PLUTO: A Test Methodology for Product Families

Antonia Bertolino, Stefania Gnesi

Istituto di Scienza e Tecnologie della Informazione, CNR
Area della Ricerca di Pisa
56100 Pisa, Italy
{a.bertolino, s.gnesi}@isti.cnr.it

Abstract. The testing stage for a product belonging to a family is a crucial and expensive part of development. Yet the derivation of test cases for product families has so far received little attention. We focus here on test planning, that is the most critical part of testing. We outline a simple methodology we are developing for this purpose, called PLUTO, relying on the early requirements specification expressed as Use Cases. We also overview the related literature.

1 Introduction

Although reuse of software artifacts has long been advocated and pursued, it is only in the last few years that software producers have started to introduce a systematic and pervasive reuse strategy throughout the development process. Reuse is put in place between applications sharing similar functionality and user requirements, but at the same time possessing each specific features that make them different from one another. These “kin” applications form what is called a product family or a product line [10], whereby [14] the first term originated within a series of European industrial-cooperation projects, while the second was introduced by the US community. We will use here the two terms interchangeably.

When building a new product line different approaches can be followed [20]: a product line can be defined from scratch, without starting from any predecessor products; several systems contemporaneously under development are integrated into the product line; or, legacy systems are reengineered into the product line.

In any case, following the Process Reference Model defined in the CAFÉ project [3], [14], product line development is characterized by two correlated processes: Domain engineering and Application engineering, as shown in Figure 1. Domain engineering is the process aiming at developing the general concept of a product line together with all the assets that are common to the whole product line, whereas Application engineering is the process aiming at designing a specific product of the family and eventually produces a customer specific application as in a traditional process.

It is possible to move from the family level to the product level by an instantiation process and on the contrary from the product level to the family level by an abstraction process.

Differently from the usual single product development, the definition process of a customer specific application that is an instance of a family is influenced not only by the requirements of the customer, but also by the capabilities of the product line. Consequently, software product families need more sophisticated requirement analysis and processing methods.

Figure 1. The CAFÉ-Process Reference Model

The most evident and perhaps most urging question is how to handle and represent *variability* [9]. Behind the many commonalities, product family instances in fact necessarily yield variable features, because these constitute precisely what allows for achieving different variants and customized applications. Seen from another perspective, variation points are a way to defer some crucial design decisions [7] until the point where these must be eventually drawn. Indeed, variability and flexibility are the top most desirable characteristics of a product family specification. But variability in a sense amounts to *ambiguity*: where we have been preached for years that a system specification should be made as precise and rigorous as possible, here we must find ways for specifying a product family by leaving some features undecided, i.e., ambiguous.

Specification of product family requirements has drawn large attention in the recent software engineering literature, e.g., [1],[8],[11],[16]. However requirement engineering is only one task of product line engineering. Little attention has been devoted instead to a closely related problem that is *how to test product families*. It is now well recognized that testing takes a predominant amount of the 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 family specification and design must tackle variability, the same need applies for testing.

As identified in [9], considering the Application engineering process, 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 Application engineering process must start from the requirements. In this view, our research ad-

dresses the testing process of product lines, based on the requirements expressed in the well-known formalism of Use Cases [5].

In Section 2, we discuss some general issues of testing for product families. In Section 3 we present the PLUTO methodology to derive specific test cases for product families. An application example is presented in Section 4. An extension of the PLUTO methodology to cover functional dependencies between requirements is given in Section 5. We briefly overview related work in Section 6 and we draw some conclusions in Section 7.

2 Testing Product Lines Starting From The Requirements

Speaking in general product line requirements can be considered as composed of a fixed and a variable part. The fixed part includes all those requirements dealing with features or functionalities common to all the products belonging to the line and that, for this reason, do not need to be modified when an application is instantiated. The variable part represents instead those functionalities that differentiate a product from another.

