A Decade of Featured Transition Systems: From Variability-intensive Model Checking to Safe Artificial Intelligence *

Maxime Cordy[1][0000−0001−8312−1358], Xavier Devroey[2][0000−0002−0831−7606], Axel Legay[3], Gilles Perrouin[4][0000−0002−8431−0377], Andreas Classen[5], Patrick Heymans[4], Pierre-Yves Schobbens[4][0000−0001−8677−4485], and Jean-François Raskin[6]

[1] SnT, University of Luxembourg, Luxembourg, Luxembourg. maxime.cordy@uni.lu
[2] Delft University of Technology, Delft, The Netherlands. x.d.m.devroey@tudelft.nl
[3] UCLouvain, Belgium axel.legay@uclouvain.be
[4] PReCISE/NaDI, Faculty of Computer Science, University of Namur, Namur, Belgium. gilles.perrouin@unamur.be
[5] INTEC Software, Belgium. Andreas.Classen@intecsoft.com
[6] ULB, Brussels, Belgium. jraskin@ulb.ac.be

Abstract. Variability-intensive systems (VIS) form a large and heterogeneous class of systems which behaviour can be modified by enabling or disabling predefined features. Variability mechanisms allows the adaptation of software to the needs of their users and the environment. However, VIS verification and validation (V&V) is challenging: the combinatorial explosion of the number of possible behaviours and undesired feature interactions are amongst such challenges. To tackle them, Featured Transitions Systems (FTS) were proposed a decade ago to model and verify the behaviours of VIS. In a FTS, each transition is annotated with a combination of features determining which variants can execute it. An FTS can model all possible behaviours of a given VIS. This compact model enabled us to create efficient V&V algorithms taking advantage of the behaviours shared amongst features resulting in a reduction of the V&V effort by several orders of magnitude. In this chapter, we will cover the formalism, its applications and the role it can play in supporting the quality assurance of the next generation of VIS powered by artificial intelligence techniques.

Keywords: variability modeling · model checking · testing · safe artificial intelligence

---

* Gilles Perrouin is a research associate at the FNRS.'This research was partially funded by the EU Project STAMP ICT-16-10 No.731529, the NIRICT 3TU.BSR (Big Software on the Run) project, the EOS project VeriLearn under FNRS Grant O05518F-RG03.

1 Introduction

Variability-intensive systems (VIS) form a vast and heterogeneous class of software systems that encompasses: *Software Product Lines* [2, 74], operating systems kernels, web development frameworks/stacks, e-commerce configurators, code generators, *Systems of Systems (SoS)*, *software ecosystems* (*e.g.,* Android's "Play Store"), autonomous systems, *etc.* While being very different in their goals and implementations, VIS see their behaviour affected by the activation or deactivation of one or more *feature(s)*, i.e. units of variability, configuration *options*. Configurable systems may involve thousands of features with complex dependencies. In order to represent the set of valid combinations of options a configurable system can have – its *variants* (which we also name *configurations*) – we can use a tree-like structure, called *feature model* [54]. Each feature may be decomposed into sub-features and additional constraints may be specified amongst the different features. Within feature models, features can be mandatory (present in every configuration) or selected depending on the groups they belong to (OR, XOR, etc.) and cross-tree constraints (dependence on or exclusion of other feature selections). To support automated reasoning, feature models have been equipped with formal semantics and in particular based on first-order logic [78]. Thanks to their formal semantics, operations on feature models such as inconsistency reasoning can be automated thanks to SAT solvers [67].

Considering that some VIS such as the Linux kernel can easily have more than 10,000 features and that the number of possible variants grows exponentially with the number of features, considering each variant independently for verification and validation (V&V) activities is intractable. As an example, Halin *et al.* [48] report their effort to perform a complete product-based testing of JHipster, an open-source generator for Web applications with 48 features. It took 8 person/month to set up the testing infrastructure, 5.2 Terabytes of disk space, and 4,376 hours (around 182 days) computation time to test all 26,256 products. It is therefore desirable to analyse VIS behaviours without requiring to build and run tests for each variant by reasoning on a behavioural model rather than on the system itself (which may not be implemented yet). However, while providing a transition system for each variant and performing model-checking [5] or model-based testing (MBT) activities allows to find bugs early in the process, this does not solve *per se* the combinatorial explosion problem due to variability, as providing a transition system for each variant is also intractable. A family-based approach is required to model all the variants in a compact manner without having to enumerate them. Almost a decade ago, Classen *et al.* [24] defined *Featured Transition Systems* (FTS) as transition systems annotated with combination of features on their transitions: each combination describe the set of products that can execute the behaviour defined by the transition. It is thus possible to model all the variants of a VIS with a unique FTS and its associated feature model. This compact formalism allows to take advantage of sharing between variants leading to drastic reduction of the analysis time, should it be for formal verifi-

cation or test generation.

This chapter reviews the foundations and application of featured transition systems, connecting them to other such as modal transition systems and considering extensions such as quantities and probabilities which are required to address V&V challenges induced by cyber-physical and learning systems. The rest of this chapter is structured as follows. Section 2 introduces the formalism and describes how the modified model-checking problem VIS can be solved efficiently. Section 3 reviews other model-checking techniques for VIS, placing FTS-based verification in a broader context. Section 4 explores how the FTS formalism was used to provide a framework for model-based testing for VIS, notably extending extending exsiting transition systems coverage criteria. Section 5 provides an outlook into the future of VIS V&V. Finally Section 6 concludes the chapter.

2 Verifying Variability-intensive Systems with FTS

The model-checking problem for VIS is more complex than for single systems because it requires to verify all the variants that can be built against a given property. More precisely, it is desired to identify exactly which VIS variants violate the property [21].

Definition 1 (VIS model checking). *Let fm be a feature model, M_v be a behavioural model of all variants in $[\![fm]\!]$, and ϕ a property. Model checking M_v against ϕ is the problem of:*

1. *Determining for each product $p \in [\![fm]\!]$ whether p satisfies ϕ in M_v, that is, whether the behaviour of p expressed in M_v satisfies ϕ.*
2. *Providing for each product p that does not satisfy ϕ a counterexample of behaviour of p that violates ϕ.*

A simple method to address this problem consists of modelling every valid product of an VIS in a separate TS, and then applying single-system model checking on each of these TS individually. This method, named *enumerative* [23] or product-based [75], immediately appears against the principles of VIS engineering: the variants should not be modelled separately. Instead, one should build a core model, which is subsequently derived into desired variants. In addition to the modelling task, performance is also a major concern. State explosion, a problem inherent to model checking, is amplified when considering VISs, especially when these consist of a huge number of variants. Being part of an VIS, these variants likely share commonalities in both their structure and their behaviour. This observation illustrates the fact that single-system verification techniques are suboptimal to address the VIS model-checking problem. Clearly, VIS model checking would benefit from models that can concisely represent the behaviour of a set of variants, and algorithms that can exploit the information about the commonalities between these variants to facilitate verification.

2.1 A Formalism to model VIS Behaviour

In [21], we proposed Featured Transition Systems (FTS) as a compact representation for a set of variants. These are an extension of TS equipped with an FM and whose every transition is annotated with the exact set of variants able to execute it. For the sake of conciseness, these sets are encoded as feature expressions.

Definition 2 (Feature expression). *Let $F = \{f_1, \ldots, f_{|F|}\}$ be a set of features. Then a feature expression over F is a Boolean formula $b \in 2^{2^F}$ whose each variable corresponds to a unique element of F, and whose semantics is a function $2^F \to \{\bot, \top\}$ encoding a set of products. A product $p \in 2^F$ is included in the set represented by b, noted $[\![b]\!]$, if and only if $b(p) = \top$, or equivalently: $\bigwedge_{f \in p} f \bigwedge_{g \in F \setminus p} \neg g \models b$. In this case, p is said to* satisfy *b, noted $p \models b$. We also denote by $\mathbb{B}(px)$ the feature expression encoding the set of products px, that is, $\mathbb{B}(px) = \bigvee_{p \in px} \big(\bigwedge_{f \in p} f \bigwedge_{g \in F \setminus p} \neg g \big)$.*

Definition 3 (Featured transition systems). *An FTS is a tuple (S, Act, $Trans$, I, AP, L, fm, γ), where*

- *S is a set of states named the state space;*
- *Act is a set of actions;*
- *$Trans \subseteq S \times Act \times S$ is the transition relation, where $(s, \alpha, s') \in Trans$ (also noted $s \xrightarrow{\alpha} s'$) means that there is a transition from state s to state s' labelled with action α;*
- *$I \subseteq S$ is a set of initial states;*
- *AP is a set of atomic propositions;*
- *$L : S \to 2^{AP}$ is a function that associates every state with the set of atomic propositions satisfied by this state.*
- *fm is a FM over a set of features F;*
- *$\gamma : Trans \to 2^{(2^F)}$ is a total function that associates a transition with a feature expression over F.*

An FTS can be seen as the merging of the TS of all the variants that compose the VIS. The TS modelling a specific product is obtained from the FTS by applying a *projection* function. In simple terms, this function prunes the FTS of all the transitions whose feature expression is not satisfied by the considered product [24], and then removes all feature expressions.

