



## On Featured Transition Systems

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

► **To cite this version:**

Gilles Perrouin, Patrick Heymans, Axel Legay, Xavier Devroey, Maxime Cordy, et al.. On Featured Transition Systems. SOFSEM 2017 - 43rd International Conference on Current Trends in Theory and Practice of Informatics, Jan 2017, Limerick Ireland. <hal-01640267>

**HAL Id: hal-01640267**

**<https://hal.inria.fr/hal-01640267>**

Submitted on 20 Nov 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# On Featured Transition Systems

Axel Legay<sup>1</sup>, Gilles Perrouin<sup>2</sup>, Xavier Devroey<sup>2</sup>, Maxime Cordy<sup>2</sup>, Pierre-Yves Schobbens<sup>2</sup>, and Patrick Heymans<sup>2</sup>

<sup>1</sup> INRIA Rennes Bretagne Atlantique, France  
axel.legay@inria.fr

<sup>2</sup> PReCISE Research Center, Faculty of Computer Science,  
University of Namur, Belgium  
{xavier.devroey, maxime.cordy, gilles.perrouin,  
pierre-yves.schobbens,patrick.heyman}@unamur.be

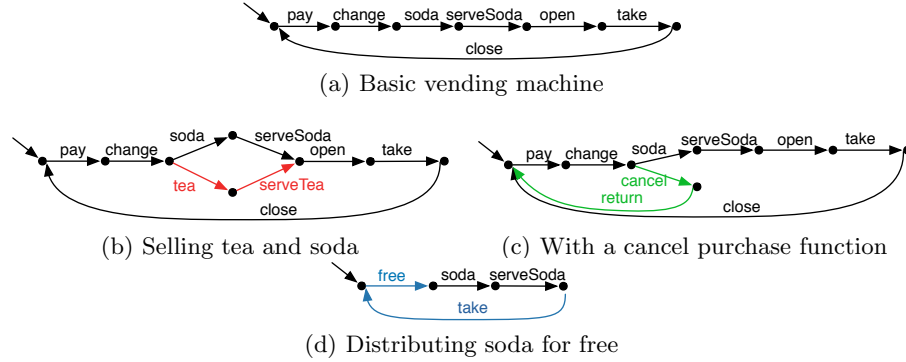
**Abstract.** Software Product Lines (SPLs) are families of similar software products built from a common set of features. As the number of products of an SPL is potentially exponential in the number of its features, analysing SPLs is harder than for single software. In this invited paper, we synthesise six years of efforts in alleviating SPL verification and testing issues. To this end, we introduced Featured Transition Systems (FTS) as a compact behavioural model for SPLs. Based on this formalism, we designed verification algorithms and tools allowing to check temporal properties on FTS, thereby assessing the correct behaviour of all the SPL products. We also used FTS to define test coverage and generation techniques for model-driven SPLs. We also successfully employed the formalism in order to foster mutation analysis. We conclude with future directions on the development of FTS for SPL analysis.

## 1 The Software Product Line Challenge

Software product line engineering (SPLE) is an increasingly popular development paradigm for highly customizable software. SPLE allows companies to achieve economies of scale by developing several similar systems together.

SPLE is now widely embraced by the industry, with applications in a variety of domains ranging from embedded systems (e.g., automotive, medical), system software (e.g., operating systems) to software products and services (e.g., e-commerce, finance). However, the benefits of SPLE come at the cost of added complexity: the (potentially large) number of systems to be considered at once, and the need for managing their variability in all activities and artifacts.

This added complexity also applies to the verification of the products' behaviour. A simple but cumbersome approach for product line verification consists in applying classical model checking algorithms [36] on each individual product of the family. However, for an SPL with  $n$  features, this would lead to  $2^n$  calls of the model checking algorithm. This solution is clearly unsatisfactory and should be replaced by new approaches that take the variability within the family into account. Those approaches often rely on compact mathematical representations



**Fig. 1.** Several variants of a vending machine.

on which a specialized model checking algorithm can be applied. The main difficulties are (1) to develop such a model checking algorithm, and (2) to propose mathematical structures that are compact and flexible enough to take the variability of the family and its specification into account.

