Towards Statistical Prioritization for Software Product Lines Testing

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

Published in:
Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems

DOI:
10.1145/2556624.2556635

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

Towards Statistical Prioritization for Software Product Lines Testing

Extended version published at VaMoS '14 (http://dx.doi.org/10.1145/2556624.2556635)

Xavier Devroey, Maxime Cordy, Gilles Perrouin[◇], Pierre-Yves Schobbens
PReCISE Research Center, Faculty of Computer Science, University of Namur, Belgium
{xde,mcr,gpe,pys}@info.fundp.ac.be

Axel Legay
INRIA Rennes Bretagne Atlantique, France
axel.legay@inria.fr

Patrick Heymans
PReCISE Research Center, Faculty of Computer Science, University of Namur, Belgium
phe@info.fundp.ac.be
INRIA Lille-Nord Europe, Université Lille 1 – LIFL – CNRS , France

***Abstract*—Software Product Lines (SPL) are inherently difficult to test due to the combinatorial explosion of the number of products to consider. To reduce the number of products to test, sampling techniques such as combinatorial interaction testing have been proposed. They usually start from a feature model and apply a coverage criterion (e.g. pairwise feature interaction or dissimilarity) to generate tractable, fault-finding, lists of configurations to be tested. Prioritization can also be used to sort/generate such lists, optimizing coverage criteria or weights assigned to features. However, current sampling/prioritization techniques barely take product behavior into account. We explore how ideas of statistical testing, based on a usage model (a Markov chain), can be used to extract configurations of interest according to the likelihood of their executions. These executions are gathered in featured transition systems, compact representation of SPL behavior. We discuss possible scenarios and give a prioritization procedure illustrated on an example.**

I. Introduction

Software Product Line (SPL) engineering is based on the idea that products of the same family may 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*, i.e., an unit of difference between products. A product member of the SPL is a valid combination of features. Individual features can be specified using languages such as UML, while their inter-relationships are organized in a *Feature Diagram* (FD) [1]. An FD thus (abstractly) describes all valid combinations of features (called *configurations* of the FD), that is all the products of the SPL.

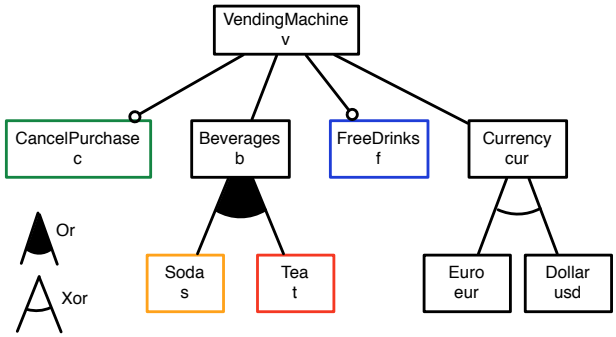
In this paper, we are interested in SPL testing. As opposed to to classical testing approaches, where the testing process only considers one software product, SPL testing is concerned about how to minimize the test effort related to a given the SPL (i.e., all the valid products of the SPL). The size of this set is roughly equal to 2^N, where N represents the number of features of the SPL. This number may vary from about 10 (2^{10} possible products) in small SPLs to thousands of features (2^{1000} possible products) in complex systems such as the Linux kernel. Automated Model-Based Testing [2] and shared execution [3], where tests can be reused amongst products, are candidates to reduce such effort.

Still, testing all products of the SPL is only possible for small SPLs, given their exponential growth with the number of features. Hence, one of the main questions arising in such a situation is: How to extract and prioritize relevant products? Existing approaches consider sampling products by using a coverage criterion on the FD (such as all valid 2-tuples of features: pairwise [4], [5]) and rank products with respect with respect to coverage satisfaction (e.g. the number of tuples covered). An alternative is to label each feature with weights and prioritize configurations accordingly [6], [7]. These methods actually help testers to scope more finely and flexibly relevant products to test than using a covering criteria alone. Yet, these approaches only sample products based on the FD which does not account for product behavior: they are just configurations of the FD. Furthermore, assigning meaningful weights on thousands of features can be tricky if no other information is available.