Definition 4 (Projection of FTS). *Let $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ be an FTS and $p \in [\![fm]\!]$ be a variant. The projection of fts onto p, noted $fts_{|p}$, is the TS (S, Act, $Trans'$, I, AP, L) where $Trans' = \{t \in Trans \mid p \models \gamma(t)\}$.*

Example 1. Figure 1 depicts an FTS modelling an VIS of vending machines, while Figure 2 shows its associated FM. This VIS consists of 12 variants, each of which has its behaviour modelled by the FTS. For instance, the transition from state 3 to state 6 is labelled with the feature expression t, meaning that it can be

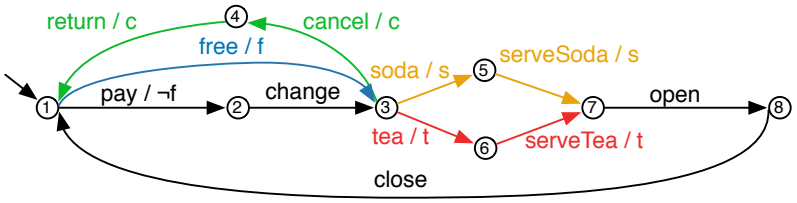

Fig. 1. The FTS modelling the vending machine VIS.

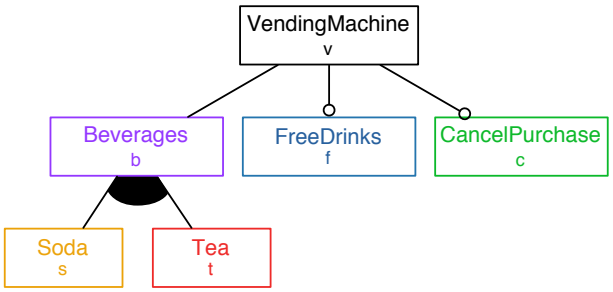

Fig. 2. The FM of the vending machine VIS.

executed only by variants including the corresponding feature Tea. Transition from state 1 to state 2 is labelled with $\neg f$, and thus can be executed only by variants that do not have the feature $Free$.

Since an FTS represents the behaviour of a *set* of variants, its semantics is defined as a function that associates a variant with the traces of the corresponding projection.

Definition 5 (FTS Semantics). *Let fts be an FTS over an FM fm. The semantics of fts is a total function $[\![fts]\!] : [\![fm]\!] \to 2^{(2^{AP})^\omega}$ such that $\forall p \in [\![fm]\!] \bullet [\![fts]\!](p) = [\![fts_{\downarrow p}]\!]$.*

2.2 FTS Model Checking

Contrary to single systems, a binary result is not sufficient to appropriately address the model-checking problem for VIS. In case of property violation, a model checker is expected to identify all variants responsible for the violation. There is thus a need for generalizing the definition of model checking. We already gave it an intuitive definition at the beginning of this chapter. Here, we rephrase this definition formally, by considering FTS as a model for VIS behaviour.

Beforehand, let us remark that a property may only be relevant for certain variants. For instance, a property may refer to characteristics that only occur

in a subset of the VIS. To address this requirement, we proposed to extend temporal logic with a *product quantifier*, i.e. a feature expression that defines for which variants the property must be checked [21]. The resulting variant of LTL is defined as follows.

Definition 6 (fLTL). *Let F be a set of features. An fLTL formula ψ is an expression $\psi = [\chi]\phi$ where χ is a feature expression over F and ϕ an LTL formula. Let fts be an FTS over an FM fm over F and let $p \in [\![fm]\!]$. Then p satisfies ψ in fts if and only if $\chi(p) \Rightarrow fts_{|p} \models \phi$.*

We are now ready to generalise the concept of satisfiability.

Definition 7 (F-satisfiability). *Let fts be a FTS over an FM fm, and $\psi = [\chi]\phi$ be an fLTL formula. Then, the variants that F-satisfy ψ in fts are encoded as the feature expression*

$$(fts \models \psi) = \neg\chi \vee \mathbb{B}(\{p \in [\![fm]\!] \mid fts_{|p} \models \phi\}).$$

Conversely, the variants that F-unsatisfy ψ in fts are encoded as the feature expression

$$(fts \not\models \psi) = \chi \wedge \mathbb{B}(\{p \in [\![fm]\!] \mid fts_{|p} \not\models \phi\}).$$

Given an FTS and an fLTL formula, an VIS model checker should thus compute the corresponding F-satisfiability, and associate each F-unsatisfying product with one of its traces that violates the formula.

2.3 Algorithms

Given its explicit notion of features, FTS constitutes a suitable formalism to concisely model behaviour subject to variability. Yet there remains a second challenge to solve, i.e. an efficient verification of the behaviour of a set of variants. To achieve that, we designed algorithms to check FTS against LTL [21] and CTL [23] formulae that exploit common transitions among variants to reduce the verification effort. As opposed to the product-based approach, a given behaviour is not always checked as many times as the number of variants in which it occurs.

Regardless of the logic used to express properties, the verification process can be reduced to the computation of reachability relations. A major difference is that the reachability of a state now depends on variability: the variant that can reach an arrival state a from an initial state i are those that can execute any sequence of transitions starting from i and ending in a. Fundamentally, the difference with single-system model checking is the definition of successor state. In TS, state s' is a successor of a given state s if and only if there exists a transition from s to s'. In FTS (resp. FBA), variability can influence the set of successors as a transition may exist for only a subset of the variants. The definition of successor has thus to be revisited according to variants as well. It is given as follows.

Definition 8 (Successors in FTS). *Let* $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ *be an FTS. The successor function in* fts *is defined as* $Post : S \to (S \to 2^{(2^F)})$ *such that:*

$$\begin{aligned} Post(s_i)(s_{i+1}) &= \mathbb{B}(\{p \in \cup_\alpha [\![\gamma(s_i, \alpha, s_{i+1})]\!]\}) \\ &= \bigvee_\alpha \gamma(s_i, \alpha, s_{i+1}) \end{aligned}$$

with $(s_i, \alpha, s_{i+1}) \in Trans$.

Intuitively, for a given couple of states (s, s') the function $Post(s)(s')$ is the feature expression encoding the variants that can execute a transition from s to s'.

From the definition of successor, one can define reachability relation in FTS. Similarly to successor, reachability takes the form of a function. It associates two states, say s_0 and s_n, to a feature expression encoding the variants able to reach s_n from s_0. These variants are those able to follow at least one path from s_0 to s_n. Let $s_0, \ldots, s_n$ be a path in a given FTS. a variant can follow this path if and only if it satisfies the feature expression $\bigwedge_{0 \leq i < n} Post(s_i)(s_{i+1})$. To obtain the variants that can reach s_n from s_0, we can existentially quantify the above expression over all the paths from s_0 to s_n.

Definition 9 (Reachability in FTS). *Let* $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ *be an FTS. Reachability in* fts *is a function* $R : S \to (S \to 2^{(2^F)})$ *such that:*

$$\begin{aligned} R(s_0)(s_n) &= \mathbb{B}(\{p \in 2^F \mid \exists s_1, \ldots, s_{n-1} \bullet p \in [\![\bigwedge_{i=0}^{n-1} Post(s_i)(s_{i+1})]\!]\}) \\ &= \mathbb{B}(\{p \in [\![\bigvee_{s_1, \ldots, s_{n-1} \in S^{n-1}} \bigwedge_{i=0}^{n-1} Post(s_i)(s_{i+1})]\!]\}) \\ &= \bigvee_{s_1, \ldots, s_{n-1}} \bigwedge_{i=0}^{n} Post(s_i)(s_{i+1}) \end{aligned}$$

where $\forall j \bullet 0 \leq j < n \bullet \exists \alpha \in Act \bullet (s_j, \alpha, s_{j+1}) \in Trans$.