In [10], we introduced *Featured Transition Systems* (FTS), an extension of transition systems used to represent the behaviour of all the products of an SPL in a single compact structure. We also showed how this representation can be exploited to perform model checking of product lines in an efficient way. In the rest of this paper, we briefly re-introduce FTS and summarize existing model checking algorithms for them. We also briefly show that FTS can be exploited to perform testing of software product lines. This is only a brief summary of the work that is presented at SOFSEM'17. More details can be found in our different papers cited below. Finally, we have to highlight that related work on product-line verification is vast and varied. To the best of our knowledge, effort in compiling related work on this topic can be found in the theses of Classen [5] and Cordy [11]. Beohar *et al.* recently compared the expressiveness of different SPL formalisms and found that FTS is the most expressive one [4].

## 2 Featured Transition Systems

Let us introduce Featured Transition Systems with a classical vending machine example. The example is a short version of the one we presented in [8]. In its basic version, the vending machine takes a coin, returns change, serves soda, and eventually opens a compartment so that the customer can take her soda, before closing it again. This behaviour is modelled by the transition system shown in Figure 1(a). There exist other variants of this vending machine. As an example, consider a machine that also sells tea, shown in Figure 1(b). Another variant lets the customer cancel her purchase after entering a coin, see Figure 1(c). A fourth one offers free drinks and has no closing beverage compartment, see Figure 1(d). This variability hints that the vending machines could be developed as an SPL,

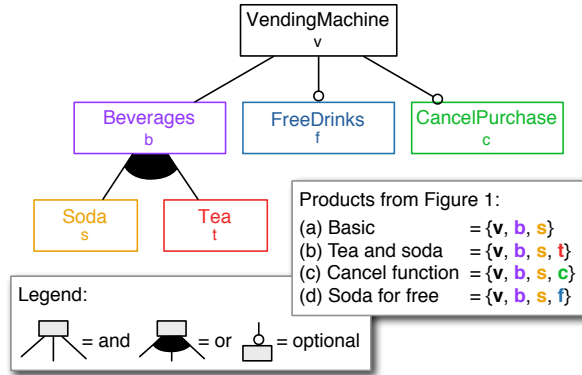


Fig. 2. FD for the vending machines of Figure 1.

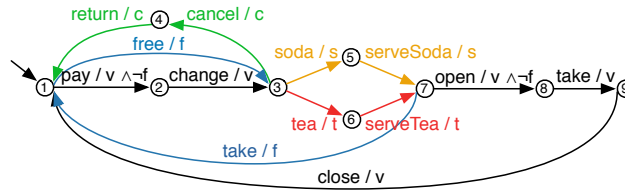


Fig. 3. FTS of the vending machine.

of which four features can be already identified: the sale of soda, the sale of tea, the ability to cancel a purchase and the ability to offer drinks for free.

By combining these features differently, yet other vending machines can be obtained. However, not every combination of features yields a valid system (e.g., a vending machine should at least sell a beverage). One can use variability models to represent the sets of valid products. In SPLE, feature diagrams [30, 35] are the most common incarnation of variability models. The feature diagram for the vending machine SPL is shown in Figure 2. This feature diagram formally describes a set of vending machines; twelve of them. A model of the behaviour of a small example such as this would already require twelve, largely identical, behavioural descriptions, four of which are shown in Figure 1.

FTS are meant to represent the behaviour of the myriad instances of an SPL in a single transition system. In fact, the main ingredient of FTS is to associate transitions with features that condition their existence. Consider again our vending machine example. Figures 1(b) and 1(c) show the impact of adding features *Tea* and *CancelPurchase* to a machine serving only soda: both add two transitions. *FreeDrinks* replaces  $\langle 1 \text{ pay} / v \wedge f \rangle \langle 2 \text{ change} / v \rangle \langle 3 \text{ free} / f \rangle$  and  $\langle 7 \text{ open} / v \wedge f \rangle \langle 8 \text{ close} / v \rangle \langle 1 \text{ take} / v \rangle$  by  $\langle 7 \text{ take} / v \rangle \langle 1 \text{ close} / v \rangle$ . The FTS of the whole vending machine SPL is given in Figure 3. The feature label of a transition is shown next to its action label, separated by a slash. In these labels (and by convenience in the refs of this paper), we use the abbreviated feature names from Figure 2. The transitions are coloured in the same way as the features in Figure 2.