We can hence say that the requirements specification of a PL comprises a common part $\mathbf{R}$, specifying the common (mandatory) requirements of all the products belonging to the PL, plus a variable part $\mathbf{R}_{\text{var}}$, which defines the variations points distinguishing the different products of the PL, i.e.:

$$\mathbf{PL\ Req}s = \mathbf{R} + \mathbf{R}_{\text{var}}$$

Considering again the CAFÉ Process Reference Model (Figure 1), the **PL Req**s constitute a family asset that is built incrementally along the Domain engineering process.

When a specific product Pr belonging to this family is instantiated in the Application engineering process, its requirements are of course not re-derived from scratch, but are obtained in principle by instantiating the variable part in the PL requirements that will lead to a set of different product-related requirements, i.e.:

$$\mathbf{Pr\ Req}s = \mathbf{R} + \mathbf{R}_{\text{ist}}$$

As previously discussed, the need to put in place a policy of asset reuse for the definition of application requirements is widely recognized, and several approaches have been proposed to tackle variability at this stage. The motivations are clearly enhancing productivity and reducing the costs of requirements specification. Our claim is that these same motivations call for applying an *asset reuse policy to the activities of test planning and management* as well, which may be even more expensive and effort-prone than requirements definition.

Intuitively, a methodology for testing PLs should work at both levels of Domain and Application engineering (Figure 1): at the Domain level it should allow a tester to describe a generic frame of test cases pertaining to the whole PL domain. In other terms, the family test plan should include a list of *mandatory test cases* $\mathbf{T}$, i.e., those tests that apply to the whole domain and correspond roughly to the mandatory requirements $\mathbf{R}$, and a variable part $\mathbf{T}_{\text{var}}$, which refers to test cases significant at the

application level, and that *need to be instantiated from $\mathbf{R}_{\text{var}}$ for each specific product*, i.e.:

$$\mathbf{PL\ Tests} = \mathbf{T} + \mathbf{T}_{\text{var}}$$

At the Application level, the methodology should then allow testers to directly instantiate the frame of test cases relative to a specific product, inclusive of generic and specific test features, i.e.:

$$\mathbf{Pr\ Tests} = \mathbf{T} + \mathbf{T}_{\text{ist}}$$

In practice, the **PL Tests** are not saved once and for all, but more plausibly are incrementally obtained as the result of an iterative process from family to product and backward from product to family. Moreover, as we will also see below in our examples, this list serves only as a conceptual reference: nobody will ever use it to test the family, which is again an abstract reference concept. It is the specific list of **Pr Tests** that is of real interest for a tester, because obviously the test cases are to be executed on a product, not on a PL.

3. PLUTO: A Simple Test Methodology for Product Families

Our proposal to address the testing of Product Lines starting from requirements is PLUTO¹ (Product Lines Use case Test Optimization), a simple and intuitive methodology for the early derivation of test cases² from PF requirements described as Product Lines Use Cases (PLUCs). PLUCs have been introduced in [1]. To make this paper self-contained, before introducing PLUTO, we provide below a short overview of PLUCs.

3.1 Use Cases for Product Families

Use Cases are a powerful tool to capture functional requirements. They provide a means to specify the interaction between a system and its environment and allow for structuring the requirements according to the user goals.

Graphical object modeling languages have become very popular in recent years. Among those, UML [9] introduces a set of graphical notation elements for Use Case modeling. UML Use Case diagrams are easy to understand and constitute a good vehicle of communication. However, they mainly serve as a sort of table of content for Use Cases, just presenting the connections between actors and Use Cases, and the dependencies between Use Cases.

¹ PLUTO has been also introduced in an extended abstract appeared in [2].

² A remark is noteworthy: although we generically speak in terms of “test cases”, this is not compliant with the common meaning of a test case as the precise specification of a test input, a sequence of events and the expected output. We deal rather with *abstract descriptions of test scenarios*: our test cases are actually scenarios of use that need to be tested for validating that the user requirements are satisfied. A refinement process from these abstract descriptions to more concrete ones is needed for obtaining executable test cases.

