An Approach to Derive Usage Models Variants for Model-based Testing

Hamza Samih, Hélène Le Guen, Ralf Bogusch, Mathieu Acher, Benoit Baudry

An Approach to Derive Usage Models Variants for Model-based Testing

Hamza Samih[1,3], Hélène Le Guen[1], Ralf Bogusch[2], Mathieu Acher[3], and Benoit Baudry[4]

[1] ALL4TEC, France {hamza.samih, helene.leguen}@all4tec.net
[2] Airbus Defence and Space, Germany ralf.bogusch@cassidian.com
[3] University of Rennes 1, IRISA/Inria mathieu.acher@irisa.fr
[4] IRISA/Inria Rennes, France benoit.baudry@inria.fr

Abstract. Testing techniques in industry are not yet adapted for product line engineering (PLE). In particular, Model-based Testing (MBT), a technique that allows to automatically generate test cases from requirements, lacks support for managing variability (differences) among a set of related product. In this paper, we present an approach to equip usage models, a widely used formalism in MBT, with variability capabilities. Formal correspondences are established between a variability model, a set of functional requirements, and a usage model. An algorithm then exploits the traceability links to automatically derive a usage model variant from a desired set of selected features. The approach is integrated into the professional MBT tool MaTeLo and is currently used in industry.

Keywords: Product Line, Model-based Testing, Usage Model, Usage Model Variant, Orthogonal Variability Model, Requirements

1 Introduction

Real world success stories of *Product Lines (PLs)* show that the effective management of a large set of products is possible [1, 2]. The factorization and exploitation of common features of the products as well as the handling of their variability is an essential success criteria for this success stories A major challenge in PL engineering is the combinatorial explosion of features, leading to potentially billions of individual products; mastering them it can lead to significant benefits. Yet configuring, deriving, and *testing* a family of products raises new problems [3–5].

Existing V&V approaches (testing, model checking, etc.) usually target validation at the single product level. They mainly consist in validating each product independently from the others and are hardly applicable to a family of products (also called *variants*). This tends to hinder expected benefits of PL engineering in terms of reuse, reduction of development cost, and shortening of time-to-market and certification cost. *Model-based Testing (MBT)* has the potential to assist practitioners in building and testing PLs with adequate abstraction and

automation. In essence MBT aims at inferring test suites from a test model that is based on the system requirements [6]. A test model can be represented using several formats such as UML state-machines, Markov chains, or a *Usage Model* (see Section 2.2). From a test model, one can define different testing strategies and derive a set of relevant test cases, accordingly to the chosen strategy [7, 8].

Although behavioural MBT is well established for single-system testing, a survey shows insufficient support of PL-based MBT [9]. In particular, usage models (a widely used formalism in MBT) are employed to test only one individual system. We want to go further and equip usage models with *variability* information in order to formally document what can vary in a usage model. For this purpose, features, as end-user visible behaviour of a system, are widely used to distinguish different behaviour variants of a PL. The idea is then to systematize the derivation of usage models variants – each variant being exploited afterwards for generating test cases of a specific product of a PL. Specifically we address the following research questions in this paper: (1) How to infuse variability concerns within a test model and build explicit relationships between a variability model and a usage model? (2) How to extract a valid product-specific test suite from a global usage model, according to a desired set of features (configurations)?

The key idea of our proposal is to establish formal correspondences between features, requirements and a usage model. The relationships are then exploited to automatically synthesize usage model variants. The synthesis algorithm has two major steps. A subset of the requirements is inferred from a specific configuration. A usage model variant is obtained from the set of requirements by pruning unnecessary transitions and correcting probabilities. The paper describes in details the overall testing approach and presents the theoretical foundations. The industrial report of the application of the approach in the aeronautic domain has been published in [10]. To summarize, we make the following contributions:

- We describe a comprehensive approach to relate a variability model, a set of functional requirements, and a usage model. The description of variability is formally defined, non intrusive (separated), and can operate over an existing usage model;
- We develop an algorithm to automatically derive a variant of a usage model from a desired selection of features. We discuss the (polynomial) complexity and the reliability of the algorithm. We integrate the algorithm into the professional MBT tool MaTeLo[5] so that variants of usage models (Markov chains) can be used afterwards to test a system.

The remainder of the paper is organized as follows. Section 2 motivates our work and introduces background information. Section 3 presents our approach to model variability of a usage model. Section 4 describes our algorithm to derive usage model variants from variability (features) and requirements. Section 5 discusses related work while Section 6 concludes the paper.

[5] MaTeLo (Markov Test Logic) is a MBT tool developed by ALL4TEC

2 Background & Example

In this section we present a car dashboard as an example of a product line (PL). The dashboard of a car provides various information such as: vehicle speed, engine speed, engine temperature, fuel level in the tank, oil pressure in the engine and turn indicator. This list may vary depending on the manufacturer, model, version, and the vehicle category. However, certain features of a dashboard are imposed by international standards. In this example of a PL, two kinds of configurations are considered: high-end (HE) and low-end (LE) products, intended for both Europe (EU) and United States (US). We will limit ourselves to some basic features of a dashboard with a couple of variants.

2.1 Overview

In this example we can define 16 valid configurations of the dashboard. In general, testing all variants is impossible due to the resources and time needed, even if the process is automated. The challenge in industry is to optimize the process for PL testing. Nevertheless, deploying a new solution in industry is a decision that needs to be justified by time and cost savings. An important step to improve the adoption of novel PL testing approaches is to propose solutions leveraging already in-use technologies. For instance, MBT is a widely used automated testing solution for embedded systems [6]. In this work we propose to extend MBT to support PL testing in order to derive specific test cases for variants of a PL. We focus on usage models as MBT formalism and choose the MaTeLo tool to develop test models. For PL variability documentation, we use OVM and extend its existing formalization [11].