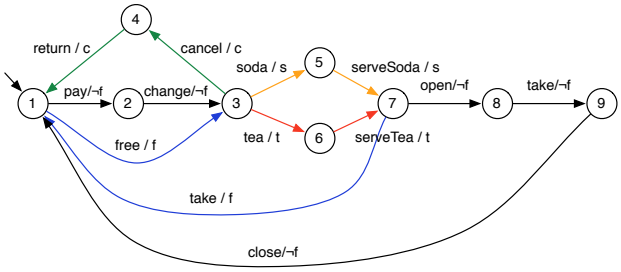
Statistical testing [8] proposes to generate test cases based on a *usage model* represented by a *Discrete Time Markov Chain* (DTMC). This usage model represents the usage scenarios of the software as well as their probability. This allows one to determine the relative importance of execution scenarios (with respect to other). This paper explores the possibility of using statistical testing to sample and prioritize products of an SPL. The basic idea is to focus on "most probable" (respectively "less probable" products), i.e. products that are able to execute highly probable (respectively. improbable) traces of the DTMC. Since black box usage scenarios may not relate directly to features, we propose to use a compact representation of SPL behaviour, *Featured Transition Systems* (FTSs) to determine which traces are legal with respect to the SPL and associate related products. In fact, we construct another FTS, which maps to the selected traces. This FTS represent the behavior of the set of products of interest and is amenable to various testing and verification techniques.

The remainder of this paper is organized as follows: Section

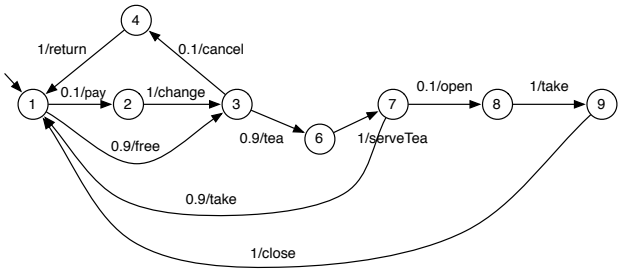
[◇]FNRS Postdoctoral Researcher



(a) Feature Diagram (FD)



(b) Featured Transition System (FTS)



(c) Usage model (DTMC)

Fig. 1. The soda vending machine example [9]

II presents the theoretical background underlying our vision presented in section III. Section IV discussed related research and Section V concludes the paper with challenges and future research directions.

II. Background

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

A. SPL Modelling

A key concern in SPL modeling is how to represent variability. To achieve this purpose, SPL engineers usually reason in terms of features. In [10] Pohl et al. define features as end-user visible characteristics of a system. Relations and constraints between features are usually represented in a Feature Diagram (FD) [1]. For example, Fig. 1a presents the FD of a soda vending machine [9]. A product derived from this diagram will correspond to a set of selected features. Here, $\{v, b, t, cur, usd\}$ corresponds to a machine that sells only tea and accept only dollar. FDs have been equipped with formal semantics [11], automated analyses and tools [1] for more than 20 years. A common semantics associated to a FD d (noted $[\![d]\!]$) is the set of all the valid products allowed by d.

1) Behavioural Modelling: Different formalisms may be used to model the behavior of a system. To allow the explicit mapping from feature to SPL behavior, Featured Transition Systems (FTSs) [9] 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 for a given FD in which products the transition may be fired. Thus it is possible to determine products that are the cause of a violation or a failed test. Formally, an FTS is a tuple $(S, Act, trans, i, d, \gamma)$ where:

- $(S, Act, trans, i)$ is a classical TS with S a set of states, Act a set of actions, $trans \subseteq S \times Act \times S$ a set of transitions (where $(s_1, \alpha, s_2) \in trans$ is sometimes noted $s_1 \xrightarrow{\alpha} s_2$) and $i \in S$ an initial state;
- d is a FD;
- $\gamma : trans \rightarrow [\![d]\!] \rightarrow \mathbb{B}$ is a total function, labeling each transition with a function that associates to each valid product a boolean expression indicating if it may fire the transition or not. This function is noted using a feature expression (which is basically a boolean expression with features' name as variables) meaning that only products that satisfy this feature expression may fire 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.