The requirements regarding the system behavior cannot be specified in detail with UML Use Case diagrams, therefore several authors have proposed to augment the UML Use Case diagrams with textual and tabular descriptions. In particular, an effective and widely used technique for specifying Use Cases was presented by Alistair Cockburn in [4]. The technique is based on natural language specification for scenarios and extensions, which are thus simply expressed by phrases in plain English language. This makes requirements documents easy to understand and communicate, even to non-technical people. We also adopt Cockburn's Use Cases, but extend them with specific annotations to handle PL variability.

A Use Case [4] describes the interaction (triggered by an external actor in order to achieve a goal) between a system and its environment. Every Use Case constitutes a goal-oriented set of interactions between external actors and the system under consideration. The term *actor* is used to describe any person or system that has a goal against the system under discussion or interacts with the system to achieve some other actor's goal.

A primary actor triggers the system behaviour in order to achieve a certain goal. A secondary actor interacts with the system, but does not trigger the Use Case. A Use Case is completed successfully when the goal associated to it is reached. 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 behaviour, error handling procedures, etc. The system is treated as a "black box": 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 actors can use the system, and therefore defines the whole required behaviour of the system.

Generally, Use Case steps are written in an easy-to-understand, structured narrative using the vocabulary of the domain. The language used for the description is usually English. Any other natural language can be used as well. A scenario is an execution path of a Use Case. The main flow is expressed, in the "Description" section, by an indexed sequence of natural language sentences, describing a sequence of actions of the system, called also "Main success scenario". It represents a single path through the Use Case that leads to success in achieving the goal. Scenarios may also be depicted in a graphical form using UML sequence diagrams.

Alternatives to the main flow are expressed in the "Extensions" section, and are linked by their index to the point of the main flow from which they branch out.

PLUCs [1] extend Cockburn's Use Cases [5], allowing variations to be easily described. Variations are explicitly enclosed into the sections of the Use Cases by means of **tags** that indicate those parts of the PL requirements to be instantiated for customizing a specific product of the line.

More specifically, the tags can represent three kinds of variability:

- **Alternative tags:** they express the possibility to instantiate the requirement by selecting an instance among a predefined set of possible choices; the selection is independent from other variation points;

- **Parametric tags:** their instantiation is connected to the actual values of a parameter in the requirements for the specific product, each of them depending on the occurrence of a condition;
- **Optional tags:** their instantiation can be done by selecting indifferently among a set of values, which are optional features for a derived product.

3.2 The test methodology

Commonalities and variabilities also affect test planning in product lines: starting from this consideration we have defined the PLUTO methodology, that is inspired by the well-known Category Partition (CP) method [19], but expands it with the capability to handle PL variabilities and to instantiate test cases for a specific customer product.

In the original CP method, 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 obtained by taking all the possible combinations of choices for all the categories.

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: *i*) either 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 contains all the necessary information for instantiating them by unfolding the constraints.

The CP method described above had to be adapted for dealing with the possible presence of the tags defined above, identifying the PL variation points. However, this can be done quite easily: we use the tags similarly to the original concept of CP constraints, i.e., in the Test Specification we associate the variability tags to the corresponding choices; then, in the process of test derivation we match the tag values in such a way to establish the combinations that are relevant with respect to a specific application. In particular, in case of:

- an Optional tag: the corresponding feature is taken into account or not depending on whether it is present in the application;

- an Alternative tag: the relevant feature is selected;
- 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 selections, but rather assign the appropriate features once some other related tags are fixed.

Another 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 the Test Specification of PLUCs will normally include a category “Scenarios”, in which all the specified scenarios are listed.

As our 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 judgment, and this is natural. However, lexical and syntactical analyzers for natural language requirements could be used to extract useful information to identify the relevant Categories.

When considering the repository of all Use Cases specified for a PL, it will often be the case that some scenarios in a Use Case depend on other scenarios in another Use Case, because of the presence of *cross-cutting features*. To handle them, we annotate the PLUC with a specific note such as “See another PLUC”. In general, whenever a test specification includes such a directive, 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.

With reference to the requirements conceptual framework in Section 2, we will have that the set T of mandatory test cases for a PL is given by those test cases that do not include variability tags, i.e., by those combinations of choices that are common throughout the family. On the other hand, all the possible combinations of choices involving tags form the set T_{var} of variable test cases. The complete set **PL Tests** of mandatory and variables test cases, which would be obtained in this way, form the family asset of test cases. As said, we do not derive actually the list of **PL Tests**, rather we derive the PL Test Specification and leave it unfolded.