2.2 The MaTeLo usage model

In this work we use MaTeLo usage models which describe the intended behaviour of a system under test (SUT) [12, 13]. MaTeLo supports the development of statistical usage based models by using extended Markov chains. Though, MaTeLo usage model is a finite state machines, where the nodes represent the major states of the system and the transitions represent the actions or operations of the SUT. A usage model is built according to textual requirements or existing specification documents of the SUT [14]. The usage model is created with MaTeLo - a MBT solution, that allows automated generation of test cases for complex systems. However, the MaTeLo approach for usage models is based on the traditional approach of MBT, that consider single systems only, whereas for a product line, we should create a model for each variant of the system. In the light of the considerable efforts to build the usage model, the test cases are generated automatically. The usage model [13] is a hierarchical model, it can be composed by multiple extended Markov chains. A chain is a part of the model referenced by a state of another chain at the above level. Each transition has a probability $p_{ss'}(\mathfrak{F})$, which corresponds to the probability of choosing the state s' when the process is in state s for a profile $\mathfrak{F}$. Profiles qualify the usage model to represent how the

system will be used statistically. Probabilities are not versus time and do not vary during generation. The usage model provides the stimulation and expected responses of the system which are extracted from the functional requirements and associated thereafter with transitions. We consider a usage model denoted $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ for a single system ($\mathcal{P}$) to be a tuple of the form (S, s_0, s_n, R, T), where:

- S: a finite set of states,
- $s_0 \in S$: unique initial state,
- $s_n \in S$: unique final state,
- R: a set of functional requirements of the SUT,
- T: a set of probabilistic transitions of the form $s \xrightarrow{p_{ss'}, R_{ss'}} s'$, with $s \in S \ / \ s_n$, $s' \in S \ / \ s_0$, $R_{ss'}$ is a set of the requirements associated with $t_{ss'}$, $p_{ss'} \in [0, 1]$ is the probability of choosing state s' from s. The sum of the probabilities associated with the outgoing transitions of state s must be equal to 1.

The initial state and final state are used to define the border of a test case. Loops are allowed in the usage model. All states except the final state s_n have at least one outgoing probabilistic transition. All states except the initial state s_0 have at least one incoming probabilistic transition. A requirement can be associated manually with transitions of the usage model in MaTeLo. Each associated requirement has a role, by default set to *necessary*. It may also have as role *sufficient* or *necessary and sufficient*. For example, the *navigation* requirement r_6 is associated with the action *Start navigation* depicted by the transition t_5 in the usage as shown in Fig. 1.

A generated test case always starts with s_0 and ends with s_n. Test case generation consists of selecting transitions according to their probabilities and the chosen algorithm. The criteria can vary between the random selection or selection of the largest probability of a transition to constitute a valid test case. A requirement is covered, when a sufficient or all the necessary transitions are present in the test case path.

Designing a usage model is not easy and requires a substantial amount of work; all information on the SUT must be integrated, such as stimuli, requirements and usage profiles. Adding a layer of variability to the test model remains tedious and it will be more complicated to track improvements and updates. Consequently, modelling variability separately with a single base model as the Orthogonal Variability Model (OVM), helps to identify inconsistencies and allows reasoning on product lines. We are interested in OVM in order to catch only the variable items and manage system variability in an efficient and consistent way, as well as to avoid a hierarchical arranged set of features.

2.3 Orthogonal Variability Model (OVM)

OVM is a flat model that documents only PL variability [1]. In the OVM model a variation point documents a variable feature. Each instance of a variation point is represented by a feature. A variation point is represented by a triangle and a rectangle depicts a feature.

Each variation point is related to at least one feature and a feature is bound to one variation point. Three types of relationships are possible between variation points and features: *mandatory*, *optional* and *alternative choices*. A *mandatory* feature is always required in the realization of a new variant. An *optional* feature can be added or not to a variant. A combination of *alternative choice* dependencies allows grouping options; this group of options is associated with a cardinality that defines a minimum number n and a maximum number m of features. The left-hand pane of Fig. 1 depicts the dashboard OVM model.

OVM uses constraints to refine relationships between features and variation points or between both, a constraint can be a *requires* or an *excludes* constraint.

OVM abstract syntax: Metzger et al. [11] have introduced a mathematical formalization for OVM, which describes the basic elements outside the meta-model. We use this formalization to describe the variability model. We consider an OVM denoted $\mathcal{M}_{\mathcal{V}}$ to be a tuple of the form $(VP, V, \mathcal{P}(V), C)$, where:

- $VP(\neq \emptyset)$, is a set of variation points.
- vp, is a variation point, where $vp \in VP$.
- $V(\neq \emptyset)$, is a set of features.
- v, is a feature[6], where $v \in V$.
- $\mathcal{P}(V) = \{V' | V' \subseteq V\}$ power set of any set V, $\mathcal{P}(V)$ is the set of all subsets of V including the empty set and V itself.
- $Req \subseteq (V \times V) \cup (V \times VP) \cup (VP \times VP)$ symbolizes the *required* constraints between V_V, V_VP and VP_VP. $Excl$ symbolizes the *excluded* constraints.
- $C = Req \cup Excl$, set of constraints, where $c \in Req$ or $c \in Excl$.

