A Decade of Featured Transition Systems

Cordy, Maxime; Devroey, Xavier; Legay, Axel; Perrouin, Gilles; Classen, Andreas; Heymans, Patrick; Schobbens, Pierre-Yves; Raskin, Jean-François

DOI
10.1007/978-3-030-30985-5_18

Publication date
2019

Document Version
Final published version

Published in
From Software Engineering to Formal Methods and Tools, and Back.

Citation (APA)
Cordy, M., Devroey, X., Legay, A., Perrouin, G., Classen, A., Heymans, P., Schobbens, P.-Y., & Raskin, J.-F. (2019). A Decade of Featured Transition Systems. In M. H. ter Beek, A. Fantechi, & L. Semini (Eds.), *From Software Engineering to Formal Methods and Tools, and Back.* (Vol. 11865, pp. 285-312). (Lecture Notes in Computer Science; Vol. 11865). Springer. https://doi.org/10.1007/978-3-030-30985-5_18

A Decade of Featured Transition Systems

Maxime Cordy[1], Xavier Devroey[2], Axel Legay[3], Gilles Perrouin[4](✉), Andreas Classen[5], Patrick Heymans[4], Pierre-Yves Schobbens[4], 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, Louvain-la-Neuve, Belgium
`axel.legay@uclouvain.be`
[4] PReCISE/NaDI, Faculty of Computer Science, University of Namur, Namur, Belgium
`{gilles.perrouin,patrick.heymans,pierre-yves.schobbens}@unamur.be`
[5] INTEC Software Engineering, St. Vith, 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 whose 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 an 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 paper, we will cover the formalism, its applications and sketch promising research directions.

Keywords: Variability-intensive systems · Modeling · Model-checking · Testing

1 Introduction

Variability-intensive systems (VISs) form a vast and heterogeneous class of software systems that encompasses: *Software Product Lines* [2,84], operating system

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.

M. H. ter Beek et al. (Eds.): Gnesi Festschrift, LNCS 11865, pp. 285–312, 2019.
https://doi.org/10.1007/978-3-030-30985-5_18

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, or configuration *options*. Configurable systems may involve thousands of features with complex dependencies. The set of valid combinations of features of a VIS can be represented in a tree-like structure, called *feature model* (FM) [60]. Each valid combination is a configuration of the VIS, which can be derived as a *variant* or a *product*, terms we will use interchangeably in this paper both at the model and code level. 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 [88]. Thanks to their formal semantics, operations on feature models such as inconsistency reasoning can be automated thanks to SAT solvers [75].

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.* [54] 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 TB of disk space, and 4,376 h (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 [6] 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.* [27] defined *Featured Transition Systems* (FTSs) as transition systems (TSs) annotated with combination of features on their transitions: each combination describes 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 reductions of the analysis time, both for formal verification or test generation.

This paper reviews the foundations and applications of featured transition systems, connecting them to other formalisms, such as modal transition systems, and considering extensions such as quantities and probabilities which are required to address V&V challenges induced by, for instance, cyber-physical or learning systems. The rest of this paper is structured as follows. Section 3 introduces the formalism and describes how the modified VIS model-checking problem can be solved efficiently. Section 3.4 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 existing coverage criteria for transition systems. Section 5 provides an outlook into the future of VIS V&V. Finally, Sect. 6 concludes the paper.

2 Grazie Mille

The content of this paper, although being a synthesis of a 10-year collective research effort, is only a very small sample of the many ways in which the work and personality of Stefania Gnesi have inspired us. The scope of this chapter lies at the intersection of formal methods and software product lines, two areas to which Stefania offered both seminal contributions and devoted a continuous effort over several decades. If that was all Stefania had done, she could already be proud of herself and happily retire without regrets. But this is actually only part of the story we are celebrating, a story that produced significant scientific contributions in a variety of other areas too, including requirements engineering, software engineering, critical systems and natural language processing; a story of relentlessly passing knowledge to her students and peers; a story of coordinating international research efforts; a story of organizing memorable scientific events and of serving research communities; and, in spite of all that, a story of remaining a humble, attentive and kind colleague with whom it is a constant pleasure to work and discuss. Stefania does not deserve a simple applause, she deserves a standing ovation. Thank you from all of us!

