Towards statistical prioritization for software product lines testing

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

Towards Statistical Prioritization for Software Product Lines Testing

Xavier Devroey, Gilles Perrouin*, Maxime Cordy†, Pierre-Yves Schobbens
PReCISE, Fac. of Computer Science, UNamur, Belgium
{xde,mcr,gpe,pys}@info.fundp.ac.be

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

Patrick Heymans
PReCISE, Fac. of Computer Science, UNamur, Belgium
INRIA Lille-Nord Europe
Université Lille 1 – LIFL – CNRS , France
phe@info.fundp.ac.be

ABSTRACT

Software Product Lines (SPLs) 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 behaviour 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 behaviour. We discuss possible scenarios and give a prioritization procedure validated on a web-based learning management software.

Keywords

Software Product Line, Software Testing, Test Prioritization, Statistical Testing

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) [17].

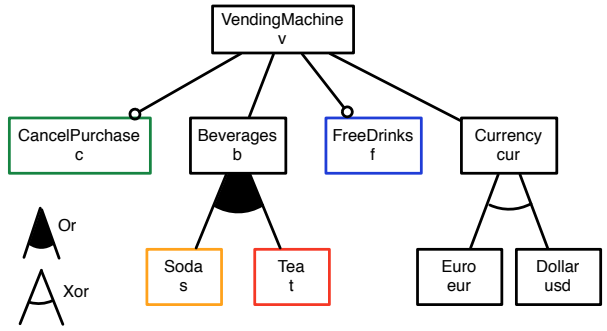
SPL testing the most common quality assurance technique in SPL engineering. As opposed to single-system testing where the testing process considers only one software product, SPL testing is concerned about minimizing the test effort for all the SPL products. Testing these products separately is clearly infeasible in real-world SPLs, which typically consist of thousands of products. Automated model-based testing [32] and shared execution [18] 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 entire as these methods still need to cover all the products.

Other approaches consist in testing only a representative sample of the products. Typical methods select these products according to some coverage criterion on the FD (e.g. all the valid couples of features must occur in at least one tested product [8,25]). An alternative method is to associate each feature with a weight and prioritize the products with the highest weight [15, 16]. This actually 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.

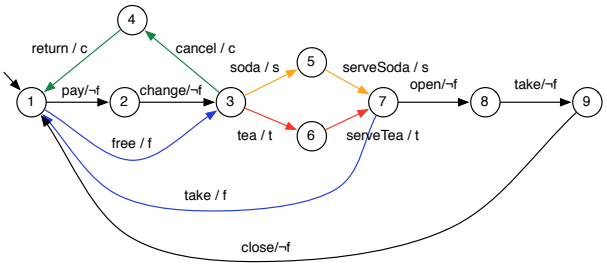
In this paper, we propose an approach to prioritize the products to test according to criteria based on the actual behaviour of the products, thereby increasing the relevancy of the testing activities and decreasing the risk of leaving errors undetected in many products. Our work leans on statistical testing [35], which generates test cases from a *usage model* represented by 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. We postulate that this ranking method can serve as a basis for prioritizing the products to test in an SPL. For instance, one may be interested only in products that can produce the executions that have the highest probability of occurrence. Usage models being extracted from running systems in a black-box manner, they are agnostic on vari-

*FNRS Postdoctoral Researcher
†FNRS Research Fellow

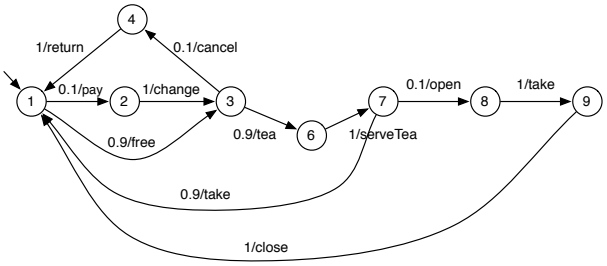
VAMOS '14 Nice, France



(a) Feature Diagram (FD)



(b) Featured Transition System (FTS)



(c) Usage model (DTMC)

Figure 1: The soda vending machine example [5]

ability and therefore cannot link a behaviour to the features needed to execute it. To overcome this, we assume the existence of design models describing the SPL behaviour. We then encode these models into a fundamental formalism – a *Featured Transition System* (FTS) – to obtain the missing link between executions and variability. The next step is to remove the behaviours of the FTS that were not selected during the analysis of the usage model. In our example, we would obtain a pruned FTS that exhibits only the most common behaviours, from which we can extract the products able to execute them.