3 Modeling Variability of Usage Models

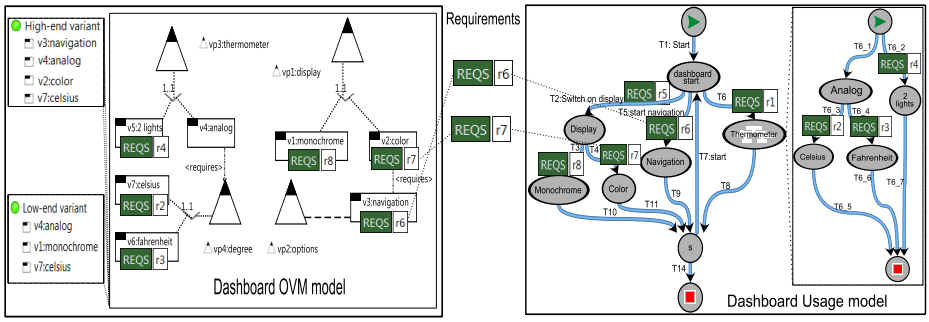

Fig. 1. Reconciliation process of variability model $\mathcal{M}_{\mathcal{V}}$ with usage model $\mathcal{M}_{\mathcal{T}}$

In this work we adapt MBT such that variant-specific test cases can be generated from derived usage model variants. In order to support PL variability, we extend both the usage model and the OVM semantics.

[6] In OVM terminology v is denoted as a "variant". In this paper, we use *feature* (instead of variant) to refer to a discriminant, user-visible characteristic of a system; variant is used for another meaning (see Section 3.3).

3.1 Relating features with requirements

During the realization of the usage model, the association between requirements and transitions of the model is performed simultaneously. To reconcile the OVM model with the usage model, it is necessary to associate requirements with features. This association is realised by users with MPLM[7] tool implemented to test our approach. During the creation of links between requirements and features, we encounter three different types of requirements:

- a requirement associated with PL commonalities as r_1. This generic requirement is generally associated only with the transition representing the variation point as vp_3 *thermometer* and not to its features.
- a requirement associated only with one feature as r_6, associated with the v_3 *navigation* feature.
- a requirement associated with a feature that is required by one or more other features. This kind of requirement is addressed specifically in Section 4.2.

We extend the **OVM abstract syntax** introduced in Section 2.3 to reuse it in our contribution to express the requirements-features association. We consider an OVM denoted $\mathcal{M}_{\mathcal{V}}$ to be a tuple of the form $(VP, V, \mathcal{P}(V), C, \delta)$, where:

- $(VP, V, \mathcal{P}(V), C)$, equivalent to $\mathcal{M}_{\mathcal{V}}$,
- $\delta : V \nrightarrow \mathcal{P}(R)$ is a partial function, labelling features with requirements.

Thus, thanks to the links made between features and requirements, we can identify transitions related to features.

3.2 The PL usage model

In our approach, we suggest to build only one usage model that describe the expected behaviour of features in a PL. That means, the PL usage model will be instantiated differently for each variant. The right-hand pane of Fig. 1 illustrates the dashboard usage model with its sub-chains. The model represents the expected behaviour of the dashboard; stimuli and expected responses are associated with transitions. For example, the thermometer function can reference a sub-chain that represents its behaviour. Test cases may cover an executable path of one or many functionalities of the dashboard. Moreover, the transition t_7 is used to loop back and include new executable paths of others features.

$T = \{t_1, ..., t_{6\_7}, ..., t_{11}\}$ is the set of transitions from $\mathcal{M}_{\mathcal{T}}$ depicted in right-hand pane of Fig. 1 which are annotated with the following requirements R:

r_1: A sensor located on the engine block or cylinder head provides temperature.

r_2: The thermometer for Europe display the temperature in C°.

r_3: The thermometer for United States display the temperature in F°.

r_4: The thermometer based on two LEDs should light the blue light when the engine is cold, red light when it is too hot and all off when everything is normal.

[7] MaTeLo Product Line Manager, aims to derive usage models variants for product line testing.

r_5: The screen must allow setting the options of the dashboard.

r_6: Navigation should be able to retrieve the position of the vehicle and assist the driver by voice and visual indications.

r_7: The high-end dashboard uses a digital display and a color display for navigation.

r_8: The low-end dashboard uses an analogical display and a monochrome display.

We extend the **MaTeLo usage model syntax** to define a PL usage model denoted $\mathcal{M}_{\mathcal{T}}$ to be a tuple of the form $(S, s_0, s_n, R, \mathcal{P}(R), T, \mathcal{P}(T), \gamma)$, where:

- (S, s_0, s_n, R, T), equivalent to $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ described in 2.2,
- $\mathcal{P}(R) = \{R' | R' \subseteq R\}$, $\mathcal{P}(R)$ power set of any set of R,
- $\mathcal{P}(T) = \{T' | T' \subseteq T\}$, $\mathcal{P}(T)$ power set of any set of T.
- $\gamma : R \nrightarrow \mathcal{P}(T)$ is a partial function, labelling requirements with transitions.

Those relationships are true only for the PL usage model. To make a valid usage model, we should observe rules presented in Section 2.2. In the following we consider a unique profile $\mathfrak{F}$ associated with the PL usage model. The $\mathfrak{F}$ profile is independent of product variants described in the usage model for multiple variants. It is used to have a valid usage model structure and help the Algorithm 1 to derive variant-specific usage model with their own profile.