The semantics of a given FTS is a function that associates each valid product with its set of finite and infinite executions, i.e. all the possible paths in the graph starting from the initial state available for this specific product. According to this definition, an FTS is actually a behavioral model of a whole SPL. Fig. 1b presents the FTS modeling a vending machine SPL. For instance, transition ③ $\xrightarrow{pay/\neg f}$ ④ is labelled with the feature expression c. This means that only the products that do have the feature $Cancel$ (c) are able to execute the transition. This definition differs from the one presented in [9], where only infinite paths are considered. In a testing context, one may also be interested in finite paths.

B. Statistical Testing

Whittaker and Thomason introduce the notion of usage model in [8] and define it as a TS where transitions are associated to probabilities. A probability p_i on a transition $t_i = (s_i, \alpha, s_j)$represents the fact that if the system is in state s_i, the transition t_i has p_i chances to be fired. Formally, a usage model will be represented by a DTMC, which is a tuple $(S, Act, trans, P, \tau)$ where :

- $(S, Act, trans)$ is a TS;
- $P : S \times S \rightarrow [0, 1]$ is the probability matrix which gives for two states (s_i, s_j) the probability for the system in state s_i to go in the state s_j;
- $\tau : S \rightarrow [0, 1]$ is the vector containing the probabilities to be in the initial state when the system starts, with the following constraint : $\exists i : (\tau(i) = 1 \wedge \forall j \neq i : \tau(j) = 0)$;

- $\forall s_i \in S : \sum_{s_j \in S} P(s_i, s_j) = 1$ the total of the probabilities of the transitions leaving a state must be equal to 1.

Actions are just labels used to annotate transitions without changing the semantics of the DTMC. In our case, they are used to relate traces of the DTMC with executions of the FTS.

III. Approach

In our approach, we consider three models: a FD d to represent the features and their constraints (in Fig. 1a), an FTS fts to represent the behaviour of the SPL (in Fig. 1b) and a usage model represented by a DTMC $dtmc$ (in Fig. 1c) with the following constraints:

- The feature diagram of fts is d;
- States, actions and transitions of $dtmc$ are included in the states, actions and transitions (respectively) of fts: $S_{dtmc} \subseteq S_{fts} \wedge Act_{dtmc} \subseteq Act_{fts} \wedge trans_{dtmc} \subseteq trans_{fts}$;
- The initial state of fts, i has a probability of 1 to be executed first in $dtmc : \tau(i) = 1$

We deliberately chose not to integrate the DMTC with the FTS in a single model. This separation of concerns is motivated as follows:

- We may want to integrate the approach with existing software which does not take variability into account such as MaTeLO, a MBT tool which uses DTMC as input model [1];
- The DTMC can be obtained from either users trying the software under test, extracted from logs, or from running code. These extractions methods are agnostic of the features of the system they are applied to;
- Since the DTMC is built from existing software executions, it may be incomplete (as in Fig. 1c). Some products (or subsets of their behaviors) may simply not be exercised in the usage model resulting in missing transitions in the DTMC. Keeping the FTS and usage models separate is helpful to identify and correct such issues.

The fact that a usage model is created from partial (i.e., finite) observations of the products without consideration of their features allows paths in the DTMC that are inconsistent for the SPL. For example in the usage model of Fig. 1c, 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). Since the DTMC is never used alone, such situations are easy to spot using the FTS.

There are now two possible testing scenarios: product based test derivation (top-down) and family based test prioritization (bottom-up). The classification product/family based comes from [12].