The remainder of this paper is organized as follows: Section 2 presents the theoretical background underlying our vision. Our approach is detailed in Section 3. The results of a preliminary experiment are presented in Section 4. Section 5 discusses related research and Section 6 concludes the paper with challenges and future research directions.

2. BACKGROUND

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

2.1 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. Relations and constraints between features are usually represented in a Feature Diagram (FD) [17]. For example, Fig. 1a presents the FD of a soda vending machine [5]. A common semantics associated to a FD d (noted $[\![d]\!]$) is the set of all the valid products allowed by d.

Different formalisms may be used to model the behaviour of a system. To allow the explicit mapping from feature to SPL behaviour, Featured Transition Systems (FTS) [5] were proposed. FTS are transition systems (TS) 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. 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 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; and $\gamma : trans \rightarrow [\![d]\!] \rightarrow \{\top, \bot\}$ 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 transition whose feature expression is not satisfied by the product.

We define the semantics of an FTS as a function that 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. 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 [5], 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 introduce the notion of usage model in [35] and define it as a TS where transitions have probability values defining their occurrence likelihood. Formally, their usage model is actually 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 defined as in FTS;
- $P : S \times Act \times S \rightarrow [0, 1]$ 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;
- $\tau : S \rightarrow [0, 1]$ is the vector containing the probabilities to be in the initial state when the system starts, with the following constraint : $\exists i : (\tau(i) = 1 \wedge \forall j \neq i : \tau(j) = 0)$;
- $\forall s_i \in S : \sum_{\alpha \in Act, s_j \in S \bullet s_i \xrightarrow{\alpha} s_j} P(s_i, \alpha, s_j) = 1$, 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. Thus, the fact that transitions are defined either as sequences of states and actions or as sequences of states is meaningless. 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 three models: a FD d to represent the features and their constraints (in Fig. 1a), an FTS fts over d (in Fig. 1b) and a usage model represented by a DTMC $dtmc$ (in Fig. 1c) whose states, actions, and transitions are subset 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 stochasticity (see, e.g., [**?**]). 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[1]) do not support DTMCs with variability. Implementing our approach on top of such tools would be made harder should we use the aforementioned unified formalism. Second, 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 on the features of the system they are applied to. Third, since the DTMC is built from existing software executions, it may be incomplete (as in Fig. 1c). Because of that, there may exist FTS executions that are not 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.

Since the usage model does not consider features, there may exist 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). The combined use of DTMCs and FTS allows us to detect such inconsistencies.

We propose two testing scenarios: product-based test derivation and family-based test prioritization.

3.1 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 : $S_{dtmc} \subseteq S_{ts} \wedge Act_{dtmc} \subseteq Act_{ts} \wedge trans_{dtmc} \subseteq trans_{ts}$. We assume that the probabilities of the removed transitions are uniformly distributed on adjacent transitions (since the probability axiom $\forall s_i \in S : \sum_{s_j \in S} P(s_i, s_j) = 1$ holds). Finally, we generate test cases using statistical testing algorithms on the DTMC [12, 35].

This scenario is proposed by Samih and Baudry [26]. 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.

3.2 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 behaviour because no user exercised it. The key idea is to generate traces from the DTMC according to their probability to happen (step 1). For example, one may be interested in analyzing the "serving teas for free" behaviour, which will correspond to the following 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 products.

