QFLan: A Tool for the Quantitative Analysis of Highly Reconfigurable Systems

Andrea Vandin, Maurice H. ter Beek, Axel Legay and Alberto Lluch Lafuente

July 20, 2018

QFLan: A Tool for the Quantitative Analysis of Highly Reconfigurable Systems

Andrea Vandin[1(✉)], Maurice H. ter Beek[2],
Axel Legay[3], and Alberto Lluch Lafuente[1]

[1] DTU, Lyngby, Denmark
{anvan,albl}@dtu.dk
[2] ISTI–CNR, Pisa, Italy
maurice.terbeek@isti.cnr.it
[3] Inria, Rennes, France
axel.legay@inria.fr

Abstract. QFLan offers modeling and analysis of highly reconfigurable systems, like product lines, which are characterized by combinatorially many system variants (or products) that can be obtained via different combinations of installed features. The tool offers a modern integrated development environment for the homonym probabilistic feature-oriented language. QFLan allows the specification of a family of products in terms of a feature model with quantitative attributes, which defines the valid feature combinations, and probabilistic behavior subject to quantitative constraints. The language's behavioral part enables dynamic installation, removal and replacement of features. QFLan has a discrete-time Markov chain semantics, permitting quantitative analyses. Thanks to a seamless integration with the statistical model checker MultiVeStA, it allows for analyses like the likelihood of specific behavior or the expected average value of non-functional aspects related to feature attributes.

1 Introduction

Product line engineering is a methodology that aims to develop and manage, in a cost-effective and time-efficient manner, a family of products or (re)configurable system variants, to allow the mass customization of individual variants. Their variability is captured by feature models, whose features represent stakeholder-relevant functionalities or system aspects [1]. The challenge when lifting successful modeling and analysis techniques for single systems to families of products or configurable systems, is to handle their variability, due to which the number of possible variants may be exponential in the number of features. This led to so-called *family-based* analyses [32]: analyse properties on an entire product line and use variability knowledge about valid feature configurations to deduce results for individual products. This is applied in, e.g., [8,10,11,15,19–23,28,30].

In [2–4], we presented various facets of the probabilistic modeling language QFLan, capable of describing a wide spectrum of aspects of (software) product lines (SPL). The type of quantitative constraints that are supported by QFLan are significantly more complex than those commonly associated to attributed feature models [9, 18, 26]. This paper presents the QFLan tool, a multi-platform tool for the specification and analysis of QFLan models, which has been implemented in the Eclipse environment using XTEXT technology, thus obtaining a state-of-the-art integrated development environment (IDE). The tool is available, together with installation and usage instructions, from `http://github.com/qflanTeam/QFLan`.

Related work The QFLan prototypes from [2, 3] were the first tools that offered statistical model checking tailored for SPL, generating approximately correct results via sampling, which is particularly useful on very large models when exact model checking is infeasible [25]. Next to dedicated exact model checkers such as VMC [6] and the tool suite ProVeLines [17], which offers the best known SPL-specific model checker, SNIP [12], also popular model checkers like mCRL2 and SPIN have been made amenable to SPL model checking [7,8,19–21]. Furthermore, the tool ProFeat [11] extends the probabilistic model checker PRISM [24] with feature-oriented concepts to be able to model families of stochastic systems and to analyze them through probabilistic model checking. QFLan scales to larger models with respect to precise probabilistic analysis techniques. In fact, we can handle (cf. `http://github.com/qflanTeam/QFLan` and [5]) significantly larger instances of the Elevator case study. Originally introduced in [29], this case study is now a benchmark for SPL analysis known to be very demanding in terms of scalability when large sizes of Elevator systems are considered (cf., e.g., [5, 10, 11, 13, 14, 18, 19]).

Outline Sect. 2 describes the tool's architecture, while Sect. 3 applies the tool to a simple family of coffee vending machines. Sect. 4 concludes the paper.

2 QFLan Architecture

The architecture of QFLan is sketched in Fig. 1. It consists of a GUI layer and a core layer, devoted to modeling and analysis tasks, respectively.

