line requirements specification by means of the tags.

This characteristic may strongly facilitate the application of our approach in the industry because it allows the use of automatic tools for the identification of variabilities. As an example, a tool can be built able to generate *all* the admissible PUCs from the PLUC, by assigning to tags all the combination of values admitted by the tool: This tool may be useful to explore the possibilities given by possible software products in a product line, before actually building them.

11.4 Using PLUCs for Derivation of Test Scenarios

We have addressed so far how PLUCs can help address variabilities and commonalities during the upfront stages of development, i.e., modeling and specification of Use Case scenarios. Commonalities and variabilities of course also affect test planning: In fact, when considering a line, a test plan consisting of a generic frame of test cases pertaining to the PL domain can be derived. In other terms, the line generic test plan includes a list of test cases that apply to the whole set of admissible products, plus other test cases which instead will vary for each specific product, depending on how the variants characteristics are instantiated. At the product level, then, a methodology should support testers in instantiating from the generic PL test frame the set of test cases relative to the specific product, inclusive of common and variable test features.

11.4.1 PLUTO: A Methodology to Derive Test Scenarios

PLUCs can provide a useful means for the above goal: Based on the PLUC formalism, we have developed a simple and intuitive methodology for the early derivation of test scenarios from the PL requirements specification, called PLUTO (Product Lines Use Case Test Optimization) [2].

The PLUTO methodology is inspired by the well-known Category Partition (CP) method [28], but expands it with the capability to handle PL variabilities and to instantiate test cases for a specific product. In the following we illustrate the CP method, and how this has been modified in PLUTO to handle PLUCs variabilities and commonalities. A remark is noteworthy: We generically speak in terms of "test cases," for readability. However, this is not compliant with the common meaning of a test case in the testing literature. A test case should consist of the precise specification of a test input, a sequence of events and the expected output. We deal rather with *abstract descriptions of test scenarios*: What we derive are not test cases, but scenarios of use that need to be tested for validating that the user requirements are satisfied. Being derived from the Use Cases description, which are high level and in natural language, both the input sequence and the expected behavior are provided at a quite high level of description (the same one in the considered scenario). A refinement process from these abstract descriptions to more concrete ones is needed for obtaining executable test cases. This is outside the scope of the current chapter, but a method for test case synthesis from test scenarios can for instance be found in Chap. 12. CP is a well-known and quite intuitive method proposed in the late eighties to derive functional tests from the specifications written in structured, semiformal language. CP provides a systematic, formalized approach to *partition testing* that is one standard functional testing methodology. Generally speaking, partition testing is based on the simple idea that the input domain is first divided into several equivalence classes (also called partitions, although to be true partitions these should be non-overlapping, which is rarely the case in practice); then one or few tests are selected from within each of the identified partitions, as representative of the behavior of the whole class.

CP is organized into a stepwise methodology. The first step is to analyze the system requirements to identify the functional units that will constitute the subjects of the test and can be considered separately. In the case of PLs the elementary units of analysis are naturally provided by the PLUCs.

Then, for each functional unit (here a PLUC), the tester identifies the environment conditions (the required system properties for a certain functional unit) and the parameters (the explicit inputs for the unit) that are relevant for testing purposes: these are called the *categories*.

For each category, the significant (from the tester's viewpoint) values that it can take are then selected, called the *choices*. A suite of test cases is finally obtained by taking all the possible combinations of choices for all the categories.

As the approach is based on structured, natural language requirements, the test derivation has to be done partially manually. In particular, the identification of relevant Categories and of the Choices to be tested is left to the tester's skill and judgment, and then this constitutes the most critical step of the approach. However, lexical and syntactical analyzers for natural language requirements [3,12] could be used to extract useful information to identify the relevant Categories. This could be augmented with pragmatic hints derived from the specific meaning of fields forming a Use Case. Moreover, this step has been empirically studied, leading to the identification of common mistakes made by testers and to the compilation of a relative checklist [4].

To prevent the construction of redundant, not meaningful, or even contradictory, combinations of choices, in CP the choices can be annotated with *constraints*, which can be of two types: either (i) properties or (ii) special conditions. In the first case, some properties are set for certain choices, and *selector* expressions related with them (in the form of simple *if* conditions) are associated with other choices: A choice marked with an *if* selector can then be combined only with those choices from other categories that fulfill the related property. The second type of constraints is useful to reduce the number of test cases: some markings, namely "error" and "single," are coupled to some choices. The choices marked with "error" and "single" refer to erroneous or special conditions, respectively, that we intend to test, but that need not to be combined with all possible choices. The list of all the choices identified for each category, with the possible addition of the constraints, forms a Test Specification. It is not yet a list of test cases, but it contains all the information necessary to instantiate them by unfolding the constraints.