3 Verifying Variability-Intensive Systems with FTS

We model the space of variants in terms for feature models following the semantics as provided by Schobbens et al. [88]. A FM fm is a tuple (F, r, DE) where F is a set of features, $r \in F$ is the root, and $DE \subseteq F \times F$ is the set of decomposition edges between features. A product (or variant) is defined as a set of features $P = \{f_1, \ldots, f_n\}$, such that $f_i \in P$ if and only if f_i is part of the product. The semantics of a FM fm, noted $[\![fm]\!]$, is the set of valid products (whose features satisfy the FM constraints), i.e. a set of sets of features: $[\![fm]\!] \in 2^{2^F}$. We also assume the presence of a behavioural model M_v for all variants – which we will formalise below – in order to introduce the VIS model-checking problem. This problem 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 [25].

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 a VIS in a separate transition system (TS), and then applying single-system model checking on each of these TS individually. This method, named *enumerative* [26] or product-based [85], violates the principles of VIS engineering: the variants should not be modelled separately. Instead, one should build a core model, which is subsequently specialized 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 a 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 speed-up verification.

3.1 A Formalism to Model VIS Behaviour

In [25,27], we proposed Featured Transition Systems (FTSs) as a compact representation of the behaviours of a set of variants. FTSs are an extension of TSs 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 $b \in 2^{2^F}$ in which 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, $\mathbb{B}(px) = \bigvee_{p \in px} \left(\bigwedge_{f \in p} f \bigwedge_{g \in F \setminus p} \neg g \right)$.*

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

- *S is a set of states named the state space;*
- *Act is a set of actions;*

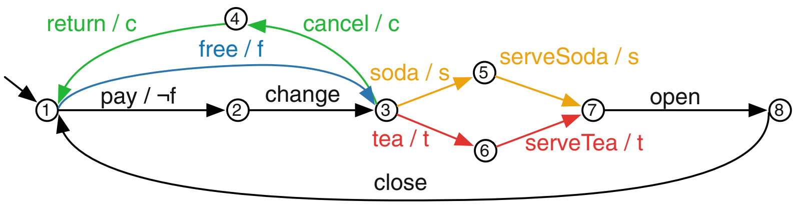

Fig. 1. The FTS modelling the vending machine VIS.

- *$Trans \subseteq S \times Act \times S$ 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 an 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 TSs of all the variants that compose the VIS. The TS model of a specific product is obtained from the FTS by applying a *projection* function. In simple terms, this function suppresses in the FTS all the transitions whose feature expression is not satisfied by the considered product [27], 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 Fig. 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 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 the fts is a total function $[\![fts]\!] : [\![fm]\!] \to 2^{(2^{AP})^\omega}$ such that $\forall p \in [\![fm]\!] \bullet [\![fts]\!](p) = [\![fts_{|p}]\!]$.*

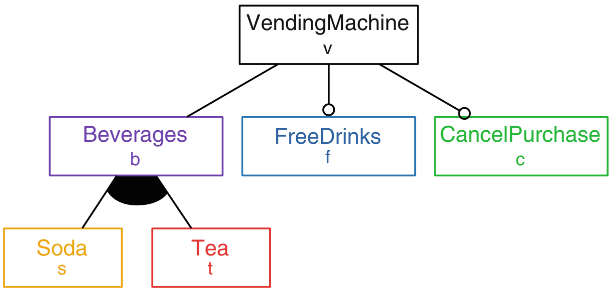

Fig. 2. The FM of the vending machine VIS.

3.2 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 an intuitive definition at the beginning of this paper. Here, we rephrase this definition formally, by considering a 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 [25]. 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, a VIS model-checker should thus compute F-satisfiability expressions, and associate each F-unsatisfying product with one of its traces that violates the formula.