To efficiently compute the reachability function in an FTS, we designed a depth-first search algorithm that accumulates the conjunction of the feature expressions of all transitions executed on a given path in order to keep track of the variants able to reach any state met along this path. This means that the algorithm separates the verification of different sets of variants only if they discover a behavioural discrepancy between them. This optimisation is called *late splitting* [3].

Algorithm 1 formalises this process to compute the reachability function of a given state s_0. The algorithm consists of a loop that iterates over a stack of couples (s, γ) where s is a state and γ is a feature expression. Initially, the stack contains only the element (s_0, fm) in order to start the search from s_0

Input: $fts = (S, Act, Trans, I, AP, L, fm, \gamma), s_0 \in S$.
Output: $R(s_0)$.

```
1  R ← ⊥;
2  Stack ← push((s_0, fm), []);
3  while Stack ≠ [] do
4  |  (s, γ) ← pop(Stack);
5  |  succ ← {(s', γ') | s' ∈ dom(Post(s)) ∧ γ' = Post(s)(s') ∧ γ};
6  |  foreach (s', γ') ∈ succ • γ' ⊭ ⊥ do
7  |  |  if s' ∈ dom(R) then
8  |  |  |  γ_new ← ¬R(s') ∧ γ';
9  |  |  |  if γ_new ⊭ ⊥ then
10 |  |  |  |  R(s') ← R(s') ∨ γ';
11 |  |  |  |  push((s', γ_new), Stack);
12 |  |  |  end
13 |  |  end
14 |  |  else
15 |  |  |  R(s') ← γ';
16 |  |  |  push((s', γ'), Stack);
17 |  |  end
18 |  end
19 end
20 return R
```

Algorithm 1: Reachables(fts, s_0)

while considering all the variants. At each iteration, the algorithm takes the top element (s, γ) of the stack, computes the successors of s and associate each successor with the variants that satisfy γ and can reach the successor from s (Lines 4–5). This results in a set of couples $(s', \gamma') \in S \times 2^{(2^F)}$. For each such couple, the algorithm first determines whether $[\![\gamma']\!]$ contains at least one valid product; otherwise it is not needed to pursue the search from s'. This verification is achieved by checking the satisfiability of γ' (Line 6). If that is the case, we enter an inner loop (Lines 7–17).

During the search, the algorithm may visit a given state more than once (Lines 7–13). In single-system model checking, it should not pursue the search since it already knows that the revisited state is reachable. In our case, however, it may happen that the algorithm discovers a new path to an already visited state s' which is executable by variants that were not known to be able to reach s'. Formally, let $R(s')$ be the feature expression encoding the set of variants that were known to reach s'. Then $\neg R(s') \wedge \gamma'$ encodes the set of variants that are newly known to reach s (noted γ_{new} at Line 8). If there is at least one valid product satisfying this feature expression, the search continues from s' considering only the variants in γ_{new} (Lines 9–12). Indeed, any state reachable from s for variants $[\![R(s')]\!]$ may have already been visited for these variants. Therefore, the paths starting from s are worth re-exploring only for the variants

in γ_{new}. Before pursuing the exploration, the feature expression $R(s')$ is updated accordingly.

The theoretical complexity of the above algorithm is given as follows.

Theorem 1. *[21] Let fts be an FTS over a set of features F. The worst-case time complexity of computing Algorithm 1 is bounded by $\mathcal{O}(|fts|.2^{2.|F|})$.*

Intuitively, in the worst-case each valid product has a different behaviour starting from the initial state. In this case, Algorithm 1 behaves as the product-based approach. Moreover, the number of valid variants is in the worst-case the size of the power set of F, i.e. $2^{|F|}$. Furthermore, there is an overhead in the FTS algorithm that does not exist in the product-by-product method: At each iteration, a satisfiability check on feature expression is performed, which also has a time complexity of $\mathcal{O}(2^{|F|})$. Although the FTS algorithm has a worse theoretical complexity, experiments tend to show that in practice it outperforms the product-based approach [19, 21, 23]. The FTS theory is thus a solid candidate solution for the VIS model-checking problem.

3 Other Verification FTS-based work

Our FTS formalism has been extended in various directions. The first of them was to consider other types of logic in order to specify product line requirements. As an example in [20], we have showed how to extend symbolic model checking of computational tree logic to FTS. We mostly showed how to encode features as extra variables in BDDs representing symbolic behaviors of multiple products without blowing up the representation. Latter, in [7], Ter Beek et al, have showed how to consider the entire mu-calculus. Their main contribution was to introduce μL_f, a logic that combines mu-calculus modalities with feature expressions. They showed how to define and model check this logic on FTS. Their work has been implemented in a tool called mCRL2.

In parallel, we have also extended our approach to conformance model checking (also known as refinement-based model checking), that is the problem of comparing the behaviors of several products. Simulation relation allows us to decide whether all behaviors of a system are covered by those of another system. In product lines, the problem reduces to check if all products from one line are covered by products from another line. One way to do so is to perform a pairwise comparison btween the products of the two lines, which is expensive. In order to avoid this enumerative comparison, we have showed how to generalize the notion of simulation from systems to family. The work, which is presented in [27], shows clear benefit in using this approach. Latter, in collaboration with University of Waterloo Canada, we have showed that this new relations can be used to quantify the impact of change when introducing or removing features from a given system. This was one of the first extensions of FTS has been used to handle problems that are not related to product lines. Indeed, here features are used to label behaviors of a system, not to distinguish products in a specific line. Results related to this topic are available in [4].

Abstraction is a technique that permits to reduce the size of a system by merging states or transitions. The resulting system is generally smaller and easier to verify. Abstraction is behaviorally conservative, but may introduce extra fake behaviors. In [28], we have showed how to abstract states and transitions of FTS. The situation is more complex than for single systems. Indeed, we need not only to merge states, but also to simplify formulas representing set of features over FTS's transitions. In order to remove fake behaviors (when needed), we have entirely redeveloped a CEGAR-based model checking for FTS. Another CEGAR procedure was developed by Wasowski for LTL and latter CTL [39–41]. Contrary to us they only focus on abstracting features, but not states. Their approach uses games and modal automata as FTS abstractions, hence showing that FTS is practically more convenient that modal automata to represent complex behavioral relations between products.

Another trend has been the one of extending FTS with quantitative information. The first attempt was when we showed how to combined FTS and timed automata in order to handled timed product lines. In [29], we have showed how to combined timed constraints of real-time clocks with feature constraints. We have then showed that the model checking procedure from [22] adapt directly to our case by using the well-known region construction from timed automata. Our timed extension has been reused an extended in various directions. As an example, in [10], Beohar and Mousavi introduced IOFTS that is an input/output extension of timed FTS for model-baed testing of software product lines. Their main contributions were to define a notion of test suite and test cases generated from an IOFTS. They also defined two notions of refinement, one at the level of IOFTS and another one at the level of test suites.

Later, probabilistic extensions of FTS were also considered. In [76], we have combined FTS with stochastic information coming from a Markov Decision Process representation of the environment. In this context, one has to compute which product satisfies a given requirement with a specified probability. We have defined family-based algorithms to analyze the resulting quantitative FTS. One of them directly extends the classical algorithm for bounded quantitative logic. The other one uses parameter synthesis in stochastic systems to extract products that do satisfy a quantitative behavior. In [35], we have showed how learning algorithms and Markov Decision processes can be used to abstract environment behaviors. We then showed how the result can be used to restrict FTS behaviors in a model-testing based approach. Our work pave also the way for compositional reasoning and analysis of probabilistic queries for software product lines. In [43], Baier et al give a clear adaptation of compositional reasoning to this context. This is implemented in the ProFeat tool [17] that uses similar techniques to those in [76]. It is worth mentioning that other research groups are also working of verifying stochastic and even quantitative behaviors of product lines. As an example, in [6], Ter Beek et al. have proposed an algebra to defined quantitative relation between features. This algebra is static in the sens that it relates features with quantitative information (cost, constraints on costs, etc) and dynamic in the sens that it allows us to specify when features can appear and disappear

in system's execution at runtime(hence opening the door to analysis of dynamic software product lines). The verification process used in this works relies on a dynamic extension of the so-called Statistical Model Checking [59].

