Statistical Prioritization for Software Product Line Testing: an Experience Report

Xavier Devroey, Gilles Perrouin, Maxime Cordy, Hamza Samih, Axel Legay, Pierre-Yves Schobbens, Patrick Heymans

▶ **To cite this version:**

Xavier Devroey, Gilles Perrouin, Maxime Cordy, Hamza Samih, Axel Legay, et al.. Statistical Prioritization for Software Product Line Testing: an Experience Report. Software and Systems Modeling, 2017, 16 (1), pp.153-171. 10.1007/s10270-015-0479-8 . hal-01642289

HAL Id: hal-01642289
https://inria.hal.science/hal-01642289

Submitted on 21 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Noname manuscript No.
(will be inserted by the editor)

Statistical Prioritization for Software Product Line Testing: An Experience Report

Xavier Devroey · Gilles Perrouin · Maxime Cordy · Hamza Samih · Axel Legay · Pierre-Yves Schobbens · Patrick Heymans

Received: date / Accepted: date

Abstract Software Product Lines (SPLs), are families of software systems sharing common assets and exhibiting variabilities specific to each product member of the family. Commonalities and variabilities are often represented as features organised in a feature model. due to combinatorial explosion of the number of products induced by possible features combinations, exhaustive testing of SPLs is intractable. Therefore, sampling and prioritisation techniques have been proposed to generate fault-finding, sorted lists of products based on coverage criteria or weights assigned to features. Solely based on the feature model, these technique do not take into account behavioural usage of such products as a source of prioritisation. In this paper we assess the feasibility of integrating usage models into the testing process to derive statistical testing approaches for SPLs. Usage models are given as a Markov chains enabling the selection of probable/rare behaviours that can then be analysed against Featured Transition Systems (FTSs), acting as design models of the SPLs, to determine which features and products are realizing these behaviours. Statistical prioritisation can achieve a significant reduction of the state space, and modelling efforts can be rewarded by easing tool integration. In particular we used MaTeLo, a statistical test cases generation suite developed at ALL4TEC. Our experience report is based on the evaluation of our feasibil-

X. Devroey, G. Perrouin (FNRS Postdoctoral Researcher), M. Cordy (FNRS Research Fellow), P.-Y. Schobbens and P. Heymans
PReCISE, University of Namur, rue de Bruxelles 61, B-5000 Namur, Belgium
E-mail: xavier.devroey@unamur.be, gilles.perrouin@unamur.be, maxime.cordy@unamur.be, pierre-yves.schobbens@unamur.be, patrick.heymans@unamur.be

H. Samih
Current affiliation: Alcatel-Lucent, IP T&R / WIRELESS TRANSMISSION
Route De VilleJust, 91620 Nozay France.
Affiliation at the time of work: ALL4TEC, France & Inria Rennes, Bretagne Atlantique, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
E-mail: hamza.samih@gmail.com

A. Legay
Inria Rennes, Bretagne Atlantique, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
E-mail: axel.legay@inria.fr

ity criteria on two different systems: Claroline, a configurable course management system, and Sferion^{TM}, dealing with an embedded helicopter landing function.

Keywords Software Product Line Testing · Prioritization · Statistical Testing

CR Subject Classification D.2.5 · D.2.7

1 Introduction

Software Product Line (SPL) engineering is based on the idea that products of the same family can be built by systematically reusing assets, some of them being common to all members whereas others are only shared by a subset of the family. Such variability is commonly captured by the notion of *feature*. Individual features can be specified using languages such as UML, while their inter-relationships are organized in a *Feature Diagram* (FD) [33].

SPL testing is the most common quality assurance technique in SPL engineering. As opposed to single-system testing, where the testing process considers only one software product, SPL testing is concerned about minimizing the test effort for the SPL products. Testing all products separately is clearly infeasible in real-world SPLs, which typically consist of myriads of possible products. Automated model-based testing [57] and shared execution [34] are established testing methods that allows test reuse across a set of software. They can thus be used to reduce the SPL testing effort. Even so, the problem remains as these methods still need to cover all the products.

Other approaches consist in testing only a representative sample of the possible products. Typical methods select this sample according to some coverage criterion on the FD (e.g. all the valid couples of features must occur in at least one tested product [16, 43]). An alternative method is to associate each feature with a weight and prioritize the products with the highest weight [31,32]. It helps testers to scope more finely and flexibly relevant products to test than a covering criteria alone. Yet, assigning meaningful weights is cumbersome in the absence of additional information regarding their behaviour.

Our effort in providing statistical testing techniques [62] for SPLs [22, 47, 49] targets selecting products and generating test cases with respect to an *usage model*, i.e., a *Discrete-Time Markov Chain* (DTMC). This usage model represents the usage scenarios of the software under test as well as their respective probability. The latter information allows one to determine the likelihood of execution scenarios, and to rank these accordingly either for one or a subset of the products of the SPL. There are two ways to associate scenarios to SPL products:

– The *family-based* approach consists in exploiting logs as a source of user information and infer the usage model using machine learning techniques. We can then extract behaviour according to a probability range and relate them to the design and feature models of the SPL [22]. The design model of the SPL is provided explicitly as a *Featured Transition System* (FTS) i.e. a fundamental formalism initially dedicated to SPL model-checking [10]. As they are extracted from the usage model, behaviours can be run on the FTS to determine which products/features are involved (which corresponds to the prioritized products/features set). To ease additional analyses, such as structural

test case generation (e.g. all-paths), our approach prunes the original FTS and derives one corresponding to only extracted behaviour.
– The *product-based* approach consists in modelling usage models from the onset, taking into account requirements and feature models as well as translating expert knowledge to probabilities in the usage model [47]. Concretely, requirements are related to features (specified in an orthogonal variability model) via a product matrix while the usage model directly relates it transitions to probabilities and requirements of the SPL [47, 49]. Then, testers have to manually specify the products they are interested in and, with the help of an industrial tool [2, 48], derive a pruned usage model corresponding to the behaviour of these products and perform automated test case generation.

In this paper we report on our previous [22] and new experiences with these two scenarios and demonstrate that these scenarios can be combined, notably to assist testers in product-based testing using family-based reasoning and prioritisation. To do so, we define *feasibility criteria*, that consider the reduction in size of the FTS prior to testing, the scalability of the prioritisation algorithm and the effort in obtaining the required models. By assessing these criteria on two very different systems (a configurable course management system and an embedded SPL) we demonstrate the feasibility of the proposed solutions and outline some future directions.

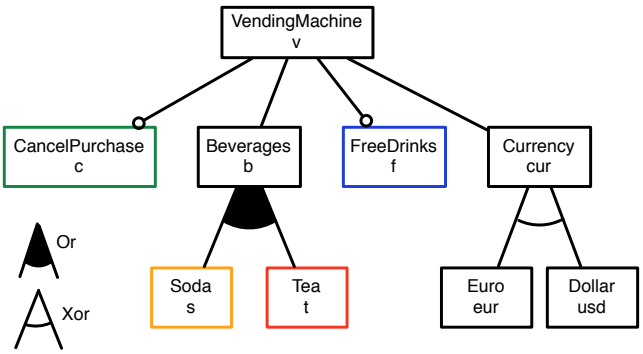
The remainder of this paper is organized as follows: Section 2 presents the theoretical background supporting our approaches. Our statistical testing techniques are detailed in Section 3. We report on our initial and novel experience with the Sferion^{TM}landing symbology function in Section 4. Section 5 discusses related research, Section 6 presents challenges and future research directions and Section 7 concludes the paper.

2 Background

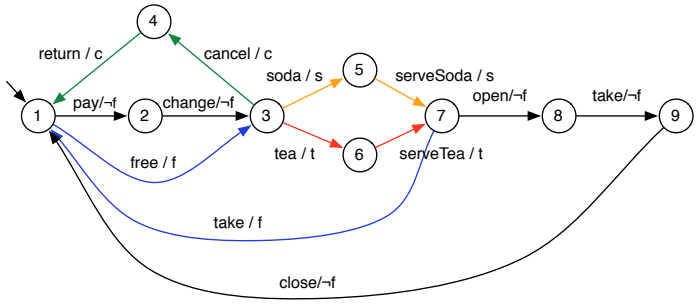
In this section, we present the foundations underlying our approach: *SPL modeling* and *statistical testing*.

2.1 SPL Modelling

Variability Modelling A key concern in SPL modelling is how to represent variability. To achieve this purpose, SPL engineers usually reason in terms of features, defined as end-user visible characteristics of a system [44]. Relations and constraints between features are represented in a Feature Diagram (FD) [33]. For example, Fig. 1a presents the FD of a beverage vending machine [10]. A common semantics associated to a FD d (noted $[\![d]\!]$) is the set of all the valid products allowed by d. A product derived from this diagram will correspond to a set of selected features. In Fig. 1a, $\{v, b, t, cur, usd\}$ corresponds to a machine that sells only tea and accept only dollar. FDs have been equipped with formal semantics [51], automated analyses and tools [33] for more than 20 years.



(a) Feature Diagram (FD)



(b) Featured Transition System (FTS)

Fig. 1 The soda vending machine example [10]

Behavioural Modelling Different formalisms may be used to model the behaviour of a system. To allow the explicit mapping from features to SPL behaviour, Featured Transition Systems (FTSs) [10] were proposed. FTSs are transition systems (TSs) where each transition is labelled with a feature expression (i.e., a boolean expression over features of the SPL), specifying which products can execute the transition. Formally, an FTS is a tuple $(S, Act, trans, i, d, \gamma)$ where :

- S is a set of states;
- Act a set of actions;
- $trans \subseteq S \times Act \times S$ is the transition relation (with $(s_1, \alpha, s_2) \in trans$ sometimes noted $s_1 \xrightarrow{\alpha} s_2$);
- $i \in S$ is the initial state;
- d is a FD;
- $\gamma : trans \rightarrow [\![d]\!] \rightarrow \{\top, \bot\}$ is a total function labelling each transition with a boolean expression over the features, which specifies the products that can execute the transition.