The test cases are derived when a specific product is being developed after having instantiated the tags in each PLUC to the appropriate values. More precisely, for each Test Specification relative to each PLUC a different set of test cases will correspond to each 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 traditional CP test method, this is also what makes PLUTO basically different from the traditional CP, in which we depict only one set of test cases corresponds directly to each Test Specification.

Figure 2. A Scheme of the PLUTO-Methodology

We point up this conceptual difference with the help of a scheme. In Figure 2 below we depict: in part a) the process of test derivation in the traditional CP method, where the test cases are directly unfolded from the test specifications; in part b) the

more articulated process of PLUTO. As shown, there are two stages: the derivation of test specifications, belonging to the family; then, a family of different sets of test scenarios can be derived from each PLUC, depending on the instantiation of tags.

4. An example

An example of a PLUC is presented in Figure 3. We propose the description of the GamePlay Use Case applicable 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.

<p>PL USE CASE GamePlay</p> <p>Goal: Play a game on a [Vo] Mobile Phone and record score</p> <p>Scope: The [Vo] Mobile Phone</p> <p>Level: Summary</p> <p>Precondition: The [Vo] Mobile Phone is on</p> <p>Trigger: Function GAMES has been selected from the main menu</p> <p>Primary actor : The Mobile Phone user</p> <p>Secondary actors: The {[V0] Mobile Phone} (the system) The Mobile Phone Company</p> <p>Main success scenario</p> <ol style="list-style-type: none"> 1. The system displays the list of the {[V1] 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 {[V 2] and sends the score to Club XXX via WAP} <p>Extensions</p> <ol style="list-style-type: none"> 1a. No game is available: <ol style="list-style-type: none"> 1a1. return to main menu 3a. The user starts the game and plays it until an incoming call arrives. See CallAnswer. <p>Variations</p> <p>V0: Alternative:</p> <ol style="list-style-type: none"> 0. Model0 1. Model1 2. Model2 <p>V1: Parametric</p> <p>if V0=0 then display msg "No game available"</p> <p>else if V0=1 then Snakell or Space Impact</p> <p>else if V0=2 then Snakell or Space Impact or Bumper.</p> <p>V2: Optional</p> <p>when V0=2</p>
--

Figure 3. Example of a Use Case in the PLUC notation

Curly brackets are introduced into the Use Case elements, and tags (here **[Vo]**, **[V1]**, and **[V2]**) indicate the variation points within the Use Case. Moreover, the possible

instantiations of the variable parts and the type of the variations are defined within the Variations section within the PLUC.

For illustration purposes, we now apply the Pluto approach to the GamePlay PLUC in Figure 3. As a first step, from an analysis of it we identify the following Categories: “Mobile Phone Model”, “Games”, “Difficulty Level”, “Scenarios”, “Club”. 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.

Now we proceed with 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.

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 family. 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 family, but only for those supporting this feature. This is specified in the GamePlay PLUC by means of the **V2** 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 Figure 4 we derive the possible choices pertaining to the “Club” category, but we annotate them with the appropriate selector, which is a simple IF condition stating that these choices are of interest only when property P2 is satisfied (which happens for Model2).

The complete Test Specification is shown below in Figure 4.

If we now apply to this test specification a generator that takes out all the possible combinations of choices, we would obtain a list of test cases, which correspond to the **PL Tests** list. This list would include all the potential test cases for all the products of the family relative to the PLUC under consideration. However, what is more interesting in our opinion, is that we can instead derive directly the **Pr Tests** list for a specific product of interest. This is obtained very easily by just instantiating the relative tags. So, for instance, if we are interested to test the Model2 product of this family, we set property P2 to true and derive all and only the combinations that remain valid.