[1] see: http://all4tec.net/index.php/en/model-based-testing/20-markov-test-logic-matelo

A. Product Based Test Derivation

Product based test derivation is straightforward: one selects one product (by selecting features in the FD), projects it onto the FTS, giving a TS with only the transitions of the product, prunes the DTMC to keep the following property true : $S_{dtmc} \subseteq S_{ts} \wedge Act_{dtmc} \subseteq Act_{ts} \wedge trans_{dtmc} \subseteq trans_{ts}$. Probabilities of the removed transitions need to be distributed on siblings (since the property $\forall s_i \in S : \sum_{s_j \in S} P(s_i, s_j) = 1$ has to hold). Finally, we generate test cases using classical statistical testing algorithms on the DTMC [8], [13]. A similar testing process is proposed by Samih and Baudry [14]. Product selection is made on an orthogonal variability model (OVM) and mapping between the OVM and the DTMC (implemented using MaTeLo) is provided via explicit traceability links to functional requirements. This process thus requires to perform selection of products of interest on the variability model and does not exploit probabilities and traces of the DTMC during such selection. Additionally, they assume that tests for all products of the SPL are modeled in the DTMC. This assumption may be too strong in certain cases and delay actual testing since designing the complete DTMC for a large SPL may take time. We thus explore a scenario where the DTMC drives product selection and prioritization.

B. Family Based Test Prioritization

Contrary to product based test derivation, our approach (in Fig. 2) only assumes partial coverage of the SPL by the usage model. For instance, the DTMC represented in Fig. 1c does not cover serving soda behavior because no user/tester exercised it. The key idea is to generate sequences of actions (i.e., finite traces) from the DTMC according to their probability to happen (step 1). For example, one may be interested in analyzing behaviors including serving teas for free, which will correspond to the sequence of actions $(free, tea, serveTea, take)$, since it is highly probable ($p = 0.729$). On the contrary, one may be interested in low probability because it can mean poorly tested or irrelevant products.

The generated sequences 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 FTS', corresponding to a pruned FTS according to the extracted sequences. Each valid sequence of actions is combined with the FTS' to generated a set of products that may effectively execute this sequence. The probability of the sequence to be executed allows to prioritize products exercising the behavior described in the FTS' (step 3).

1) Trace Selection in the DTMC: The first step is to extract sequences of actions (i.e., finite traces) from the DTMC according to desired parameters provided by the tester. We define a finite trace as a finite path (i.e., a finite sequence of labels) in a TS. This may differ from the standard notion of trace but since we are in our context, infinite traces are not really useful. Formally, a finite trace t corresponds to a tuple of labels : $t = (\alpha_1, ..., \alpha_n)$ such as $\exists s_i, s_j \in S_{TS} \wedge s_i \overset{(\alpha_1,...,\alpha_n)}{\Longrightarrow} s_j$ where $s_i \overset{(\alpha_1,...,\alpha_n)}{\Longrightarrow} s_j$ denotes the existence of a path

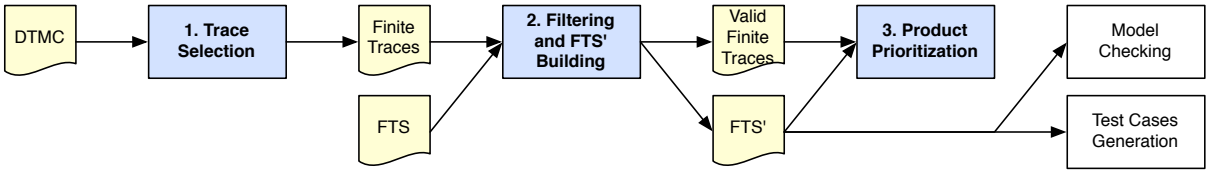


Fig. 2. Family based test prioritization approach

$(s_i \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} s_j)$ in the TS starting from s_i and ending in s_j with transitions labelled as $(\alpha_1, ..., \alpha_n)$.