### 3 Verifying SPLs with FTS

Over the years, we have developed a series of model checking algorithms that exploit the compact structure of FTS to verify sets of requirements on product lines. We first recap the meaning for product line requirements, and then briefly summarise our results.

#### 3.1 What are Product Lines requirements?

The requirements of an SPL are requirements imposed over a subset of its products. As such, they can be represented as a formula in temporal logic preceded by a Boolean formula over the SPL features, which represents the set of products whose behaviour must satisfy the temporal formula. As an example, in single systems one can check that “the system can never reach a bad state”. The product-line counterpart of this property would be: “all valid products can never reach a bad state”. The objective of an SPL verification algorithm is thus to discover all products that do not satisfy a given property, and a proof of violation (i.e. a counterexample) for each of them. One can also extend our queries to quantitative, real-time or even stochastic requirements. For example, the single-product property “is the probability to satisfy the safety requirement greater than 0.5” becomes “what are the products for which the probability to satisfy the safety requirement is greater than 0.5” in the product-line realm.

#### 3.2 How to exploit the FTS structure to model check requirements: a sketch

Let us now illustrate how one can exploit the FTS structure to reason on a classical verification problem: finding all the reachable states. Consider again the vending machine FTS of Figure 3. State ① is an initial state, and thus reachable by all products. From there, the transition  $\textcircled{1} \xrightarrow{\text{pay}} \textcircled{2}$  can only be fired by products in  $\llbracket v \wedge \neg f \rrbracket$ . Transition  $\textcircled{2} \xrightarrow{\text{change}} \textcircled{3}$  can be fired for all products in  $\llbracket v \rrbracket$ , and so state ③ is reachable by the same products as state ②. Proceeding in this way, we compute in one step the reachability relation of  $s$  for all the products. The presence of the feature diagrams permits us to ignore products that are not part of the product line. We also observe that if we find a state  $s$  reachable by a set of products  $A$  and discover later that  $s$  can also be reached by a set of product  $B \subseteq A$ , then it is enough to consider the superset  $A$ .

Considering sets of products during the verification makes us move from an enumerative approach to a product line approach, which benefits from the common behaviour of the products. Interestingly, the theoretical complexity of our algorithms is higher than that of their enumerative counterpart. However, due to the structure of the FTS, experiments show that in practice the former is faster (see, e.g., [8] for extended comparisons). Observe that the efficiency of our approach also rely on efficient representation of sets of products. In [8], we showed that the best way is to represent them with Boolean formulas. We also showed that our approach remains efficient in quantitative settings, e.g. when properties are real-time [15] or in case the features are not Boolean [16].

### 3.3 Summary of our results

Let us now briefly summarize the results we have obtained over those six last years. Our first algorithms have mainly focused on extending model checking properties of Linear Temporal Logic (LTL) to FTS [8,10]. We then moved to CTL and symbolic algorithms [9]. We have then proposed extensions of FTS that allow us to reason on more quantitative aspect of systems. This includes real time to specify timing constraints in a timed automaton fashion [15], and probabilities that allows us to make quantitative hypotheses via a combination of FTS and Markov chains [33]. Behavioural relations such as simulation were also extended to FTS. There, one tries to compute the set of products for which two states are in simulation [12]. This allows us, among many other possibilities, to define a CEGAR-based abstraction for FTS [14]. In all those algorithms, the root has always been to efficiently represents pairs of (state,product).