In a series of recent works [70], we have also tried to extend FTS to handle quantitative problems such as long run average. Quantitative problems were already handled at feature diagram level, but not yet at behavioral level. Unfortunately, the family-based approach advantages decreases in the context of those challenges. Indeed, those weighted automata-based problems require to compute specific quantities that differ from products to product. A solution to this problem could be to use abstraction-based approaches over quantities.

4 Testing Variability-Intensive Systems with FTSs

In this section, we focus on *Model-Based Testing* (MBT) [82]. MBT requires to define a model of the expected behaviour of the System Under Test (SUT), *i.e.,* a specification, that serves as input to an automated test suite selection tool. The model should be small enough to be cheaper than the analysis of the actual system, but accurate enough to describe the characteristics to test. The tool uses this model to generate a sequence of input (*i.e.,* a *test case*) and an oracle for each one of those sequences. For most systems, selecting all the possible test cases from the model is intractable. The test engineer relies on selection algorithms that maximize a given *coverage criterion*, measuring the adequacy of a test suite [65].

4.1 Test Concepts for FTSs

Adapting the ideas of model-based testing in the context of software product lines was the focus of X. Devroey in the frame of his PhD thesis completed in 2017 [34]. Since FTSs are derived from transition systems, a natural starting point is to consider existing coverage criteria for transition systems [82] and extend them to make them meaningful with respect to FTSs.

Abstract test case over an FTS In an MBT approach, test cases are automatically selected from a model of the system under test. This derivation is done in several steps: first, *abstract test case* are selected from the model, an FTS in our case, using a given criterion; those abstract test cases are then refined, using additional information in order to be executable by the SUT. This section, and the remainder of this chapter cover the first step: abstract test case selection.

First, let us define the notion of abstract test case for FTS. We define an abstract test case over an FTS as a sequence of *actions* from this FTS, such that there exists a sequence of *transitions* in this FTS with the given actions.

Definition 10 (abstract test case). *Let fts* $= (S, Act, Trans, I, AP, L, fm, \gamma)$ *be an FTS. An abstract test case* t *is a finite sequence* $(\alpha_1, \ldots, \alpha_n)$*, where* $\alpha_1, \ldots, \alpha_n \in Act$ *and there exists a sequence of transitions in trans such that*

$$i \xrightarrow{\alpha_1} s_k \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_l$$

For instance, an abstract test case for the soda vending machine software product line (SPL) described in Figures 1 and 2 may be *(pay, change, soda, serveSoda, open, close)*. This abstract test case is a sequence of actions in the FTS representing a behaviour of this SPL.

Positive and negative abstract test cases. We distinguish two kinds of test cases: *positive test cases* trigger a desired/expected behaviour of the system under test; and *negative test cases* trigger an undesired behaviour of the system under test [82]. In our case, a positive abstract test case is defined as a sequence of actions executable by the *fts*, while a negative abstract test case is a sequence of actions not executable by the *fts*. Once concretized (*i.e.,* transformed into executable code) [84], negative abstract test cases typically represent sequences of actions that every product of the product line should forbid.

In a LTS (*lts*), an abstract test case $t = (\alpha_1, \ldots, \alpha_n)$ is executable, denoted $lts \stackrel{t}{\Longrightarrow}$, if there exists a sequence of transitions starting from the initial state and labelled with $\alpha_1, \ldots, \alpha_n$ [80, 81]. For an FTS (*fts*), to be executable, the sequence of transitions must moreover have compatible feature expressions. In other words, a sequence of actions is executable by *fts* if there exists at least one product (p) which, when *fts* is projected onto p (denoted $fts_{|p}$), is able to execute it:

$$\left(fts \stackrel{\alpha_1,\ldots,\alpha_n}{\Longrightarrow}\right) \Leftrightarrow \left(\exists p \in [\![fm]\!] : fts_{|p} \stackrel{\alpha_1,\ldots,\alpha_n}{\Longrightarrow}\right)$$

In testing, unlike model checking [5], we only consider finite sequences of actions. Since FTS (as LTS) do not have final/accepting state *per se*, in order to decide if a sequence of actions represents a desired behaviour of the system, we chose to consider the initial state of an FTS as a final state. Positive abstract test cases have to end their execution in the initial state (*e.g.,* state 1 in the soda vending machine FTS).

Definition 11 (positive abstract test case). *Let* $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ *be an FTS. A positive abstract test case* $t = (\alpha_1, \ldots, \alpha_n)$*, where* $\alpha_1, \ldots, \alpha_n \in Act$*, is a finite sequence of actions such as there is at least one product from* fm *able to execute* t*, and this execution ends in the initial state:*

$$\exists p \in [\![fm]\!] : fts_{|p} \stackrel{t}{\Rightarrow} i$$

Definition 12 (negative abstract test case). *Let* $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ *be an FTS. A negative abstract test case* $t = (\alpha_1, \ldots, \alpha_n)$*, where* $\alpha_1, \ldots, \alpha_n \in Act$*, is a finite sequence of actions such as for every product from* fm*, the product is not able to execute* t *or this execution does not end in the initial state:*

$$\forall p \in [\![fm]\!] : fts_{|p} \stackrel{t}{\not\Rightarrow} i$$

When derived from the soda vending machine FTS, a positive abstract test case has to start from 1 and end in 1 and only fire transitions with compatible feature expressions. For instance, abstract test case *(free, soda, serveSoda, open, close)* is a positive abstract test case, while *(free, soda, serveSoda)* is a negative

abstract test cases as it does not end in the initial state when it is executed on the FTS. Other negative abstract test cases include sequences of actions that mixes the behaviour of two incompatible products.

In the remainder, we mainly focus on positive abstract test cases and simply write *test case*. A *test suite*, defined for a SUT, is a set of test cases.

Test suite product selection When abstract test cases are concretized, the result (*i.e.,* concrete test cases, represented as a sequence of operations on the system) has to be executed on one or more products of the SPL. The set of products able to execute a test case may be calculated from the FTS (and the FM). It corresponds to all the products (*i.e.,* set of features) of the FM that satisfy all the feature expressions associated to the transitions fired by the abstract test case when it is executed on the FTS:

Definition 13 (Test case product selection). *Given an FTS* $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ *and a positive abstract test case* $t = (\alpha_1, \ldots, \alpha_n)$ *with* $(\alpha_1, \ldots, \alpha_n) \in Act$*, the set of products able to execute* t *is defined as:*

$$prod(fts, t) = \{p \in [\![fm]\!] \mid fts_{|p} \stackrel{t}{\Longrightarrow} i\}$$

From a practical point of view, the set of products contains all the products satisfying the conjunction of the feature expressions $\gamma(s_k \xrightarrow{\alpha_i} s_{k+1})$ on the path(s) of t and the FM fm. When fm is Boolean, it may be transformed to a Boolean formula in CNF where features become variables [33]. The existence of a product for a test case is equivalent to the satisfiability of the following formula, that can be checked by a SAT solver:

$$\bigvee_{pt \in paths} \left(\bigwedge_{i=1}^{n_{pt}} \left(\gamma(s_k \xrightarrow{\alpha_i} s_l) \right) \right) \wedge CNF(fm)$$

For instance, the set of products for the test case *(free, soda, serveSoda, open, close)*, derived from the vending machine FTS, contains all the products of the SPL that offer free soda. Similarly, for a test suite, we have:

Definition 14 (Test suite product selection). *Given an FTS* $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ *and a test suite* $s = \{t_1, \ldots, t_n\}$*, where* $t_1, \ldots, t_n$ *are positive abstract test cases, the set of products able to execute the test suite:*

$$prod(fts, s) = \bigcup_{t_i \in s} prod(fts, t_i)$$

If we have a test suite (s) with two test cases *(free, soda, serveSoda, open, close)* and *(free, tea, serveTea, open, close)*, the set of products contains all the products of the SPL that offers free soda or free tea.

