Coverage Criteria for Behavioural Testing of Software Product Lines

Xavier Devroey[1], Gilles Perrouin[1⋆], Axel Legay[2], Maxime Cordy[1⋆⋆], Pierre-Yves Schobbens[1], and Patrick Heymans[1]

[1] PReCISE Research Center, Faculty of Computer Science, University of Namur, Belgium
{xavier.devroey, maxime.cordy, gilles.perrouin, pierre-yves.schobbens, patrick.heymans}@unamur.be
[2] INRIA Rennes Bretagne Atlantique, France axel.legay@inria.fr

Abstract. Featured Transition Systems (FTS) is a mathematical structure to represent the behaviour of software product line in a concise way. The combination of the well-known transition systems approach to formal behavioural modelling with feature expressions was pivotal to the design of efficient verification approaches. Such approaches indeed avoid to consider products' behaviour independently, leading to often exponential savings. Building on this successful structure, we lay the foundations of model-based testing approach to SPLs. We define several FTS-aware coverage criteria and report on our experience combining FTS with usage-based testing for configurable websites.

Keywords: Coverage Criteria, Model Based Testing, Software Product Line Engineering

1 Introduction

A *Software Product Line* (SPL) "is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [8]. Features are thus the key to the discrimination of SPL members by showing their commonalities and differences. Such features are commonly organized in a *Feature Model* (FM) [18] which represents all the possible products of the SPL by expressing relationships and constraints between such features.

As for any software engineering paradigm, providing efficient Quality Assurance (QA) (e.g. model-checking and testing) techniques is essential to SPL engineering success. Devising an approach to SPLs QA requires to deal with the well-known *combinatorial explosion* problem as the number of products to consider for validation is growing exponentially with the number of features. In

⋆ FNRS Postdoctoral Researcher
⋆⋆ FNRS Research Fellow

The final authenticated publication is available online at https://doi.org/10.1007/978-3-662-45234-9_24

the worst case, no more than 270 features are needed to derive as many products as there are atoms in the universe. Industry reports dealing regularly with thousands of features in their product lines [14, 2] and the Linux kernel model is now roughly composed of 8,000 features. Thus, combinatorial explosion poses both theoretical and practical challenges for SPL QA. Depending on the QA approach (model checking or testing) and abstraction level (model, code) several strategies have been designed, which can be positioned on various edges of the product-line analysis cube [26]. Our research strives to provide generic solutions at the model level, both for verification and testing. In [7, 5, 6], we have proposed model-checking algorithms for *Featured Transition Systems* (FTSs), a variability-aware extension of transition systems. Contrary to enumerating approaches that would visit the state space of each product, our algorithms exploit the structure of the FTS in order to explore common behaviours only once. All those model-checking results have been implemented in ProVeLines,[10] that is a product line of model checkers for FTS.

Automated model-based testing [29] and shared execution [20] 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. To address this issue, we previously developed ideas based on sampling and prioritization principles [25, 16]. Typical methods in this area define a coverage criterion on an FM (e.g. all the valid couples of features must occur in at least one tested product [25, 9]) and extract configurations of interest to be validated. *Combinatorial interaction testing* allows drastic reduction of the configuration space from billions to few dozens or hundreds of products. It is possible to prioritize extracted configurations using coverage metrics or by assigning weights to features [16, 17], eventually leading multi-objective SPL testing [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 line with our preliminary vision [11], we believe that FTS are also suitable to establish a model-based testing framework for SPLs enabling both family-based and product-based strategies and benefiting from the experience gained by the model-checking community. In this paper, we are currently concerned with the definition of various coverage criteria to support FTS-based testing. We adapt existing concepts and structural coverage criteria known for transition systems to the FTS formalism. We then report on our previous experience [12] defining a *usage-based* [31] coverage approach, based on the extraction of Discrete Time Markov Chain (DTMC) from an Apache log of an online course management system. Behaviours of interests (selected according to a given probability interval) in the DTMC are then run on an FTS (assumed to be provided by SPL designers), enabling the projection of associated products and features related to those behaviours and test case generation.

Section 2 provides the background to FTS-based modelling and verification, required to define what FTS-based testing is as well as structural and usage

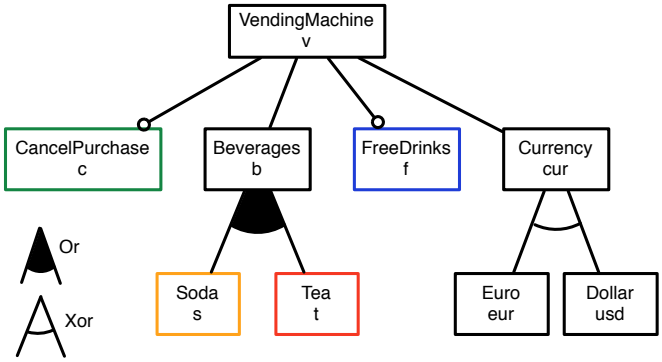

Fig. 1. Soda Vending Machine Feature Diagram [5]

coverage criteria in section 3. Section 5 concludes the paper and outlines future directions.