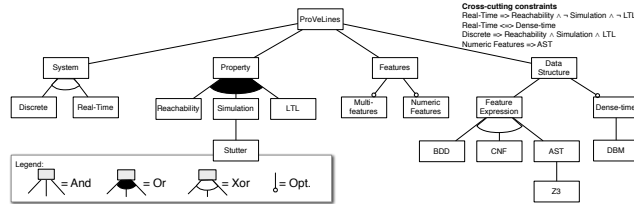


Fig. 4. Features of Provelines

*Tool* Some of our results have been implemented in ProVeLines: a product line of verification tools for QA on different types of product lines<sup>3</sup>. The structure of the tool is that of an SPL, whose corresponding feature diagram is presented in Figure 4 (taken from [17]). One can observe that the tool provides several opportunities to describe both systems (discrete, real-time) and requirements (reachability, simulation, LTL). The constraints on the top left of Figure 4 informs us that using the real-time specification for systems disables the possibility to use LTL and simulation algorithms. Otherwise, this would require the use of dense-time verification algorithms.

Any ProVeLines variant requires at least two artefacts from the user: an FD and an fPromela model. For the former, we use TVL [6,16], one of the latest incarnations of FDs, due to some of its advantages: high expressiveness, formal semantics and tool support. fPromela is a feature-oriented extension of Promela [27], which we defined as a high-level language on top of FTS. An fPromela model thus describes the behaviour of all the products defined by the FD [7,16].

<sup>3</sup> Note that prototype tools exist for other results we developed.

## 4 SPL Testing with FTS

FTSs, as concise models of SPLs behaviours, can also support model-based testing (MBT) activities. In our research we investigated two directions: *i*) coverage and *ii*) synergies between SPL testing and mutation analysis.

### 4.1 SPL Coverage Analysis

**Extending Usual Coverage Criteria.** Since FTS are extensions of transition systems, a natural research direction was to consider “usual” coverage criteria (e.g., all-states, all-transitions) for product-line test generation [21]. In our work, we modelled test cases in terms of sequences of actions. There are thus *abstract* by nature since in the FTS formalism, actions are simple labels without any input or output. Additionally, an Abstract Test Case (ATC) may not be executable. As we have seen, each transition can only be executed by the set of products that match the associated feature expression. If we consider a sequence of actions, we have to conjunct these feature expressions and check the satisfiability of the resulting expression to know which product(s) can execute this abstract test case. If the formula is not satisfiable, there is no product that can execute the behaviour described in this abstract test case. For example,  $ATC = \{pay, change, tea, serveTea, take\}$  leading to the run  $\textcircled{1}pay \rightarrow \textcircled{2}change \rightarrow \textcircled{3}tea \rightarrow \textcircled{6}serveTea \rightarrow \textcircled{7}take \rightarrow \textcircled{1}$ , is executable by products in  $\llbracket v \wedge \neg f \wedge v \wedge t \wedge t \wedge f \rrbracket$ , which in turn trivially maps to the empty set. Such a negative test case can be useful to ensure whether an implementation does not allow more products than specified.

Thus, to be executable, an ATC can be executed by at least one product of the product line. We then extend this definition to executable test suites, by stating that they should contain only executable test cases. Equipped with such notions, we can define product-line coverage as a function that takes a FTS and abstract test suite as parameters and returns a value between 0 and 1. This value represents the ratio between the number of actually covered elements (states, transitions, etc.) and the number of possible ones in the FTS, if the value is 1 then we obtain all-X coverage, where X is the set of elements under consideration for this coverage criterion. When such elements involve transitions, we impose that these transitions are executable by at least one product (see [21] for formal definitions). In our coffee machine, the following test suite both satisfies all-states and all-transitions coverage:

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

We also experimented using another criteria that is not based on the model structure but on the capture of usage model that describes usages of the system [19]. There are two ways to capture such usages: either by extracting them from logs (such as Apache logs) [20], and assign more importance to more frequent usages or by assigning them directly using a dedicated modelling tool such as

MaTeLO [2]. Technically, these usage models take the form of a Markov chain that can be used to derive the most frequent test cases. There are then run on the FTS to derive the associated product-line coverage metrics. This scenario complements the one proposed by Samih *et al.* [34] who start by selecting a product prior to generate test cases using statistical testing techniques [37]. For more information about these dual scenarios, see [19].