We will consider that for a given test suite (s), a set of products (M) is adequate, if M contains enough products to execute the test cases in s:

Definition 15 (s-adequate set of products). *Let fts be an FTS and* $s = \{t_1, \ldots, t_n\}$ *be an abstract test suite where* $t_1, \ldots, t_n$ *are positive abstract test cases. The set of products* M *is* s*-adequate, denoted* $M \overset{s}{\Longrightarrow}$*, if each test case in* s *may be executed by at least one product in* M*:*

$$\forall t \in s : \exists p \in M, fts_{|p} \overset{t}{\Longrightarrow} i$$

Since one of the main concerns in SPL testing is to reduce the number of products needed to execute the tests, we also define the selection of the minimal s-adequate set of products required to execute a test suite:

Definition 16 (P-Minimal test suite product selection). *Let fts be an FTS and* $s = \{t_1, \ldots, t_n\}$ *be an abstract test suite where* $t_1, \ldots, t_n$ *are positive abstract test cases. A minimal* s*-adequate set of products needed to execute the test suite, denoted* $mprod(fts, s) = M$*, is a subset of* $prod(fts, s)$ *such that* M *is* s*-adequate and there is no subset of* M *that is* s*-adequate:*

$$\left(M \overset{s}{\Longrightarrow}\right) \wedge \left(\forall M' \subset M, \left(M' \overset{s}{\not\Longrightarrow}\right)\right)$$

For instance, there are two products able to execute all the test cases in the test suite s: one that allows to cancel purchase and one that doesn't. The p-Minimal set of products for s is a set with only one of those two products. The decision of the products to include (or not) should be taken by the test engineer, depending for instance on the cost linked to the derivation of each product.

4.2 Selection criteria

In order to efficiently select test cases, the test engineer has to provide *selection criteria* [65, 82], defined hereafter for *connected* FTSs as a function, returning for a given FTS and a test suite, 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.

Definition 17 (coverage criterion). *A coverage criterion is a function cov that associates an FTS and a test suite over this FTS to a real value in* $[0, 1]$*.*

Structural coverage Classical structural coverage criteria are expressed using the structural elements of the model [65, 82] (in this case, FTSs) covered by the execution of a test case.

Definition 18 (State/All-states coverage). *The state coverage criterion is related to the ratio between the states visited by the test cases pertaining to the test suite and all the states of the FTS. When the value of the function equals to* 1*, the test suite satisfies the* all-states coverage*.*

Definition 19 (Action/All-actions coverage). *The action coverage criterion is related to the ratio between the actions triggered by the test cases pertaining to the test suite and all the actions of the FTS defined. When the value of the function equals* 1*, the test suite satisfies* all-actions coverage*.*

Definition 20 (Transition/All-transitions coverage). *Transition coverage is related to the ratio between transitions covered when running test cases on the FTS and the total number of transitions of the FTS. When this ratio equals to* 1*, then the test suite satisfies* all-transitions *coverage.*

Definition 21 (Transition-pair/All-pairs coverage). *The transition-pairs coverage considers adjacent transitions successively entering and leaving a given state. When the coverage function reaches the value of* 1*, then the test suite covers* all-transition-pairs*.*

Definition 22 (Path/All-paths coverage). *Path coverage takes into account simple executable paths (i.e., paths that does not fire the same transition twice), that is sequences of transitions starting from and ending in the initial state. If the coverage function value computing the ratio between the number of simple executable paths covered by the test cases and total number of simple executable paths in the FTS is* 1*,* all-paths *coverage has been reached.*

The all-path coverage is the strongest coverage criterion. It specifies that each simple executable path in the FTS should be followed at least once when executing the test suite. Depending on the FTS, this coverage criterion might not be scalable.

Dissimilarity-based coverage Dissimilarity testing is a technique used to select a test suite among all possible test cases, which aims to maximise the fault detection rate by increasing diversity among test cases [16, 49]. This diversity is characterized by a *dissimilarity distance* defined over the different test cases. For instance, Henard *et al.* [51] applied dissimilarity testing to SPL in order to sample and prioritize products to test. The idea was to mimic the combinatorial interaction testing (CIT) sampling for SPLs [62,73], in which valid combinations of features are covered at least once.

Applied to FTS, dissimilarity-based coverage extends Henard *et al.*'s [51] work by formulating the abstract test case selection as a bi-objective problem [36] where one wants to maximize dissimilarity between the products, but also the excercised behaviors. Formally, we define the dissimilarity between two test cases as follows:

Definition 23 (Test cases dissimilarity). *Given an FTS fts and two test cases* $t_1 = (\alpha_1, \ldots, \alpha_n)$ *and* $t_2 = (\beta_1, \ldots, \beta_n)$ *derived from fts, the dissimilarity between* t_1 *and* t_2 *is defined as:*

$$diss(fts, t_1, t_2) = diss_p(prod(fts, t_1), prod(fts, t_2)) \otimes diss_a((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n))$$

Where $diss_p : [\![d]\!] \times [\![d]\!] \to [0, 1.0]$ *computes a dissimilarity distance between the products,* $diss_a : Act^+ \times Act^+ \to [0, 1.0]$ *computes a dissimilarity distance between the actions of the test cases, and* $\otimes : [0, 1.0] \times [0, 1.0] \to [0, 1.0]$ *is an operator combining the products and actions distances to return a global dissimilarity distance between the two test cases.*

The dissimilarity between products ($diss_p$) may for instance be the Jaccard index (*i.e.,* the the ratio between the number of products common to $prod(fts, t_1)$ and $prod(fts, t_2)$, and the total number of products in both) [51,52]. In our previous work [36], we defined several dissimilarity distances $diss_a$ for the actions executed by two test cases (including the Jaccard index which gave best results in our evaluation) and used Definition 23 to drive the selection of abstract test cases using a (1+1) evolutionary algorithm [42].

4.3 Test case and test suite minimality

Usually, when performing test case selection, one wants to have a test suite as small as possible while ensuring the best coverage. Contrary to single systems where only the size of the test suite is taken into account, when performing SPL testing, we also have to consider the number of products needed to execute the test suite. We define the *size* of a test suite as the number of transitions triggered by its test cases.

Definition 24 (Test suite size). *The size of a test suite s corresponds to the number of transitions triggered in a FTS fts when executing the test cases of s on fts, denoted*

$$fts \xRightarrow{s}$$

This allows to differentiate a test suite s_1 with test cases only triggering a minimal set of transitions to satisfy a coverage criterion from a test suite s_2 also satisfying this coverage criterion, but with longer test cases triggering transitions that do not contribute to the coverage. For a given FTS *fts*, we denote $s_1 < s_2$ if

$$\left(fts \xRightarrow{s_1}\right) < \left(fts \xRightarrow{s_2}\right)$$

As opposed to current practice, the size of the test suite does not take the number of test cases into account. Two test suites with the same size may have different number of test case. This metric is more representative of the behaviour of the SPL covered by a test suite. As for test suites, we define the size of a test case as the number of transitions triggered by this test case.

Definition 25 (Test case size). *The size of a test case t corresponds to the number of transitions triggered in a FTS fts when executing t on fts, denoted*

$$fts \xRightarrow{t}$$

Depending on the product line under test, the test engineer decides if the test suite has to contain lots of small test cases, to ease the debugging process when a test case fails for instance, or few longer test cases, if the setup required to execute each test is expensive for instance.

For such a distribution of test cases sizes in a test suite, the selection process compromises between the size of the test suite and the number of products needed to execute this test suite. We define the former as the *minimal* test suite property, and the latter as the *P-minimal* test suite property.

Property 1 (Minimal test suite). A test suite s over a given FTS $fts = (S, Act, trans, i, d, \gamma)$ is minimal *w.r.t.* a selection criteria cov iff $\nexists\, s'$ such that $s' < s$ and $cov(fts, s') \geq cov(fts, s)$.

Property 2 (P-minimal test suite). A test suite s over a given FTS $fts = (S, Act, trans, i, d, \gamma)$ is product-minimal (p-minimal) regarding a selection criteria cov iff $\nexists\, s'$ such that $(cov(fts, s') \geq cov(fts, s)) \wedge (\#\, mprod(fts, s') < \#\, mprod(fts, s))$.

In other words, a test suite is minimal if there exists no smaller test suite with a better coverage, and a p-minimal test suite represents the minimal set of test cases (with the best coverage) such that the number of products needed to execute all of them is minimal.