For instance: $\neg f$ in Fig. 1b indicates that only products that have not the $free$ feature may fire the $pay, change, open, take$ and $close$ transitions. A TS modelling the behaviour of a given product is obtained by removing the transitions whose feature expression is not satisfied by the product (i.e., set of features selected as part of the product). We define the semantics of an FTS as a function that

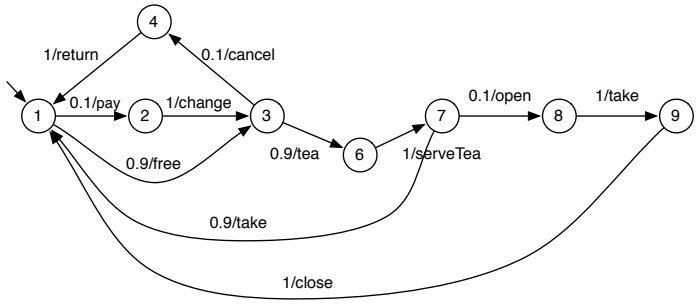


Fig. 2 The soda vending machine example usage model (DTMC)

associates each valid product with its set of finite and infinite traces, i.e. all the alternating sequences of states and actions starting from the initial state available, satisfying the transition relation and such that its transitions are available to that product. According to this definition, an FTS is indeed a behavioural model of a whole SPL. Fig. 1b presents the FTS modeling a vending machine SPL. This definition differs from the one presented by Classen et al. [10], where only infinite paths are considered. In a testing context, one may also be interested in finite paths.

2.2 Statistical Testing

Whittaker and Thomason introduced the notion of usage model [62] and define it as a TS where transitions have probability values corresponding to their occurrence likelihood. Formally, a usage model is equivalent to a DTMC where transitions are additionally decorated with actions, i.e. a tuple $(S, Act, trans, P, \tau)$ where :

- $(S, Act, trans)$ are respectively a set of states, a set of actions, and a set of transitions $(S \times Act \times S)$;
- $P : S \times Act \times S \to [0, 1]$ is the probability function that associates each transition (s_i, α, s_j) the probability for the system in state s_i to execute action α and reach state s_j;
- $\tau : S \to [0, 1]$ is the vector containing the probabilities to be in the initial state when the system starts;
- $\forall s_i \in S : \sum_{\alpha \in Act, s_j \in S \bullet s_i \xrightarrow{\alpha} s_j} P(s_i, \alpha, s_j) = 1$, that is, the total of the probabilities of the transitions leaving a state must be equal to 1.

Note that in their original definition, DTMCs have no action. We need them here to relate transitions in a DTMC with their counterpart in an FTS. Still, our alternate definition remains in accordance with the action-less variant. Fundamentally, in both cases, the probability function associates probability values to transitions, regardless of the definition of transitions (either as sequences of states and actions or as sequences of states). Also, we consider that there is a single initial state i, that is, $\tau(i) = 1$.

3 Statistical Prioritization in SPL Testing

In our approach, we consider 3 models:

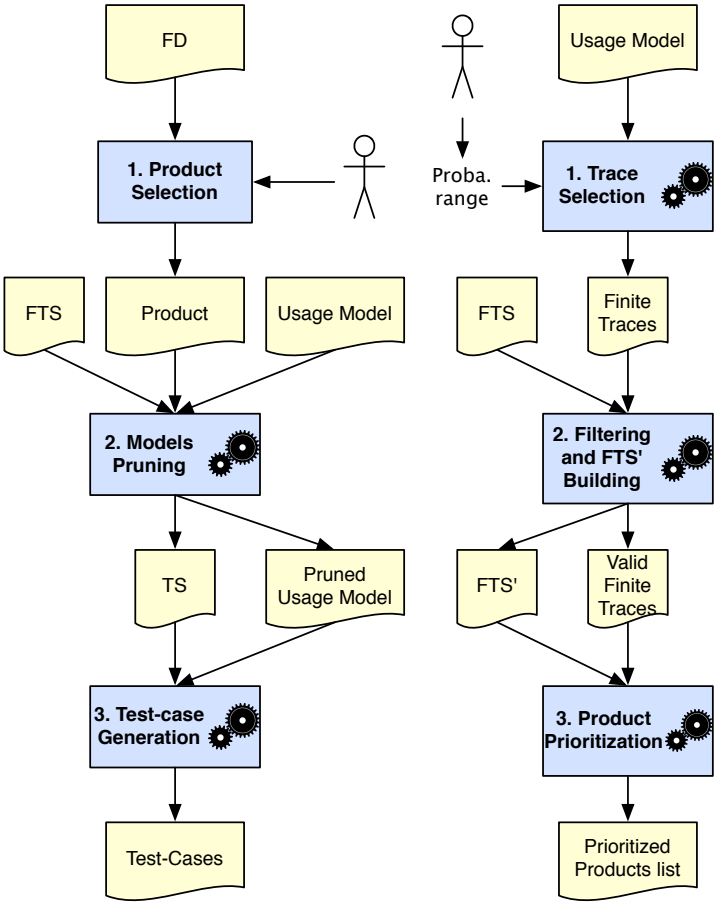
1. a FD d to represent the features and their constraints (in Fig. 1a);
2. an FTS fts over d (in Fig. 1b);
3. and a usage model represented by a DTMC $dtmc$ (in Fig. 2.2) whose states, actions, and transitions are subsets of their counterpart in fts:

$$S_{dtmc} \subseteq S_{fts} \wedge Act_{dtmc} \subseteq Act_{fts} \wedge trans_{dtmc} \subseteq trans_{fts}$$

Moreover, the initial state i of fts is in S_{dtmc} and has an initial probability of 1, that is, $\tau(i) = 1$.

An alternative would have been to rely on a unified formalism able to represent both variability and probability (see, e.g., Cordy et al. [18]) in the same model. However, we believe it is more appropriate to maintain a separation of concerns for several reasons: first, existing model-based testing tools relying on usage models (e.g., MaTeLo [2]) do not support usage models with variability. Implementing our approach on top of such tools would be made harder should we use the aforementioned unified formalism (see also discussion in Section 4.4).
Second, the usage model can be defined by a system expert, but also obtained from users trying the software under test, extracted from logs, or from running code. These extractions methods are agnostic regarding SPL features and the resulting usage model will represents the behaviour of one or more products of the product line. One may merge logs of different products of the same product line and use model inference techniques to build the usage model [54,59]. The correctness of the usage model (i.e., the usage model only represents valid behaviour of the SPL) depends on the inference method used and the number and size of logs used as entry of this method. Methods producing correct usage models may be interesting but are more computationally expensive and may not scale for large logs [54]. Less accurate methods (e.g., 2-gram inference) scales for very large logs and real data [59] but may produce usage models with errors in the description of the behaviour of the SPL. For example, in the usage model of Fig. 2.2 one can follow the path *pay*, *change*, *tea*, *serveTea*, *take*. This path actually mixes "pay machine" (feature f not enabled) and "free machine" (feature f enabled). The combined use of usage models and FTSs allows us to detect such inconsistencies. For large and/or numerous logs, building the usage model will result from a trade off between correctness and "budget" (time, computing resources).
Third, since the usage model may be built from existing software executions, it may be incomplete (as in Fig. 2.2). There may exist FTS executions that are not exercised in the usage model, resulting in missing transitions in the usage model. Keeping the FTS and usage models in isolation is helpful to identify and correct such issues.
Finally, classical statistical testing tools (like MaTeLo [2]) allows to define multiple usage profiles, corresponding to different probability assignations to transitions, e.g., to represent usages of different user roles (administrator, registered user, etc.). In our approach, those usage profiles are represented by different usage models that may be used with the same FTS.

Using the 3 models, we distinguish 2 possible test scenarios: *product-based test derivation* (Fig. 3a) [46] and *family-based test prioritization* (Fig. 3b). *Product-based*



(a) Product-Based Test Derivation (b) Family-Based Test Prioritization

Fig. 3 Family-Based and Product-Based Approaches

test derivation makes the assumption that a model of the product line may be pruned in order to get a valid model for each product [7,47,49,61]: the test engineer selects a product to test from the FD; the usage model is automatically pruned using the FTS, giving a TS and a pruned usage model representing the behaviour of this product; the test cases are generated from the pruned usage model using existing test case generation tools (e.g., MaTeLo [2] in our case). The process may be repeated for a set of representative products (e.g., selected using pairwise testing [30,31,36,43]). We propose *Family-based test prioritization*, which prioritizes products to test in a SPL according to their behaviour. The idea is to build a usage model, automatically select most representative traces in this model according to their probability to occur (a probability interval defined by the engineer in our case), and see (using the FTS) which products may execute those traces. The list of valid traces is sorted in order to determine the products to test in priority.

3.1 Product-Based Test Derivation

Product-based test derivation in Fig. 3a is straightforward: the engineer selects one product (by selecting features in the FD) to test, the tool then automatically extracts from the FTS a TS corresponding to the product, and prunes the usage model to keep the following property true : $S_{dtmc} \subseteq S_{ts} \wedge Act_{dtmc} \subseteq Act_{ts} \wedge trans_{dtmc} \subseteq trans_{ts}$. We assume that the probabilities of the removed transitions are proportionally distributed on adjacent transitions, so that the probability axiom $\forall s_i \in S : \sum_{s_j \in S} P(s_i, s_j) = 1$ holds and balance between the probabilities of the transitions with a same source state are kept [49]. Finally, the tool generates abstract test cases (i.e., sequences of actions) [23] using statistical testing algorithms on the usage model [25, 62]. Those abstract test cases will have to be refined by providing inputs and expected outputs for the different actions [57].

This scenario is proposed by Samih et al. [47, 49] in the MaTeLo Product Line Manager (MPLM) tool. Product selection is made on an orthogonal variability model (OVM) and mapping between the OVM and the usage model (build by a system expert using MaTeLo [2]) is provided via explicit traceability links to functional requirements. This process requires to perform selection of products of interest on the variability model and does not exploit probabilities and traces of the usage model during such selection. Additionally, they assume that tests for all products of the SPL are modeled in the usage model. This assumption may be too strong in certain cases and delay actual testing since designing the complete usage model for a large SPL may take time.