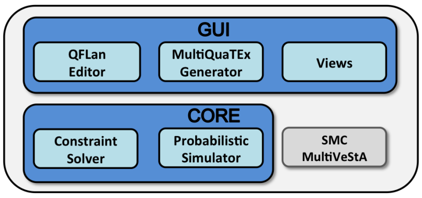

Fig. 1. QFLan's architecture

GUI layer The components of the GUI layer are depicted in Fig. 2. The QFLan editor provides state-of-the-art editing support, including auto-completion, syntax and error highlighting, and fix suggestions. This was obtained using XTEXT technology. For instance, the editor does on-the-fly error-detection on the structure of the feature model. The editor also offers support for MultiQuaTEx, the query language of the statistical model checker MultiVeStA [31] integrated in the tool. In particular, the user can specify properties to be analyzed with a high-level language consisting of QFLan ingredients only, from which MultiQuaTEx queries are automatically generated. In addition, the GUI layer offers a number of views, including the *project explorer* to navigate across different QFLan models, the tree-like *outline* to navigate in the elements of QFLan model, the *console view* to display diagnostic information, and the *plot view* to display analysis results.

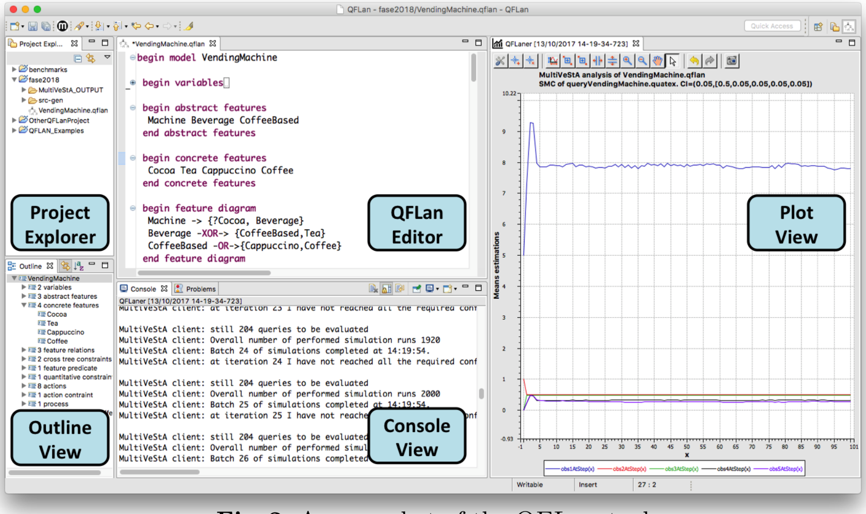

Fig. 2. A screenshot of the QFLan tool

Core layer The main component of the core layer is a probabilistic simulator. According to QFLan's semantics, each state can have a set of outgoing admissible transitions, each labeled with a weight to compute the probability distribution of the transitions outgoing from each state. This leads to a discrete-time Markov chain semantics. Starting from the initial configuration specified by the modeler, the simulator iteratively computes all one-step transitions allowed in the state, and probabilistically selects one according to the probability distribution that results from normalizing the rates of the generated transitions. In particular, to check whether a transition is admissible the tool uses an ad-hoc constraint solver to guarantee that the transition does not violate any of the constraints specified by the modeler.

In [3, 4], we presented a prototypical implementation of a QFLan simulator based on the Maude toolkit [16] and Microsoft's SMT solver Z3 [27], integrated with MultiVeStA. The mature QFLan tool presented in this paper has been redeveloped from scratch using Java-based technologies in order to obtain a multi-platform modern IDE for QFLan instead of a command-line prototype. Furthermore, this led to an analysis speedup of several orders of magnitude.

3 QFLan at Work: Coffee Vending Machine

We consider a family of vending machines inspired by examples from the literature (e.g., [2,6–8,15,28]). For illustration purposes, we consider a simple version. Larger case studies can be found at `http://github.com/qflanTeam/QFLan`, including the bike-sharing case study used as running example in [2–4], the above mentioned SPL benchmark Elevator case study used in [5] to evaluate QFLan's scalability, and a case study concerning risk analysis of a safe lock system with variability used in [5] to illustrate QFLan's applicability in a non-SPL setting.

This family of vending machines sells either tea, or the coffee-based beverages coffee, cappuccino, and cappuccino with cocoa (chocaccino). Its feature model is depicted in Fig. 3. Listing 1 shows its QFLan specification in 1:1 correspondence.