A specific characteristic of test cases derived from Use Cases is the presence of several scenarios, i.e., the main success scenario and in addition the possible extensions. Of course all of them must be exercised during testing. Therefore a Test Specification derived from PLUCs will normally include a category "Scenarios," in which all the specified scenarios are listed as choices.

Finally, when considering PLs, the CP method described above must be adapted for dealing with the presence of the tags included in the PLUC to identify the PL variation points. However, this can be done in a quite intuitive way: We use the tags similarly to the original concept of CP constraints, i.e., in the Test Specification we associate to the corresponding choices the variability tags; then, in the process of test case derivation we match the tag values in such a way to establish the combinations that are significant with respect to a specific product. In particular, in case of:

- An alternative tag: the relevant feature is selected
- An optional tag: the corresponding feature is taken into account or not depending on whether it is present in the product
- A parametric tag: the feature corresponding to the pertinent value is taken

Note that actually parametric tags do not directly contribute to the task of identifying the test scenarios: In fact, they do not identify possible points of selection, but rather assign the appropriate values once some other related tags are fixed.

When dealing with PLUCs, to express the *selectors*, since these are here used to express relations over tag values, we continue to adopt the formalism of the logical expressions introduced in Sect. 11.3.1. Hence properties over categories in PLUTO are expressed as constraints over tags.

Conceptually, the suite of all potential test cases for a PL encompasses all those combinations of choices that are common throughout the product line and are given by those test cases that do not include variability tags. In addition to these, all the possible combinations of choices involving tags form a set of variable test cases. The complete set of mandatory and variables test cases, which would be obtained in this way, form the asset of test scenarios for the line.

In PLUTO we do not derive the list of all admissible PL test cases; rather we derive the PL Test Specification and leave it unfolded. The test cases are actually derived for a specific product after having instantiated the tags in each PLUC to the appropriate values.

More precisely, for each Test Specification relative to a PLUC, a different set of test cases will correspond to every specific product of the PL, depending on the tag values. We observe that this intermediate step of tag instantiation between the definition of the Test Specification and the derivation of the test sets is the means by which in PLUTO we tackle variability. For readers familiar with the CP test method, this is also what makes PLUTO basically different from the traditional CP. In the latter, only one set of test cases directly correspond to each Test Specification. In PLUTO, from each Test Specification several different sets of test cases can be instantiated, depending on the tag values.

Considering the testing process, the PLUTO approach addresses the stage of testing for validation of user requirements, i.e., it can be used to support Acceptance testing against the documented usage scenarios during application engineering to make certain that the application works according to the expectations of the targeted users. Such test cases are executed as Input/Output black box tests on the completed system. Along the application engineering process, they should be complemented with other test stages addressing unit and integration testing.

PLUTO could nicely be complemented with the ScenTED approach described in Chap. 13. Such an approach is conceived to derive application test cases for system and integration test levels. Moreover, unit test techniques should also be considered for components.

11.4.2 An Example

For illustration purposes, we now apply the PLUTO approach to the GamePlay PLUC in Fig. 11.2. As a first step, from an analysis of it we identify the following Categories: “Mobile Phone Model,” “Games,” “Difficulty Level,” and “Club,” plus of course “Scenarios,” which is always present. These identify the relevant characteristics to be varied when testing the Mobile Phone system for validating the user requirements with respect to the functionality of playing games.

We proceed by partitioning these categories into the relevant choices, i.e., we single out for each of the categories the values that are the relevant cases to be considered in specific

tests. As said, when applying the CP method to PLs, in general we will have that some of the choices will be available for all the products of the product line. On the other hand, some of the categories are specialized into choices that depend on the specific product considered. For instance, the category "Club," which relates to the capability to exchange the achieved game score with other Club affiliates, is relevant only for those models that support WAP connection. Hence it cannot be tested for any potential applications of the product line, but only for those supporting this feature. This is specified in the GamePlay PLUC by means of the GP2 optional tag. Hence, when the test cases are being derived, we make use of this tag similarly to the "constraint" formalism of the CP method. As shown in Fig. 11.4 we derive the two possible choices pertaining to the "Club" category, but we annotate them with an appropriate selector, which is a simple condition stating that these choices are of interest only when the tag GP0 takes value 2, i.e., the Mobile Phone is Model 2. The complete Test Specification is shown below in Fig. 11.4.