PLUC GAMEPLAY TEST SPECIFICATION	
[V0]: Mobile Phone Model:	
0. Model0	[Property P0]
1. Model1	
2. Model2	[Property P2]
Games:	
None	[if P0]
Snakell	[if NOT P0]
Space Impact	[if NOT P0]
Bumper	[if NOT P0] [if P2]
Difficulty Level:	
easy	[if NOT P0]
medium	
expert	
Scenarios:	
Main	[if NOT P0]
ext: no game available	[if P0]
ext: a call arrives	see CallAnswer [single]
[V2]: Club:	
WAP connection on	[if P2]
WAP connection off	[if P2]

Figure 4. Main Test Categories for the GamePlay PLUC

As an example, we list below in Figure 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.

In Figure 5 the test cases T_i , T_{j1} , T_{j2} all refer to a simpler situation in which the features in a PLUC do not depend on the features of another PLUC. Test T_k 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. 3.2. We now see below how this can be handled in the Pluto methodology.

Tag V0=0 Ti: Mobile Phone Model: Model 0 Games: None Scenarios: ext: no game available
Tag V0=2 Tj1: Mobile Phone Model: Model 2 Games: Snakell 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 1 Games: Space Impact Difficulty Level: medium Scenarios: ext: a call arrives - <i>see CallAnswer</i>

Figure 5. Some Test Scenarios

5. Extending the Methodology

When considering the repository of all Use Cases specified for a PL, it will often be the case that some scenarios in a Use Case depend on other scenarios in another Use Case, because of the presence of cross-cutting features. 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.

Figure 6. Another PLUC example

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 referred in Figure 3. The description of the CallAnswer PLUC is reported in Figure 6.

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 Figure 7.

PLUC CALLANSWER TEST SPECIFICATION	
[V0]: Mobile Phone Model:	
0. Model0	Property P0
1. Model1	Property P1
2. Model2	Property P1
Saving:	
a. game status is not saved	[if P1]
b. game status is saved	[if P1]
Scenarios:	
Main:	Call is accepted
ext:	Call is refused

Figure 7. 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 Use Case.

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 Figure 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 Figure 5 the test case Tk for instance selects this choice (we report it again below):

Tk:
Mobile Phone Model: Model 1
Games: Space Impact
Difficulty Level: medium
Scenarios: ext: a call arrives

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 (Figure 7), we get at least four refined test cases as follows:

Tk-1:
Mobile Phone Model: Model 1
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 1
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 1
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 1
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, described in Section 3.2. In fact, 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.

6. Related Work

The field of Product Lines testing is very young. We quickly overview related work, for the purpose of identifying relevant differences and commonalities with our ongoing research.

In [15] 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 [4]. 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 [15] and [4],

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 [12] the authors propose that variability is introduced in the domain-level test cases, correspondingly 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 approach 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 automatized. 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 [18] an approach to expressing test requirements and to formally validate them in a UML-based development process which takes into account product lines (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 analysis early stages. Perhaps the two approaches could be considered in combination, as addressing different concerns of PL life cycle.

Product line testing is also addressed in RITA [13], an environment under development at the University of Helsinki. RITA is orthogonal to our work, in that it is specifically designed for framework and framelet-based PLs, and does not assist the generation of test cases from requirements. Instead, assuming that the test cases supplied in input, the environment is conceived for supporting test scripting, execution, result evaluation and more in general for helping with all 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 [17] those challenges are discussed as well as the opportunities for addressing them. The Component+ architecture [6] defines instead standardized test interfaces that minimize the effort needed to verify the components by extending software components with configurations.

7. Conclusions

We have discussed the important issue of test planning for product families starting from the PL requirements, expressed as PLUCs, which are an extension of Cockburn's Use Cases.

After analyzing the need for handling variability and reusing Test Specifications across products of a same family, we have proposed the PLUTO methodology, that is inspired by the Category Partition method, but expands it with the capability to handle PL variabilities and to instantiate test cases for a specific customer product.

As our 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 judgment, and this is natural. However, lexical and syntactical analyzers for natural language requirements could be used to extract useful information to identify the relevant Categories.