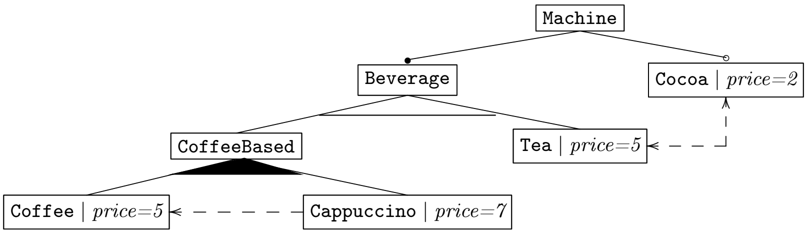

Fig. 3. Feature model of vending machine product line

Each node is a feature, while edges denote constraints defining admissible combinations of installed features. As is common in feature models, we distinguish between *concrete* and *abstract* features. The former are the tree's leaves, and can be explicitly (un)installed, whereas the latter are internal nodes, used mainly to group features. The root denotes a product, i.e. a specific vending machine. To instantiate a machine, one may install the *optional* feature `Cocoa` (its optionality is denoted by a circle in Fig. 3 and with a `?` in Listing 1), while it must contain the *mandatory* feature `Beverage` (as denoted by a filled circle in Fig. 3 and by the absence of a `?` in Listing 1). Finer constraints on the presence of features other than mandatory or optional also can be imposed. The machine may come equipped with either a `Tea` dispenser or with one for `CoffeeBased` beverages. This is specified by the `XOR` edges connecting `Beverage` to `Tea` and to `CoffeeBased`. The `CoffeeBased` dispenser can be used to pour `Coffee`, `Cappuccino`, or both, as denoted by the `OR` edges.

Features can also be subject to *cross-tree constraints*. The arrow from `Cappuccino` to `Coffee` denotes that the former *requires* the latter, the rationale being that coffee is a prerequisite for preparing cappuccino. Instead, the double-headed arrow connecting `Cocoa` and `Tea` denotes that they *exclude* each other, since cocoa only serves to prepare chocaccino. Such constraints are specified in QFLan as shown in Lines 15–18 of Listing 1. Finally, features can have quantitative predicates as attributes. In the example, all concrete features have *price* as attribute (Lines 20–22 of Listing 1). Abstract features implicitly inherit all predicates from concrete ones, with the cumulative value of all descendant concrete features actually installed.

```
1  begin abstract features
2   Machine Beverage Coffee-based
3  end abstract features
4
5  begin concrete features
6   Cocoa Tea Cappuccino Coffee
7  end concrete features
8
9  begin feature diagram
10  Machine -> {?Cocoa,Beverage}
11  Beverage -XOR-> {CoffeeBased,Tea}
12  CoffeeBased -OR-> {Cappuccino,Coffee}
13 end feature diagram
14
15 begin cross-tree constraints
16  Cappuccino requires Coffee
17  Tea excludes Cocoa
18 end cross-tree constraints
19
20 begin feature predicates
21  price = {Cocoa=2,Tea=5,Cappuccino=7,
        Coffee=5}
22 end feature predicates
23
24 begin quantitative constraints
25  price(Machine) <= 10
26 end quantitative constraints
```

Listing 1. QFLan encoding of feature model displayed in Fig. 3

Lines 24–26 of Listing 1 show that QFLan supports another family of constraints regarding feature predicates, the *quantitative constraints*, used in this case study to exclude machines with a cumulative *price* that is superior to 10.

The dynamics of our example family is sketched in Fig. 4, while Listing 2 shows a textual QFLan specification in close correspondence. The machine, initialized with a `Coffee` dispenser only, is pre-configured in the FACTORY by possibly *installing* any admissible feature configuration. After this, one can *sell* the machine to a company, in whose DEPOSIT minor customizations can be done before *deployment*. Once in OPERATION, the machine can *serve* customer requests, depending on the installed features, or be *reconfigured* in the DEPOSIT.