4.4 Related work

With the coming of age of SPL as a development paradigm, the question of how to test them was raised. The first approaches considered the impact of the intertwined domain engineering and application engineering processes on test planning, design and execution activities [66, 74]. Early contributions focussed notably on the relationship between SPL use cases [13] and tests [14]. In the latter, Bertolino and Gnesi adapt the SPL use cases into test plans with tags. These tags allows to specify which scenarios and which properties must be tested depending on the activated features (mandatory, alternative, optional, etc.). Another approach is to combine high-level "test patterns" during product derivation and synthesize such scenarios as LTS in order to take advantage of model-based test generation techniques [68]. Incremental testing approaches have also been more recently adapted in the SPL context [57,63,71,83]. For example, Lochau *et al.* [61,63] proposed a model-based approach that shifts from one product to another by applying *deltas* to statemachine models. These deltas enable automatic reuse and adaptation of the test model and derivation of retest obligations. Oster *et al.* [71] extend combinatorial interaction testing with the possibility to specify a predefined subset of products in the set of products to test. These approaches assume that *we know already* which products to test.

Sampling techniques, such as t-wise approaches [25, 26, 53, 73], strive to answer to this question by exploring configurations allowed by the feature model. These techniques are based on the systematic coverage of the interaction of two more features, a criteria that has been shown empirically to cover 80% of

bugs [58]. T-wise sampling being NP-complete in the presence of constraints, various heuristics have been proposed [64], from greedy algorithms [25, 53] to meta-heuristics [44,45]. Meta-heuristics are also at the heart of dissimilarity sampling techniques that maximize distances between configurations [1, 51]. There are also approaches that combines several objectives (coverage, cost of configurations, *etc.*) [47, 50, 77].

Efforts to combine sampling techniques with modelling ones (*e.g.,* [62]) exist. These approaches are product-based, meaning that they may miss opportunities to reuse tests amongst sampled products [75]. There are also approaches focused on the SPL code by building variability-aware interpreters for various languages [55]. Based on symbolic execution techniques such interpreters are able to run a very large set of products with respect to one given test case [69]. Cichos *et al.* [18] 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. At the code level, Li *et al.* [60] focuses on test specification and values reuse from one product to another by using a genetic algorithm that integrates software fault localization techniques and structural coverage of the program. Finally, Beohar *et al.* [9, 11, 12] propose to adapt the *ioco* framework proposed by Tretmans [80] to FTSs.

As we have seen, the FTS formalism offers an ideal language to study model-based testing of SPLs. Though we initially focused on family-based approaches to exploit the sharing opportunities amongst test cases, the impact of sampling techniques can be assessed and we can envision both in a multi-objective setting [36]. We believe that the FTS formalism, natively equipped with features as a first-class concept, is pivotal to inter-model verification support and supports combination of quality assurance techniques both at the domain and application engineering levels.

5 Perspectives

Ten years after the inception of featured transition systems, we (and others) demonstrated its relevance to lay the foundations for model-based and formal quality assurance of variability-intensive systems. This in turn enabled us to derive efficient algorithms and to integrate them in the ProVeLines and ViBES frameworks [31, 38]. In this section, we would like to discuss some perspectives that would possibly lead us to work on and extend FTS for next decade.

5.1 Grand verification challenges: Cyber-Physical and Learning Systems

The last decade has seen a tremendous increase in the integration of hardware and software in number of connected devices and sensors, leading to the advent of the internet-of-things (IoT). IoT has pervaded every domain of our lives from the most useless gadget to more safety-critical Cyber-Physical Systems (CPS)

embedded in cars and planes. According to Briand *et al.*, even a simple car controller can be *untestable* [15]. Indeed, the large input space and the necessity to evaluate the outputs continuously over a time period is not adapted to discrete testing and verification approaches. The fact that CPS are also VIS, in the sense that they can dynamically adapt to their environment and the difficulty to predict this environment precisely forms an additional challenge.

Connected devices and sensors produce an enormous amount of data that are processed by intelligent systems increasingly relying on artificial intelligence (AI) algorithms. AI-enabled systems have shown their power in a variety of domains from the game of Go to autonoumous driving. "With great power comes great responsibility" is a cliché that perfectly applies to artificial intelligence (AI). As technology is progressing faster than ever before towards software with more and more abilities, adaptation and autonomy, the risks are becoming increasingly apparent. Adversarial machine learning [46] has shown how to have a given AI algorithm to misclassify a panda as a gibbon thanks to a few transformations to the image, sometimes invisible. Slight changes in luminosity may lead to the wrong turn on the road [79]. With recent work showing that learnability may be undecidable, the hope of fully verifiable AI vanishes [8].

5.2 Extended FTS for Cyber-Physical and AI-ready Systems

The aforementioned challenges suggest two research directions in order to extend the FTS formalism and its verification and validation algorithms for these highly complex, dynamic and configurable systems.

Anytime FTS. FTS were initially thought in the usual product line setting where all the features and their relationships could be specified in advance ans were not allowed to change. Adaptation to the environment and learning imply that this assumption does not hold anymore: features will disappear and new ones may appear as normal operation of the system. We previously envisioned the scenario where cars receive new features such as autopilot that can be downloaded and activated on demand via a software marketplace [72]. Since these cars dynamically adapt - the behaviour of the car is itself variability-aware and context-dependent - verifying if the introduction of a new feature is safe, the car should itself embeds its FTS and model-checker. To be efficient, on-the-fly reduction techniques of the verification space must be employed: for example, Kim *et al.* prune statically configurations that cannot violate a given property, reducing the number of configurations to monitor at runtime [56]. Cordy *et al.* have proposed incremental verification of software product lines to deal with partial configurations [30], though this technique has not been extended to runtime scenarios yet. These challenges lead us to conjecture that the upcoming techniques should be able to mix design time and runtime V&V techniques in a seamless manner.

Stochastic FTS. The uncertain nature of the targeted systems lead us to pursue the work on stochastic FTS and their relation with other formalisms such as markov chains, markov decision processes or modal transition systems (see Section 3). As we have seen, there is no predetermined winning strategy between family-based and product-based scenarios. It has to be noted that stochasticity does not only concern behaviour but also decisions as it is the nature of machine learning algorithms to take decisions on probabilities rather than on logic. In other words, V&V algorithms will have to deal with feature models that are themselves probabilistic [32].

6 Conclusion

In this chapter, we covered almost a decade of VIS modelling, verification and testing for and with Featured Transition Systems. Initially dedicated to model-checking it also demonstrated it suitability for model-based testing and supported applications even beyond VIS such as offering solutions to speed up the analysis of mutants [37]. We believe that the universality and simplicity of FTS contributed to this diversity of FTS-related contributions. From a more personal perspective, it enabled the dialogue between the authors issued from the formal verification and testing communities, yielding fruitful collaborations. Given the challenges ahead, we are convinced that the combination of techniques and the removal of the frontiers between these communities is a prerequisite to future advances in VIS V&V and we look forward to it.

References

1. Al-Hajjaji, M., Thüm, T., Meinicke, J., Lochau, M., Saake, G.: Similarity-based prioritization in software product-line testing. In: 18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014. pp. 197–206 (2014). https://doi.org/10.1145/2648511.2648532, http://doi.acm.org/10.1145/2648511.2648532
2. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer-Verlag (2013)
3. Apel, S., von Rhein, A., Wendler, P., Größlinger, A., Beyer, D.: Strategies for product-line verification: case studies and experiments. In: ICSE'13. pp. 482–491 (2013)
4. Atlee, J.M., Beidu, S., Fahrenberg, U., Legay, A.: Merging features in featured transition systems. In: Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, MoDeVVa@MoDELS 2015, Ottawa, Canada, September 29, 2015. pp. 38–43. CEUR-WS.org (2015)
5. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
6. ter Beek, M.H., Legay, A., Lluch-Lafuente, A., Vandin, A.: Statistical analysis of probabilistic models of software product lines with quantitative constraints. In: Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015. pp. 11–15. ACM (2015)