Semantics. Each $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ derived describes the expected behaviour (see Section 2.2) of a usage model variant. The semantics of a $\mathcal{M}_{\mathcal{T}}$ is thus the union of the behaviours of all valid configurations: $\mathcal{M}_{\mathcal{T}} = \bigcup_{\mathcal{P} \in \mathbf{P}} [\![\mathcal{M}_{\mathcal{T}_\mathcal{P}}]\!]$.

3.3 Variant semantics

A variant is a valid configuration composed of a set of features that considers a set of constraints defined in the $\mathcal{M}_{\mathcal{V}}$. A variant denoted $\mathcal{P}$ is a tuple of the form $(V_\mathcal{P}, R_\mathcal{P}, \mathcal{M}_{\mathcal{T}_\mathcal{P}})$ used to identify $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ from $\mathcal{M}_{\mathcal{T}}$, where:

- $V_\mathcal{P}$ set of features that compose $\mathcal{P}$,
- $R_\mathcal{P}$ set of requirements related to $\mathcal{P}$,
- $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ usage model variant.
- $\mathbf{P} = \{\mathcal{P}_1, ..., \mathcal{P}_n\}$ denotes PL variants.

The proposed formalization allows deriving usage model variants for generation of product-specific test cases.

4 Deriving Usage Model Variants

In this section we present the second part of our contribution. The automated derivation of a usage model variant consists of projecting a set of OVM features composing a valid configuration onto the PL usage model. The bindings between features, PL requirements and the transitions of the PL usage model help to derive $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ with only the transitions and requirements of variant $\mathcal{P}$. The results of the reconciliation process depicted in Fig. 1 (see items 1, 2 and 3) are taken as input for the derivation process shown in Fig. 2.

In our approach we consider four steps to reach the automated derivation of usage model variants:

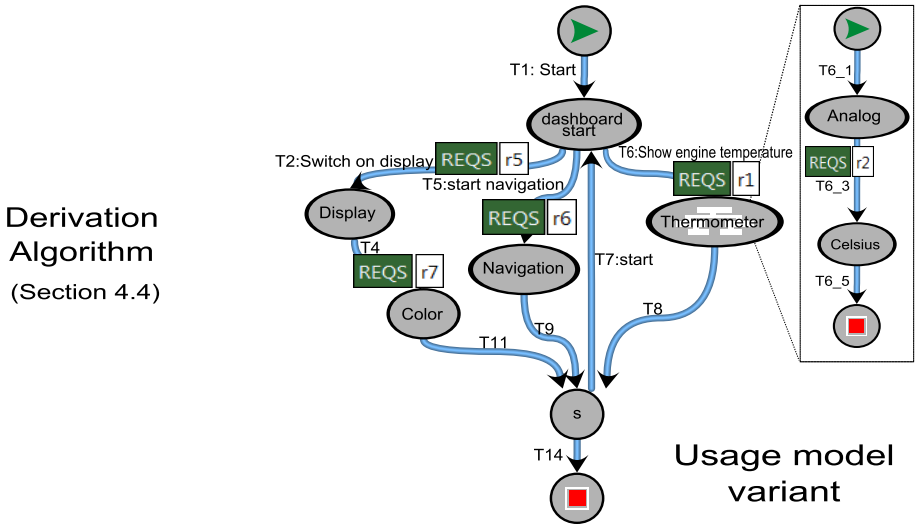

Fig. 2. The derivation process of usage model variants

Step A Identify and select features that compose the configuration under test and remove features not used or not required.

Step B Extract and classify according to the selected features the requirements to keep and to delete.

Step C Identify and select according to the classified requirements, the transitions to keep and to delete as well as identify incoherent cases.

Step D Derive a usage model variant $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ from $\mathcal{M}_\mathcal{T}$ by removing transitions not mapped straightforwardly to the selected variant, while keeping correctness of the derived model as defined in Section 2.2.

For illustration, we use high-end variant $\mathcal{P}_1$ to derive its corresponding usage model variant, rather than low-end variant $\mathcal{P}_2$. As a reminder, $V_{\mathcal{P}_1}$ is equal to $\{v_2, v_3, v_4, v_7\}$, and $V_{\mathcal{P}_2}$ is equal to $\{v_1, v_4, v_7\}$, see left side of Fig. 1.

4.1 Step A:

The first step consists of identifying the features that should not be considered for the derived usage model variant. This implies to select all features in the OVM model except the selected features that compose the variant under test. Let $V_{\bar{\mathcal{P}}}$ be the set of features not associated with $\mathcal{P}$.

$$V_{\bar{\mathcal{P}}} = V \setminus V_\mathcal{P} \tag{1}$$

Thanks to the traceability function δ defined in Section 3.1, we can extract the corresponding requirements for both $V_\mathcal{P}$ and $V_{\bar{\mathcal{P}}}$.