If we now applied to this Test Specification a generator that takes out all the possible combinations of choices, we would obtain a long list of test cases. This list would include all the potential test cases for all the products of the line relative to the PLUC under consideration. However, what is more interesting in our opinion is that we can instead derive directly a list of test cases for a specific product of interest. This is obtained easily by just instantiating the relative tags. So, for instance, if we are interested to test the Model 2 product of this line, we set the related optional tag to true (recall from Sect. 11.3.1 that this is modeled by Implication) and derive all and only the combinations that remain valid.

PLUC GAMEPLAY TEST SPECIFICATION
[GP0]: Mobile Phone Model:
0. Model 0
1. Model 1
2. Model 2

Games:

None	GP0 == 0
Snake II	GP0 <> 0
Space Impact	GP0 <> 0
Bumper	GP0 == 2

Difficulty Level: GP0 <> 0
easy
medium
expert

Scenarios:

Main	GP0 <> 0
ext: no game available	GP0 == 0
ext: a call arrives	see CallAnswer [single]

[GP2]: Club:

WAP connection on	GP0 == 2
WAP connection off	GP0 == 2

Fig. 11.4. Main test categories for the GamePlay PLUC

As an example, we list below in Fig. 11.5 some of the test cases that would be thus so obtained for different products, i.e., for different tag assignments. We show these as abstract descriptions and leave to the reader the obvious transformation of these into the corresponding functional test scenarios.

GP0 == 2

Tj1:
Mobile Phone Model: Model 2
Games: Snake II
Difficulty Level: easy
Scenarios: main
Club: WAP connection on

Tj2:
Mobile Phone Model: Model 2
Games: Bumper
Difficulty Level: expert
Scenarios: main
Club: WAP connection on

……..

Tk:
Mobile Phone Model: Model 2
Games: Space Impact
Difficulty Level: medium
Scenarios: ext: a call arrives - *see CallAnswer*

Fig. 11.5. Some test scenarios

In Fig. 11.5 the test cases Tj1, Tj2 refer to a simpler situation in which the features in a PLUC do not depend on the features of another PLUC. Test Tk instead needs further consideration. It considers the choice "a call arrives" of the Scenarios category, which has a specific "See CallAnswer" annotation. This is an example of a cross-cutting feature, whose notion we have introduced in Sect. 11.2. We now see below how this can be handled in the PLUTO methodology.

11.4.3 Extending the Methodology

Referring to the example used so far, let us suppose that the Mobile Phone PL under consideration provides for some applications the capability to save the current status of a game being played in the case that an incoming call arrives. The user may answer or refuse the call. Then, after the communication is closed, the game can be resumed from the status in which it was interrupted.

This case depicts a cross-cutting feature arising from a functional dependency between the GamePlay PLUC and another Use Case, the CallAnswer PLUC, that describes the handling of incoming calls and that we have already presented in Fig. 11.3. Considering now the CallAnswer PLUC (independently from the GamePlay PLUC), we assume we

have already derived a Test Specification by applying to it the PLUTO methodology, as shown in Fig. 11.6.

PLUC CALLANSWER TEST SPECIFICATION
[CA0]: Mobile Phone Model:

0. Model 0
1. Model 1
2. Model 2

Saving:
a. game status is not saved CA0 <> 0
b. game status is saved CA0 <> 0

Scenarios:
Main: Call is accepted
ext: Call is refused

Fig. 11.6. Main test categories for the CallAnswer PLUC

Similarly to what we have done for GamePlay, if we take all the potential combinations of choices in the CallAnswer Test Specification, in respect of the associated constraints, we would obtain the list of test scenarios relative to this PLUC. It is clear however that the PLUCs GamePlay and CallAnswer are related with respect to the possibility to interrupt and then retrieve a game play because a call arrives. To identify that a dependency exists, as said, when we elicited the Use Cases we have annotated the related scenario in the GamePlay PLUC with the note "See CallAnswer." Correspondingly, in the process of deriving the test cases from the GamePlay Test Specification (see Fig. 11.4) the case that a call arrives is contemplated in all those tests in which for the "Scenarios" category the choice "ext: a call arrives" is taken. In Fig. 11.5 the test case Tk for instance selects this choice (we report it again below):

Tk:
Mobile Phone Model: Model 2
Games: Space Impact
Difficulty Level: medium
Scenarios: ext: a call arrives - *see CallAnswer*