3.2 Family-Based Test Prioritization

Contrary to product-based test derivation, our prioritization process (in Fig. 3b) supports partial coverage of the SPL by the usage model. For instance, the usage model represented in Fig. 2.2 does not cover serving soda behaviour because no user exercised it. The key idea is to generate traces from the usage model according to their probability to happen using a interval given by the engineer (step 1). Only traces in the model with a probability in this interval will be considered. E.g., one may be interested in analysing highly probable behaviours (interval $[0.5, 1]$). Only one trace has a probability in this range: $Pr(free, tea, serveTea, take) = 0.729$, which corresponds to the behaviour "serving teas for free". The generated traces are filtered using the FTS in order to keep only sequences that may be executed by at least one product of the SPL (step 2). The result will be a pruned FTS, named FTS', according to the extracted traces. Each valid trace is executed on FTS' to determine a set of products that may effectively execute this behaviour. The probability of the trace to be executed allows us to prioritize products exercising the behaviour described in FTS' (step 3).

3.2.1 Trace Selection in the usage model

The first step is to extract traces from the usage model according to desired parameters provided by the tester. Formally, a finite trace t is a finite alternating sequence $t = (i = s_0, \alpha_0, ..., \alpha_{n-1}, s_n)$ such that $(s_j, \alpha_{j+1}, s_{j+1}) \in trans, \forall j \in [0, n-1]$. To perform trace selection in a usage model *dtmc*, we apply a Depth-First Search

(DFS) algorithm parametrized with a maximum length l_{max} for finite traces and an interval $[Pr_{min}, Pr_{max}]$ specifying the minimal and maximal values for the probabilities of selected traces. Formally:

$$DFS(l_{max}, Pr_{min}, Pr_{max}, dtmc) = \{(i, \alpha_1, ..., \alpha_n, i) \mid \\ n < lmax \wedge (\nexists k : 0 < k < n \bullet i = s_k) \wedge (Pr_{min} \leq Pr(i, \alpha_1, ..., \alpha_n, i) \leq Pr_{max})\}$$

where $\tau_{dtmc}(i) = 1$ and

$$Pr(i, \alpha_0, \ldots, s_n) = \tau_{dtmc}(i) \times \Pi_{j=0}^{n-1} P_{dtmc}(s_j, \alpha_j, s_{j+1}).$$

We initially consider only finite traces starting from and ending in the initial state i (assimilate to an accepting state) without passing by i in between. Theses correspond to a coherent execution scenario in the usage model. With respect to partial finite traces (i.e., finite traces not ending in i), our trace definition involve a smaller state space to explore in the usage model than a all-paths execution [37]. This is due to the fact that the exploration of a part of the graph may be stopped, without further checks of the existence of partial finite traces, as long as the partial trace is higher than l_{max}. The l_{max} parameter allows the DFS algorithm to scale to large usage models [22].

The interval $[Pr_{min}, Pr_{max}]$ is provided by the engineer based on his knowledge of the SPL and the test prioritization purpose: an interval with high values (e.g., $[0.5, 1]$) will be more likely to give highly probable behaviours of the SPL. This is often desired for non-regression testing scenario where the engineer wants to ensure that the main functionalities of a SPL are still reliable after an update [37]. Assuming that the usage model has been built using logs of running products, the engineer may also be interested in testing behaviours with a low probability as they may find rare bugs not discovered by the users of the products. Such strategies can be used, e.g., for intrusion detection [27].

The interval, its relevance, and generated test cases will depend on the usage model source (e.g., built from running products, manually built by an engineer, etc.) and the usage model shape (i.e., number of states, transitions, average states degree, etc.). To have an idea of the interval to choose and the number of traces that will be selected, the engineer may use random walks in the usage model to generate random traces with their probability and see how they are distributed.

Practically, this algorithm will build a tree where a node represents a state with the probability to reach it and the branches are the α_k labels of the transitions taken from the state associated to the node. The root node corresponds to the initial state i and has a probability of 1. Since we are only interested in finite traces ending in the initial state, the exploration of a branch of the tree is stopped when the depth is higher than then maximal path l_{max}. This parameter is provided to the algorithm by the test engineer and is only used to avoid infinite loops during the exploration of the usage model. Its value will depend on the size of the usage model and should be higher than the maximal "loop free" path in the usage model in order to get coherent finite traces.

For instance, the execution of the algorithm on the soda vending machine (vm) example presented in Fig. 1b, with a l_{max} value of 7 (the size of the maximal loop free path) and an interval $[0, 0.1]$ to capture the least probable traces, gives 5 finite

Require: $traces, fts$
Ensure: $traces, fts'$
1: $S_{fts'} \leftarrow \{i_{fts}\}$; $i_{fts'} \leftarrow i_{fts}$; $d_{fts'} \leftarrow d_{fts}$
2: **for all** $t \in traces$ **do**
3: **if** $accept(fts, t)$ **then**
4: $S_{fts'} \leftarrow S_{fts'} \cup states(fts, t)$
5: $Act_{fts'} \leftarrow Act_{fts'} \cup t$
6: $trans_{fts'} \leftarrow trans_{fts'} \cup transitions(fts, t)$
7: $\gamma_{fts'} \leftarrow fLabels(fts, t)\gamma_{fts'}$
8: **else**
9: $traces \leftarrow traces \setminus \{t\}$
10: **end if**
11: **end for**
12: **return** fts'

Fig. 4 FTS' building algorithm

traces:

$$DFS(7; 0; 0.1; dtmc_{vm}) = \{$$
$$(pay, change, cancel, return); (free, cancel, return);$$
$$(pay, change, tea, serveTea, open, take, close);$$
$$(pay, change, tea, serveTea, take);$$
$$(free, tea, serveTea, open, take, close)\}$$

During the execution of the algorithm, the trace ($free$, tea, $serveTea$, $take$) has been rejected since its probability (0.729) is not between 0 and 0.1.

The downside is that the algorithm will possibly enumerate all the paths in the usage model depending on the l_{max} value. This can be problematic and we plan in our future work to use symbolic executions techniques inspired by work in the probabilistic model checking area, especially automata-based representations in order to avoid a complete state space exploration [12].

3.2.2 FTS-Based Trace Filtering and FTS Pruning

We do not make any assumptions about the source of the usage model. Therefore, step 2 serves as a sanity check to ensure that selected traces correspond to behaviour that may be executed at least one valid product of the SPL. The set of products able of executing a trace t may be calculated from the FTS (and its associated FD). It corresponds to all the products (i.e., set of features) of the FD ($[\![d]\!]$) that satisfy all the feature expressions associated to the transitions of t; i.e., $prod(t, fts) = \bigcap_{k=1}^{n} \{p \mid \gamma_{fts}(s_k \xrightarrow{\alpha_k} s_{k+1})(p) = true\}$. From a practical point of view, the set of products corresponds to the products satisfying the conjunction of the feature expressions $\gamma_{fts}(s_k \xrightarrow{\alpha_k} s_{k+1})$ on the path of t and the FD d_{fts}. As d_{fts} may be transformed to a boolean formula where features become variables [19], the following formula can be calculated using a SAT solver: $\bigwedge_{k=1}^{n}(\gamma_{fts}(s_k \xrightarrow{\alpha_k} s_{k+1})) \wedge booleanForm(d_{fts})$.

To be valid, a trace has to be executable by at least one product of the SPL ($prod(t, fts) \neq \emptyset$), if this is not the case, there is an error in the usage model or in the FTS. Depending on how the model has been built, the error may come from the engineer who built the model (e.g., missing transition/state, extra transition/state,

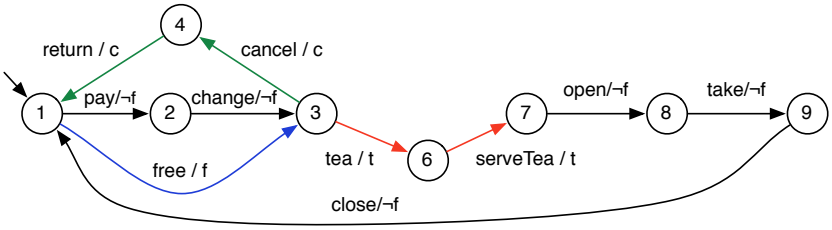


Fig. 5 FTS' of the soda vending machine

wrong feature expression on a transition of the FTS, etc.) or, if the error is in the usage model, from the model inference method used to generate the model from a set of execution traces. Such errors have to be detected and reported to the engineer who will decide what to do: either correct the usage model or the FTS in order to avoid illegal behaviours; or ignore the error if it is not significant.

In our approach, the set of generated finite traces has to be filtered using the FTS such that the following property holds: for a given FTS fts and a usage model $dtmc$, a finite trace t generated from $dtmc$ represents a valid behaviour for the product line pl modelled by fts iff there exists a product p in pl such that $t \subseteq [\![fts_{|p}]\!]_{TS}$, where $fts_{|p}$ represents the projection of fts using product p and $[\![ts]\!]_{TS}$ represents all the possible traces and their prefixes for a TS ts. The idea here is to use the FTS to detect invalid finite traces by running them on it.

Practically, we will build a second *FTS'* which will represent only the behaviour of the SPL appearing in the valid finite traces generated from the usage model. This FTS' represents a prioritized subset of the original FTS that may be used to generate test cases [23]. Fig. 4 presents the algorithm used to build an fts' from a set of *traces* (filtered during the algorithm) and a fts. The initial state of fts' corresponds to the initial state of the fts (line 1) and d in fts' is the same as for fts (line 1). If a given trace is accepted by the fts (line 3), then the states, actions and transitions visited in fts when executing the trace t are added to fts' (line 4 to 6). The $accept(fts, t)$ function on line 3 will return true if there exists at least one product in d_{fts} that has t as one of its behaviours. On line 7, the $fLabels(fts, t)$ function is used to enrich the $\gamma_{fts'}$ function with the feature expressions of the transitions visited when executing t on the fts. It has the following signature: $fLabels : (FTS, trace) \rightarrow (trans \rightarrow [\![d]\!] \rightarrow \{\top, \bot\}) \rightarrow (trans \rightarrow [\![d]\!] \rightarrow \{\top, \bot\})$ and $fLabels(fts, t)\gamma_{fts'}$ will return a new function $\gamma'_{fts'}$ which will for a given transition $tr = (s_i \xrightarrow{\alpha_k} s_j)$ return $\gamma_{fts}(tr)$ if $\alpha_k \in t$ or $\gamma_{fts'}(tr)$ otherwise.