3.3 Algorithms

Given its explicit notion of features, FTSs constitute 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 [25] and CTL [26] 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 variants that can reach a target 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 FTSs, 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 $Post : S \rightarrow (S \rightarrow 2^{(2^F)})$ 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 pair 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 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 $R : S \rightarrow (S \rightarrow 2^{(2^F)})$ 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. 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 the computation of the reachability function of a given state s_0. The algorithm consists of a loop that iterates over a stack of pairs (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 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 associates 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 pair, 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.

```
Input: $fts = (S, Act, Trans, I, AP, L, fm, \gamma), s_0 \in S$.
Output: $R(s_0)$.
$R \leftarrow \bot$;
$Stack \leftarrow push((s_0, fm), [])$;
while $Stack \neq []$ do
    $(s, \gamma) \leftarrow pop(Stack)$;
    $succ \leftarrow \{(s', \gamma') \mid s' \in dom(Post(s)) \wedge \gamma' = Post(s)(s') \wedge \gamma\}$;
    foreach $(s', \gamma') \in succ \bullet \gamma' \not\models \bot$ do
        if $s' \in dom(R)$ then
            $\gamma_{new} \leftarrow \neg R(s') \wedge \gamma'$;
            if $\gamma_{new} \not\models \bot$ then
                $R(s') \leftarrow R(s') \vee \gamma'$;
                $push((s', \gamma_{new}), Stack)$;
            end
        end
        else
            $R(s') \leftarrow \gamma'$;
            $push((s', \gamma'), Stack)$;
        end
    end
end
return $R$
```

Algorithm 1: Reachables(fts, s_0)

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

Theorem 1. *[25] 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 [23,25,26]. The FTS theory is thus a solid candidate solution for the VIS model-checking problem.

3.4 Related FTS-Based Verfication Work

Modal Automata, i.e., automata with optional and compulsory transitions, precede FTS as a formal model for software product lines. As an example, in [49], Gnesi and Fantechi proposed a behavioural model, namely the Extended Modal Labeled Transition Systems (EMLTS), as a basis for the formalisation of the different notions of variability usually present in the definitions of product families. In particular, an EMLTS is able to define a family of products by telling at any state of the system whether transitions are optional or compulsory for the products of the family. The work was then pursued by Leucker and co-authors [52] and compared with FTS in [4]. One of the main drawbacks of EMLTS is that there is no causality on transition choice from state to state. This causality is captured by FTS constraints and also by a constraint-based extension of EMLTS proposed in [7,8], but without the family-based analysis. It should be also noted that, contrary to FTS, EMLTS have not been extended to the quantitative setting.

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 [24], we have showed how to extend symbolic model-checking of computational tree logic to FTS. We showed how to encode features as extra variables in BDDs representing symbolic behaviors of multiple products without blowing up the representation. Later, in [10], 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 [96].

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 between 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 [31], shows clear benefit in using this approach. Branching bisimulation for FTS was also studied by Belder *et al.* [11]. Later, in collaboration with University of Waterloo Canada, we have showed that these 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 [5].

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 [32], 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 [43–45]. 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 than 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 combine FTS and timed automata in order to handle timed product lines. In [34], we have showed how to combine timed constraints of real-time clocks with feature constraints. We have then showed that the model-checking procedure from [25] applies directly to our case by using the well-known region construction from timed automata. Our timed extension has been reused and extended in various directions. As an example, Beohar and Mousavi introduced IOFTS that is an input/output extension of timed FTS for model-based testing of software product lines [14]. 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 [86], 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 [39], 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 also paved the way for compositional reasoning and analysis of probabilistic queries for software product lines. In [47], Baier *et al.* give a clear adaptation of compositional reasoning to this context. This is implemented in the ProFeat tool [21] that uses similar techniques to those in [86]. 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 [9,89], Ter Beek *et al.* have proposed an algebra to defined quantitative relations between features. This algebra is static in the sense that it relates features with quantitative information (cost, constraints on costs, *etc.*) and dynamic in the sense that it allows us to specify when features can appear and disappear in system's execution at runtime (hence opening the door to the analysis of dynamic software product lines). The verification process used in these works relies on a dynamic extension of statistical model-checking [66].

In a series of recent works *e.g.,* [79], we have also extended 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 decrease in this context. 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) [93] at the SPL level. Test cases are defined during domain engineering [84] for the SPL by associating each test to the set of products able to execute it. Intuitively, if one wants to test a particular product, she will consider only the tests associated to that particular product. In the other way around, if one wants to test an SPL, she will start by building the product with the highest number of associated tests and execute those tests on that product.

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 [73].