However, as described in the CallAnswer PLUC, when a call arrives several behaviors are possible. This test hence is not complete: It must be further refined into several related test cases, considering each of the possible combinations of choices offered in its turn by the CallAnswer Test Specification. Hence for example from the above Tk, considering the Test Specification relative to the CallAnswer PLUC (Fig. 11.6), we get at least four refined test cases as follows:

Tk-1:
Mobile Phone Model: Model 2
Games: Space Impact
Difficulty Level: medium
Scenarios: ext: a call arrives
Saving: game status is not saved
Scenarios: Call is accepted

Tk-2:
Mobile Phone Model: Model 2
Games: Space Impact
Difficulty Level: medium
Scenarios: ext: a call arrives
Saving: game status is saved
Scenarios: Call is accepted

Tk-3:
Mobile Phone Model: Model 2
Games: Space Impact
Difficulty Level: medium
Scenarios: ext: a call arrives
Saving: game status is not saved
Scenarios: Call is refused

Tk-4:
Mobile Phone Model: Model 2
Games: Space Impact
Difficulty Level: medium
Scenarios: ext: a call arrives
Saving: game status is saved
Scenarios: Call is refused

More in general, whenever a test specification includes a directive "See another PLUC," the derivation of test cases is made by combining the relevant choices from the two related PLUCs. Note that the annotation is made in the PLUC that triggers the test cases, in our example the GamePlay PLUC. Note also that in the GamePlay Test Specification we have marked the choice "ext: a call arrives" with the [single] constraint. As described above the common heuristic in the CP method is that special, unusual, or redundant conditions are not combined with all possible choices, and to recognize them, they are marked as [single]. This heuristic reduces the total number of test cases, while assuring that one frame will be anyhow created with the marked choice. As explained in [28] the decision to use a [single] marking is a judgment by the tester that the marked choice can be adequately tested with only one test case. It is an attempt to trade-off between exhaustive testing of combinations (which is unfeasible) against the pragmatic testing resource limitations. Accordingly, to reduce the number of test scenarios, we have decided not to test separately the arrival of a call together with all possible combinations of GamePlay choices (that are being tested already along the main scenario). Instead we select one representative combination (as the Tk example above) on the side of GamePlay, and from this we then derive as many tests as are the possible refinements when considering the CallAnswer Test Specification.

11.5 Related Work

The problem of the PL modeling and scoping has been approached following different approaches [13,15,29]. Our approach, aiming at introducing constraints on the variabilities inside PLUCs, is based on the proposal by Mannion [24] that addresses general product line model requirements: He presents a way to describe the relationships between product line requirements in order to formally analyze them and to extract information about the internal consistency of the requirements (i.e., they provide a valid template for at least one single product) and of the single products derived from the product line model (i.e., they satisfy all constraints of product line requirements).

We adopt a similar approach and we apply it to the PLUCs, by transforming the described relationships between PL requirements into relationships between PLUC tags and between different PLUCs, and we also extend the set of basic relationships with some composed new ones. The fact that we define a specific notation within which to embed such constraints and relationships provides the product line engineering with a more concrete technique, which can be supported by automatic tools as well.

Chapter 15 exploits UML diagrams and their transformation to address product derivation. The fact that we base our product derivation approach on Use Cases (instead of UML statechart diagrams) means that we focus on the early stages of the development process, that is, requirement elicitation. Addressing product derivation at an early stage has the advantage of early detection of problems and early derivation of test cases, as shown in Sect. 11.4, advantage paid in terms of a higher level of abstraction.

For what concerns the field of product line testing, we quickly overview related work, for the purpose of identifying relevant differences and commonalities with our ongoing research. For the first time, a whole workshop has been devoted to PL testing at SPLC 2004 [11], recognizing the urgent need for testing to keep pace with PLE development productivity gains. Some papers presented in that workshop [20, 23, 30] are particularly interesting because they address the problem of test cases generation starting from the PL variability.