Another interesting aspect that differs from “usual” coverage for single systems is the notion of *P-coverage*. P-coverage represent the ratio between the set of products executable by a given abstract test suite and the set of products derivable in the feature diagram that is  $\llbracket FD \rrbracket$ . Since ATCs relates the two types of coverage (products and their behaviours), their generation is de-facto a *multi-objective* problem. The compactness of the FTS formalism makes it easy for the SPL testing community to study different multi-objective scenarios and compare different criteria.

**Multi-objective Coverage.** Continuing previous line of work that considered coverage only at the structural (feature diagram) level [25, 26, 32], we initially started with a rather strange question: “what is the behavioural coverage of structural coverage ?” [22]. The idea behind this question is that as some behavioural coverage criteria may be difficult to compute in practice because of their complexity, approximating them with less computationally expensive approaches at the feature diagram level can be of interest. To investigate this question, we measured the behavioural coverage (state, transitions and actions) of two FD coverage criteria: (i) pairwise coverage [29, 32] that covers any two combination of features and (ii) similarity coverage that maximises distances between configurations [1, 25]. Results [22] shown that it was indeed possible to cover large parts of behaviour by sampling few configurations (e.g. only 2 products were necessary to achieve all-transitions coverage for the Claroline SPL allowing more than 5,000,000 products). Nevertheless, the resulting test suites are not optimal and more experiments are needed to generalise our results.

Recently, we considered extending similarity at the behavioural level to design search algorithms that maximize both distances between configurations at the FD level and distance between test cases [23]. We considered various distances (Hamming, Anti-Dice, Jaccard, Levhenstein) and both single objective (operating on an initial random set of test cases) and bi-objective (also taking into account distance between products). We seeded our models with random faults to compare the various algorithms. In our models, being bi-objective is not necessary an advantage, and the efficiency seems largely influenced by the choice of the distance function we make. A threat to validity to these conclusions is the fact that our feature diagrams are not heavily constrained, favouring the accidental discovery of dissimilar products.

## 4.2 Mutation Analysis.

A less expected application of FTSs in the field of software testing is mutation analysis [18, 23]. Mutation analysis (see [28] for a comprehensive survey) is a



technique that assess the quality of test suites by mutating software artifacts (program, models) called *mutants* and measuring the ability of such test suites to distinguish them from the original (we also say that a test case *kills* a mutant) ones. The underlying idea is to mimic a “competent programmer” that would introduce a few mistakes in the implementation of a system. The mutation score measures the ratio of the number of killed mutants divided by the number total mutants for a given test suite. The contribution of FTS to mutation testing is first to model mutants as families [18] and then to exploit the FTS formalism to perform shared execution of the mutants “all-at-once” to speed up analysis [23].

**Mutants as SPL Variants.** We studied mutation analysis at the model level, where the original system and its mutants can be expressed as transition systems. Model-based mutation complements program-based mutation as they tend to exercise different faults. To generate mutants automatically we design so-called mutation operators. For a transition system, these operators are model transformations that for example remove a transition or replace an action by another one. As we have seen, the features in a FTS add or remove transitions in a similar way. Building on this analogy, we sketched a vision of managing (model) mutants as a SPL to bring all the advantages of FTS and variability modelling to mutation analysis [18]. We describe mutations as features and organise them in a feature diagram, which allows a precise control on the type and number of mutants we allow for analysis. From a behavioural perspective, all the mutants are represented in a centralised model (the FTS), which eases their management and storage.