2 FTSs: Background

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) [18]. For example, Fig. 1 presents the FD of a soda vending machine [6]. 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) [6] 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. Thus it is possible to determine products that are the cause of a violation or a failed test.

Definition 1 (Featured Transition System (FTS)). *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* $\gamma : trans \to [\![d]\!] \to \{\top, \bot\}$ *is a total function labelling each transition with a boolean expression over the features, which specifies the products that can execute the transition.*

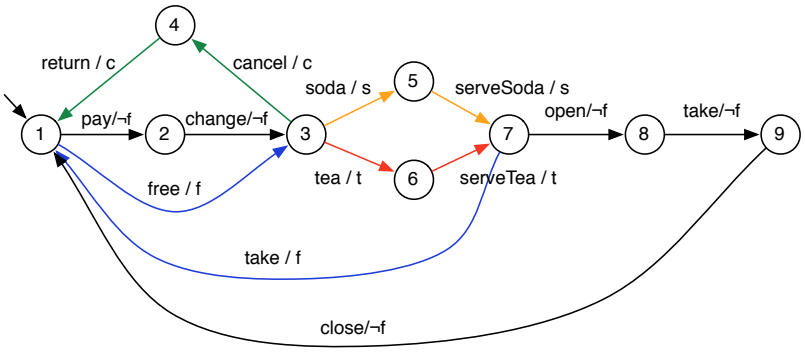

Fig. 2. Soda Vending Machine FTS [5]

Additionally, we consider the *projection* of an FTS onto a product $p \in [\![d]\!]$ noted $fts_{|p}$, the syntactical transformation of removing all transitions labelled with features not in p, thus resulting in the TS representing the behaviour of this particular product (see [6]). For instance: $\neg f$ in Fig. 2 indicates that only products that have not the $free$ feature may fire the $pay, change, open, take$ and *close* transitions. Thus, only those transitions will appear in their respective projections.

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 sequences of 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. 2 presents the FTS modeling a vending machine SPL. Without loss of generality, we consider FTSs in which the only allowed loops go through the initial state. This is useful to deal with finite traces in practice [12].

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. Other works on modeling software product lines can be found, e.g., in [15, 13].

3 SPL Behavioural testing using FTSs

Fig. 3 presents the classical testing process in a Model-Based Testing approach for single systems [29]. First, the test engineer builds a test model of the System Under Test (SUT) from its requirements. Then, according to some selection criteria, an abstract test suite (i.e., set of abstract test cases) is automatically generated. For instance, if using Transitions Systems in order to model the behaviour of a SUT, abstract test cases will represent sequences of abstract actions

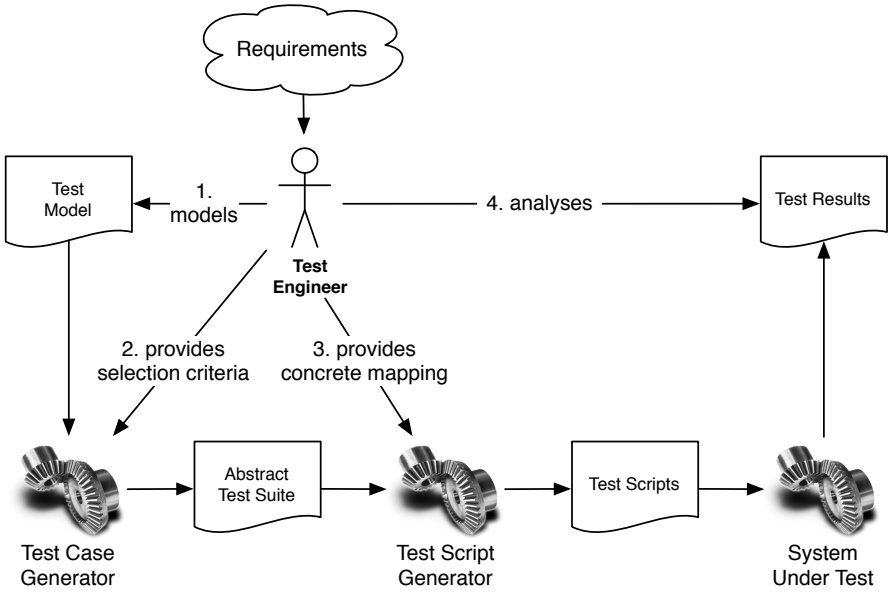

Fig. 3. Model-Based Testing general approach [29]

that should be executed on the SUT [28]. The abstract test cases are concretized using a mapping provided by the test engineer in order to match actions (with input values) of the system before being executed on the SUT. This execution may be manual or automated depending on the formalism of the concrete test cases (e.g., textual description of the operations to perform, automated scripts, etc.). Finally, the results of the tests executions are analysed by the test engineer.