4.2 Step B:

This step prunes R, to identify requirements that cover the variant $\mathcal{P}$ and requirements to remove. R is composed of R_V a set of requirements associated with the PL variability and $R_{\bar{V}}$ a set of requirements associated with commonalities. In the example, $R_{\bar{V}}$ corresponds to $\{r_1, r_5\}$ and $R_V = \{r_2, r_3, r_4, r_6, r_7, r_8\}$.

$$R = R_V \cup R_{\bar{V}} \tag{2}$$

The identification of requirements to keep, to delete and common requirements between both, requires refining R_V, i. e., identifying requirements labelling $V_{\mathcal{P}}$ and $V_{\bar{\mathcal{P}}}$.

$$R_V = R_{V\_\mathcal{P}} \cup R_{V\_\bar{\mathcal{P}}} \tag{3}$$

Where $R_{V\_\mathcal{P}}$ is the set of requirements labelling $V_{\mathcal{P}}$ while $R_{V\_\bar{\mathcal{P}}}$ is the set of requirements labelling $V_{\bar{\mathcal{P}}}$. Nevertheless, some requirements may belong to $R_{V\_\mathcal{P}}$ and at the same time to $R_{V\_\bar{\mathcal{P}}}$. R_{Inter} represents the common requirements between variants, where:

$$R_{Inter} = R_{V\_\mathcal{P}} \cap R_{V\_\bar{\mathcal{P}}} \tag{4}$$

R_{Inter} helps to refine $R_{V\_\mathcal{P}}$ and $R_{V\_\bar{\mathcal{P}}}$ and select accurately the requirements to delete and to keep. Furthermore, R_{Inter} can assist in the detection of the incoherent cases described in 4.3. R_{Inter} of the dashboard example is empty.

$$R_{\|V\_\mathcal{P}\|} = R_{V\_\mathcal{P}} \setminus R_{Inter} \tag{5}$$

$R_{\|V\_\mathcal{P}\|}$, set of requirements labelling only features of the selected variant without common requirements. For $\mathcal{P}_1$, $R_{\|V\_\mathcal{P}_1\|}$ is equal to $\{r_2, r_6, r_7\}$.

$$R_{\|V\_\bar{\mathcal{P}}\|} = R_{V\_\bar{\mathcal{P}}} \setminus R_{Inter} \tag{6}$$

Where $R_{\|V\_\bar{\mathcal{P}}\|}$ is the set of requirements that are not labelling $\mathcal{P}$ features. For $\mathcal{P}_1$, $R_{\|V\_\bar{\mathcal{P}}_1\|}$ is equal to $\{r_3, r_4, r_8\}$.

The objective is to identify R_{Inter}, $R_{\|V\_\mathcal{P}\|}$ and $R_{\|V\_\bar{\mathcal{P}}\|}$ based on $V_{\mathcal{P}}$ and $V_{\bar{\mathcal{P}}}$ as input.

4.3 Step C:

We remind that a transition in a usage model can be labelled by several requirements ensured by the traceability function γ. Three subsets of transitions are identified:

Selection of transitions to be deleted T_D: it consists of selecting each transition labelled by a requirement of $R_{\|V\_\bar{\mathcal{P}}\|}$ must be removed definitively, i. e., each transition associated with a subset of requirements R' that satisfies the following condition should be deleted:

$$R' \subseteq R_{\|V\_\bar{\mathcal{P}}\|} \tag{7}$$

For $\mathcal{P}_1$, T_D is equal to $\{t_3, t_{6\_2}, t_{6\_4}\}$.

Selection of transitions to be kept T_K: it consists of selecting each transition labelled by a requirement of $R_{V\_\mathcal{P}}$ must be kept, i. e., each transition covering requirements that label features of the selected variant should be kept:

$$R' \subseteq R_{V\_\mathcal{P}} \tag{8}$$

To recap, $R_{V\_\mathcal{P}}$ is the union of $R_{\|V\_\mathcal{P}\|}$ and R_{Inter}. For $\mathcal{P}_1$, T_K is equal to $\{t_4, t_5, t_{6\_3}\}$. If all transitions of the usage model are selected in T_K, it means the PL usage model describes only the usage of a system, and it is by no means a PL usage model. In this case, all transitions will be kept.

Detection of incoherent case $T_{incoherent}$: it consists of identifying all incoherent transitions, i.e. identify the transitions annotated both by a set of requirements to be kept $R_{V\_\mathcal{P}}$ and a set of requirements to be removed $R_{\|V\_\bar{\mathcal{P}}\|}$. These cases must be detected during the selection and classification of transitions. Incoherent cases are all incoherencies occurred during the construction of the PL usage model.

The usage model contains transitions not labelled by requirements, but needed to complete the usage model. Some of these transitions may contain also input data and expected responses. In some cases, such transitions can be removed if necessary in order to keep the correctness of the derived usage model variant, otherwise they will be kept.

$R_{V\_\mathcal{P}}$ and $R_{\|V\_\bar{\mathcal{P}}\|}$ will serve to extract and prune transitions of the usage model $\mathcal{M}_{\mathcal{T}}$, corresponding to requirements associated with the variant $\mathcal{P}$. We use $R_{V\_\mathcal{P}}$, because it represents the specific requirements of variant $\mathcal{P}$ to be kept.

Table 1, summarizes the classification rules of each transition associated to one of these subsets of requirements.

$R_{\|V\_\mathcal{P}\|}$	R_{Inter}	$R_{\|V\_\bar{\mathcal{P}}\|}$	
x			$t \in T_K$
x	x		$t \in T_K$
x		x	$t \in T_{incoherent}$
	x		$t \in T_K$
x	x	x	$t \in T_{incoherent}$
	x	x	$t \in T_{incoherent}$
		x	$t \in T_D$

Table 1. Summary for the classification of transitions

4.4 Step D:

After pruning transitions, Algorithm 1 proceeds to derive a usage model variant. This phase consists of deleting all transitions and states that do not correspond to the variant $\mathcal{P}$. In practice, the generation may result in one of the following states: a usage model with broken branches, a complete usage model with incorrect profile probabilities. We discuss these cases in the following sections. The expected result of Algorithm 1 is a valid usage model as defined in Section 2.2.