4.1 Test Concepts for FTSs

Since FTSs are derived from TSs, a natural starting point to adapt model-based testing in the context of software product lines is to consider existing coverage criteria for transition systems [93] 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. The remainder of this section 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*

$$\exists i \in I : i \xrightarrow{\alpha_1} s_k \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_l$$

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 [93]. At the SPL level, a positive abstract test case is defined as a sequence of actions executable by the *fts* (*i.e.,* executable by at least one product), while a negative abstract test case is a sequence of actions not executable by the *fts* (*i.e.,* not executable by any product). Once concretized (*i.e.,* transformed into executable code) [95], negative abstract test cases typically represent sequences of actions that every product of the product line should forbid. Note that a positive test case for a SPL may become a negative test case for one particular product of the SPL if this product is not allowed to exercise the behavior described in the test case. In the remainder of this section, we focus on test case selection at the SPL level.

In a LTS (*lts*), an abstract test case $t = (\alpha_1, \ldots, \alpha_n)$ is executable, denoted $\mathit{lts} \overset{t}{\Longrightarrow}$, if there exists a sequence of transitions starting from an initial state and labelled with $\alpha_1, \ldots, \alpha_n$ [91,92]. For an FTS (*fts*), to be executable, the sequence of transitions must moreover have feature expressions compatible with the associated FM (or its projection on a subset of the product line if one wants to test only a given set of products). In other words, when selecting test cases for a product line, a sequence of actions is executable by *fts* if there exists at least one product (p) which, when *fts* is projected onto p (denoted $\mathit{fts}_{|p}$), is able to execute it:

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

In testing, unlike model-checking [6], we only consider finite sequences of actions. Since FTS (as LTS) do not have an observable 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 states of an FTS as accepting states, observable for the tester (contrarily to Tretmans *et al.* [91,92], we do not partition the set of actions into inputs and observable outputs, this will be part of our future work). Positive abstract test cases have to end their execution in an initial state (*e.g.,* state 1 in the soda vending machine FTS) in order to observe that the test case was executed successfully.

Definition 11 (Positive abstract test case). *Let fts* $= (S, \mathit{Act}, \mathit{Trans}, I, \mathit{AP}, L, fm, \gamma)$ *be an FTS. A positive abstract test case* $t = (\alpha_1, \ldots, \alpha_n)$*, where* $\alpha_1, \ldots, \alpha_n \in \mathit{Act}$*, 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 an initial state:*

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

Definition 12 (Negative abstract test case). *Let fts* $= (S, \mathit{Act}, \mathit{Trans}, I, \mathit{AP}, L, fm, \gamma)$ *be an FTS. A negative abstract test case* $t = (\alpha_1, \ldots, \alpha_n)$*, where* $\alpha_1, \ldots, \alpha_n \in \mathit{Act}$*, 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 an initial state:*

$$\forall p \in [\![fm]\!], \nexists i \in I : fts_{|p} \overset{t}{\not\Rightarrow} i$$

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 an initial state when it is executed on the FTS (and hence one cannot observe if the test is successfull or not). Other negative abstract test cases include sequences of actions that mix 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:*

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

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 [38]. 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:

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

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:*

$$\forall t \in s : \exists p \in M, \exists i \in I, \mathit{fts}_{|p} \overset{t}{\Longrightarrow} i$$

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 $mprod(\mathit{fts}, s) = M$, is a subset of $prod(\mathit{fts}, s)$ such that M is s-adequate and there is no subset of M that is s-adequate:*

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

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* [73,93], defined hereafter 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 [73,93] (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 an 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 [20,55]. This diversity is characterized by a *dissimilarity distance* defined over the different test cases. For instance, Henard *et al.* [57] 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 [69,83], in which valid combinations of features are covered at least once.

Applied to FTS, dissimilarity-based coverage extends Henard *et al.*'s work [57] by formulating the abstract test case selection as a bi-objective problem [40] where one wants to maximize dissimilarity between the products, but also the exercised 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:*

$$\begin{aligned} \mathit{diss}(\mathit{fts}, t_1, t_2) = {} & \mathit{diss}_p(\mathit{prod}(fts, t_1), \mathit{prod}(fts, t_2)) \\ & \otimes \ \mathit{diss}_a((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n)) \end{aligned}$$

Where $diss_p : [\![d]\!] \times [\![d]\!] \rightarrow [0,1]$ *computes a dissimilarity distance between the products,* $diss_a : Act^+ \times Act^+ \rightarrow [0,1]$ *computes a dissimilarity distance between the actions of the test cases, and* $\otimes : [0,1] \times [0,1] \rightarrow [0,1]$ *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) [57,58]. In our previous work [40], 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 [46].