In order to efficiently test SPLs, we propose to adopt FTSs as the formalism to represent SPLs behaviour as the test model of a MBT approach. In [11], we sketched a Quality Assessment (QA) framework with FTSs as the shared behavioural model representation for SPLs. As illustrated in figure 4, FTSs serve as input for QA activities (roughly decomposed in Model-Checking and Testing). Other processing oriented models (e.g., Markov Chain [12], LTL formula [6]) may be joined to the FTS for specific QA activities (e.g., test case prioritization [12]). FTS and computation oriented models are not meant to be used by QA engineer. They are the results of a model to model transformation from abstracted representations of the Feature Diagram (FD), SPL behaviour, formula and/or coverage criteria used by the QA activities. The framework will offer a language (with abstraction and composition mechanisms), *State Diagram Variability Analysis (SDVA)* based on UML state machines to model the behaviour of the SPL. Once the input models are transformed into a FTS and processing oriented models, they can be used to perform model-checking and/or

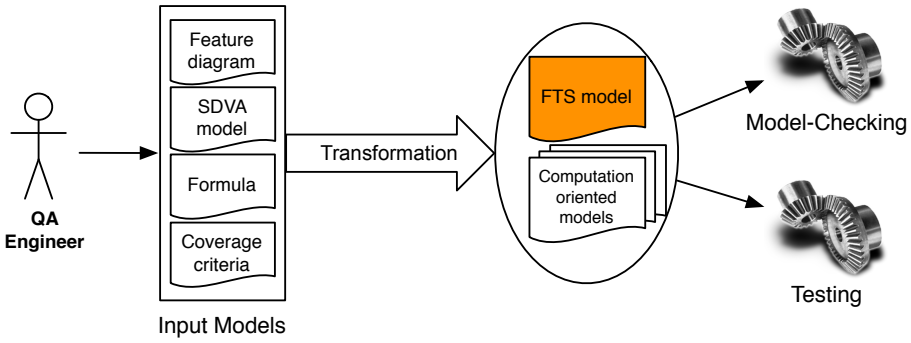

Fig. 4. Framework overview

testing activities (e.g., test case generation). By using a common representation (i.e., FTSs), we believe that the testing community may benefits from the last advances made in model-checking SPLs [6]. In [12] we present a first step in this direction by combining statistical testing techniques with FTSs in order to prioritize product testing.

3.1 Transition Systems MBT applied to FTSs

In order to select relevant test cases, proper coverage criteria have to be defined at the SPL level. A coverage criteria is an adequacy measure to qualify if a test objective is reached when executing a test suite on a SUT. In classical MBT approaches, when working with state-transitions models (e.g., TS), most commonly used selection criteria are structural criteria: state, transition, transition-pair and path coverage [22, 29]. The state/transition coverage criteria specifies that when executing a test suite on the SUT, all the states /transitions (resp.) of the test model are visited/fired (resp.) at least once. The transition-pair coverage specifies that for each state, all the ingoing-outgoing transitions pairs are fired at least once. The path coverage criteria specifies that each path in the test model has to be executed at least once. In the following, we define the notion of test case for a FTS, transpose the classical structural criteria, discuss some observations and redefine the test case selection problem for FTSs.

Abstract Test Case and Test Suite An abstract test case corresponds to a finite trace (i.e., finite sequence of actions) in the FTS.

Definition 2 (Abstract Test Case). *Let* $fts = (S, Act, trans, i, d, \gamma)$ *be an FTS, let* $atc = (\alpha_1, \ldots, \alpha_n)$ *where* $\alpha_1, \ldots, \alpha_n \in Act$ *be a finite sequence of actions. The abstract test case atc is valid iff :*

$$fts \stackrel{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$$

Where $fts \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$ *is equivalent to* $i \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$*, meaning that there exists a state* $s_k \in S$ *with sequence of transitions labelled* $(\alpha_1, \ldots, \alpha_n)$ *from* i *to* s_k*.*

This definition is similar to classical test case definitions for TS test models [22]. However, for an FTS, it is possible to extract sequences of actions that cannot be executed by any product of the SPL (e.g., the related transitions contains mutually exclusive features). This leads us to the definition of an executable abstract test case:

Definition 3 (Executable Abstract Test Case). *Let* $fts = (S, Act, trans, i, d, \gamma)$ *be an FTS, let* $atc = (\alpha_1, \ldots, \alpha_n)$ *where* $\alpha_1, \ldots, \alpha_n \in Act$ *represents a sequence of actions in* fts *be an abstract test case. An abstract test case* atc *is* executable *if it can be executed by at least one product of the product line:*

$$\exists p \in [\![d]\!] : fts_{|p} \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$$

Where $fts_{|p} \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$ *is equivalent to* $i \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$*, meaning that there exists a state* $s_k \in S$ *with sequence of transitions labelled* $(\alpha_1, \ldots, \alpha_n)$ *from* i *to* s_k *in the projection of the FTS onto* p*.*

We make a difference between abstract test case and *executable* abstract test case. Contrary to executable abstract test case, an abstract test case has not to be necessary executable by at least one product of the SPL. Since the FTS is a model of the behaviour of the SPL, it may be interesting to use abstract test cases which may not (according to the model) be executed by any product in order to do mutation testing (by mutating feature expressions), security testing (to detect undesired behaviours), etc. If this abstract test case can be executed on a concrete implementation, it reveals a modelling issue or an implementation error. Similarly:

Definition 4 (Executable Abstract Test Suite). *An abstract test suite is a (possibly empty) finite set of abstract test cases. An executable abstract test suite is an abstract test suite that contains only executable abstract test cases.*

An "empty" abstract test suite has no practical value, but it can be the result of a too restrictive or inconsistent selection process. However, we keep this liberal definition to support the definition of selection procedures in the general case.

3.2 Coverage Criteria

In order to efficiently select abstract test cases, the test engineer has to provide selection criteria [29]. We redefine hereafter classical structural selection criteria as a function that, for a given FTS and an executable abstract test suite, returns a value between 0 and 1 specifying the coverage degree of the executable abstract test suite over the FTS (0 meaning no coverage and 1 the maximal coverage). As for TS [29], we consider only coverage criteria for states reachable from the initial state (a state s_i is reachable iff $\exists p \in [\![d]\!] \wedge \exists \alpha_1, \ldots, \alpha_n \in Act$ such as $fts_{|p} \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow} s_i$). Formally :

Definition 5 (Coverage Criterion). *A coverage criterion is a function cov that associates an FTS and an abstract test suite over this FTS to a real value in* $[0, 1]$

Classical structural coverage criteria are defined as follow, we illustrate each coverage criteria with test suites satisfying the criteria for the Soda Vending Machine FTS [5] defined in figure 2 :

Definition 6 (State/All-States Coverage). *The state coverage criterion relates to the ratio between the states visited by the test cases pertaining to the abstract test suite and all the states of the FTS. When the value of the function equals to* 1*, the abstract test suite satisfies* ***all-states coverage****.*

The all-states coverage criteria is the weakest structural coverage criteria, it specifies that when executing the test suite, each state has to be visited at least once. On the Soda Vending Machine, an all-states covering abstract test suite may be:

$$\{(pay, change, soda, serveSoda, open, take, close)$$
$$(free, tea, serveTea, take); (free, cancel, return)\}$$

Definition 7 (Transition/All-Transitions Coverage). *Transition coverage relates to the ratio between transitions covered when running abstract test cases on the FTS and the total number of transitions of the FTS that are executable by at least one valid product. When this ratio equals to* 1*, then the abstract test suite satisfies* ***all-transitions*** *coverage.*

The all-transitions coverage specifies that, ideally, each transition is fired at least once when executing the abstract test suite on the FTS. In this case, a satisfying abstract test suite for a coverage of 1 on the Soda Vending Machine may be the same as the one defined for the all-state coverage.

Definition 8 (Transition-Pairs/All-Transition-Pairs Coverage). *The transition-pairs coverage considers adjacent transitions successively entering and leaving a given state. As for transition coverage, only pairs that are executable by at least one product are considered in the ratio. When the coverage function reaches the value of* 1*, then the abstract test suite covers* ***all-transition-pairs****.*

The all-transition-pairs coverage specifies that for each state, each couple of entering/leaving transition has to be fired at least once. On the soda vending machine, an abstract test suite with a all-transitions-pairs coverage of 1 may be:

$$\{(pay, change, soda, serveSoda, open, take, close); (pay, changecancel, return);$$
$$(pay, change, tea, serveTea, open, take, close); (free, soda, serveSoda, take);$$
$$(free, tea, serveTea, take); (free, cancel, return)\}$$

Definition 9 (Path/All-Paths Coverage). *Path coverage takes into account executable paths, that is sequence of actions* $(\alpha_1, \ldots, \alpha_n)$ *from* i *to* i *such that* $\exists p \in [\![d]\!] : fts_{|p} \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow} i$*. If the coverage function value computing the ratio between executable paths covered by the test cases runs on the FTS and total executable paths in the FTS is* 1*,* ***all-paths*** *coverage has been reached.*

The all-path coverage specifies that each executable path in the FTS should be followed at least once when executing the abstract test suite on it. On the soda vending machine, it gives an executable abstract test suite equal to the one defined for all-transitions-pair coverage.