**Accelerating Mutation Analysis.** As noted by Jia and Harman [28], one of the practical obstacles to the development of mutation testing is the cost associated to mutation analysis. Traditional mutation testing proceeds by running every test case on every mutant. Since we need to have a large number of mutants to assess test suites’ sensitivity in a meaningful way, analysis time can be huge. In fact, this is equivalent as processing all mutants in isolation like the naive approach is doing for product line model-checking. Of course, an important justification of using the FTS formalism to model mutations is to avoid this naive approach and perform family-based mutation analysis [18]. We implemented this featured model-based mutation analysis recently [23]. The *Featured Mutant Model* (FMM) is thus comprised of a FTS modelling the mutant family and a feature diagram representing all the mutations supported by this family. To perform mutation analysis, we simply run test cases on the FTS. As we have seen, this yields a boolean formula describing all the mutants (in  $\llbracket FD \rrbracket$ ) that are killed by this test case. Therefore, we only need to run each test case once on the FTS, rather than on the  $2^n$  individual transition systems associated to this mutant family. Our experiments showed gains between 2.5 and 1,000 times than previous approaches. Additionally, the FMM favours *higher-order mutation*. Higher-order mutation consists in applying several mutation operators on the same model. In the FMM scheme, higher-order mutation is supported allow-

ing certain features to be selected together in a mutant. If we want to restrict ourselves to the first order, we then need to specify that each feature excludes all the other ones. Computing the mutation score require enumerating the instances satisfying the union of formulas gathered for each test case and computing the total number of mutants from the feature diagram. Computing such values may be tricky for large models (even with BDD solvers) and optimisations require to be investigated [23].

### 4.3 ViBES: A model-based framework for SPL testing.

All our research on model-based testing has been integrated in a framework called ViBES [24]. We designed an XML representation for FTS while the feature diagrams are encoded in TVL [6]. The framework is implemented in JAVA and provides a domain-specific language to create mutations operators and mutant families in a programmer-friendly way. The framework also contains the implementations of test coverage and generation techniques discussed above. Finally, the framework is open-source (MIT Licence) and can be downloaded here: <https://projects.info.unamur.be/vibes/>.

## 5 Conclusion

In this paper, we summarised six years of efforts in harnessing the central problem of SPL analysis: the combinatorial explosion of the number of products to consider. To this end, we introduced featured transition systems as a compact and efficient representation of the whole behaviour of a SPL. This unique representation of all the products served as a support to a family of verification algorithms itself implemented as a software product-line [17]. We also employed FTS for model-based testing activities such as coverage and test generation and prioritisation. The FTS formalism demonstrated its universality to readily be applide for mutation analysis of single systems, with substantial analysis speedups.

After having had a look on the past, let us have a look in the future. There are several research directions worth of investigation. First we would like to extend our verification algorithms to quantitative software product lines. This requires to extend the FTS formalism to specify quantities [31]. Another interesting information to specify in FTS is probabilities, in order to perform statistical model-checking activities [33]. Such extended FTS formalism is also of interest for testing [19, 20]. As is the addition of inputs and outputs for ioco conformance [3]. With respect to mutation, we would like to formally investigate the mutant equivalence problem using exact or approximate simulation techniques [13].

## References

1. M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. Similarity-based prioritization in software product-line testing. In *SPLC*, pages 197–206. ACM, 2014.