In QFLan, one first specifies real-valued *variables* (Lines 1–3 of Listing 2) that can be used in the guards of constraints or to facilitate the analysis phase. Variables can be updated as side effects of the execution of *actions*, defined in the `actions` block (Lines 5–8). In addition, also installed concrete features can be executed as actions, meaning a user is using them. Note that `Chocaccino` is not a feature (like `Cappuccino` or `Coffee`) but an action. The rationale is that any machine provided with `Cappuccino` and `Cocoa` dispensers can serve `Chocaccino`. This is expressed in QFLan using yet another family of constraints, the *action constraints* (Lines 10–12). Finally, Lines 14–32 specify the actual behavior in terms of a process named `dynamics` with sets of states (Line 15) and transitions (Lines 16–31), corresponding to the nodes and edges, respectively, of Fig. 4. Each transition is labeled with an action, including the (un)installation or replacement of features, and its weight, used to calculate the probability that it is executed.

The model is completed by specifying an initial configuration (Lines 34–37).

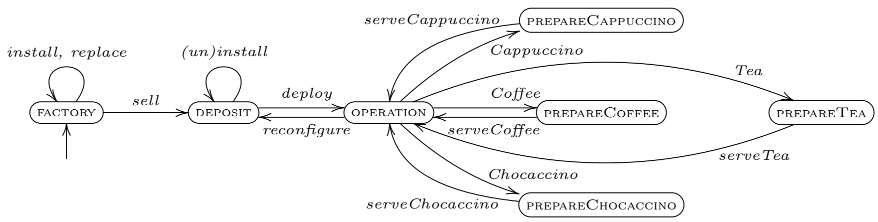

Fig. 4. Sketch of vending machine behavior

```
1  begin variables
2   sold = 0 deploys = 0
3  end variables
4
5  begin actions
6   sell deploy reconfigure chocaccino serveTea
7   serveCoffee serveCappuccino serveChocaccino
8  end actions
9
10 begin action constraints
11  do(chocaccino) -> (has(cappuccino) and has(cocoa))
12 end action constraints
13
14 begin process dynamics
15  states = factory,deposit,operation,prepareCoffee,
          prepareCappuccino,prepareTea,prepareChocaccino
16  transitions =
17   //Factory
18   factory -(replace(coffee,tea),20)-> factory,
19   factory -(install(cocoa),10)-> factory,
20   factory -(install(cappuccino),10)-> factory,
20   factory -(sell,1,{sold=1})-> deposit,
21   //Deposit
22   deposit -(install(cappuccino),2)-> deposit,
23   deposit -(uninstall(cappuccino),2)-> deposit,
24   deposit -(install(cocoa),2)-> deposit,
25   deposit -(uninstall(cocoa),2)-> deposit,
26   deposit -(deploy,2,{deploys+=1})-> operation,
27   //Operation
28   operation -(coffee,3)-> prepareCoffee,
29   prepareCoffee -(serveCoffee,1)-> operation,
30   //Tea, Cappuccino & Chocaccino are similar...
31   operation -(reconfigure,1)-> deposit
32 end process
33
34 begin init
35  installedFeatures = { coffee }
36  initialProcesses = dynamics
37 end init
```

Listing 2. QFLan encoding of vending machine behavior sketched in Fig. 4

Note that, as side-effect of executing an action (`sell`), the transition in Line 20 sets the variable `sold` to 1 when a machine is sold. Pinpointing this precise moment allows to study, e.g., the average price of sold machines, or the probability that they have dispensers for `Coffee`, `Tea`, `Cappuccino`, and `Cocoa`.

These five properties (average price and four distinct probabilities) can be expressed as in Listing 3. The `query` specifies the properties to `eval`uate in the first state that satisfies `sold == 1`. The expected value x of each property is estimated by MultiVeStA as the mean value $\overline{x}$ of n samples (obtained from n independent simulations), with n large enough such that with probability $(1-\alpha)$ we have $x \in [\overline{x} - {}^{\delta}/_{2}, \overline{x} + {}^{\delta}/_{2}]$. Default values of α and δ are provided by the user via the keywords `alpha` and `delta` (these can be overruled for a specific property by providing its new value in square brackets, cf. Line 3). Finally, keyword `parallelism = p` allows to distribute simulations across `p` processes to be allocated on different cores. The analysis of the five properties in Listing 3 with QFLan required $\pm$ 2000 simulations, for which it ran in under a second on a laptop with a 2.4 GHz Intel Core i5 processor and 4 GB of RAM.

```
1 begin analysis
2  query = eval when {sold == 1 } :
3    { price(Machine) [delta = 0.5],
4      Coffee,Tea,Cappuccino,Cocoa }
5  default delta = 0.05 alpha = 0.05
6  parallelism = 1
7 end analysis
```