3.3 Test Case Product Selection

Once the test cases are selected, they have to be concretized (step 3 in Fig. 3) in order to (i) get the implementation of the products on which the concrete test cases will be executed and (ii) get the concrete actions to perform with the adequate input values for each test case. This last point may be done using existing concretization techniques once the products are selected [22, 29]. For (i), the implementations able to execute a given abstract test case *atc* in a FTS corresponds to all the products (i.e., valid configurations) of the FD ($[\![d]\!]$) that satisfy all the feature expressions associated to the transitions fired in the FTS when executing *atc* :

Definition 10 (Abstract Test Case Product Selection). *Given a FTS* $fts = (S, Act, trans, i, d, \gamma)$ *and an abstract test case* $atc = (\alpha_1, \ldots, \alpha_n)$ *with* $(\alpha_1, \ldots, \alpha_n) \in Act$*, the set of products able to execute atc is defined as:*

$$prod(fts, atc) = \{p \in [\![d]\!] \mid fts_{|p} \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}\}$$

It corresponds to all the products able to execute the sequence of actions in the abstract test case.

Similarly, for an abstract test suite, we have:

Definition 11 (Abstract Test Suite Product Selection). *Given a FTS* $fts = (S, Act, trans, i, d, \gamma)$ *and an abstract test suite* $ats = \{atc_1, \ldots, atc_n\}$*, the set of products able/needed to execute ats:*

$$prods(fts, ats) = \bigcup_{k=1}^{n} prod(fts, atc_i)$$

Since the main interest in SPL testing is to reduce the number of products to test, we also define the minimal set of products needed to execute an abstract test suite.

Definition 12 (P-Minimal Abstract Test Suite Product Selection). *Let* fts *be an FTS and* ats *be an abstract test suite. A minimal set of products needed to execute* ats *over* fts *is a minimal subset* $pMinProd(fts, ats)$ *of* $prods(fts, ats)$ *such that* $\forall atc \in ats : \exists p \in pMinProd(fts, ats)$ *such that* $fts_{|p} \overset{atc}{\Longrightarrow}$

Test Case Minimality Since FTSs represents all the TSs of all the possible products of the SPL, it is possible that some coverage criteria may not be completely achieved ($\forall atc : cov(fts, ats) < 1$). For instance, some states may not be reachable during the execution of any valid product behaviour. From the definitions here above, we derive the following properties :

Property 1 (Minimal Test Suite). An executable abstract test suite ats over a given FTS $fts = (S, Act, trans, i, d, \gamma)$ is minimal w.r.t. a coverage criteria cov iff $\nexists ats'$ such that ats' is executable and $\#ats' < \#ats$ and $cov(ats', fts) \geq cov(ats, fts)$. In other words, an executable abstract test suite is minimal if there exists no smaller executable abstract test suite with a better coverage.

Property 2 (P-Minimal Test Suite). An executable abstract test suite ats over a given FTS $fts = (S, Act, trans, i, d, \gamma)$ is product-minimal (p-minimal) regarding a coverage criteria cov iff $\nexists ats'$ such as ats' is executable and $(cov(ats', fts) \geq cov(ats, fts)) \wedge (\#pMinProd(ats', fts) < \#pMinProd(ats, fts))$. A p-minimal executable abstract test suite for a given coverage criteria over an FTS represents the minimal set of executable abstract test cases (with the best coverage) such as the number of products needed to execute all of them is minimal.

For instance, the abstract test suite {(pay, $change$, $soda$, $serveSoda$, $open$, $take$, $close$); ($free$, tea, $serveTea$, $take$); ($free$, $cancel$, $return$)} is minimal for the all-states-coverage criteria but not p-minimal since it needs at least two different products (i.e., *free* and *not free* machines) to be executed on the FTS in Fig. 1. A p-minimal abstract test suite satisfying the all-paths coverage could be: {(pay, $change$, $soda$, $serveSoda$, $open$, $take$, $close$); (pay, $change$, tea, $serveTea$, $open$, $take$, $close$); (pay, $change$, $cancel$, $return$)}. Which only needs one product to execute the abstract test suite.