Removal of transitions First, the model $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ is initialized to $\mathcal{M}_\mathcal{T}$. Afterwards, we remove all transitions in T_D from $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$. At this step no incoherent case exists. So, $T_{incoherent}$ is empty. Nevertheless, some inconsistency may appear in the usage model, such as broken branches. To have a valid model, Algorithm 1 is detecting broken branches to be removed. Fig. 3 illustrates the extracted $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ model with broken branches. For $\mathcal{P}_1$, the broken branches are $\{t_{10}, t_{6\_6}, t_{6\_7}\}$. In the tool implementation, an execution log is intended to help users to identify all deleted transitions and requirements from the PL usage model.

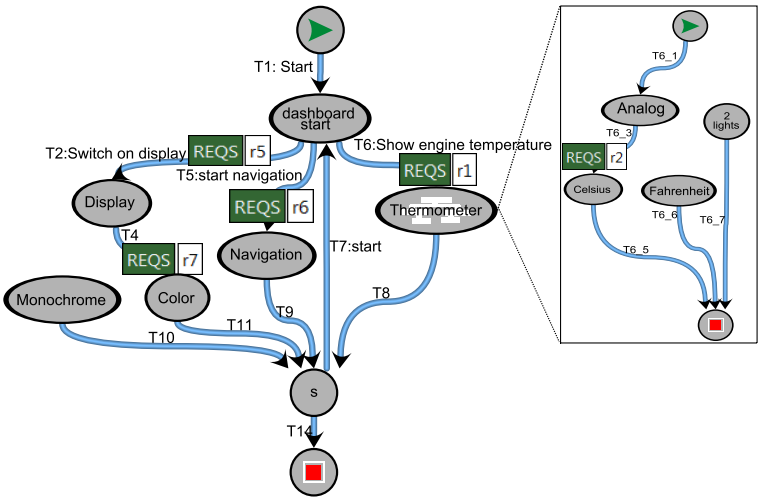

Fig. 3. Usage model of $\mathcal{P}_1$ with its sub-chains

Detection of broken branches and unreachable states deletion In the previous step two problems can occur. Firstly, some states can be unreachable from the initial state s_0 (in case of all incoming states have been removed or they are included in an "island"). Secondly, the situation when it is not possible to reach the final state s_n from another state. So this step consists in detecting all incomplete paths that start in s_0 and where it is not possible to reach s_n.

We denote by $\mathbf{Q}$ the sub-stochastic matrix of size $n+1$ corresponding to the derived usage model, where:

- $q_{ij} = p_{ij}$, if transition $t_{ij} \in T_K$. p_{ij} is the probability related to t_{ij},
- 0 if $t_{ij} \in T_D$,
- 0 if $i = n$, if it is not possible to reach another state from s_n,
- 0 if $j = 0$, if it is not possible to reach s_0 from another state.

Let $\mathbf{B}$ be the matrix such that $\mathbf{B} = (\mathbf{I} - \mathbf{Q})^{-1}$. The probability to visit s_j when process is in s_i is [15]: $f_{ij} = b_{ij}/b_{jj}$.

State s_i and all incoming and outgoing transitions are removed from the model $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ in two specific situations: if $f_{0i} = 0$, in this case s_i is inaccessible, or if $f_{in} = 0$, it is not possible to reach s_n when the process is in state s_i.

After identifying the broken branches, Algorithm 1 proceeds with the suppression of detected unreachable states and the related transitions. However, it is possible that some transitions to be removed belong to the set T_K. This case could result from a bad construction of the PL usage model.

Adjustment of probabilities After removing all broken branches from the extracted usage model, we update the associated usage profile. We choose to distribute the probabilities of the removed transitions proportionally on adjacent transitions. This entails applying the following relation: $q_{ij}(\mathfrak{F}) \longleftarrow \frac{q_{ij}}{\sum_{k \in S} p_{ik}}$.

The derived usage model variant for $\mathcal{P}_1$ is depicted in Fig. 2.

Algorithm 1 Derivation of usage model variant

Input: $\mathcal{M}_{\mathcal{T}}$, T_D, T_K, $T_{incoherent}$
Output: $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$

$\mathcal{M}_{\mathcal{T}_\mathcal{P}} \longleftarrow \mathcal{M}_{\mathcal{T}}$
{Removal of T_D}
remove_transitions($T_D, \mathcal{M}_{\mathcal{T}_\mathcal{P}}$)
{Detection of unreachable states in $\mathcal{M}_{\mathcal{T}}$ see Section 4.4 }
$S \leftarrow getUnreachableStates(\mathcal{M}_{\mathcal{T}_\mathcal{P}})$
{Retrieve all t outgoing and incoming from the state s}
for s **of** S **do**
  $T \longleftarrow s \rightarrow transitions[*]$
  for t **of** T **do**
    if $t \in T_K$ **then**
      $show\_problems(t)$
    else if t has DATA **then**
      $show\_warning(t)$
    end if
  end for
  $remove\_transitions(T, \mathcal{M}_{\mathcal{T}_\mathcal{P}})$
end for
{Removal of all unreachable s}
$remove\_states(S, \mathcal{M}_{\mathcal{T}_\mathcal{P}})$
{Correct probabilities to get a valid Markov chains see Section 4.4}
$Adjust\_probabilities(\mathcal{M}_{\mathcal{T}_\mathcal{P}})$