In our vm example, the set of finite traces with a probability between 0 and 0.1 selected in step 1 contains two illegal traces: (pay, $change$, tea, $serveTea$, $take$) and ($free$, tea, $serveTea$, $open$, $take$, $close$), which both violate the $(free \wedge \neg free)$ expression. Those 2 traces (mixing free and not free vending machines) cannot be executed on the fts_{vm} and will be rejected in step 2. The generated fts'_{vm} is presented in Fig. 5.

3.2.3 Product Prioritization

At the end of step 2 in Fig. 3b, we have an FTS' and a set of finite traces in this FTS'. This set of finite traces (coming from the usage model) covers all the

valid behaviours of the FTS'. It is thus possible to order them according to their probability to happen. This probability corresponds to the the cumulated individual probabilities of the transitions fired when executing the finite trace in the usage model. A valid finite trace $t = (i, \alpha_1, \ldots, \alpha_n, i)$ corresponding to a path $(i \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} i)$ in the usage model (and in the FTS') has a probability $Pr(t)$ (calculated as in step 1) to be executed. In order to improve the implementation, we may retain the probabilities of the traces ($Pr(t)$) from step 1.

At this step, each valid finite trace t is associated to the set of products $prod(t, fts')$ that can actually execute t with a probability $Pr(t)$. Product prioritization may be done by classifying the finite traces according to their probability to be executed, giving t-behaviourally equivalent classes of products for each finite trace t. For instance, for the trace $t_{vm} = (pay, change, tea, serveTea, open, take, close)$, generated for our vm example, the products will have to satisfy:

$$\neg f \wedge t \wedge booleanForm(d_{vm})$$

This gives us a set of 8 products (amongst 32 possible):

$$\{(v, b, cur, t, eur); (v, b, cur, t, usd); (v, b, cur, t, c, eur);$$
$$(v, b, cur, t, c, usd); (v, b, cur, t, s, eur); (v, b, cur, t, s, usd);$$
$$(v, b, cur, t, s, c, eur); (v, b, cur, t, s, c, usd)\}$$

Each of them executing t_{vm} with a probability $Pr(t_{vm}) = 0.009$, which is the behaviour of the soda vending machine with the lowest probability.

4 Feasibility Study

In this section, we report on the feasibility of using and combining product-based and family-based scenarios. By applying them on two systems: the first one is *Claroline* [8], an open-source web-based application dedicated to learning and online collaborative work; the second one is the landing symbology function, part of Sferion^{TM}, an industrial software supporting helicopter pilots during the landing approach in degraded visual environment [1, 48].

4.1 Feasibility Criteria

We assess the ***feasibility*** of our approach using the following criteria:

1. ***FTS pruning***: What are the reductions gains (model pruning) achieved by applying statistical prioritization?
2. ***Modelling***: What is the modelling effort induced by our approach and what are the consequences of modelling choices?
3. ***Scalability***: How does prioritization scale to increasing probability ranges and what are the implications for testing?

It is difficult to provide precise thresholds for these criteria. Testing should fit a given budget, which is a complex trade-off involving testing time, human and infrastructure resources, level of system coverage desired, etc. Statistical approaches covered in this paper are flexible to meet such a tradeoff. We argue that fixing

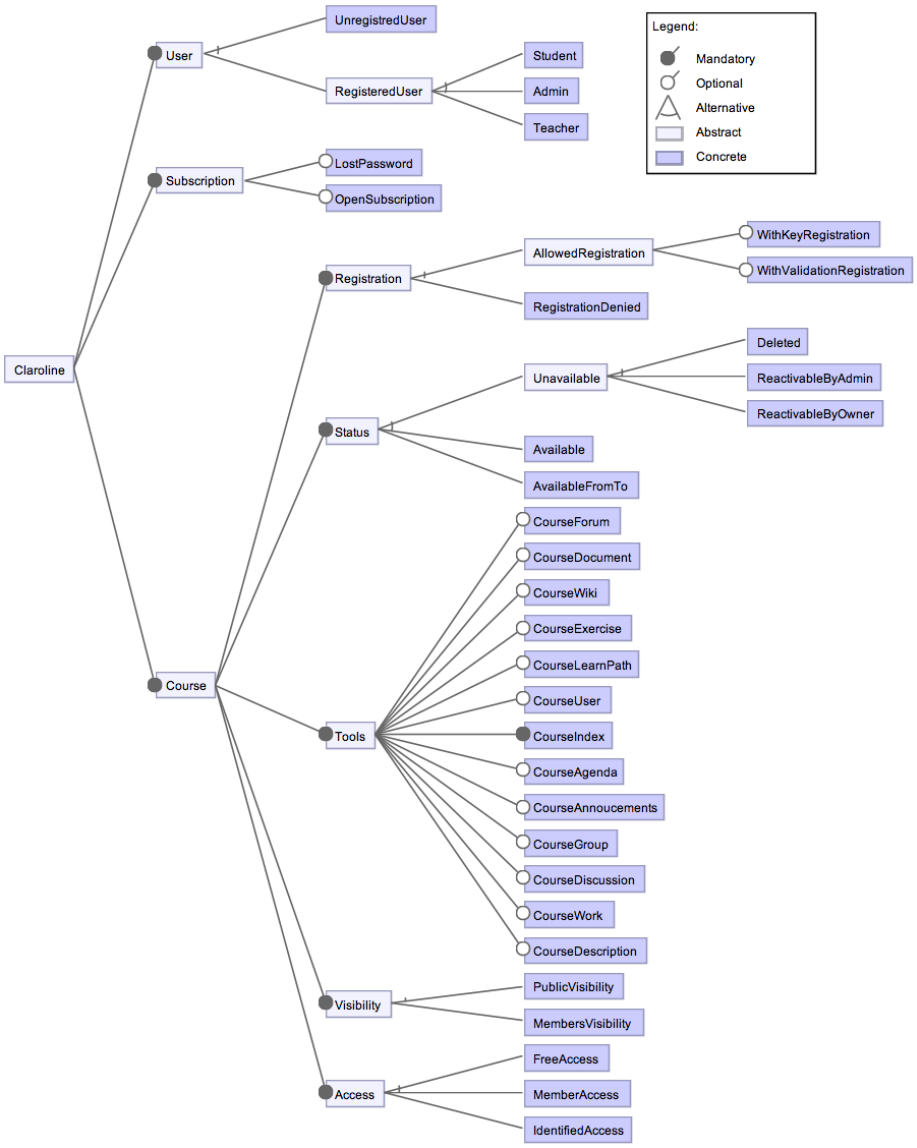


Fig. 6 Claroline Feature Diagram

meaningful thresholds values requires additional experience, especially in industrial settings where they both can be set and assessed. We therefore leave this issue for future work, giving both quantitative and qualitative information stemming from our experience applying our techniques on Claroline and Sferion^{TM}.

4.2 Claroline, an online course management system

The Claroline SPL[1] has already been presented in our previous work [22]. The instance of Claroline at University of Namur[2] is the main communication channel between students and lecturers and is used by approximately 7000 users. Students may register to courses and download documents, receive announcements, submit their assignments, perform online exercises, etc. Claroline is a configurable system [14]. Contrary to classical SPL, the selection of the features does not occur during the development of the software [44] (design time), but during its execution (runtime). Thus, a configuration can dynamically evolve while the system is running: this requires the system architecture to be able to accommodate evolutions, by following plugin-based or component-based architectural styles. Thanks to the versatility of the feature concept [11], it is possible to represent design time and runtime configurations using the same formalism (FD), as configuration semantics is ultimately given through the mapping with the FTS. In the Claroline case, features represent installation parameters. A configuration represents a running Claroline instance with a minimal set of data.

Usage Model Inference We derived the usage model[3], from an anonymized Apache access log containing 12.689.030 HTTP requests provided by the IT support team of the university (5,26 Go), using *n-gram* (contiguous sequence of n elements) inference technique [28, 50, 54]. We were able to process the entirety of our log database in approximately two hours, ending up in a usage model formed of 96 states and 2149 transitions.