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 FTS', corresponding to a pruned FTS according to the extracted traces. Each valid trace is combined with the FTS' to generated 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 the FTS' (step 3).

3.2.1 Trace Selection in the DTMC

The first step is to extract traces from the DTMC 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 as $(s_j, \alpha_{j+1}, s_{j+1}) \in trans, \forall j \in [0, n-1]$.

To perform trace selection in a DTMC $dtmc$, 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:

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

where $\tau_{dtmc}(s_0) = 1$ and

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

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. Finite traces starting from and ending in i corresponds to a coherent execution scenario

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

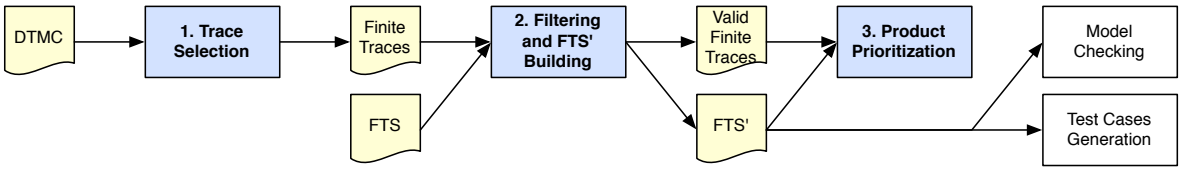


Figure 2: Family-based test prioritization approach

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:

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

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 [3] in order to avoid a complete state space exploration.

3.2.2 FTS-Based Trace Filtering and FTS Pruning

Generated finite traces from the DTMC may be illegal: behaviour that cannot be executed 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

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'

Figure 3: FTS' building algorithm

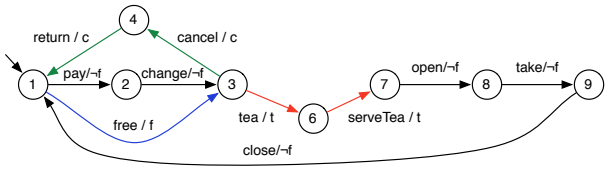


Figure 4: FTS' of the soda vending machine

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 finite traces generated from the DTMC. Fig. 3 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 : $fLabels : (FTS, trace) \rightarrow \gamma \rightarrow \gamma$ 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.

3.2.3 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 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 DTMC. 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 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 [10], 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-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 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. CASE STUDY

To assess the feasibility of our proposed approach, we decided to apply it on *Claroline* [4], an open-source web-based application dedicated to learning and on-line collaborative work. Analysing such systems using Markov Chains has already been investigated [28] (e.g, for link prediction) suggesting that our approach apply to this domain. The instance of Claroline at University of Namur[2] is the main communication channel between students and lecturers. Students may register to courses and download documents, receive announcements, submit their assignments, perform online exercises, etc. Due to space constraint, the complete models are not shown in this paper but may be downloaded at `http://info.fundp.ac.be/~xde/fts-testing`.

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

4.1 Inferring the Usage Model

To derive the DTMC usage model, we used the anonymized Apache access log provided by the IT support team of the university. This file (5,26 Go) contains all the HTTP requests made to access the Claroline instance from January 1st to October 1st 2013. We chose to consider only requests to access PHP pages accessible to users. It represents a total of 12.689.030 HTTP requests. We setup an automated way to infer the usage model from the log, as it would be intractable to do so manually. First, Claroline Log was translated into a MySQL database to ease processing of requests. Indeed, each entry of the table represents a single access to a page: we had to reconstruct the behaviour of an user by grouping requests. We followed a similar approach as Sampath [27]: we group visited PHP pages by the user (according to their line numbers in the log) to Claroline pages for a given IP and within a 45 minutes timeframe. Once these user traces are computed, we can learn an usage model from them. There are various machines learning approaches to learn such underlying DTMCs [33]: we elected n-grams (contiguous sequence of n elements) for their close relationship to DTMCs and their reported success in practice for real instances [33]. Our implementation can be thought of a 2-gram one without smoothing (we do not consider unseen 2-grams as we focus only on observed behaviour). Concretely, each page is mapped to a state in the usage model and each trace is completed so that it ends in the start state. In the obtained DTMC, the set Act contains actions that indicate the target state of the transition (since they correspond to request to a page). In that setting, they cannot be more than one transition for any two given states. We can therefore ignore actions in our probability computation method. Then, the probability of moving from state s_i to another s_j is given as