With regard to the derivation of the test cases from the Test Specification, instead, this task can be easily automated, and we are currently working at a PLUTO tool implementation. We plan also to investigate the integration of some of the available lexical/syntactical analyzers in the PLUTO tool to further automate the test generation process.

Our objective here was to propose a simple and intuitive methodology to manage the testing process of product lines, based on the Use Case requirements. The proposed methodology helps derive a set of Test Specifications, each relative to a PL Use Case, as an asset of the family; and a suite of application specific test cases, when a customized application is defined within the product line.

The work is clearly a first step towards a more comprehensive testing strategy for families. Further refinement of the approach is ongoing, as well as the development of automated support tools. On the other hand, as we have seen, the topic of product families testing is relatively new, and therefore this paper is also intended as a contribution to trigger further research.

Acknowledgments

This work was partially supported by the Eureka $\square$!2023 Programme, ITEA (ip00004, Project CAFÉ).

References

1. Bertolino, A., Fantechi, A., Gnesi, S., Lami, G., Maccari, A. Use Case Description of Requirements for Product Lines, REPL'02, Essen, Germany, Avaya Labs Tech. Rep. ALR-2002-033, September 2002.
2. Bertolino, A., Gnesi, S., Use Case-based Testing of Product Lines, Proc. ESEC/FSE 2003, p. 355-358, ACM Press.
3. Café project homepage, at: <http://www.extra.research.philips.com/euprojects/cafe/>.
4. Clements, P.C., Northrop, L. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley, August 2001.
5. Cockburn, A. Writing Effective Use Cases. Addison Wesley, 2001.
6. Component+, "D4 – BIT Case studies", www.componentplus.org, October 2002

7. van Gurp, J., Bosch, J., and Svahnberg, M. On the Notion of Variability in Software Product Lines, Proc. of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), p. 45-54.
8. Halmans, G., Pohl K., Communicating the Variability of a Software-Product Family to Customers, Journal of Software and Systems Modeling, Vol. 2, p.15-36 Springer, March 2003.
9. Jaring, M., and Bosch, J. Representing Variability in Software Product Lines: A Case Study. In Chastek G. J. (Ed.): Proc. Software Product Lines, 2nd Int. Conf, SPLC 2, San Diego, CA, USA, August 19-22, 2002, LNCS 2379, p.15-36.
10. Jazayeri, M., Ran, A., van der Linden, F. Software Architecture for Product Families: Principles and Practice, Publishers: Addison-Wesley, Reading, Mass. and London, 1998.
11. John, I., Muthig, D. Tailoring Use Cases for Product Line Modeling, International workshop on requirements Engineering for Product Line (REPL'02), 2, Avaya Labs Technical Report ALR-2002-033, September 2002.
12. E. Kamsties, K. Pohl, S. Reis, A. Reuys; "Testing Variabilities in Use Case Model", 5th International Workshop on Product Family Engineering, Siena, November 2003.
13. Kauppinen, R., Taina J., RITA Environment for Testing Framework-based Software Product Lines. *Proceedings of the Eighth Symposium on Programming Languages and Software Tools (SPLST'2003)*, Kuopio, Finland, June 2003, 58-69, University of Kuopio, 2003.
14. van der Linden, F. Software Product Families in Europe: The ESAPS & Café Projects, IEEE Software July/August 2002.
15. MacGregor, J. D. Testing a Software Product Line, *Technical Report*, CMU/SEI-2001-TR-022.
16. 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
17. Muccini H., van der Hoek A., Towards Testing Product Line Architectures, Electronic Notes in Theoretical Computer Science 82 No. 6, 2003.
18. Nebut C., Pickin S., Le Traon Y. and Jézéquel, J.-M. Reusable Test Requirements for UML-Modeled Product line, REPL'02, Essen, Germany, Avaya Labs Technical Report ALR-2002-033, September 2002.
19. Ostrand, T.J., and Balcer, M.J. The Category Partition Method For Specifying and Generating Functional Tests. ACM Comm. 31 (6), June 1988, pp. 676-686.
20. Schmid K., Verlage, M. The Economic Impact of Product Line Adoption and Evolution. IEEE Software, 19(4): 50--57, July-August 2002.