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*

$$fts \overset{s}{\Longrightarrow}$$

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

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

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*

$$fts \overset{t}{\Longrightarrow}$$

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

The first approaches of SPL testing considered the impact of the intertwined domain engineering and application engineering processes on test planning, design and execution activities [74,84]. Early contributions focused notably on the relationship between SPL use cases [17] and tests [18]. 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 [76]. Incremental testing approaches have also been more recently adapted in the SPL context [63,70,81,94]. For example, Lochau *et al.* [68,70] proposed a model-based approach that shifts from one product to another by applying *deltas* to state machine models. These deltas enable automatic reuse and adaptation of the test model and derivation of retest obligations. Oster *et al.* [81] 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 [28,29,59,83], 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 [64]. T-wise sampling being NP-complete in the presence of constraints, various heuristics have been proposed [71], from greedy algorithms [28,59] to meta-heuristics [48,50]. Meta-heuristics are also at the heart of dissimilarity sampling techniques that maximize distances between configurations [1,57]. There are also approaches that combines several objectives (coverage, cost of configurations, *etc.*) [53,56,87].

Efforts to combine sampling techniques with modelling ones (*e.g.,* [69]) exist. These approaches are product-based, meaning that they may miss opportunities to reuse tests among sampled products [85]. There are also approaches focused on the SPL code by building variability-aware interpreters for various languages [61]. Based on symbolic execution techniques such interpreters are able to run a very large set of products with respect to one given test case [77]. Cichos *et al.* [22] 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.* [67] 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.* [13,15,16] propose to adapt the *ioco* framework proposed by Tretmans [91] 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 [40]. 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 [36,42]. 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 Optimisation of Quality Requirements

The initial endeavour surrounding FTS and the work presented in this paper mainly targets the verification and validation of functional requirements in VIS. FTS-based approaches for checking non-functional (aka quality) requirements have also been targeted in the recent years, most of them focusing on one particular non-functional aspect (e.g. execution time [30,72], reliability [86], income [80], quality of service [78]). Our recent work [65] proposes an end-to-end framework to efficiently assess multiple quality attributes across all variants and find the variant optimizing the trade-off between those attributes.

This quest for optimum paves the way for future research that exploit sampling techniques to efficiently search for such optimal variants. Our preliminary work [33] shows that this problem is non trivial and call for new endeavour lying at the intersection of VIS verification, configuration sampling and statistical model checking.

5.2 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* [19]. 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 [51] 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 [90]. With recent work showing that learnability may be undecidable, the hope of fully verifiable AI vanishes [12].

5.3 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 [82]. 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 [62]. Cordy *et al.* have proposed incremental verification of software product lines to deal with partial configurations [35], 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 Sect. 3.4). 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 [37].

6 Conclusion

In this paper, 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 [41]. 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 2014, Florence, Italy, 15–19 September 2014, pp. 197–206 (2014). https://doi.org/10.1145/2648511.2648532
2. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37521-7