To perform trace selection in a DTMC $dtmc$, we use a classical 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:

$$DFS(l_{max}, Pr_{min}, Pr_{max}, dtmc) = \{(\alpha_1, ..., \alpha_n) \mid i \overset{(\alpha_1,...,\alpha_n)}{\Longrightarrow} i \wedge (\nexists k : 1 < k < n \wedge i \overset{(\alpha_1,...,\alpha_k)}{\Longrightarrow} i) \wedge (Pr_{min} \leq Pr(\alpha_1, ..., \alpha_n) \leq Pr_{max})\}$$

where $\tau_{dtmc}(i) = 1$ and $Pr(\alpha_1, ..., \alpha_n) = \Pi_{k=1}^{n} P_{dtmc}(s_i, s_j)$ such as $s_i \xrightarrow{\alpha_k} s_j \in (i \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} i)$. 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. Finite sequences starting from and ending in i corresponds to a coherent execution scenario in the DTMC. 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 DTMC. 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 DFS algorithm is hence easier to implement and may better scale to large DTMCs.

Practically, this algorithm will build a n-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 DTMC. Its value will depend on the size of the DTMC and should be higher than the maximal "loop free" path in the DTMC 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 gives 5 finite traces:

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

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. 3. FTS' building algorithm

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

The downside is that the algorithm will possibly enumerate all the paths in the DTMC 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 [15] in order to avoid a complete state space exploration.

2) Traces Filtering using the FTS and Building the FTS': Generated finite traces from the DTMC may contain illegal sequences of actions (i.e., sequences of actions which can not be performed by any valid product of the SPL). 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 if there exists a product p in pl such as $t \subseteq [\![fts_{|p}]\!]_{TS}$, 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 behavior of the SPL appearing in the finite traces generated from the DTMC. Fig. 3 presents the algorithm

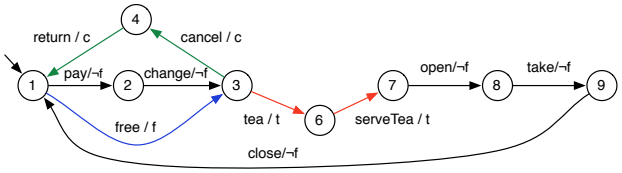


Fig. 4. FTS' of the soda vending machine

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 can execute the sequence of actions in t. 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 : $fLabels : (FTS, trace) \rightarrow \gamma \rightarrow \gamma$ 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 generated from step 1 contains two illegal traces: $(pay, change, tea, serveTea, take)$ and $(free, tea, serveTea, open, take, close)$. 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. 4.

3) Product Prioritization: At the end of step 2 in Fig. 2, we have an FTS' and a set of finite traces in this FTS'. This set of finite traces (coming from the DTMC) covers all the valid behaviors 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 DTMC. A valid finite trace $t = (\alpha_1, \ldots, \alpha_n)$ corresponding to a path $(s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} s_{n+1})$ in the DTMC (and in the FTS') has a probability $Pr(t)$ (calculated as in step 1) to be executed. We may perform bookkeeping of $Pr(t)$.

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. Formally, for t and a FTS' fts', the set of products $prod(t, fts') = \bigcap_{k=1}^{n} \{p \mid \gamma_{fts'}(s_k \xrightarrow{\alpha_k} s_{k+1})p = true\}$. 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 [16], the following formula can be calculated using a SAT solver: $\bigwedge_{k=1}^{n}(\gamma_{fts'}(s_k \xrightarrow{\alpha_k} s_{k+1})) \wedge booleanForm(d_{fts'})$.

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 priortization may be done by classifying the finite traces according to their probability to be executed, giving t-behaviorally 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 the $\neg f \wedge t$ feature expression and d_{vm}. This gives us a set of 8 products (amongst 32 possible):

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

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

4) Model Checking and Test Case Generation: Since the FTS' represents the set of valid products capable of executing the valid finite traces generated from the DTMC. It represents a subset of the behavior of the SPL that has to be assessed in priority according to the provided $[Pr_{min}, Pr_{max}]$ bounds. The FTS' may be verified using existing algorithms [9] and/or be used to generate test cases [17].