$$P(s_i, s_j) = \frac{occ(s_i, s_j)}{\sum_{s \in S} occ(s_i, s)}$$

where $occ(s_i, s_j)$ is the occurrence count of the related 2-gram. We were able to process the entirety of our log database, ending up in an usage model formed of 96 states and 2149 transitions. The whole inference process took approximately 2 hours on a Ubuntu Linux machine with a Intel Core i3 (3.10GHz) processor and 4GB of memory.

4.2 Building Family Models

To obtain the Claroline FD and FTS we proceeded the following way. The FD was built manually by inspecting a locally installed Claroline instance (Claroline 1.11.7 [3]). The FD describes Claroline with 3 main features decomposed in sub-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 ($Available$, $AvailableFromTo$

[3] `http://sourceforge.net/projects/claroline/files/Claroline/Claroline_1.11.7/`

Table 1: Traces generation

	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

Table 2: Most frequent features in valid traces

Run 1	CourseDocument, CourseForum, CourseAnnoucements, CourseExercise, RegisteredUser
Run 2	CourseDocument, CourseAnnoucements, CourseForum, RegisteredUser, CourseExercise
Run 3	CourseDocument, RegisteredUser, CourseForum, CourseAnnoucements, CourseWork
Run 4	CourseDocument, RegisteredUser, CourseForum, CourseAnnoucements, CourseWork

or $Unavailable$), may be publicly visible ($PublicVisibility$) or not ($MembersVisibility$), may authorize registration to identified users ($AllowedRegistration$) 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 to different teaching purposes, e.g., an agenda ($opt\ CourseAgenda$), an announcement panel ($opt\ CourseAnnoucements$), a document download section where teacher may post documents and students may download them ($opt\ CourseDocument$), an on-line exercise section ($opt\ CourseExercise$), etc. Since we are in a testing context, one instance of the FD does not represents a complete Claroline instance but rather only the minimal instance needed to play a test-set (i.e., test cases formed as sequences of HTTP requests). Basically, it will correspond to a Claroline instance with its subscription features, one particular user and one particular course. In order to represent a complete Claroline instance, we need to introduce cardinalities on the $Course$ and $User$ features [22]. Eventually we obtained a FD with 44 features.

Regarding the FTS, we employed a web crawler (i.e., a Python bot that systematically browse and record information about a website) [29] on our local Claroline instance to discover states of the FTS which as for the DTMC represent visited page on the website. 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: $S_{dtmc} \subseteq S_{fts} \wedge Act_{dtmc} \subseteq Act_{fts} \wedge trans_{dtmc} \subseteq trans_{fts}$. Moreover, 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). 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, 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.

4.3 Setup and Results

The DFS algorithm has been applied 4 times to the Claroline DTMC with a maximal length of 98, which corresponds to the number of states, a maximal probability of 1 and 4 different minimal probabilities: 10^{-4}, 10^{-5}, 10^{-6} and 10^{-7} to see if a pattern emerge. The execution time took from less than a minute for the first run to ± 8 hours for run 4. Additionally to those values, the algorithm has been parametrized to consider each transition only once. This modification has been made since we discovered after a few tests that the algorithm produced lot of traces with repeated actions which is of little interest in the product prioritization context. Those repetitions were due to the huge number of loops in the Claroline DTMC.

Table 1 presents the parameters provided to the DFS and the resulting traces before (# DTMC tr.) and after checking with the FTS (# Valid tr.). Surprisingly, all traces generated from the DTMC 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. As expected, the average size of the traces increase as the Pr_{min} decrease to tend to $9,88$, the average size of the traces used to generate the DTMC.