7. ter Beek, M.H., de Vink, E.P., Willemse, T.A.C.: Family-based model checking with mcrl2. In: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10202, pp. 387–405. Springer (2017)
8. Ben-David, S., Hrubeš, P., Moran, S., Shpilka, A., Yehudayoff, A.: Learnability can be undecidable. Nature Machine Intelligence **1**(1), 44–48 (2019). https://doi.org/10.1038/s42256-018-0002-3, https://doi.org/10.1038/s42256-018-0002-3
9. Beohar, H., Mousavi, M.R.: Input-output Conformance Testing Based on Featured Transition Systems. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1272–1278. SAC '14, ACM Press (2014), http://doi.acm.org/10.1145/2554850.2554949
10. Beohar, H., Mousavi, M.R.: Input-output conformance testing for software product lines. J. Log. Algebr. Meth. Program. **85**(6), 1131–1153 (2016). https://doi.org/10.1016/j.jlamp.2016.09.007, https://doi.org/10.1016/j.jlamp.2016.09.007
11. Beohar, H., Mousavi, M.: Spinal Test Suites for Software Product Lines. ArXiv e-prints (2014)
12. Beohar, H., Varshosaz, M., Mousavi, M.R.: Basic behavioral models for software product lines: Expressiveness and testing pre-orders. Science of Computer Programming (jul 2015), http://www.sciencedirect.com/science/article/pii/S0167642315001288
13. Bertolino, A., Fantechi, A., Gnesi, S., Lami, G.: Product Line Use Cases: Scenario-Based Specification and Testing of Requirements, pp. 425–445. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
14. Bertolino, A., Gnesi, S.: Pluto: A test methodology for product families. In: van der Linden, F.J. (ed.) Software Product-Family Engineering. pp. 181–197. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
15. Briand, L., Nejati, S., Sabetzadeh, M., Bianculli, D.: Testing the untestable: Model testing of complex software-intensive systems. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 789–792. ICSE '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2889160.2889212, http://doi.acm.org/10.1145/2889160.2889212
16. Cartaxo, E.G., Machado, P.D.L., Neto, F.G.O.: On the use of a similarity function for test case selection in the context of model-based testing. Software Testing, Verification and Reliability **21**(2), 75–100 (2011), http://dx.doi.org/10.1002/stvr.413
17. Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C.: Profeat: feature-oriented engineering for family-based probabilistic model checking. Formal Asp. Comput. **30**(1), 45–75 (2018). https://doi.org/10.1007/s00165-017-0432-4, https://doi.org/10.1007/s00165-017-0432-4
18. Cichos, H., Oster, S., Lochau, M., Schürr, A.: Model-based Coverage-driven Test Suite Generation for Software Product Lines. In: Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems. pp. 425–439. MODELS'11, Springer (2011), http://dl.acm.org/citation.cfm?id=2050655.2050698
19. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with SNIP. STTT **14**(5), 589–612 (2012)

20. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.: Formal semantics, modular specification, and symbolic verification of product-line behaviour. Sci. Comput. Program. **80**, 416–439 (2014). https://doi.org/10.1016/j.scico.2013.09.019, https://doi.org/10.1016/j.scico.2013.09.019
21. Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., cois Raskin, J.F.: Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. Transactions on Software Engineering pp. 1069–1089 (2013)
22. Classen, A., Cordy, M., Schobbens, P., Heymans, P., Legay, A., Raskin, J.: Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. IEEE Trans. Software Eng. **39**(8), 1069–1089 (2013). https://doi.org/10.1109/TSE.2012.86, https://doi.org/10.1109/TSE.2012.86
23. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: ICSE'11. pp. 321–330. ACM (2011)
24. 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)
25. Cohen, M., Dwyer, M., Jiangfan Shi: Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. IEEE Transactions on Software Engineering **34**(5), 633–650 (sep 2008), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4564473 http://ieeexplore.ieee.org/document/4564473/
26. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis - ROSATEA '06 pp. 53–63 (2006), http://portal.acm.org/citation.cfm?doid=1147249.1147257
27. Cordy, M., Classen, A., Perrouin, G., Schobbens, P., Heymans, P., Legay, A.: Simulation-based abstractions for software product-line model checking. In: 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. pp. 672–682. IEEE Computer Society (2012)
28. Cordy, M., Heymans, P., Legay, A., Schobbens, P., Dawagne, B., Leucker, M.: Counterexample guided abstraction refinement of product-line behavioural models. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. pp. 190–201. ACM (2014)
29. Cordy, M., Schobbens, P., Heymans, P., Legay, A.: Behavioural modelling and verification of real-time software product lines. In: 16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1. pp. 66–75. ACM (2012)
30. Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A.: Towards an incremental automata-based approach for software product-line model checking. In: Proceedings of the 16th International Software Product Line Conference-Volume 2. pp. 74–81. ACM (2012)
31. Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A.: Provelines: A product-line of verifiers for software product lines. In: SPLC'13. pp. 141–146. ACM (2013)
32. Czarnecki, K., She, S., Wasowski, A.: Sample spaces and feature models: There and back again. In: Proceedings of the 2008 12th International Software Product Line Conference. pp. 22–31. SPLC '08, IEEE Computer So-

ciety, Washington, DC, USA (2008). https://doi.org/10.1109/SPLC.2008.49, https://doi.org/10.1109/SPLC.2008.49
33. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC '07. pp. 23–34. IEEE Computer Society (2007)
34. Devroey, X.: Behavioural model-based testing of software product lines. Ph.D. thesis, University of Namur, Namur, Belgium (2017), https://researchportal.unamur.be/en/studentTheses/behavioural-model-based-testing-of-software-product-lines
35. Devroey, X., Perrouin, G., Cordy, M., Samih, H., Legay, A., Schobbens, P., Heymans, P.: Statistical prioritization for software product line testing: an experience report. Software and System Modeling **16**(1), 153–171 (2017). https://doi.org/10.1007/s10270-015-0479-8, https://doi.org/10.1007/s10270-015-0479-8
36. Devroey, X., Perrouin, G., Legay, A., Schobbens, P.Y., Heymans, P.: Search-based Similarity-driven Behavioural SPL Testing. In: Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16. pp. 89–96. ACM Press, Salvador, Brazil (jan 2016), http://0-doi.acm.org.fama.us.es/10.1145/2866614.2866627 http://dl.acm.org/citation.cfm?doid=2866614.2866627
37. Devroey, X., Perrouin, G., Papadakis, M., Legay, A., Schobbens, P., Heymans, P.: Featured model-based mutation analysis. In: Dillon, L.K., Visser, W., Williams, L. (eds.) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. pp. 655–666. ACM (2016). https://doi.org/10.1145/2884781.2884821, https://doi.org/10.1145/2884781.2884821
38. Devroey, X., Perrouin, G., Schobbens, P.Y., Heymans, P.: Poster: VIBeS, Transition System Mutation Made Easy. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. ICSE '15, vol. 2, pp. 817–818. IEEE, Florence, Italy (may 2015). https://doi.org/10.1109/ICSE.2015.263, http://ieeexplore.ieee.org/document/7203084/
39. Dimovski, A.S., Legay, A., Wasowski, A.: Variability abstraction and refinement for game-based lifted model checking of full CTL. In: Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11424, pp. 192–209. Springer (2019)
40. Dimovski, A.S., Wasowski, A.: From transition systems to variability models and from lifted model checking back to UPPAAL. In: Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10460, pp. 249–268. Springer (2017)
41. Dimovski, A.S., Wasowski, A.: Variability-specific abstraction refinement for family-based model checking. In: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10202, pp. 406–423. Springer (2017)
42. Droste, S., Jansen, T., Wegener, I.: On the analysis of the (1+1) evolutionary algorithm. Theoretical Computer Science **276**(1), 51–81 (apr 2002), http://www.sciencedirect.com/science/article/pii/S0304397501001827