Building Family Models To obtain the Claroline FD and FTS we proceeded the following way. The FD was built manually from the Claroline documentation and by inspecting a locally installed Claroline instance (Claroline 1.11.7) in approximately 3 days (by one person). The FD in Fig. 6 (additional constraints have been omitted due to space constraint) describes Claroline with three main features: $User$, $Course$ and $Subscription$. $Subscription$ may be open to everyone (*opt Open Subscription*) and may have a password recovery mechanism (*opt Lost Password*). User corresponds to the different possible user types provided by default with a basic Claroline installation: unregistered users ($UnregisteredUser$) who may access courses open to everyone and registered users ($RegisteredUser$) who may access different functionalities of the courses according to their privilege level (*Student*, *Teacher* or *Admin*). The last main feature, $Course$, corresponds to the page dedicated to a course where students and teacher may interact. A course has a status ($Available$, $AvailableFromTo$ or $Unavailable$), may be publicly visible ($PublicVisibility$) or not ($MembersVisibility$), may authorize registration to identified users (*Allowed Registration*) or not (*Registration Denied*) and may be accessed by everyone ($FreeAccess$), identified users ($IdentifiedAccess$) or members of the course only ($MembersAccess$). Moreover, a course may have a list of tools ($Tools$) used for different teaching purposes, e.g., an agenda (*opt CourseAgenda*),

[1] Complete models are downloadable at `https://projects.info.unamur.be/vibes/`

[2] `http://webcampus.unamur.be`

[3] Tool implementation may be downloaded at `https://projects.info.unamur.be/yami/`

Table 1 Claroline Traces Selection

	Run 1	Run 2	Run 3	Run 4
l_{max}	98	98	98	98
Pr_{min}	$1E^{-4}$	$1E^{-5}$	$1E^{-6}$	$1E^{-7}$
Pr_{max}	1	1	1	1
#DTMC tr.	211	1389	9287	62112
#Valid tr.	211	1389	9287	62112
Avg. size	$4,82$	$5,51$	$6,35$	$7,17$
σ size	$1,54$	$1,54$	$1,62$	$1,66$
Avg. proba.	$2,06E^{-3}$	$3,36E^{-4}$	$5,26E^{-5}$	$8,10E^{-6}$
σ proba	$1,39E^{-2}$	$5,46E^{-3}$	$2,12E^{-3}$	$8,18E^{-4}$
#FTS' st.	16	36	50	69
#FTS' tr.	66	224	442	844

an announcement panel (*opt CourseAnnoucements*), a document download section where lecturers may post documents and students may download them (*opt CourseDocument*), an online exercise section (*opt CourseExercise*), etc. Since we are in a testing context, one configuration of the FD does not represent a complete Claroline instance, but the minimal instance needed to play a set of test cases. Basically, it maps to a Claroline instance with one particular user and one particular course. This is similar to the technique presented by Segura et al. [53] used to represent the testing entry domain of a e-commerce web site. In order to represent a complete Claroline instance (with all its users and courses), we need to introduce cardinalities [39] on the $User$ and $Course$ features in order to have multiple users and multiple courses. Eventually we obtained a FD with 44 features.

Regarding the FTS, we employed a web crawler (Scrapy [52], a Python bot that systematically browse and record information about a website) on our local Claroline instance to discover states of the FTS, which, as for the usage model, represent visited pages on the website. The crawler has been parametrized to login as a visitor, a student, a teacher, and an administrator and to record accessible pages. Internal states of the pages (e.g., checkboxes, forms, etc.) have been ignored. Transitions have been added in such a way that every state may be accessed from anywhere. This simplification, used only to ease the FTS building, is consistent with the Web nature of the application and satisfy the inclusion property: $S_{dtmc} \subseteq S_{fts} \wedge Act_{dtmc} \subseteq Act_{fts} \wedge trans_{dtmc} \subseteq trans_{fts}$. First, we tried to build the FTS using the navigation model from the web crawler but found out that some user traces show that the navigation in Claroline is not always as obvious as it seems (e.g., if the users access the website from an external URL sent by e-mail or use tabs in their browser). To remain general we decide to adopt the mentioned simplification. Finally, transitions have been tagged manually with feature expressions based on the knowledge of the system (via the documentation and the local Claroline instance). To simulate a web browser access, both to the root page or directly to a sub-page of the website (e.g., from a direct link sent in an email), we added a "0" initial state connected to and accessible from all states in the FTS. The final FTS consists of 107 states and 11236 transitions and has been built in approximately 4 days (by one person).

4.2.1 Results

The DFS algorithm has been applied four times to the Claroline usage model (see Tab. 1) with a maximal length of 98 (the maximal path length without any loop in the usage model), which corresponds to the number of states, a maximal probability of 1 and four different minimal probabilities: 10^{-4}, 10^{-5}, 10^{-6} and 10^{-7} to observe patterns. Execution times range from less than a minute for the first run with 211 selected traces to ±8 hours for run 4 with 62112 selected traces. Additionally, the algorithm has been parametrized to consider each transition only once (i.e., a transition does not appear more than once in a selected trace). This modification has been made since we discovered after a few runs that the algorithm produced a lot of traces with repeated actions, which is of little interest for product prioritization. Repetitions were due to the huge number of loops in the Claroline usage model. All traces generated from the usage model are valid, this is caused by the nature of the Claroline FD: most of the features are independent from each other and few of them have exclusive constraints. The 2-gram solution used to generate the usage model fits well in this case, as there is no trace selected in the usage model that has been rejected. Sprenkle et al. [54] experimentally demonstrate that increasing the n in $n-gram$ generation of the usage model does increase the size of the generated model in a non linear way (as long as n is between 2 and 10). Increasing the n value in our case would just result in an unnecessary increase of the model complexity. As expected, the average size of the traces increases as the Pr_{min} decrease (a lower probability allows longer traces to be selected). The average size of the traces used to generate the usage model is $9,88$. As explained in section 3, it is possible to prune the original FTS using the valid traces in order to consider only the valid products capable of executing those traces. In this case, it eventually reduces the number of states and transitions from 107 and 11236 (resp.) to 16 and 66 (resp.) in run 1 and to 69 and 844 (resp.) in run 4. As expected, by controlling the interval size we can reduce the number of traces to be considered and yield easily analysable FTS'.

4.3 Sferion^{TM}Landing Symbology Function

Sferion^{TM}is an industrial situational awareness suite for helicopters flying in degraded visual environments [1, 48]. The landing symbology function supports the pilot during the landing approach by marking the intended lading position on ground using a head-tracked Helmet Mounted Sight and Display (HMS/D) and Hands On Collective And Stick (HOCAS). The spatial awareness is enhanced during the final landing approach by displaying 3D conformal visual cues on the helmet. Obstacles in the landing zone are detected and classified using a real-time OWS (Obstacle Warning System). Depending on the customer and the helicopter platform, the landing symbology function may have different features (Fig. 7) selected: $ELOP$ or $HELLAS$ OWS, SI_sensor_based or SI_from_DB as slope indication provider for landing position, etc..

The models have been designed by engineers using MaTeLo [2] tool, OVM and Matelo Product Line Management (MPLM) [49]. They have originally been presented by Samih et al. [48]. MaTeLo supports the description of statistical usage models by using extended Markov chains. MaTeLo's usage model is a DTMC,

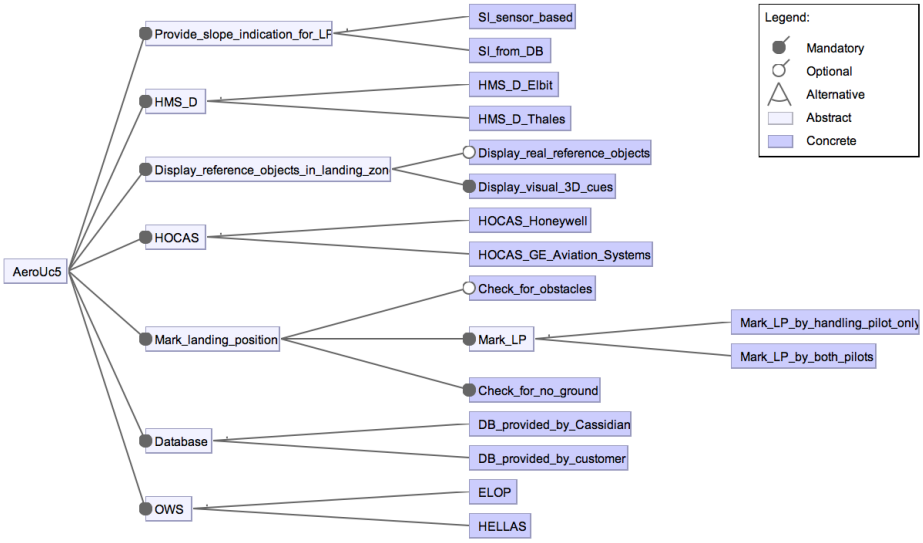


Fig. 7 Sferion^{TM}Landing Symbology Function Feature Diagram

where the nodes represent the major states of the system and the transitions are labelled with the actions or operations of the SUT with their probability to be fired. In the Sferion^{TM}landing symbology function model, the transitions are tagged with a probability representing the likelihood, when we are in one state, to execute the transition, and an action performed when the transition is executed; and each action is associated to zero, one or more requirements. The variability is described using OVM (Orthogonal Variability Model), each variation point is associated to zero, one or more requirement(s). The mapping, encoded in MPLM, between the variation points and the usage model transitions is made through the requirements. MPLM and MaTeLo tools support the product-based test derivation approach (Fig. 3a).

4.3.1 Results

Before applying family-based prioritization, we encoded the Sferion^{TM}landing symbology function models using our formalisms: the usage model has been flattened to remove hierarchy (by hand in 1/2 day); the OVM model has been translated to TVL [9] (by hand in 1/2 day); and the mapping between features and behaviour has been encoded using a FTS, generated from the MaTeLo usage model, the OVM model and the MPLM mapping model (in 1 day). We obtained a TVL model with 25 features (256 possible products), a usage model with 25 states, 12 actions and 46 transitions, and a FTS with the same numbers of states, actions and transitions.