3. Apel, S., von Rhein, A., Wendler, P., Größlinger, A., Beyer, D.: Strategies for product-line verification: case studies and experiments. In: ICSE 2013, pp. 482–491 (2013)
4. Asirelli, P., ter Beek, M.H., Gnesi, S., Fantechi, A.: Formal description of variability in product families. In: 15th International Conference on Software Product Lines, SPLC 2011, Munich, Germany, 22–26 August 2011, pp. 130–139 (2011)
5. 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, 29 September 2015, pp. 38–43. CEUR-WS.org (2015)
6. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
7. ter Beek, M.H., Damiani, F., Gnesi, S., Mazzanti, F., Paolini, L.: From featured transition systems to modal transition systems with variability constraints. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 344–359. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_24
8. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Modelling and analysing variability in product families: model checking of modal transition systems with variability constraints. J. Log. Algebr. Meth. Program. **85**(2), 287–315 (2016). https://doi.org/10.1016/j.jlamp.2015.11.006
9. 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, 20–24 July 2015, pp. 11–15. ACM (2015)
10. Ter Beek, M.H., de Vink, E.P., Willemse, T.A.C.: Family-based model checking with mCRL2. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 387–405. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_23
11. Belder, T., ter Beek, M.H., de Vink, E.P.: Coherent branching feature bisimulation. In: Atlee, J.M., Gnesi, S. (eds.) Proceedings 6th Workshop on Formal Methods and Analysis in SPL Engineering, FMSPLE@ETAPS 2015, London, UK, 11 April 2015. EPTCS, vol. 182, pp. 14–30 (2015). https://doi.org/10.4204/EPTCS.182.2
12. Ben-David, S., Hrubeš, P., Moran, S., Shpilka, A., Yehudayoff, A.: Learnability can be undecidable. Nat. Mach. Intell. **1**(1), 44–48 (2019). https://doi.org/10.1038/s42256-018-0002-3
13. 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, SAC 2014, pp. 1272–1278. ACM Press (2014). https://doi.org/10.1145/2554850.2554949
14. 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
15. Beohar, H., Mousavi, M.: Spinal test suites for software product lines. ArXiv e-prints (2014)
16. Beohar, H., Varshosaz, M., Mousavi, M.R.: Basic behavioral models for software product lines: expressiveness and testing pre-orders. Sci. Comput. Program., July 2015. http://www.sciencedirect.com/science/article/pii/S0167642315001288

17. Bertolino, A., Fantechi, A., Gnesi, S., Lami, G.: Product line use cases: scenario-based specification and testing of requirements. In: Käköla, T., Duenas, J.C. (eds.) Software Product Lines, pp. 425–445. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-33253-4_11
18. Bertolino, A., Gnesi, S.: PLUTO: a test methodology for product families. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 181–197. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24667-1_14
19. 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, ICSE 2016, pp. 789–792. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2889160.2889212
20. 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. Softw. Test. Verification Reliab. **21**(2), 75–100 (2011). https://doi.org/10.1002/stvr.413
21. 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
22. Cichos, H., Oster, S., Lochau, M., Schürr, A.: Model-based coverage-driven test suite generation for software product lines. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 425–439. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_31
23. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with SNIP. STTT **14**(5), 589–612 (2012)
24. 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
25. 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
26. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: ICSE 2011, pp. 321–330. ACM (2011)
27. 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 2010, pp. 335–344. ACM (2010)
28. Cohen, M., Dwyer, M., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. IEEE Trans. Software Eng. **34**(5), 633–650 (2008)
29. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. In: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis - ROSATEA 2006, pp. 53–63 (2006). http://portal.acm.org/citation.cfm?doid=1147249.1147257
30. Cordy, M., Classen, A., Perrouin, G., Heymans, P., Schobbens, P.Y., Legay, A.: Simulation-based abstractions for software product-line model checking. In: ICSE 2012, pp. 672–682. IEEE (2012)
31. 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, 2–9 June 2012, Zurich, Switzerland, pp. 672–682. IEEE Computer Society (2012)

32. 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, 16–22 November 2014, pp. 190–201. ACM (2014)
33. Cordy, M., Legay, A., Lazreg, S., Collet, P.: Towards sampling and simulation-based analysis of featured weighted automata. In: FORMALISE@ICSE 2019, pp. 61–64 (2019)
34. 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 2012, Salvador, Brazil, 2–7 September 2012, vol. 1, pp. 66–75. ACM (2012)
35. 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, vol. 2, pp. 74–81. ACM (2012)
36. Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A.: ProVeLines: a product-line of verifiers for software product lines. In: SPLC 2013, pp. 141–146. ACM (2013)
37. 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, SPLC 2008, pp. 22–31. IEEE Computer Society, Washington, DC, USA (2008). https://doi.org/10.1109/SPLC.2008.49
38. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: there and back again. In: SPLC 2007, pp. 23–34. IEEE Computer Society (2007)
39. Devroey, X., et al.: Statistical prioritization for software product line testing: an experience report. Softw. Syst. Model. **16**(1), 153–171 (2017). http://link.springer.com/10.1007/s10270-015-0479-8
40. 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 2016, pp. 89–96. ACM Press, Salvador, Brazil, January 2016
41. 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, 14–22 May 2016, pp. 655–666. ACM (2016). https://doi.org/10.1145/2884781.2884821
42. 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 2015, vol. 2, pp. 817–818. IEEE, Florence, Italy, May 2015. https://doi.org/10.1109/ICSE.2015.263, http://ieeexplore.ieee.org/document/7203084/
43. Dimovski, A.S., Legay, A., Wasowski, A.: Variability abstraction and refinement for game-based lifted model checking of full CTL. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 192–209. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_11
44. Dimovski, A.S., Wąsowski, A.: From transition systems to variability models and from lifted model checking back to UPPAAL. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 249–268. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_13