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 reduce 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'.

Table 2 presents the 5 most frequents features appearing in the feature expressions associated to the transitions of the valid traces. In all the runs, $CourseDocument$ appears the most frequently, from 47% in run 1 to 53% in run 4.

4.4 Discussion

After running our approach on the Claroline case study, we made the following observations :

First, the independence of the features in the FD and the low size of valid FTS traces does not allow to reduce significantly the number of products associate to each trace. This is caused by the web nature of the application which allows a very free way of travelling in the FTS with lots of independent feature expressions (i.e., most feature expressions concern optional features with few dependencies to other features). In order to get a more concrete product, one could generate longer traces in the DTMC by coupling probabilistic approach to more classical state/transition coverage criteria [21]. This is left for future work.

Although it is possible to generate the list of products for each trace by taking the conjunction of the feature expressions of the transitions of the trace and the FD, this seems of little interest in this case. The valid FTS traces are short and the FD has too much variability to get a reasonable list of products to test. However, we can tackle this problem by taking only needed features to ensure the execution of a valid

trace on the FTS. For instance, if we take the less probable trace in run 1 (probability=1.0149×10^{-4}) and its associated feature expression $RegisteredUser \wedge CourseDocument$, the minimal feature configuration to execute the trace on the FTS is done by conjunction the feature expression with the FD and negation all optional features except *Course Document* and $RegisteredUser$. This gives us a set of 260 possible products. This set is reduced to 20 products if we consider only courses available to student with an id, which is the most classical scenario in the UNamur Claroline instance. The tester, with knowledge of the application domain, may also use the feature frequency (presented in table 2) in order to reduce the number of products associated to each trace.

Thirdly, Claroline has a fine grained access control system. In its basic setup, it comes with three user roles: Administrator, Teacher and User. According to its role, a user may or may not perform different actions and access different functionalities. Claroline also allows public access to some pages and functionalities (e.g., a course description) to anonymous users. We think that since those 4 user roles have very different usage profiles, a better approach would be to create 4 different usage models, one for each user role. This confirm our design choice to keep usage model and FTS separated. Unfortunately, it was not possible in our case with the provided Apache access log since the user roles have been erased in the anonymisation process.

Finally, we also tried to implement other selection criteria to apply the DTMC. One may want to get the most probable traces in the DTMC (max). On the Claroline case, this gives us a lot of small traces, most of them with 2 or 3 transitions implying each time the `download.php` page. After a few investigations, we discovered that this page is used to download documents, text, images, etc. from the platform and that it is the most frequent HTTP request in the Apache access log that we processed. This is coherent with the fact that all the states in the DTMC may return to the initial state, allowing very small traces. On the contrary, getting the less probable traces (min) is hard since it requires a full exploration of the DTMC. Other abstraction/simulation techniques will be investigate in future work in order to allow such criteria and to improve the actual DFS algorithm.

The main threat to our case study is the nature of the considered application. This first case study has been performed on a very particular kind of application: a web application accessible trough PHP pages in a web browser. 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 our future work, we will apply our approach on other cases from various domains (e.g., system engineering) where the transitions from state to state are more constrained.

5. RELATED WORK

To the best of our knowledge, there is no approach prioritizing behaviours statistically for testing SPLs in a family-based manner. The most related proposal (outlined in section 3) has been devised by Samih and Baudry [26]. 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. [8, 9, 25]). More recently sampling was combined with prioritization thanks to the addition of weights on feature models and the definition of multiple objectives [15,16]. However, these approaches do not consider SPL behaviour in their analyses.

To consider behaviour in an abstract way, a full-fledged MBT approach [32] is required. Although behavioural MBT is well established for single-system testing [31], a survey [24] shows insufficient support of SPL-based MBT. However, there have been efforts to combine sampling techniques with modelling ones (e.g. [20]). These approaches are also product-based, meaning that may miss opportunities for test reuse amongst sampled products [34]. We believe that benefiting from the recent advances in behavioural modelling provided by the model checking community [1–3,6,7,13,19], sound MBT approaches for SPL can be derived and interesting family-based scenarios combining verification and testing can be devised [11].