2. ALL4TEC - MaTeLo. <http://all4tec.net/index.php/en/model-based-testing/20-markov-test-logic-matelo>.
3. H. Beohar and M. Mousavi. Input-output conformance testing based on featured transition systems. In *29th Annual ACM Symposium on Applied Computing*, pages 1272–1278. ACM, 2014.
4. H. Beohar, M. Varshosaz, and M. R. Mousavi. Basic behavioral models for software product lines: Expressiveness and testing pre-orders. *Science of Computer Programming*, 123:42 – 60, 2016.
5. A. Classen. *Modelling and Model Checking Variability-Intensive Systems*. PhD thesis, University of Namur (FUNDP), 2011.
6. A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *SCP*, 76:1130–1143, December 2011.
7. A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with SNIP. *STTT*, 14(5):589–612, 2012.
8. A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. cois Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. *Transactions on Software Engineering (in press)*, 2013.
9. A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *ICSE’11*, pages 321–330. ACM, 2011.
10. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE’10*, pages 335–344. ACM, 2010.
11. M. Cordy. *Model Checking for the Masses*. PhD thesis, University of Namur, 2014.
12. M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Managing evolution in software product lines : A model-checking perspective. In *VaMoS’12*, pages 183–191. ACM, 2012.
13. M. Cordy, A. Classen, G. Perrouin, P. Heymans, P.-Y. Schobbens, and A. Legay. Simulation-based abstractions for software product-line model checking. In *ICSE’12*, pages 672–682. IEEE, 2012.
14. M. Cordy, P. Heymans, A. Legay, P.-Y. Schobbens, B. Dawagne, and M. Leucker. Counterexample guided abstraction refinement of product-line behavioural models. In *FSE’14*. ACM, 2014.
15. M. Cordy, P. Heymans, P.-Y. Schobbens, and A. Legay. Behavioural modelling and verification of real-time software product lines. In *SPLC’12*. ACM, 2012.
16. M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *ICSE’13*, pages 472–481. IEEE, 2013.
17. M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Provelines: A product-line of verifiers for software product lines. In *SPLC’13*, pages 141–146. ACM, 2013.
18. X. Devroey, G. Perrouin, M. Cordy, M. Papadakis, A. Legay, and P. Schobbens. A variability perspective of mutation analysis. In *SIGSOFT FSE*, pages 841–844. ACM, 2014.
19. X. Devroey, G. Perrouin, M. Cordy, H. Samih, A. Legay, P.-Y. Schobbens, and P. Heymans. Statistical prioritization for software product line testing: an experience report. *Software & Systems Modeling*, pages 1–19, 2015.
20. X. Devroey, G. Perrouin, M. Cordy, P. Schobbens, A. Legay, and P. Heymans. Towards statistical prioritization for software product lines testing. In *VaMoS*, pages 10:1–10:7. ACM, 2014.

21. X. Devroey, G. Perrouin, A. Legay, M. Cordy, P. Schobbens, and P. Heymans. Coverage criteria for behavioural testing of software product lines. In *ISoLA (1)*, volume 8802 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2014.
22. X. Devroey, G. Perrouin, A. Legay, P. Schobbens, and P. Heymans. Covering SPL behaviour with sampled configurations: An initial assessment. In *VaMoS*, page 59. ACM, 2015.
23. X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, and P. Heymans. Featured model-based mutation analysis. In *ICSE*, pages 655–666. ACM, 2016.
24. X. Devroey, G. Perrouin, P. Schobbens, and P. Heymans. Poster: Vibes, transition system mutation made easy. In *ICSE (2)*, pages 817–818. IEEE Computer Society, 2015.
25. C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Software Eng.*, 40(7):650–670, 2014.
26. C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Multi-objective test generation for software product lines. In *SPLC*, pages 62–71. ACM, 2013.
27. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
28. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE TSE*, 37(5):649–678, 2011.
29. M. F. Johansen, Ø. Haugen, and F. Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 638–652. Springer, 2011.
30. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990.
31. R. Olaechea, U. Fahrenberg, J. M. Atlee, and A. Legay. Long-term average cost in featured transition systems. In *Proceedings of the 20th International Systems and Software Product Line Conference, SPLC '16*, pages 109–118, New York, NY, USA, 2016. ACM.
32. G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. L. Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *ICST*, pages 459–468. IEEE Computer Society, 2010.
33. G. N. Rodrigues, V. Alves, V. Nunes, A. Lanna, M. Cordy, P. Schobbens, A. M. Sharifloo, and A. Legay. Modeling and verification for probabilistic properties in software product lines. In *HASE*, pages 173–180. IEEE Computer Society, 2015.
34. H. Samih and R. Bogusch. MPLM - MaTeLo Product Line Manager. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2, SPLC '14*, pages 138–142, New York, NY, USA, 2014. ACM.
35. P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE'06*, pages 139–148, 2006.
36. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS'86*, pages 332–344. IEEE CS, 1986.
37. J. Whittaker and G. Thomason, Michael. A Markov chain model for statistical software testing. *Software Engineering, IEEE Transactions on*, 20(10):812–824, Oct 1994.