IV. Related Work

To the best of our knowledge, there is no approach prioritizing behaviors statistically for testing SPLs in a family-based manner. The most related proposal (outlined in section III) has been devised by Samih and Baudry [14]. This is a product-based approach and therefore requires selecting one or more products to test at the beginning of the method. One also needs that the DTMC covers all products of the SPL, which is not our assumption here.

There have been SPL test efforts to sample products for testing such as t-wise approaches (e.g. [4], [5], [18]). More recently sampling was combined with prioritization thanks to the addition of weights on feature models and the definition of multiple objectives [6], [7]. However, these approaches do not consider SPL behavior in their analyses.

To consider behavior in an abstract way, a full-fledged MBT approach [2] is required. Although behavioural MBT is well established for single-system testing [19], a survey [20] shows insufficient support of SPL-based MBT. However, there have been efforts to combine sampling techniques with modeling ones (e.g. [21]). These approaches are also product-based, meaning that may miss opportunities for test reuse amongst sampled products [22]. We believe that benefiting from the recent advances in behavioral modeling provided by the model checking community [15], [23]–[29], sound MBT approaches for SPL can be derived and interesting family-based scenarios combining verification and testing can be devised [17].

Our will is to apply ideas stemming from statistical testing and adapt them in an SPL context. For example, combining structural criteria with statistical testing has been discussed in [30], [31]. We do not make any assumption on the way the DTMC is obtained: via an operational profile [32] or by

analyzing the source code or the specification [31]. However, an uniform distribution of probabilities over the DTMC would probably be less interesting. As noted by Witthaker [8], 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 [13].

V. Conclusion

In this paper, we combine concepts stemming from statistical testing with SPL sampling to extract products of interest according to the probability of their execution traces gathered in a discrete-time markov chain representing their usages. As opposed to product-based sampling approaches, we select a subset of the full SPL behavior given as Featured Transition Systems (FTS). This allows us to construct a new FTS representing only the executions of relevant products. This such pruned FTS can be analyzed all at once, to enable allow test reuse amongst products and/or to scale model-checking techniques for testing and verification activities. Future work will naturally proceed to the full implementation of the approach presented here and its validation on concrete systems. This raise a number of challenges, including inference of usage models using various techniques such as the analysis of systems logs or symbolic execution of the software product line, as well as the design of efficient algorithms for trace extraction and FTS pruning. We also would like to consider partial traces (traces that do not need to end in the initial state). Although making prioritization less scalable, they may prove useful when the discrepancies between the behavioral and usage models are too important (partial execution can cope such situations easily) or to focus on specific feature interactions.

References

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. Spencer Peterson, "Feature-Oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 1990.

[2] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.

[3] C. H. P. Kim, S. Khurshid, and D. S. Batory, "Shared execution for efficiently testing product lines," in *ISSRE '12*, 2012, pp. 221–230.

[4] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. L. Traon, "Pairwise testing for software product lines: comparison of two approaches," *Software Quality Journal*, vol. 20, no. 3-4, pp. 605–643, 2012.

[5] M. Cohen, M. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *ISSTA 07*, 2007, pp. 129–139.

[6] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon, "Multi-objective test generation for software product lines," in *SPLC '13 (to appear)*, 2013.

[7] M. F. Johansen, Ø. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen, "Generating better partial covering arrays by modeling weights on sub-product lines," in *MoDELS '12*, 2012, pp. 269–284.

[8] J. A. Whittaker and M. G. Thomason, "A markov chain model for statistical software testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 812–824, 1994.

[9] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin, "Featured Transition Systems : Foundations for Verifying Variability-Intensive Systems and their Application to LTL Model Checking," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–22, 2013.

[10] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York Inc, 2005.

[11] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Computer Networks*, vol. 51, no. 2, pp. 456–479, 2007.