As explained in section 3.2.1, engineers will probably have to run the algorithm several times using different minimal and maximal probabilities intervals in order to refine the selection. In our first attempt, we applied our trace selection algorithm 10 times with a maximal probability value of 1 and a minimal probability value ranging from 10^{-1} to 10^{-10} and a maximal length of 50. The results of the

Table 2 Landing Symbology Function Traces Selection

	Run 1	Run 2	Run 3	Run 4	Run 5
Pr_{min}	$1E^{-1}$	$1E^{-2}$	$1E^{-3}$	$1E^{-4}$	$1E^{-5}$
Pr_{max}	1	1	1	1	1
#UM tr.	0	0	8	50	306
#Valid tr.	0	0	8	50	306
Avg. size	0	0	15	$16,68$	$18,58$
σ size	0	0	$0,76$	$1,17$	$1,39$
Avg. proba.	0	0	$2,19E^{-3}$	$5,67E^{-4}$	$1,19E^{-4}$
σ proba	0	0	$1,51E^{-3}$	$9,30E^{-4}$	$4,23E^{-4}$
#FTS' st.	0	0	18	23	23
#FTS' tr.	0	0	12	12	12
#FTS' act.	0	0	27	40	42

	Run 6	Run 7	Run 8	Run 9	Run 10
Pr_{min}	$1E^{-6}$	$1E^{-7}$	$1E^{-8}$	$1E^{-9}$	$1E^{-10}$
Pr_{max}	1	1	1	1	1
#UM tr.	1870	8622	36582	123534	*err*
#Valid tr.	1870	8622	36582	123534	*err*
Avg. size	$20,85$	$22,99$	$25,15$	$27,17$	*err*
σ size	$1,62$	$1,77$	$1,88$	$1,97$	*err*
Avg. proba.	$2,20E^{-5}$	$5,02E^{-6}$	$1,21E^{-6}$	$3,59E^{-7}$	*err*
σ proba	$1,76E^{-4}$	$8,25E^{-5}$	$4,01E^{-5}$	$2,18E^{-5}$	*err*
#FTS' st.	23	23	23	23	*err*
#FTS' tr.	12	12	12	12	*err*
#FTS' act.	42	42	42	42	*err*

execution are showed in table 2. The run 10 did not return any results due to the too wide range of considered probabilities, giving too many possibles paths in the usage model. This is not a problem for our approach as a wide range of probabilities is not very useful for prioritization. According to those results, the interval with the most probable traces is between $1E^{-3}$ and $1E^{-2}$. We re-run the algorithm with the minimal probabilities $5E^{-3}$ and $2.5E^{-3}$: the execution with an interval between $[0; 5E^{-3}]$ returned no traces; the execution with an interval between $[0; 2,5E^{-3}]$ returned 2 traces (*trace 1* and (*trace 2*)) with an average probability of $4,62E^{-3}$ and a length of 14. Those two traces are the most probable behaviours of the landing symbology function and may be executed by all the products of the product line.

In order to get a more concrete product, we generate longer traces in the usage model by using the classical state-coverage criterion [37]. This criteria specifies that, when executing a test set on the system, all the states of the system have to be visited at least once. Generating a trace from the usage model using this criteria gives us one trace visiting all states (*trace 3*). As for previously generated traces, we execute it on the FTS to ensure that there exists at least one product able to exercise this behaviour: this gives us a set of 64 products.

4.3.2 Test case generation with MaTeLo

All products can execute *trace 1*, only products with the `Display_real_reference_objects` feature may execute *trace 2*. In those products, we selected a configuration (i.e., a product) p based on the risk associate to features and the customers satisfaction of previously commercialised products. With the help of MPLM, we

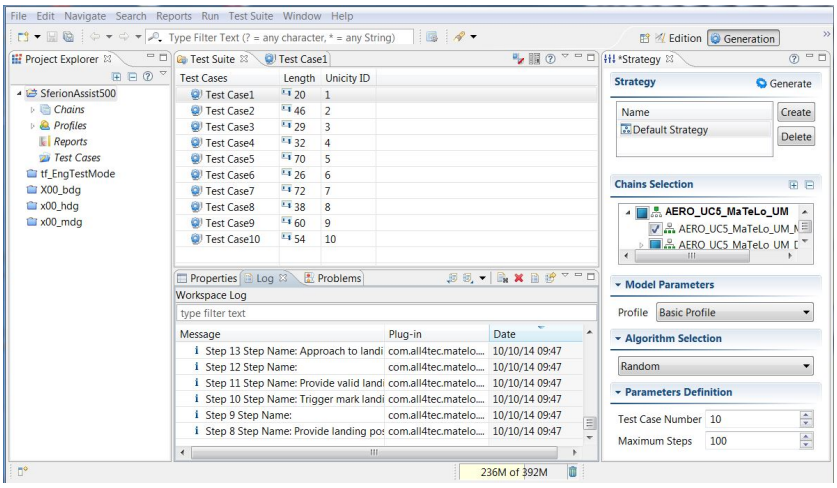


Fig. 8 MaTeLo Test-Case Generation View

generate the test model for p from the usage model of the SPL, that contains 16 states and 28 transitions, from which four test cases are generated. Fig. 8 presents the result of the test-case generation process for this test model: there are 10 test-cases with at most 72 steps (in the main tab of Fig. 8). Right tab in Fig. 8 presents the parameters used for the test case generation: the default strategy; the chains selection corresponds to the DTMCs used for test case generation (MaTeLo allows definition of hierarchical states but not orthogonal states, selected DTMCs corresponds to the main DTMC and sub-states DTMCs); the basic profile contains probabilities for the different transitions as defined by the engineer; the random selection algorithm has been used to generate 10 test cases with at most 100 steps (i.e., transitions in the DTMCs). This random selection algorithm is the default strategy in MaTeLo. The tools allows for the definition of custom strategies. These strategies involve the definition of the scope (portion of the usage model to be covered) of the usage model, probability profile, and alternative generation algorithms.

4.4 Discussion

We organise our discussion on the final results regarding feasibility for statistical prioritization SPL testing according to the criteria mentioned above: i) FTS pruning; ii) modelling; and iii) scalability.

4.4.1 FTS pruning.

In both cases, it was possible to substantially prune the FTS models according to frequent behaviours: from 28% to 85% reduction w.r.t. the number of states (for Sferion^{TM}and Claroline) and up to 99,994% reduction w.r.t to transitions (for Claroline). These important reduction factors are interesting in the sense that it is possible to use statistical selection to deal with additional computationally

expensive coverage criteria that would not be directly applicable on the original model (e.g. all-paths coverage on the claroline FTS [24]). Regarding the number of products associated with the selected traces, the situation is less favourable. In the claroline case, the least probable trace in run 1 is already associated to 260 products [22]. The main reason is that traces are small in size yielding short associated feature expressions. Most Claroline users therefore seems to visit few pages after the login one. Because the source Apache Log is anonymised, it is impossible to investigate further in this direction. The Symbology function exhibits more complex behaviours as witnessed by traces' sizes. For Sferion^{TM}, there are traces that can be executed by all the products of the SPL. While from pure product selection perspective this is a bad result, two additional observations need to be made. First, the usage model was provided by experts to focus on the most relevant behaviours: it seems they did perform correctly this task as most part of the described behaviour concern all the products of the SPL. Second, there are opportunities to reuse test cases amongst products: these two traces can be used to derive a small number of concrete test cases covering all configurations. This strategy can be used to explore interaction problems [41]. Finally, feature models of our considered systems have very few constraints (e.g., $Mark_landing_position \Rightarrow HOCAS$, $Check_for_obstacles \Rightarrow OWS$, etc.) amongst features, which clearly influence product reduction ability. While such an open feature model is not surprising for a web based system, this is more unusual for an embedded SPL.

4.4.2 Modelling

Using statistical prioritisation in both case studies involved some modelling: the family based scenario we experienced for Claroline allowed us to extract automatically the usage model using a machine learning technique while the Sferion^{TM}product one relied on SPL and testing experts to explicitly provide the required usage model. However what is the common to both scenarios is the necessity to provide variability models (in OVM or TVL) and mapping from features to behaviours either by means of FTS or mapping matrices [47, 49]. Both approaches try to keep requirements from test models separated. Such a separation of concerns does not guarantee that these models are correct (learning behaviours from anonymous logs entails approximations and hand-made usage model are not free from biases either) but helps finding discrepancies as they are generally provided by stakeholders having different perspectives and skills. Keeping these models separated was also useful to integrate our approach with tools like MaTeLo that do not take into account natively features in their usage models but provide additional facilities such as risk management or customer satisfaction during test case generation. As discussed above, keeping the usage model and the FTS separated may be detrimental to the analysis as some invalid traces may be first extracted from the usage model and then removed. Even if this was not the case on the considered SPLs, this may happen in more constrained SPLs. One strategy could be to start with separated models and to merge them in a feature-aware usage model once enough confidence is gained on both models. This is left for future work.

The effort spent in modelling activities depends on the case study: for the Claroline case study, the usage model has been automatically generated, the FTS has been semi-automatically generated and the variability model has been hand

crafted. Giving the size of Claroline (442.399 LOC), the total effort spent in modelling activities is reasonable (around 7 days). The Sferion^{TM}case study models a critical system, the modelling and testing efforts are important but have supported by the company in order to guaranty a safe and sound product. The additional effort required to derive the FTS from the Sferion^{TM}models is small (around 2 days).

4.4.3 Scalability

Final results show that the scalability of our implementation mainly depends on the $[Pr_{min}, Pr_{max}]$ interval and the shape of the usage model. We notice that if the model is large (Claroline FTS) and/or the probability interval very large computation time obviously increases and may even lead to no result at all (Run 10, table 2). In this case we encountered memory overflows. The DFS algorithm used in our implementation seems to perform well as long as the $[Pr_{min}, Pr_{max}]$ interval is not too wide, even on large usage model (like the Claroline case). Thus, we rely upon the tester to choose a relatively small probability interval in order to extract behaviours that results in the desired amount of traces and products. So far, we explored these intervals manually to find tradeoffs. This exploration can be automated if additional criteria (such as the maximum number of products desired) are specified. Other state space exploration techniques will have to be investigated to improve the algorithm (e.g., limit the length of the selected traces is amongst the simplest, or use simulation techniques [5]). It should be noted that, for more specialized explorations, such as finding the most probable path, dedicated algorithms like the one proposed by Viterbi [60] may be used.

4.5 Threats to Validity

Internal Validity The approach has only been applied on 2 systems. In order to mitigate this risk, we chose two different kinds of systems: the first one is a web application with a very few constraints and the other one is an embedded system with a more constrained behaviour. Usage models have different sizes and come from different sources: the first one has been generated from an Apache web log and the second one has been designed by an expert.

Construct Validity To implement our approach, we choose to use a DFS algorithm with some restrictions (maximal length of the selected traces) in order to avoid infinite executions. This choice may be not optimal but the DFS exploration ensure (in worst case) a complete exploration of the usage model. The input models (usage model as a usage model, FTS and TVL) are not the only possibilities to represent usages, behaviour, and variability of the SPL. In our second SPL, we showed how we translate other input models in order to apply our approach.