4.5 Analysis of algorithm complexity & reliability

Complexity (Algorithm 1). To extract the usage model of variant $\mathcal{P}$, the Step D is based on inputs provided by Step C that involved Step A and Step B. The complexity of the algorithms behind these steps A, B and C are not reported here. We focus only on Algorithm 1 of Step D. The preliminary step of Algorithm 1 consists of seeking all non-reachable states from state s_0 to state s_n. We use matrix to enumerate efficiently all unreachable states in a chain. For this method the worst case is a usage model with a large number of states

$(n \times n) \times n$. Therefore the order of complexity is $O(n^3)$, where n is the number of $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ states. The second part of Algorithm 1 removes from $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ all transitions belonging to unreachable states. Subsequently, Algorithm 1 performs a consistency check of the usage model. The order of complexity is $O(n)$, where n is the number of $\mathcal{M}_{\mathcal{T}_\mathcal{P}}$ states.

Proof of Algorithm 1. A test case generation is considered as a random walk on a Markov chain until the end state s_n is reached. We are interested in the visiting probability: this probability must be strictly positive from s_0 to every other states of the model and the final state s_n must be accessible from every state of the chain. This visiting probability is calculated on a chain with absorbing state and means the probability to visit a state before absorption.

So we add a new absorbing state in the model and the probability to go from s_n to its new state s_{n+1} is equal to 1. This construction is required, because we consider the probability to reach the final state in case where an incoming transition of this state has been deleted by the previous step of the Algorithm 1.

To calculate the probability of visiting the states before absorption, we remove the line and the column related to this state and we obtain the matrix $\mathbf{Q}$. The matrix $\mathbf{Q}$ has the following form:

$$\mathbf{Q} = \begin{pmatrix} \mathbf{Q'} & \mathbf{q}_n \\ \mathbf{0} & 0 \end{pmatrix} \tag{9}$$

Where the sub-matrix $\mathbf{Q'}$ is the lines and the columns related to states except the final state s_n and $\mathbf{q}_n$ is the vector of the probabilities to go from states $s_0, ..., s_{n-1}$ to the final state s_n.

The inverse matrix $\mathbf{B} = (\mathbf{I}-\mathbf{Q})^{-1}$ always exists, because $\mathbf{Q}$ is a sub-stochastic matrix, i. e., all elements are included between 0 and 1 and the sum of each line is inferior or equal to 1. It is possible to denote $\mathbf{B}$ on the following form:

$$\mathbf{B} = \begin{pmatrix} \mathbf{B} & \mathbf{b} \\ \mathbf{0} & 1 \end{pmatrix} \tag{10}$$

Where $\mathbf{b}$ is composed of values equal to 0 or 1. Indeed, as we have considered only one absorbing state, either the process is absorbed by this state or it cannot reach it. Consequently, the states to be removed are identified. If we delete all lines and columns where $b_{0i} = 0$ and $b_{in} = 0$, the visiting probabilities could be calculated without difficulty and the model is "clean" without broken branches. Eventually, we add the property that the sum of the probabilities associated to the outgoing transitions of each states is equal to 1 (see line 16 of Algorithm 1).

We can conclude that the usage model is an extended Markov chain, where test cases can be generated by random walks.

4.6 Implementation

The theoretical foundations of the approach are implemented in MaTeLo Product Line Manager [8] (MPLM) tool [16]. MPLM is an Eclipse-based extension of

[8] http://people.irisa.fr/Hamza.Samih/mplm

the MaTeLo tool suite, which supports deriving usage models variants from a PL usage model. MPLM realizes the overall variability testing approach while all the described algorithms are part of the tool. Users can import an OVM model and a PL usage model, link features with requirements, configure variants for testing, and derive variant-specific usage model. In particular, the generated usage models variants can be exploited by the MaTeLo tool to produce automatically test cases for a given variant (product). We report in [10] an experimental case study conducted with the industrial partner Airbus Defence and Space in the frame of the ARTEMIS Joint Undertaking research project MBAT[9] in order to validate the approach from an industrial point of view.

5 Related Work

Our contribution relates to the automatic generation of test cases for a product line (PL). Pohl et al. [1] presented a PL framework, resulting outcome of European projects *Café*, *FAMILIES*, *ESAPS*. The framework is composed of two distinct phases. The *domain engineering* phase aims to define commonality and variability of a reusable set of artefacts. The OVM language has been proposed to document variability. The second phase, called *application engineering*, aims to derive new applications based on a desired combination of features and the actual reuse of artefacts.

Our PL approach promotes the use of functional requirements as an intermediate layer to link the variability model with the test model. The usage model and functional requirements act as reusable artefacts while the derivation phase produces specific test cases. It has the merit to enforce separation of concerns and to be non invasive with current practice – we simply reuse the specification of usage models and requirements. Metzger et al. distinguish software variability (hidden from customers and internal to implementation) from PL variability (visible to customers and external) [11]. They used an OVM model and a feature model for describing the two kinds of variability. We also separate the variability description in a distinct variability model. A notable difference is that we map the OVM model to a set of functional requirements itself connected to a usage model. A key contribution of the paper is to properly define the formal correspondences and to develop automated techniques to derive variants.

Validation and testing of PLs. Numerous research studies have focused on the validation of product lines, offering techniques to optimize and improve this costly phase. Thüm et al. [5] surveyed existing kinds of verification strategies for various kinds of artefacts, e.g., from the checking of feature-related code in isolation to the exploitation of variability information during analysis of the PL. Recent advances in behavioural modelling have also been provided by the model checking community (e.g., see [2]). As argued in [8], testing and model checking techniques can be combined to enforce quality assurance of PLs.