45. Dimovski, A.S., Wąsowski, A.: Variability-specific abstraction refinement for family-based model checking. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 406–423. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_24
46. Droste, S., Jansen, T., Wegener, I.: On the analysis of the (1+1) evolutionary algorithm. Theoret. Comput. Sci. **276**(1), 51–81 (2002). http://www.sciencedirect.com/science/article/pii/S0304397501001827
47. Dubslaff, C., Klüppelholz, S., Baier, C.: Probabilistic model checking for energy analysis in software product lines. In: 13th International Conference on Modularity, MODULARITY 2014, Lugano, Switzerland, 22–26 April 2014, pp. 169–180. ACM (2014)
48. 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.) CAiSE 2012. LNCS, vol. 7328, pp. 613–628. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31095-9_40
49. Fantechi, A., Gnesi, S.: A behavioural model for product families. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, 3–7 September 2007, pp. 521–524 (2007)
50. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empir. Softw. Eng. **16**(1), 61–102 (2011)
51. Goodfellow, I., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: International Conference on Learning Representations (2015). http://arxiv.org/abs/1412.6572
52. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 113–131. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68863-1_8
53. Guo, J., et al.: SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines. Softw. Syst. Model., July 2017. https://doi.org/10.1007/s10270-017-0610-0
54. 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. Empir. Softw. Eng. **24**(2), 674–717 (2019). https://doi.org/10.1007/s10664-018-9635-4
55. Hemmati, H., Arcuri, A., Briand, L.: Achieving scalable model-based testing through test case diversity. ACM Trans. Softw. Eng. Methodol. **22**(1), 1–42 (2013). http://dl.acm.org/citation.cfm?id=2430536.2430540
56. 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, vol. 1, ICSE 2015, pp. 517–528. IEEE Press, Piscataway, NJ, USA (2015). http://dl.acm.org/citation.cfm?id=2818754.2818819
57. 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 Trans. Softw. Eng. **40**(7), 650–670 (2014)
58. 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)

59. 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 2012, vol. 1, p. 46. ACM Press (2012)
60. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-21, Carnegie Mellon University (1990)
61. Kästner, C., et al.: Toward variability-aware testing. In: Proceedings of the 4th International Workshop on Feature-Oriented Software Development, FOSD 2012, pp. 1–8. ACM Press (2012). http://doi.acm.org/10.1145/2377816.2377817
62. Kim, C.H.P., Bodden, E., Batory, D., Khurshid, S.: Reducing configurations to monitor in a software product line. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 285–299. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_22
63. 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, vol. 1, SPLC 2014, pp. 247–251. ACM Press (2014). http://doi.acm.org/10.1145/2648511.2648539
64. Kuhn, D., Wallace, D., Gallo, A.: Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. **30**(6), 418–421 (2004)
65. Lazreg, S., Cordy, M., Collet, P., Heymans, P., Mosser, S.: Multifaceted automated analyses for variability-intensive embedded systems. In: ICSE 2019, pp. 854–865 (2019)
66. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_11
67. 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. Empir. Softw. Eng., pp. 1–51 (2017). https://doi.org/10.1007/s10664-016-9494-9
68. Lochau, M., Lity, S., Lachmann, R., Schaefer, I., Goltz, U.: Delta-oriented model-based integration testing of large-scale systems. J. Syst. Softw. **91**, 63–84 (2014). https://doi.org/10.1016/j.jss.2013.11.1096. http://linkinghub.elsevier.com/retrieve/pii/S0164121213002781
69. Lochau, M., Oster, S., Goltz, U., Schürr, A.: Model-based pairwise testing for feature interaction coverage in software product line engineering. Software Qual. J. **20**(3–4), 567–604 (2012). http://www.springerlink.com/index/10.1007/s11219-011-9165-4
70. Lochau, M., Schaefer, I., Kamischke, J., Lity, S.: Incremental model-based testing of delta-oriented software product lines. In: Brucker, A.D., Julliand, J. (eds.) TAP 2012. LNCS, vol. 7305, pp. 67–82. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30473-6_7
71. 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)
72. Luthmann, L., Stephan, A., Bürdek, J., Lochau, M.: Modeling and testing product lines with unbounded parametric real-time constraints. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume A, SPLC 2017, pp. 104–113. ACM, New York, NY, USA (2017). http://doi.acm.org/10.1145/3106195.3106204