In [22] test-related activities in a product line organization are described. Test-related activities are organized into a test process that is purposely designed to take advantage of the economies of scope and scale that are present in a product line organization. These activities are sequenced and scheduled so that a test activity expands on the testing practice area described by Clements and Northrop [5]. Here we present a test case derivation strategy for PLs described starting from a very general description like the Use Cases are. We can say therefore that the main difference between [22] and [5] and our work stays in the focus, which is there on the process while here is on the methodology. A mutual influence between these two directions of work would certainly be desirable and beneficial. In [18] the authors propose that variability is introduced in the domain-level test cases corresponding to the variabilities present in the Use Cases and that application specific test cases are then derived from them. The derivation strategy depends on how the variability is expressed, and different approaches, including Abstraction, Parameterization, Segmentation, Fragmentation, and Instantiation are overviewed. It is envisaged that a combination of these approaches needs to be used. The approach is still preliminary and details are missing, in particular it is not clear to what extent it can be automated. However, the idea of combining several derivation approaches is interesting and our approach could probably be incorporated in this general framework as one of the derivation strategies (in

particular the Parameterization one). In [27] an approach to expressing test requirements and to formally validate them in a UML-based development process which takes into account PL specificities is presented. Behavioral test patterns (i.e., the test requirements) are built as combinations of use-case scenarios, these scenarios being product-independent and therefore constituting reusable PL assets. The difference between this approach and ours is that from a methodological point of view they propose a whole process from early modeling of requirements to test cases starting from UML specifications, whereas we instead exploit the description of a PL given in natural language and work at the early analysis stages. Perhaps the two approaches could be considered in combination, as addressing different concerns of the PL life cycle. Product line testing is also addressed in RITA [19], an environment under development at the University of Helsinki. RITA is orthogonal to our work, in that it is specifically designed for framework and frameletbased PLs, and does not assist the generation of test cases from requirements. Instead, assuming that the test cases are supplied in input, the environment is conceived for supporting test scripting, execution, result evaluation and more in general for helping with the test process management activities. Different from ours finally are some recent approaches that attack the testing problem based on the product line software architectures. Indeed, the increased use of product line architectures in today's software development poses several challenges for existing testing techniques. In [26] those challenges are discussed as well as the opportunities for addressing them. The Component+ architecture [8] defines instead standardized test interfaces that minimize the effort needed to verify the components by extending software components with configurations.

11.6 Conclusions and Future Research

We have presented the PLUC notation for the description of product lines requirements and shown how this notation allows several kinds of analysis to be performed over such documents, which are extremely useful in the development of products of a software product line. We have concentrated in this chapter over the analysis of PLUCs to derive Product Use Cases and to derive test cases for a product line and its products. In [10] we have applied PLUCs in the process of product line elicitation, that is, how to define a line of products by generalization of some similar products.

In order to support our belief that PLUCs can meet industrial expectations for a notation which is at the same time rigorous and easy to understand, we plan to validate the methodology through extensive industrial case studies. Another important direction we are currently working on is the development of a suite of tools that can support both product derivation from a line and test case derivation for the products of a line.

Moreover, PLUCs could complement the graphical and intuitive but abstract notation of UML Use Cases. Defining a UML profile for PLUCs in order to include variabilities in the diagrams and to associate them with the textual, more detailed descriptions using our notation could be a step toward a standardized version of PLUCs. When we have completed the validation of our methodology, we will thus initiate the international standardization process for PLUCs, facilitating wide industrial adoption and application of the PLUC notation.

Acknowledgments

This work was partially supported by the Eureka Σ!2023 Programme, ITEA (ip00004, Project CAFÉ). We wish to thank in particular Alessandro Maccari from NOKIA, Isabel John from IESE, and Emiliano Nesti from University of Florence for their contributions on the research activity summarized in this chapter. The reviews of Erwin Engelsma, Erik Kamsties, Timo Käkölä, Antti Tevanlinna, and Tewfik Ziadi significantly improved the quality of this chapter.