To the best of our knowledge, the only proposal to handle the specific formalism of usage models has been devised by Devroey et al. [17]. A key difference is

[9] http://www.mbat-artemis.eu

that the authors assume the specification of a so-called feature transition system (FTS) to describe the variability of a usage model. The elaboration of a FTS requires a significant amount of work and is another formalism that practitioners need to handle. In our approach, we simply map variability to a set of functional requirements – it has the merit of reusing existing artefacts and current practice is slightly impacted. Another difference is that some variability expressed in the FTS may not be covered in the usage model. In our approach, practitioners start with the usage model and express the necessary and sufficient variability.

Combinatorial testing aims at reducing testing costs when dealing with large and complex systems with many input combinations to test. Different approaches have been proposed to help in this task. For instance, pair-wise techniques aim at minimizing the number of *feature configurations* (i.e., combinations of features) to test while covering each pair of features. Constraint Satisfaction Problem (CSP) [18] and algorithmic [19, 20] approaches have been proposed to obtain coverage configuration sets from a feature model – a widely used formalism for modelling variability.

In our context, the current practice is to manually choose configurations. A natural alternative is to apply combinatorial techniques for automatically generating a subset of configurations from the OVM model. Metzger et al. [11] showed that an OVM model can be translated into a feature model so that efficient automated techniques developed in the context of feature modelling can be reused. With our approach, we can thus envision a fully automated process for deriving usage model variants and test cases.

6 Conclusion

We presented a solution to use model-based testing in the context of product line engineering. The generation of test cases from a usage model (roughly a Markov chain test model) can be performed not only for one variant (product), but for many variants with specific features. The proposed approach augments the description of a usage model with variability information. Variability is described in a separate model in terms of features which are linked to functional requirements of a testable system. Practitioners can project the variability onto a usage model and automatically synthesize usage model variants.

The theoretical foundations of the approach are implemented in an industrial model-based testing tool (MaTeLo). An experimental case study was performed with the industrial partner Airbus Defence and Space in the frame of the ARTEMIS Joint Undertaking research project MBAT. Practitioners report a reduction of the cost for test case development and highlight the minimal invasiveness of the solution so that established requirements and usage models can be reused. A detailed description of the industrial report can be found in [10].

Future work. We are now continuing the experiments of applying the tool-supported approach with other industrial uses cases in other domains, in order to complete our work with concrete quantitative and qualitative results of the industrial case studies. A study is underway to explore the possibility of achiev-

ing multiple profiles for each variant.

Acknowledgments. The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 269335 ARTEMIS project MBAT, French DGCIS and German BMBF.

References

1. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc. (2005)
2. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: ICSE. (2010)
3. Lochau, M., Schaefer, I., Kamischke, J., Lity, S.: Incremental model-based testing of delta-oriented software product lines. In: TAP. (2012)
4. Weißleder, S., Lackner, H.: Top-down and bottom-up approach for model-based testing of product lines. In: MBT. (2013)
5. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Computing Surveys (2014)
6. Utting, M., Legeard, B.: Practical Model-based Testing. Morgan-Kaufmann (2007)
7. Le Guen, H., Thelin, T.: Practical experiences with statistical usage testing. In: Software Technology and Engineering Practice. (2003)
8. Devroey, X., Cordy, M., Perrouin, G., Kang, E.Y., Schobbens, P.Y., Heymans, P., Legay, A., Baudry, B.: A vision for behavioural model-driven validation of software product lines. In: ISoLA'12. (2012) 208–222
9. Sebastian Oster, Andreas Wubbeke, Gregor Engels, Andy Schürr: A survey of model-based software product lines testing. In: MBT for embedded systems. (2011)
10. Samih, H., Acher, M., Bogusch, R., Le Guen, H., Baudry, B.: Deriving Usage Model Variants for Model-based Testing: An Industrial Case Study. In IEEE, ed.: ICECCS'14, Tianjin, Chine (2014)
11. Metzger, A., Pohl, K., Heymans, P., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: RE'07. (2007) 243–253
12. Zander-Nowicka, J.: Model-Based Testing of Real-Time Embedded Systems in the Automotive Domain. Fraunhofer-IRB-Verlag (2009)
13. Le Guen, H., Marie, R., Thelin, T.: Reliability estimation for statistical usage testing using markov chains. In: ISSRE. (2004)
14. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test., Verif. Reliab. **22** (2012) 297–312
15. Çinlar, E.: Introduction to stochastic processes. [nachdr.] edn. Prentice-Hall, Englewood Cliffs, NJ (1975)
16. Samih, H., Bogusch, R.: MPLM – MaTeLo Product Line Manager. In: SPLC'14 Demonstrations and Tools track. (2014, to appear)
17. Devroey, X., Perrouin, G., Cordy, M., Schobbens, P.Y., Legay, A., Heymans, P.: Towards statistical prioritization for software product lines testing. (In: VaMoS'14)
18. Gotlieb, A., Hervieu, A., Baudry, B.: Minimum pairwise coverage using constraint programming techniques. In: ICST'12. (2012) 773–774
19. Johansen, M.F., Haugen, O.y., Fleurey, F.: An algorithm for generating t-wise covering arrays from large feature models. SPLC '12 (2012) 46
20. Perrouin, G., Sen, S., Klein, J., Baudry, B., Traon, Y.l.: Automated and scalable t-wise test case generation strategies for software product lines. In: ICST'10, (IEEE)