43. Dubslaff, C., Klüppelholz, S., Baier, C.: Probabilistic model checking for energy analysis in software product lines. In: 13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014. pp. 169–180. ACM (2014)
44. Ensan, F., Bagheri, E., Gašević, D.: Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In: Ralyté, J., Franch, X., Brinkkemper, S., Wrycza, S. (eds.) Advanced Information Systems Engineering: 24th International Conference, CAiSE '12. pp. 613–628. Springer (2012)
45. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empirical Software Engineering **16**(1), 61–102 (2011)
46. Goodfellow, I., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: International Conference on Learning Representations (2015), http://arxiv.org/abs/1412.6572
47. Guo, J., Liang, J.H., Shi, K., Yang, D., Zhang, J., Czarnecki, K., Ganesh, V., Yu, H.: SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines. Software & Systems Modeling (Jul 2017). https://doi.org/10.1007/s10270-017-0610-0, https://doi.org/10.1007/s10270-017-0610-0
48. Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B.: Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. Empirical Software Engineering **24**(2), 674–717 (2019). https://doi.org/10.1007/s10664-018-9635-4, https://doi.org/10.1007/s10664-018-9635-4
49. Hemmati, H., Arcuri, A., Briand, L.: Achieving scalable model-based testing through test case diversity. ACM Transactions on Software Engineering and Methodology **22**(1), 1–42 (feb 2013), http://dl.acm.org/citation.cfm?id=2430536.2430540
50. Henard, C., Papadakis, M., Harman, M., Le Traon, Y.: Combining multi-objective search and constraint solving for configuring large software product lines. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1. pp. 517–528. ICSE '15, IEEE Press, Piscataway, NJ, USA (2015), http://dl.acm.org/citation.cfm?id=2818754.2818819
51. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Le Traon, Y.: Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. IEEE Transactions on Software Engineering **40**(7), 650–670 (jul 2014), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6823132 http://ieeexplore.ieee.org/document/6823132/
52. Jaccard, P.: Étude comparative de la distribution florale dans une portion des Alpes et des Jura. Bulletin del la Société Vaudoise des Sciences Naturelles **37**, 547–579 (1901)
53. Johansen, M.F., Haugen, Ø., Fleurey, F.: An algorithm for generating t-wise covering arrays from large feature models. In: Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1. SPLC '12, vol. 1, p. 46. ACM Press (2012), http://doi.acm.org/10.1145/2362536.2362547 http://dl.acm.org/citation.cfm?doid=2362536.2362547
54. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Carnegie Mellon University (1990)

55. Kästner, C., von Rhein, A., Erdweg, S., Pusch, J., Apel, S., Rendel, T., Ostermann, K.: Toward Variability-aware Testing. In: Proceedings of the 4th International Workshop on Feature-Oriented Software Development. pp. 1–8. FOSD '12, ACM Press (2012), http://doi.acm.org/10.1145/2377816.2377817
56. Kim, C.H.P., Bodden, E., Batory, D.S., Khurshid, S.: Reducing configurations to monitor in a software product line. In: Runtime Verification. Lecture Notes in Computer Science, vol. 6418, pp. 285–299. Springer (2010)
57. Knapp, A., Roggenbach, M., Schlingloff, B.H.: On the Use of Test Cases in Model-based Software Product Line Development. In: Proceedings of the 18th International Software Product Line Conference - Volume 1. pp. 247–251. SPLC '14, ACM Press (2014), http://doi.acm.org/10.1145/2648511.2648539
58. Kuhn, D., Wallace, D., Gallo, A.: Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering **30**(6), 418–421 (jun 2004)
59. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 122–135. Springer (2010)
60. Li, X., Wong, W.E., Gao, R., Hu, L., Hosono, S.: Genetic Algorithm-based Test Generation for Software Product Line with the Integration of Fault Localization Techniques. Empirical Software Engineering pp. 1–51 (2017), http://dx.doi.org/10.1007/s10664-016-9494-9
61. Lochau, M., Lity, S., Lachmann, R., Schaefer, I., Goltz, U.: Delta-oriented model-based integration testing of large-scale systems. Journal of Systems and Software **91**, 63–84 (may 2014). https://doi.org/10.1016/j.jss.2013.11.1096, http://linkinghub.elsevier.com/retrieve/pii/S0164121213002781
62. Lochau, M., Oster, S., Goltz, U., Schürr, A.: Model-based pairwise testing for feature interaction coverage in software product line engineering. Software Quality Journal **20**(3-4), 567–604 (sep 2012), http://www.springerlink.com/index/10.1007/s11219-011-9165-4 http://link.springer.com/10.1007/s11219-011-9165-4
63. 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), http://dx.doi.org/10.1007/978-3-642-30473-6_7 http://link.springer.com/10.1007/978-3-642-30473-6_7
64. Lopez-Herrejon, R.E., Fischer, S., Ramler, R., Egyed, A.: A first systematic mapping study on combinatorial interaction testing for software product lines. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 1–10. IEEE (2015)
65. Mathur, A.P.: Foundations of software testing. Pearson Education, India (2008)
66. McGregor, J.D.: Testing a Software Product Line, pp. 104–140. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
67. Mendonca, M., Branco, M., Cowan, D.: S.p.l.o.t.: Software product lines online tools. In: Proceedings of OOPSLA '09. pp. 761–762. ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1639950.1640002
68. Nebut, C., Pickin, S., Le Traon, Y., Jézéquel, J.M.: Automated requirements-based generation of test cases for product families. In: 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings. pp. 263–266. IEEE (2003)

69. Nguyen, H.V., Kästner, C., Nguyen, T.N.: Exploring variability-aware execution for testing plugin-based web applications. In: Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. pp. 907–918. ICSE '14, ACM Press (2014), http://dl.acm.org/citation.cfm?doid=2568225.2568300
70. Olaechea, R., Fahrenberg, U., Atlee, J.M., Legay, A.: Long-term average cost in featured transition systems. In: Proceedings of the 20th International Systems and Software Product Line Conference, SPLC 2016, Beijing, China, September 16-23, 2016. pp. 109–118. ACM (2016)
71. Oster, S., Markert, F., Ritter, P.: Automated Incremental Pairwise Testing of Software Product Lines. In: Bosch, J., Lee, J. (eds.) Proceedings of the 14th International Software Product Lines Conference: Going Beyond. pp. 196–210. SPLC'10, Springer, Jeju Island, South Korea (2010), http://dx.doi.org/10.1007/978-3-642-15579-6_14
72. Perrouin, G., Acher, M., Davril, J., Legay, A., Heymans, P.: A complexity tale: web configurators. In: Proceedings of the 1st International Workshop on Variability and Complexity in Software Design, VACE@ICSE 2016, Austin, Texas, USA, May 14-22, 2016. pp. 28–31. ACM (2016). https://doi.org/10.1145/2897045.2897051, https://doi.org/10.1145/2897045.2897051
73. Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., le Traon, Y.: Pairwise testing for software product lines: Comparison of two approaches. Software Quality Journal **20**(3-4), 605–643 (2011), http://dx.doi.org/10.1007/s11219-011-9160-9
74. Pohl, K., Böckle, G., van der Linden, F.: Software product line engineering - foundations, principles, and techniques. Springer (2005)
75. von Rhein, A., Apel, S., Kästner, C., Thüm, T., Schaefer, I.: The pla model: on the combination of product-line analyses. In: VaMoS. p. 14 (2013)
76. Rodrigues, G.N., Alves, V., Nunes, V., Lanna, A., Cordy, M., Schobbens, P., Sharifloo, A.M., Legay, A.: Modeling and verification for probabilistic properties in software product lines. In: 16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015, Daytona Beach, FL, USA, January 8-10, 2015. pp. 173–180. IEEE Computer Society (2015)
77. Sayyad, A.S., Menzies, T., Ammar, H.: On the value of user preferences in search-based software engineering: a case study in software product lines. In: ICSE'13. pp. 492–501 (2013)
78. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Feature Diagrams: A Survey and A Formal Semantics. In: RE'06. pp. 139–148 (2006)
79. Tian, Y., Pei, K., Jana, S., Ray, B.: Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In: ICSE '18. pp. 303–314. ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3180155.3180220
80. Tretmans, J.: Model based testing with labelled transition systems. Formal methods and testing pp. 1–38 (2008), http://www.springerlink.com/index/y390356226x154j0.pdf
81. Tretmans, J.: Model-Based Testing and Some Steps towards Test-Based Modelling. In: Bernardo, M., Issarny, V. (eds.) Formal Methods for Eternal Networked Software Systems, LNCS, vol. 6659, chap. 9, pp. 297–326. Springer (2011). https://doi.org/10.1007/978-3-642-21455-4_9, http://dx.doi.org/10.1007/978-3-642-21455-4_9
82. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann (2007)
83. Uzuncaova, E., Khurshid, S., Batory, D.: Incremental Test Generation for Software Product Lines. Software Engineering, IEEE Transactions on **36**(3), 309–322 (2010)

84. Vanhecke, J., Devroey, X., Perrouin, G.: AbsCon : A Test Concretizer for Model-based Testing. In: Software Testing, Verification and Validation Workshops (ICSTW), 2019 IEEE Twelfth International Conference on. A-MOST '19, IEEE, Xi'an, China (2019)