[12] A. von Rhein, S. Apel, C. Kästner, T. Thüm, and I. Schaefer, "The PLA model: on the combination of product-line analyses," in *VaMoS '13*. Pisa, Italy: ACM Press, 2013, p. 1.

[13] A. Feliachi and H. Le Guen, "Generating transition probabilities for automatic model-based test generation," in *ICST '10*. IEEE, 2010, pp. 99–102.

[14] H. Samih and B. Baudry, "Relating variability modelling and model-based testing for software product lines testing," in *ICTSS '12 Doctoral Symposium*, 2012.

[15] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT Press, 2008.

[16] K. Czarnecki and A. Wasowski, "Feature Diagrams and Logics: There and Back Again," in *SPLC 2007*. IEEE, Sep. 2007, pp. 23–34.

[17] X. Devroey, M. Cordy, G. Perrouin, E.-Y. Kang, P.-Y. Schobbens, P. Heymans, A. Legay, and B. Baudry, "A Vision for Behavioural Model-Driven Validation of Software Product Lines," in *ISoLA 2012, Part I*, ser. LNCS 7609, Margaria T., Steffen B., and Merten M., Eds. Crete: Springer-Verlag, 2012, pp. 208–222.

[18] M. B. Cohen, M. B. Dwyer, and J. Shi, "Coverage and adequacy in software product line testing," in *ROSATEA '06*, 2006, pp. 53–63.

[19] J. Tretmans, "Model based testing with labelled transition systems," in *Formal methods and testing*, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 1–38.

[20] S. Oster, A. Wöbbeke, G. Engels, and A. Schürr, "Model-based software product lines testing survey," in *Model-Based Testing for Embedded Systems*, ser. Computational Analysis, Synthesis, and Design of Dynamic Systems, J. Zander, I. Schieferdecker, and P. J. Mosterman, Eds. CRC Press, September 2011, pp. 339–382.

[21] M. Lochau, S. Oster, U. Goltz, and A. Schürr, "Model-based pairwise testing for feature interaction coverage in software product line engineering," *Software Quality Journal*, vol. 20, no. 3-4, pp. 567–604, 2012.

[22] A. von Rhein, S. Apel, C. Kästner, T. Thüm, and I. Schaefer, "The pla model: on the combination of product-line analyses," in *VaMoS '13*, 2013, p. 14.

[23] P. Asirelli, M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti, "Design and validation of variability in product lines," in *PLEASE '11*. New York, NY, USA: ACM, 2011, pp. 25–30.

[24] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi, "Formal description of variability in product families," in *SPLC '11*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 130–139.

[25] A. Classen, P. Heymans, P. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *ICSE '11*, 2011.

[26] A. Classen, P. Heymans, P. Schobbens, A. Legay, and J. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *ICSE '10*. New York, NY, USA: ACM, 2010, pp. 335–344.

[27] D. Fischbein, S. Uchitel, and V. Braberman, "A foundation for behavioural conformance in software product line architectures," in *ROSATEA '06*. New York, NY, USA: ACM, 2006, pp. 39–48.

[28] A. Gruler, M. Leucker, and K. Scheidemann, "Modeling and model checking software product lines," in *Formal Methods for Open Object-Based Distributed Systems*, G. Barthe and F. S. Boer, Eds., vol. 5051. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 113–131.

[29] K. Lauenroth, K. Pohl, and S. Toehning, "Model checking of domain artifacts in product line engineering," in *ASE '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 269–280.

[30] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre, "A new way of automating statistical testing methods," in *ASE '01*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 5–.

[31] P. Thévenod-Fosse and H. Waeselynck, "An investigation of statistical software testing," *Softw. Test., Verif. Reliab.*, vol. 1, no. 2, pp. 5–25, 1991.

[32] J. D. Musa, G. Fuoco, N. Irving, D. Kropfl, and B. Juhlin, "The operational profile," *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, vol. 154, pp. 333–344, 1996.