When designing an abstract test suite using a coverage criteria, the most interesting product to select in order to execute this abstract test suite is the one who will achieve the best coverage using this abstract test suite. We define here the p-coverage as the coverage reached by the execution of an executable abstract test suite for a given product and p-coverage upper bound as the product which will be able to execute the subset of an abstract test suite with the best coverage.

Definition 13 (P-Coverage). *Let ats be an abstract test suite over fts, a given FTS, and a covering criteria $cov(ats, fts)$. Given a product $p \in prod(ats, fts)$ and $ats_p \subseteq ats$ the set of all abstract test cases of ats executable by p. The p-coverage is the coverage reached when executing ats_p :*

$$p - coverage = cov(fts, ats_p).$$

Equipped with this notion of product coverage by a subset of the test suites, we may look for the product(s) that optimize(s) a given coverage function.

Definition 14 (P-Coverage Upper Bound). *Given an executable abstract test suite ats over a given FTS $fts = (S, Act, trans, i, d, \gamma)$ and a covering criteria $cov(ats, fts)$. Given a product $p \in prod(ats, fts)$ and $ats_p \subseteq ats$*

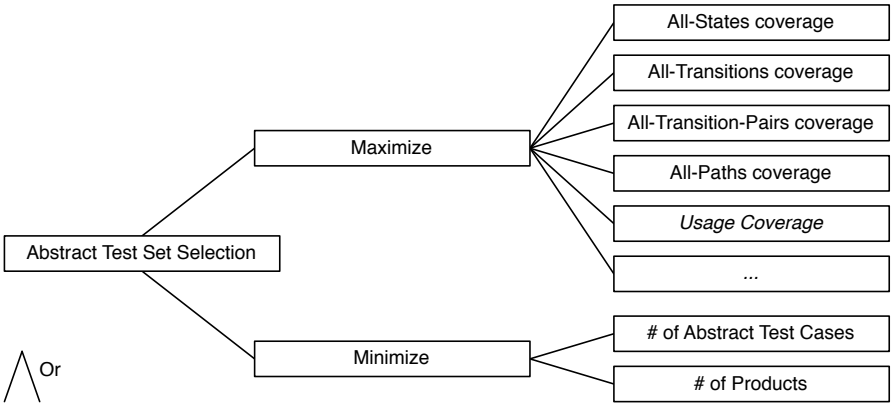

Fig. 5. Abstract Test Suite Selection problem

the abstract test suite executable by p. The product p will be the p-cov upper bound iff $\nexists ats'_p \subset ats$ *executable by* $p' \in prod(ats, fts)$ *such as* $cov(ats'_p, fts) > cov(ats_p, fts)$.

For instance, the p-all-transition-pairs upper bounds products are :

- $\{VendingMachine, CancelPurchase, Beverage, Soda, Tea, Currency, Euro\}$
- $\{VendingMachine, CancelPurchase, Beverage, Soda, Tea, Currency, Dollar\}$

Each one with a all-transition-pairs coverage of 68.75%. It means that concretizing the abstract test suite derived to achieve a all-transition-pairs coverage using one of those products and executing the concretized test cases on the selected product will achieve a all-transition-pairs coverage of 68.75% for the *behaviour of the SPL*.

3.4 SPL Test Case Selection

As illustrated in Fig. 5, the abstract test case selection problem may be formulated as an optimisation problem. In its most simple expression, the considered selection criteria has to be maximised and either the size of the executable abstract test suite (minimal test suite) or the number of product needed to execute the test suite (p-minimal test suite) has to be minimized. Of course, in reality we expect more complex situations where a finer grained objective function will be designed. For instance by adding weights to features in the FD [16] and/or transitions [1] in the FTS and try to minimize the total cost of the test suite.

Adding weight to transitions is a classical approach developed in statistical testing where weight (between 0 and 1) represents the probability of a transition

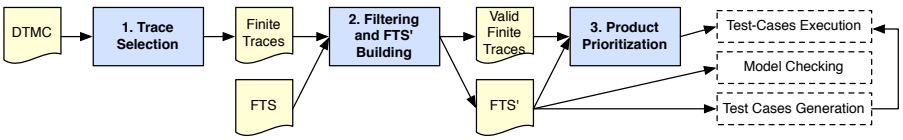

Fig. 6. Test-Case Prioritization process [12]

to be fired. In our previous work [12], we apply this idea to product lines in order to prioritize behaviours and products to test. In the following section, we define one more additional coverage criteria: the *Usage Coverage*, specifying that the selected abstract test suite should cover the most or least common behaviours of the SPL.