External Validity The main threat is the nature of the considered applications as it influences the shape of the different models: average degree in the FTS and usage model, number of features in the FD, number and nature of the constraints in the FD... The first considered system [22] is a very particular kind of application: a web application accessible through PHP pages in a web browser with a small number

of states and a huge number of transitions. This kind of application allows a very flexible navigation from page to page either by clicking on the links in the different pages or by a direct access with a link in a bookmark or an e-mail for instance. In order to mitigate this risk, we applied our approach to the Sferion^{TM}landing symbology function, an embedded system. The diversity of the considered systems and models gives us a good confidence in the possibilities of generalisation of our approach.

5 Related Work

To the best of our knowledge, there is no approach prioritizing behaviours statistically for testing SPLs in a family-based manner. There have been SPL test efforts to sample products for testing such as t-wise approaches [15, 16, 43]. More recently sampling was combined with prioritization thanks to the addition of weights on feature models and the definition of multiple objectives [30, 32]. However, these approaches do not consider SPL behaviour in their analyses.

Efforts to combine sampling techniques with modelling ones (e.g. [36]) exist. These approaches are product-based, meaning that they may miss opportunities to reuse tests amongst sampled products [46]. We believe that benefiting from the recent advances in behavioural modelling provided by the model checking community [3, 4, 12, 13, 26, 35], sound MBT approaches for SPL can be derived and interesting scenarios combining verification and testing can be devised [20].

To consider behaviour in an abstract way, a full-fledged MBT approach [57] is required. Although behavioural MBT is well established for single-system testing [56], a survey [42] shows insufficient support of SPL-based MBT. Metzger and Pohl further emphasizes the need for inter-model consistency and minimizing test redundancy across the lifecycle (domain and application engineering) [38]. We believe that the FTS formalism, natively equipped with features as a first-class concept, is pivotal to inter-model verification support and supports combination of quality assurance techniques both at the domain and application engineering levels as our integration between family-based and product-based statistical test selection illustrates.

Our will is to apply ideas stemming from statistical testing [54] and adapt them in an SPL context. For example, combining structural criteria with statistical testing has been discussed by Gouraud et al. [29] and Thévenod-Fosse and Waeselynck [55]. We do not make any assumption on the way the usage model is obtained: via an operational profile [40] or by analysing the source code or the specification [55]. However, a uniform distribution of probabilities over the usage model would probably be less interesting. As noted by Witthaker [62], in such case only the structure of traces would be considered and therefore basing their selection on their probabilities would just be a means to limit their number in a mainly random-testing approach. In such cases, structural test generation has to be employed [25].

We use MaTeLo [2] to generate test cases from a product model. Other tools like JUMBL [45] would have qualified. Both are model-based statistical testing tools, supporting the development of statistical usage models using Markov chains, the analysis of models, and the generation of test cases [58]. However, none of them are able to natively handle SPL models. We use the MaTeLo Product Line Manager

(MPLM) tool [47,49] to generate models for a product of the SPL, which are then used to generate test cases.

6 Future Work

Recently, with the explosion of model size (and particularly in a SPL context), the model checking community has explored techniques to reduce the state space of the models to check [6]. Our approach is used to prioritize products to test in a SPL using a usage model of this SPL. The FTS' built at step 2 represents a subset of the behaviour of the SPL (i.e., the complete FTS) able to execute the traces selected in the usage model. We believe that this FTS' model may be used during model checking [17] to verify in priority a subset of the behaviour of a SPL. This possibility has to be explored and discussed regarding the loss of completeness and its impact on the verification for the complete FTS.

The presented prioritization approach has been implemented in our Variability Intensive Behavioural teSting (VIBeS) framework [21]. There are several possible improvements for this current implementation: the algorithm, currently implemented as a DFS, may be improved in several ways (e.g., using the algorithm proposed by Viterbi [60] or using algorithms from the model checking community [5]) ; the trace selection algorithm may be combined with other coverage criteria to improve the selection ability in our approach (e.g., one may be interested to select highly probable behaviour but also to cover all states in the usage model); keeping usage model and FTS separated allows a better integration with tools like MaTeLo but requires to execute selected traces on the FTS, one way to reduce the computation cost could be to merge the usage model and the FTS.

7 Conclusion

In this paper, we reported on our experiences in applying both family-based and product-based statistical testing for SPLs. We proposed family-based prioritisation in our previous work to extract configurations of interest according to the probability of their execution traces gathered in a discrete-time Markov chain representing their usages an mined from a log [22]. We thus select a subset of the full SPL behaviour given as Featured Transition Systems (FTS). This allows us to construct a new FTS representing only the executions of relevant products. This pruned FTS can be analysed all at once to enable test reuse amongst products to scale during testing activities. Product-based statistical testing requires the testers to select a product interest before the usage model is pruned, leaving only executions associated to it. The approach followed by ALL4TEC [47–49] is also to rely on the experts to provide the usage model.

Though these approaches may seem antagonistic, family-based prioritisation can gracefully complement product-based one by suggesting configurations of interest. This can help testers finding opportunities for test reuse since we demonstrated that a given behaviour can in fact be executed by many configurations. We also noticed the influence of variability models in the discrimination power of prioritisation and reported on ways to cope with such a situations either by using additional coverage criteria [23,24] and ultimately relying on tester expertise.

Indeed, as for sampling configuration from feature models, statistical testing of SPLs is constrained by multiple objectives [31].

Acknowledgements We would like to Jean-Roch Meurisse and Didier Belhomme from the University of Namur for providing the Webcampus Apache access log.

References

1. Airbus Defence & Space - Sferion: http://www.defenceandsecurity-airbusds.com/fr/sferion
2. ALL4TEC - MaTeLo: http://all4tec.net/index.php/en/model-based-testing/20-markov-test-logic-matelo
3. Asirelli, P., ter Beek, M., Gnesi, S., Fantechi, A.: Formal description of variability in product families. In: Software Product Line Conference (SPLC), 2011 15th International, pp. 130–139 (2011). DOI 10.1109/SPLC.2011.34
4. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Design and validation of variability in product lines. In: Proceedings of the 2Nd International Workshop on Product Line Approaches in Software Engineering, PLEASE '11, pp. 25–30. ACM, New York, NY, USA (2011). DOI 10.1145/1985484.1985492. URL http://doi.acm.org/10.1145/1985484.1985492
5. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
6. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10$\hat{2}$0 states and beyond. Inf. Comput. **98**(2), 142–170 (1992). DOI 10.1016/0890-5401(92)90017-A. URL http://dx.doi.org/10.1016/0890-5401(92)90017-A
7. Cichos, H., Oster, S., Lochau, M., Schürr, A.: Model-based Coverage-driven Test Suite Generation for Software Product Lines. In: Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems, MODELS'11, pp. 425–439. Springer-Verlag, Berlin, Heidelberg (2011). URL http://dl.acm.org/citation.cfm?id=2050655.2050698
8. Claroline: http://www.claroline.net/
9. Classen, A., Boucher, Q., Heymans, P.: A Text-based Approach to Feature Modelling: Syntax and Semantics of {TVL}. Science of Computer Programming **76**(12), 1130–1143 (2011). DOI 10.1016/j.scico.2010.10.005. URL http://dx.doi.org/10.1016/j.scico.2010.10.005http://linkinghub.elsevier.com/retrieve/pii/S0167642310001899
10. Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F.: Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. Software Engineering, IEEE Transactions on **39**(8), 1069–1089 (2013). DOI 10.1109/TSE.2012.86
11. Classen, A., Heymans, P., Schobbens, P.Y.: What's in a Feature: A Requirements Engineering Perspective. In: J.L. Fiadeiro, P. Inverardi (eds.) Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'08), *LNCS*, vol. 4961, pp. 16–30. Springer (2008). URL http://www.cs.le.ac.uk/events/fase2008/
12. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 321–330. ACM, New York, NY, USA (2011). DOI 10.1145/1985793.1985838. URL http://doi.acm.org/10.1145/1985793.1985838
13. 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: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pp. 335–344. ACM, New York, NY, USA (2010). DOI 10.1145/1806799.1806850. URL http://doi.acm.org/10.1145/1806799.1806850
14. Cohen, M., Dwyer, M., Shi, J.: Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. IEEE Transactions on Software Engineering **34**(5), 633–650 (2008). DOI 10.1109/TSE.2008.50. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4564473

15. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. In: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, ROSATEA '06, pp. 53–63. ACM, New York, NY, USA (2006). DOI 10.1145/1147249.1147257. URL http://doi.acm.org/10.1145/1147249.1147257
16. Cohen, M.B., Dwyer, M.B., Shi, J.: Interaction testing of highly-configurable systems in the presence of constraints. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07, pp. 129–139. ACM, New York, NY, USA (2007). DOI 10.1145/1273463.1273482. URL http://doi.acm.org/10.1145/1273463.1273482
17. Cordy, M., Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Provelines: A product line of verifiers for software product lines. In: Proceedings of the 17th International Software Product Line Conference Co-located Workshops, SPLC '13 Workshops, pp. 141–146. ACM, New York, NY, USA (2013). DOI 10.1145/2499777.2499781. URL http://doi.acm.org/10.1145/2499777.2499781
18. Cordy, M., Heymans, P., Schobbens, P.Y., Sharifloo, A.M., Ghezzi, C., Legay, A.: Verification for reliable product lines. arXiv:1311.1343 (2013)
19. Czarnecki, K., Wasowski, A.: Feature Diagrams and Logics: There and Back Again. In: SPLC '07, pp. 23–34. IEEE (2007). DOI 10.1109/SPLINE.2007.24
20. 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: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, *Lecture Notes in Computer Science*, vol. 7609, pp. 208–222. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-34026-0_16. URL http://dx.doi.org/10.1007/978-3-642-34026-0_16
21. Devroey, X., Perrouin, G.: Variability Intensive system Behavioural teSting framework (VIBeS) (2014). URL https://projects.info.unamur.be/vibes/
22. Devroey, X., Perrouin, G., Cordy, M., Schobbens, P., Legay, A., Heymans, P.: Towards statistical prioritization for software product lines testing. In: P. Collet, A. Wasowski, T. Weyer (eds.) The Eighth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014, p. 10. ACM (2014). DOI 10.1145/2556624.2556635. URL http://doi.acm.org/10.1145/2556624.2556635
23. Devroey, X., Perrouin, G., Legay, A., Cordy, M., Schobbens, P., Heymans, P.: Coverage criteria for behavioural testing of software product lines. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 8802, pp. 336–350. Springer (2014). DOI 10.1007/978-3-662-45234-9_24. URL http://dx.doi.org/10.1007/978-3-662-45234-9_24
24. Devroey, X., Perrouin, G., Schobbens, P.: Abstract test case generation for behavioural testing of software product lines. In: S. Gnesi, A. Fantechi, M.H. ter Beek, G. Botterweck, M. Becker (eds.) 18th International Software Product Lines Conference - Companion Volume for Workshop, Tools and Demo papers, SPLC '14, Florence, Italy, September 15-19, 2014, pp. 86–93. ACM (2014). DOI 10.1145/2647908.2655971. URL http://doi.acm.org/10.1145/2647908.2655971
25. Feliachi, A., Le Guen, H.: Generating transition probabilities for automatic model-based test generation. In: Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, pp. 99–102 (2010). DOI 10.1109/ICST.2010.26
26. Fischbein, D., Uchitel, S., Braberman, V.: A foundation for behavioural conformance in software product line architectures. In: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, ROSATEA '06, pp. 39–48. ACM, New York, NY, USA (2006). DOI 10.1145/1147249.1147254. URL http://doi.acm.org/10.1145/1147249.1147254
27. Garca-Teodoro, P., Daz-Verdejo, J., Maci-Fernndez, G., Vzquez, E.: Anomaly-based network intrusion detection: Techniques, systems and challenges. Computers & Security **28**(12), 18 – 28 (2009). DOI http://dx.doi.org/10.1016/j.cose.2008.08.003. URL http://www.sciencedirect.com/science/article/pii/S0167404808000692
28. Ghezzi, C., Pezzè, M., Sama, M., Tamburrelli, G.: Mining Behavior Models from User-Intensive Web Applications Categories and Subject Descriptors. In: 36th International Conference on Software Engineering, ICSE '14. ACM, Hyderabad, India (2014)
29. Gouraud, S.D., Denise, A., Gaudel, M.C., Marre, B.: A new way of automating statistical testing methods. In: Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on, pp. 5–12 (2001). DOI 10.1109/ASE.2001.989785

30. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Le Traon, Y.: Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. IEEE Transactions on Software Engineering **40**(7), 650–670 (2014). DOI 10.1109/TSE.2014.2327020. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6823132
31. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Traon, Y.L.: Multi-objective Test Generation for Software Product Lines. In: Proceedings of the 17th International Software Product Line Conference, SPLC '13, pp. 62–71. ACM, New York, NY, USA (2013). DOI 10.1145/2491627.2491635. URL http://doi.acm.org/10.1145/2491627.2491635
32. Johansen, M., Haugen, O., Fleurey, F., Eldegard, A., Syversen, T.: Generating better partial covering arrays by modeling weights on sub-product lines. In: R. France, J. Kazmeier, R. Breu, C. Atkinson (eds.) Model Driven Engineering Languages and Systems, *Lecture Notes in Computer Science*, vol. 7590, pp. 269–284. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-33666-9_18. URL http://dx.doi.org/10.1007/978-3-642-33666-9_18
33. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Spencer Peterson, A.: Feature-Oriented domain analysis (FODA) feasibility study. Tech. rep., Soft. Eng. Inst., Carnegie Mellon Univ. (1990)
34. Kim, C., Khurshid, S., Batory, D.: Shared execution for efficiently testing product lines. In: Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on, pp. 221–230 (2012). DOI 10.1109/ISSRE.2012.23
35. Lauenroth, K., Pohl, K., Toehning, S.: Model checking of domain artifacts in product line engineering. In: Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on, pp. 269–280 (2009). DOI 10.1109/ASE.2009.16
36. Lochau, M., Oster, S., Goltz, U., Schürr, A.: Model-based pairwise testing for feature interaction coverage in software product line engineering. Software Quality Journal **20**(3-4), 567–604 (2012). DOI 10.1007/s11219-011-9165-4
37. Mathur, A.P.: Foundations of software testing. Pearson Education (2008)
38. Metzger, A., Pohl, K.: Software product line engineering and variability management: Achievements and challenges. In: Proceedings of the on Future of Software Engineering, FOSE 2014, pp. 70–84. ACM, New York, NY, USA (2014). DOI 10.1145/2593882.2593888. URL http://doi.acm.org/10.1145/2593882.2593888
39. Michel, R., Classen, A., Hubaux, A., Boucher, Q.: A formal semantics for feature cardinalities in feature diagrams. In: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '11, pp. 82–89. ACM, New York, NY, USA (2011). DOI 10.1145/1944892.1944902. URL http://doi.acm.org/10.1145/1944892.1944902
40. Musa, J.D., Fuoco, G., Irving, N., Kropfl, D., Juhlin, B.: The operational profile. NATO ASI series F Comp. and Syst. Sc. **154**, 333–344 (1996)
41. Nguyen, H.V., Kästner, C., Nguyen, T.N.: Exploring variability-aware execution for testing plugin-based web applications. In: P. Jalote, L.C. Briand, A. van der Hoek (eds.) 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, pp. 907–918. ACM (2014). DOI 10.1145/2568225.2568300. URL http://doi.acm.org/10.1145/2568225.2568300
42. Oster, S., Wöbbeke, A., Engels, G., Schürr, A.: Model-based software product lines testing survey. In: Model-Based Testing for Embedded Systems, pp. 339–382. CRC Press (2011)
43. Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., le Traon, Y.: Pairwise testing for software product lines: comparison of two approaches. Software Quality Journal **20**(3-4), 605–643 (2012). DOI 10.1007/s11219-011-9160-9. URL http://dx.doi.org/10.1007/s11219-011-9160-9
44. Pohl, K., Böckle, G., Van Der Linden, F.: Software product line engineering: foundations, principles, and techniques. Springer-Verlag New York Inc (2005)
45. Prowell, S.J.: JUMBL: a tool for model-based statistical testing. In: System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on, pp. 9 pp.– (2003). DOI 10.1109/HICSS.2003.1174916
46. von Rhein, A., Apel, S., Kästner, C., Thüm, T., Schaefer, I.: The PLA model: On the combination of product-line analyses. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, pp. 14:1–14:8. ACM, New York, NY, USA (2013). DOI 10.1145/2430502.2430522. URL http://doi.acm.org/10.1145/2430502.2430522
47. 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.) 2014 19th

International Conference on Engineering of Complex Computer Systems (ICECCS 2014). Tianjin, Chine (2014). URL http://hal.inria.fr/hal-01002099
48. Samih, H., Bogusch, R.: MPLM - MaTeLo Product Line Manager. In: Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2, SPLC '14, pp. 138–142. ACM, New York, NY, USA (2014). DOI 10.1145/2647908.2655980. URL http://doi.acm.org/10.1145/2647908.2655980
49. Samih, H., Le Guen, H., Bogusch, R., Acher, M., Baudry, B.: An Approach to Derive Usage Models Variants for Model-based Testing. In: The 26th IFIP International Conference on Testing Software and Systems (2014). Springer, Madrid, Espagne (2014). URL http://hal.inria.fr/hal-01025124
50. Sampath, S., Bryce, R.C., Viswanath, G., Kandimalla, V., Koru, a.G.: Prioritizing User-Session-Based Test Cases for Web Applications Testing. In: Software Testing, Verification, and Validation, 2008 1st International Conference on, pp. 141–150. IEEE (2008). DOI 10.1109/ICST.2008.42. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4539541
51. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks **51**(2), 456 – 479 (2007). DOI http://dx.doi.org/10.1016/j.comnet.2006.08.008. URL http://www.sciencedirect.com/science/article/pii/S1389128606002179. Feature Interaction
52. Scrapy: http://scrapy.org/
53. Segura, S., Sánchez, A.B., Ruiz-Cortés, A.: Automated Variability Analysis and Testing of an E-commerce Site.: An Experience Report. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pp. 139–150. ACM, New York, NY, USA (2014). DOI 10.1145/2642937.2642939. URL http://doi.acm.org/10.1145/2642937.2642939
54. Sprenkle, S.E., Pollock, L.L., Simko, L.M.: Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. Software Testing, Verification and Reliability **23**(6), 439–464 (2013). DOI 10.1002/stvr.1496. URL http://dx.doi.org/10.1002/stvr.1496
55. Thévenod-Fosse, P., Waeselynck, H.: An investigation of statistical software testing. Softw. Test., Verif. Reliab. **1**(2), 5–25 (1991)
56. Tretmans, J.: Model based testing with labelled transition systems. In: R. Hierons, J. Bowen, M. Harman (eds.) Formal Methods and Testing, *Lecture Notes in Computer Science*, vol. 4949, pp. 1–38. Springer Berlin Heidelberg (2008). DOI 10.1007/978-3-540-78917-8_1. URL http://dx.doi.org/10.1007/978-3-540-78917-8_1
57. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann (2007)
58. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches (April 2011), 297–312 (2012). DOI 10.1002/stvr
59. Verwer, S., Eyraud, R., De La Higuera, C.: PAutomaC: a probabilistic automata and hidden Markov models learning competition. Machine Learning pp. 1–26 (2013). DOI 10.1007/s10994-013-5409-9. URL https://hal.archives-ouvertes.fr/hal-00873981
60. Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. Information Theory, IEEE Transactions on **13**(2), 260–269 (1967)
61. Weiß leder, S., Sokenou, D., Schlingloff, B.: Reusing State Machines for Automatic Test Generation in Product Lines. In: 1st Workshop on Model-based Testing in Practice (MoTiP 2008), p. 19. Citeseer, Berlin, Germany (2008). URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.169.5699\&rep=rep1\&type=pdf\#page=21
62. Whittaker, J., Thomason Michael, G.: A markov chain model for statistical software testing. Software Engineering, IEEE Transactions on **20**(10), 812–824 (1994). DOI 10.1109/32.328991