73. Mathur, A.P.: Foundations of Software Testing. Pearson Education, India (2008)
74. McGregor, J.D.: Testing a software product line. In: Borba, P., Cavalcanti, A., Sampaio, A., Woodcook, J. (eds.) PSSE 2007. LNCS, vol. 6153, pp. 104–140. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14335-9_4
75. Mendonca, M., Branco, M., Cowan, D.: S.P.L.O.T.: Software product lines online tools. In: Proceedings of OOPSLA 2009, pp. 761–762. ACM, New York, NY, USA (2009). http://doi.acm.org/10.1145/1639950.1640002
76. Nebut, C., Pickin, S., Le Traon, Y., Jézéquel, J.M.: Automated requirements-based generation of test cases for product families. In: 2003 Proceedings of 18th IEEE International Conference on Automated Software Engineering, pp. 263–266. IEEE (2003)
77. 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. ACM Press (2014). http://dl.acm.org/citation.cfm?doid=2568225.2568300
78. Olaechea, R., Atlee, J., Legay, A., Fahrenberg, U.: Trace checking for dynamic software product lines. In: Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2018, pp. 69–75. ACM (2018)
79. 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, 16–23 September 2016, pp. 109–118. ACM (2016)
80. 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, pp. 109–118. ACM, New York, NY, USA (2016). http://doi.acm.org/10.1145/2934466.2934473
81. Oster, S., Markert, F., Ritter, P.: Automated incremental pairwise testing of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 196–210. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15579-6_14
82. 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, 14–22 May 2016, pp. 28–31. ACM (2016). https://doi.org/10.1145/2897045.2897051
83. Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., le Traon, Y.: Pairwise testing for software product lines: comparison of two approaches. Softw. Qual. J. **20**(3–4), 605–643 (2011). http://dx.doi.org/10.1007/s11219-011-9160-9
84. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer, Heidelberg (2005). https://doi.org/10.1007/3-540-28901-1
85. 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)
86. Rodrigues, G.N., et al.: 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, 8–10 January 2015, pp. 173–180 (2015)
87. 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 2013, pp. 492–501 (2013)

88. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Feature diagrams: a survey and a formal semantics. In: RE 2006, pp. 139–148 (2006)
89. Ter Beek, M., Legay, A., Lluch Lafuente, A., Vandin, A.: A framework for quantitative modeling and analysis of highly (re)configurable systems. IEEE Trans. Softw. Eng., p. 1 (2018). https://doi.org/10.1109/TSE.2018.2853726
90. Tian, Y., Pei, K., Jana, S., Ray, B.: DeepTest: automated testing of deep-neural-network-driven autonomous cars. In: ICSE 2018, pp. 303–314. ACM, New York, NY, USA (2018)
91. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78917-8_1. http://www.springerlink.com/index/y390356226x154j0.pdf
92. Tretmans, J.: Model-based testing and some steps towards test-based modelling. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 297–326. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_9
93. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco (2007)
94. Uzuncaova, E., Khurshid, S., Batory, D.: Incremental test generation for software product lines. IEEE Trans. Softw. Eng. **36**(3), 309–322 (2010)
95. Vanhecke, J., Devroey, X., Perrouin, G.: AbsCon : a test concretizer for model-based testing. In: 2019 IEEE Twelfth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), A-MOST 2019. IEEE, Xi'an, China (2019)
96. Vojnar, T., Zhang, L. (eds.): TACAS 2019. LNCS, vol. 11428. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1