3.5 Usage Coverage Criteria

In [12], FTSs are combined to a Deterministic Timed Markov Chain (DTMC) in order to select and prioritize SPL behaviours to assess. The result of the process is a FTS' representing a subset of the original FTS that has to be assessed (using testing and/or model checking) in priority. The complete process is presented in figure 6. First, traces are selected in the DTMC according to their probability to happen (step 1), e.g., probability between a lower and upper bound, most probable traces, etc. A trace represents a sequence of actions in the DTMC. Since the DTMC does not have any notion of features (which allows us to use existing statistical testing tools like MeTeLo [1]), the selected traces have to be filtered using the FTS to ensure they can be performed by at least one valid product (step 2). By pruning the FTS and keeping only transitions fired when executing selected traces, we get a FTS', representing a (priority) subset of the original FTS. Optionally, in step 3, valid traces, and FTS' are used to generate valid configurations (i.e., products) which have to be tested in priority. We assessed the feasibility of this process on an existing system (see section 4 of [12]), the local Claroline instance at the University of Namur (an online course management system) using a 5Go Apache access log to build the DTMC, a FTS with 107 states and 11236 transitions and a feature digram with 44 features. In this first version, we run a depth first search algorithm 4 times in order to get behaviours with probability between $[10^{-4}; 1]$, $[10^{-5}; 1]$, $[10^{-6}; 1]$ and $[10^{-7}; 1]$. The average probabilities of the traces selected in the DTMC and the size of the generated FTS' are presented in Tab. 1.

We intend to combine the usage coverage criteria with other structural coverage criteria in order to asses the behaviour of a SPL. The classical scenario we imagine would be:

1. The test engineer select the lowest or highest probable usage of the system based on a DTMC and build a FTS' (using [12]), prioritized subset of the

Table 1. Claroline Feasability Assessment Results [12]

Proba. interval	Traces avg. proba.	σ	# FTS' states	# FTS' transitions
$[10^{-4}; 1]$	$2,06E^{-3}$	$1,39E^{-2}$	16	66
$[10^{-5}; 1]$	$3,36E^{-4}$	$5,46E^{-3}$	36	224
$[10^{-6}; 1]$	$5,26E^{-5}$	$2,12E^{-3}$	50	442
$[10^{-7}; 1]$	$8,10E^{-6}$	$8,18E^{-4}$	69	844

original FTS. We did not make any assumption on how the usage model is build. In the experiment presented in [12], the usage model was obtain from actual usage of the system using an Apache log. It could also be done manually by a system expert who will tag the transitions in the DTMC with probabilities based on its own knowledge of the system [27].

2. The test engineer select a minimal or p-minimal (executable or not) abstract test suite in the FTS' using a structural coverage criteria.
3. The test engineer concretize this abstract test suite and execute it on one product of the SPL.

4 Related Work

Other strategies to perform SPLs testing have been proposed. One of those considers incremental testing in the SPL context [30, 24, 21]. For example, Lochau et al. [21] proposed a model-based approach that shifts from one product to another by applying"deltas" to statemachine models. These deltas enable automatic reuse/adaptation of test model and derivation of retest obligations. Oster et al. [24] extend combinatorial interaction testing with the possibility to specify a predefined set of products in the configuration suite to be tested. There are also approaches focused on the SPL code by building variability-aware interpreters for various languages [19]. Based on symbolic execution techniques such interpreters are able to run a very large set of products with respect to one given test case [23]. In [4], Cichos et al. use the notion of 150% test model (i.e., a test model of the behaviour of a product line) and test goal to derive test cases for a product line but do not redefine coverage criteria at the SPL level. In [3], Beohar et al. propose to adapt the *ioco* framework proposed by Tretmans [28] to FTSs. Contrary to this approach, we do not seek exhaustive testing of an implementation but rather to select relevant abstract test cases based on the criteria provided by the test engineer.

5 Conclusion & Perspectives

In this paper, we have established the preliminary foundations to support SPL testing using FTS by defining dedicated testing concepts and providing several coverage criteria to support test generation. Next steps naturally include the design of strategies that realize the extraction of behaviours based on such criteria. Experience and optimisation realised for model-checking algorithms will

be key to the design of efficient and scalable FTS traversals. We also plan to devise an FTS-aware random test generation strategy (e.g. systematically producing random executable abstract test cases). Finally, we also plan to combine such criteria with each other and with test case selection based on temporal properties. Our approach will be integrated with the ProVeLines family of SPL model-checkers [10].