References

1. Bertolino, A., Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Use case description of requirements for product lines, REPL'02, Essen, Germany (September 2002)
2. Bertolino, A., Gnesi, S.: Use case-based testing of product lines. Proceedings of ESEC/FSE 2003 (ACM, New York) pp 355–358
3. Cascini, G., Fantechi, A., Spinicci, E.: Natural language processing of patents and technical documentation. Proceedings of DAS 2004, 6th IAPR International Workshop on Document Analysis Systems, Firenze, Italy, September 2004. Lecture Notes in Computer Science, vol 3163 (Springer, Berlin Heidelberg New York 2004)
4. Chen, T.Y. et al: On the identification of categories and choices for specification-based test case generation. Inform. Softw. Technol. **46**: 887–898 (2004)
5. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. *SEI Series in Software Engineering* (Addison-Wesley, Reading, MA August 2001)
6. Cockburn, A.: Structuring use cases with goals. J. Object-Oriented Program. Sept–Oct 1997 (part I) and Nov–Dec 1997 (part II)
7. Cockburn, A., *Writing Effective Use Cases* (Addison-Wesley, Reading, MA 2001)
8. Component+, "D4 – BIT Case studies". http://www.component-plus.org (October 2002)
9. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Linguistic techniques for use cases analysis. Proceedings of the IEEE Joint International Requirements Engineering Conference – RE02, Essen, Germany, 9–13 September 2002
10. Fantechi, A., Gnesi, S., John, I., Lami, G., Dörr, J.: Elicitation of use cases for product lines. 5th International Workshop on Product Family Engineering, PFE-5, Siena, 4–6 November 2003. Lecture Notes in Computer Science, vol 3014 (Springer, Berlin Heidelberg New York 2004)
11. Geppert, B., Krueger, C., Li, J.J. (eds): Proceedings of SPLiT 2004, International Workshop on Software Product Line Testing, co-located with SPLC 2004, Boston, MA, USA, August 2004, Avaya Labs Research Tech. Rep. series ALR-2004-031. http://www.research.avayalabs.com/techreport.html
12. Gnesi, S. et al: An automatic tool for the analysis of natural language requirements. Int. J. Comput. Syst. Sci. Eng. **20**(1), 53–62 (2005)
13. van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), pp 45–54
14. Halmans, G., Pohl, K.: *Communicating the Variability of a Software-Product Family to Customers Journal of Software and Systems Modeling* (Springer, Berlin Heidelberg New York 2003)
15. Jaring, M., Bosch, J.: Representing variability in software product lines: a case study. In: *Software Product Lines*, ed by Chastek, G.J., 2nd International Conference, SPLC 2, San Diego, CA, USA, 19–22 August 2002. Lecture Notes in Computer Science, vol 2379, pp 15–36
16. Jazayeri, M., Ran, A., van der Linden, F.: *Software Architecture for Product Families: Principles and Practice* (Addison-Wesley, Reading, MA 1998)
17. John, I., Muthig, D.: Tailoring use cases for product line modeling, REPL'02, Essen, Germany (September 2002)

18. Kamsties, E., Pohl, K., Reis, S., Reuys, A.: Testing variabilities in use case model. 5th International Workshop on Product Family Engineering, Siena, November 2003
19. Kauppinen, R., Taina, J.: RITA environment for testing framework-based software product lines. Proceedings of the 8th Symposium on Programming Languages and Software Tools (SPLST'2003), Kuopio, Finland, June 2003 (University of Kuopio 2003) pp 58–69
20. Knauber, P., Schneider, J.: Tracing variability from implementation to test using aspect-oriented programming. International Workshop on Software Product Lines Testing, Boston, MA, 31 August 2004
21. van der Linden, F.: Software product families in Europe: the ESAPS & CAFÉ projects. IEEE Software (July/August 2002)
22. MacGregor, J.D.: Testing a software product line. Technical report, CMU/SEI-2001-TR-022
23. MacGregor, J.D., Sodhani, P., Madhavapeddi, S.: Testing variability in a software product line. International Workshop on Software Product Lines Testing, Boston, MA, 31 August 2004
24. Mannion, M., Camara, J.: Theorem proving for product line model verification. 5th International Workshop on Product Family Engineering, PFE-5, Siena, 4–6 November 2003. Lecture Notes in Computer Science, vol 3014 (Springer, Berlin Heidelberg New York 2004)
25. von der Massen, S., Lichter, H.: Modeling variability by UML use case diagram. International Workshop on Requirements Engineering for Product Line (REPL'02), Avaya Labs Technical Report, ALR-2002-033 (September 2002)
26. Muccini, H., van der Hoek, A.: Towards testing product line architectures. Electron. Notes Theor. Comput. Sci. **82**(6) (2003)
27. Nebut, C., Pickin, S., Le Traon, Y., Jézéquel, J.-M.: Reusable test requirements for UML-modeled product line, REPL'02, Essen, Germany, Avaya Labs technical report, ALR-2002-033 (September 2002)
28. Ostrand, T.J., Balcer, M.J.: The category partition method for specifying and generating functional tests. ACM Commun. **31**(6), 676–686 (June 1988)
29. Schmid, K.: A comprehensive product line scoping approach and its validation. 24th International Conference on Software Engineering, Orlando, FL, 2002
30. Stephenson, Z., Zhan, Y., Clark, J., McDermid, J.: Test data generation for product lines – a mutation testing approach. International Workshop on Software Product Lines Testing, Boston, MA, 31 August 2004