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 [14, 30]. We do not make any assumption on the way the DTMC is obtained: via an operational profile [23] or by analyzing the source code or the specification [30]. However, an uniform distribution of probabilities over the DTMC would probably be less interesting. As noted by Witthaker [35], 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 [12].

6. 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 behaviour 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 proceed to the improvement of the first implementation and its validation on other cases from various domains. There remains a number of challenges, the main one being an efficient trace extraction algorithm using abstraction and/or simulation techniques. We will also associate statistical prioritization to more classical test selection criteria like transition coverage and state coverage in order to improve the quality of the extracted traces.

Acknowledgments

Thanks to Jean-Roch Meurisse and Didier Belhomme for providing the Webcampus' Apache access log.

7. REFERENCES

[1] P. Asirelli, M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. Design and validation of variability in

product lines. In *PLEASE '11*, pages 25–30. ACM, 2011.

[2] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi. Formal description of variability in product families. In *SPLC '11*, pages 130–139. IEEE, 2011.

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

[4] Claroline. `http://www.claroline.net/`.

[5] 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. *TSE*, PP(99):1–22, 2013.

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

[7] 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*, pages 335–344. ACM, 2010.

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

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

[10] K. Czarnecki and A. Wasowski. Feature Diagrams and Logics: There and Back Again. In *SPLC '07*, pages 23–34. IEEE, Sept. 2007.

[11] 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. ISoLA '12, pages 208–222. Springer-Verlag, 2012.

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

[13] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA '06*, pages 39–48. ACM, 2006.

[14] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *ASE '01*. IEEE Computer Society, 2001.

[15] 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.

[16] 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*, pages 269–284, 2012.

[17] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. Spencer Peterson. Feature-Oriented domain analysis (FODA) feasibility study. Technical report, Soft. Eng. Inst., Carnegie Mellon Univ., 1990.

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

[19] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *ASE '09*, pages 269–280. IEEE, 2009.

[20] 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*, 20(3-4):567–604, 2012.

[21] A. P. Mathur. *Foundations of software testing*. Pearson Education, 2008.

[22] R. Michel, A. Classen, A. Hubaux, and Q. Boucher. A formal semantics for feature cardinalities in feature diagrams. VaMoS '11, pages 82–89. ACM, 2011.

[23] J. D. Musa, G. Fuoco, N. Irving, D. Kropfl, and B. Juhlin. The operational profile. *NATO ASI series F Comp. and Syst. Sc.*, 154:333–344, 1996.

[24] S. Oster, A. Wöbbeke, G. Engels, and A. Schürr. Model-based software product lines testing survey. In *Model-Based Testing for Embedded Systems*, pages 339–382. CRC Press, 2011.

[25] 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*, 20(3-4):605–643, 2012.

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

[27] S. Sampath, R. Bryce, G. Viswanath, V. Kandimalla, and A. Koru. Prioritizing user-session-based test cases for web applications testing. In *ICST '08*, pages 141–150, 2008.

[28] R. R. Sarukkai. Link prediction and path analysis using markov chains. *Computer Networks*, 33(1-6):377–386, 2000.

[29] Scrapy. `http://scrapy.org/`.

[30] P. Thévenod-Fosse and H. Waeselynck. An investigation of statistical software testing. *Softw. Test., Verif. Reliab.*, 1(2):5–25, 1991.

[31] J. Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer-Verlag, 2008.

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

[33] S. Verwer, R. Eyraud, and C. Higuera. Pautomac: a probabilistic automata and hidden markov models learning competition. *Machine Learning*, pages 1–26, 2013.

[34] 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*. ACM, 2013.

[35] J. A. Whittaker and M. G. Thomason. A markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, 1994.