References

1. ALL4TEC: MaTeLo, `http://www.all4tec.net/index.php/en/model-based-testing` (last visit 31/01/2014)
2. Astesana, J.M., Cosserat, L., Fargier, H.: Constraint-based vehicle configuration: A case study. In: Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on. vol. 1, pp. 68–75 (Oct 2010)
3. Beohar, H., Mousavi, M.R.: Spinal Test Suites for Software Product Lines. ArXiv e-prints (2014)
4. Cichos, H., Oster, S., Lochau, M., Schürr, A.: Model-based Coverage-driven Test Suite Generation for Software Product Lines. pp. 425–439. MODELS '11, Springer (2011)
5. Classen, A.: Modelling and Model Checking Variability-Intensive Systems. Ph.D. thesis, PReCISE Research Center, Faculty of Computer Science, University of Namur (FUNDP) (2011)
6. 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. TSE PP(99), 1–22 (2013)
7. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: ICSE'10. pp. 335–344. ACM (2010)
8. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison Wesley, Reading, MA, USA (2001)
9. Cohen, M., Dwyer, M., Shi, J.: Interaction testing of highly-configurable systems in the presence of constraints. In: ISSTA '07. pp. 129–139 (2007)
10. Cordy, M., Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Provelines: A product-line of verifiers for software product lines. In: SPLC '13 Workshops. pp. 141–146. ACM (2013)
11. Devroey, X., Cordy, M., Perrouin, G., Kang, E.Y., Schobbens, P.Y., Heymans, P., Legay, A., Baudry, B.: A vision for behavioural model-driven validation of software product lines. In: ISoLA '12. LNCS, vol. 7609, pp. 208–222. Springer (2012)
12. Devroey, X., Perrouin, G., Cordy, M., Schobbens, P.Y., Legay, A., Heymans, P.: Towards statistical prioritization for software product lines testing. pp. 10:1–10:7. VaMoS '14, ACM (2013)
13. Fantechi, A., Gnesi, S.: Formal modeling for product families engineering. In: SPLC '08. pp. 193–202 (2008)
14. Flores, R., Krueger, C., Clements, P.: Mega-scale product line engineering at general motors. pp. 259–268. SPLC '12, ACM (2012)
15. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: FMOODS '08. pp. 113–131. Springer (2008)
16. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Traon, Y.L.: Multi-objective test generation for software product lines. pp. 62–71. SPLC '13, ACM (2013)

17. Johansen, M.F., Haugen, Ø., Fleurey, F., Eldegard, A.G., Syversen, T.: Generating better partial covering arrays by modeling weights on sub-product lines. In: MoDELS '12. pp. 269–284 (2012)
18. 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)
19. Kästner, C., von Rhein, A., Erdweg, S., Pusch, J., Apel, S., Rendel, T., Ostermann, K.: Toward variability-aware testing. pp. 1–8. FOSD '12, ACM (2012)
20. Kim, C.H.P., Khurshid, S., Batory, D.S.: Shared execution for efficiently testing product lines. In: ISSRE '12. pp. 221–230 (2012)
21. Lochau, M., Schaefer, I., Kamischke, J., Lity, S.: Incremental model-based testing of delta-oriented software product lines. In: Brucker, A., Julliand, J. (eds.) Tests and Proofs. LNCS, vol. 7305, pp. 67–82. Springer (2012)
22. Mathur, A.P.: Foundations of software testing. Pearson Education (2008)
23. Nguyen, H.V., Kästner, C., Nguyen, T.N.: Exploring variability-aware execution for testing plugin-based web applications. ICSE '14, IEEE (2014)
24. Oster, S., Markert, F., Ritter, P.: Automated incremental pairwise testing of software product lines. In: Software Product Lines: Going Beyond. pp. 196–210. Springer (2010)
25. Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., Traon, Y.L.: Pairwise testing for software product lines: comparison of two approaches. Software Quality Journal 20(3-4), 605–643 (2012)
26. von Rhein, A., Apel, S., Kästner, C., Thüm, T., Schaefer, I.: The pla model: on the combination of product-line analyses. p. 14. VaMoS '13, ACM (2013)
27. Samih, H.: Relating Variability Modeling and Model-Based Testing for Software Product Lines Testing. In: Weise, C., Nielsen, B. (eds.) Proceedings of the ICTSS 2012 Ph.D. Workshop. pp. 18–22. Aalborg University, Department of Computer Science, Aalborg, Denmark (2012)
28. Tretmans, J.: Model based testing with labelled transition systems. Formal methods and testing pp. 1–38 (2008)
29. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann (2007)
30. Uzuncaova, E., Khurshid, S., Batory, D.: Incremental test generation for software product lines. Software Engineering, IEEE Transactions on 36(3), 309–322 (2010)
31. Whittaker, J.A., Thomason, M.G.: A markov chain model for statistical software testing. TSE 20(10), 812–824 (1994)