Listing 3. QFLan properties

The probability of having `Cappuccino` dispensers in machines sold is zero: any machine always has at least either a `Tea` or `Coffee` dispenser, with `price` 5, so installing `Cappuccino` would violate the constraint `price(Machine) <= 10` because `Cappuccino` has `price` 7. For the same reason, the probability to have `Coffee` installed is 0.33, which is roughly $\frac{1+10}{1+10+20}$, since in Lines 18–20 of Listing 2 we see that 1, 10, and 20 are the weights assigned to `sell`, `install(Cocoa)`, and `replace(Coffee,Tea)`, respectively (the weight of installing `Cappuccino` is ignored as it is not allowed). Note that `Coffee` is installed in machines sold if `replace(Coffee,Tea)` is not executed, which happens if `sell` is executed first, or if `Cocoa` gets installed, which prevents the execution of `replace(Coffee,Tea)` due to constraint `Tea excludes Cocoa`. The probability of installing `Cappuccino` becomes 0.46 if we use the more permissive constraint `price(Machine) <= 15`.

QFLan also allows to study parametric properties as time progresses. For example, by replacing "`when {sold == 1.0}`" with "`from 0 to 100 by 1`" in Listing 3, we study the five properties at each of the first 100 simulation steps, for a total of 500 properties. Analysis results of parametric properties are visualized in interactive plots, in this case the one in Fig. 2, computed in a few seconds.

4 Outlook

We presented QFLan, a quantitative modeling and verification environment for highly (re)configurable systems, like SPL, including Eclipse-based tool support. QFLan offers an IDE for specifying system configurations and their probabilistic behavior in a high-level language as well as advanced statistical analyses of non-functional properties based on a discrete-time Markov chain semantics.

In the future, we envision a stochastic semantics based on continuous-time Markov chains for the analysis of time-related properties and a semantics based on featured transition systems to interface with the ProVeLines tool suite [17].

References

1. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer (2013). https://doi.org/10.1007/978-3-642-37521-7
2. ter Beek, M.H., Legay, A., Lluch Lafuente, A., Vandin, A.: Quantitative Analysis of Probabilistic Models of Software Product Lines with Statistical Model Checking. In: FMSPLE 2015. EPTCS, vol. 182, pp. 56–70 (2015). https://doi.org/10.4204/EPTCS.182.5
3. ter Beek, M.H., Legay, A., Lluch Lafuente, A., Vandin, A.: Statistical Analysis of Probabilistic Models of Software Product Lines with Quantitative Constraints. In: SPLC 2015. pp. 11–15. ACM (2015). https://doi.org/10.1145/2791060.2791087
4. ter Beek, M.H., Legay, A., Lluch Lafuente, A., Vandin, A.: Statistical Model Checking for Product Lines. In: ISoLA 2016. LNCS, vol. 9952, pp. 114–133 (2016). https://doi.org/10.1007/978-3-319-47166-2_8
5. ter Beek, M.H., Legay, A., Lluch Lafuente, A., Vandin, A.: A framework for quantitative modeling and analysis of highly (re)configurable systems (2018), `http://arxiv.org/abs/1707.08411`
6. ter Beek, M.H., Mazzanti, F., Sulova, A.: VMC: A Tool for Product Variability Analysis. In: FM 2012. LNCS, vol. 7436, pp. 450–454 (2012). https://doi.org/10.1007/978-3-642-32759-9_36
7. ter Beek, M.H., de Vink, E.P.: Using mCRL2 for the Analysis of Software Product Lines. In: FormaliSE 2014. pp. 31–37. ACM (2014). https://doi.org/10.1145/2593489.2593493
8. ter Beek, M.H., de Vink, E.P., Willemse, T.A.C.: Family-Based Model Checking with mCRL2. In: FASE 2017. LNCS, vol. 10202, pp. 387–405 (2017). https://doi.org/10.1007/978-3-662-54494-5_23
9. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. **35**(6), 615–636 (2010). https://doi.org/10.1016/j.is.2010.01.001
10. Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C.: Family-based modeling and analysis for probabilistic systems – featuring ProFeat. In: FASE 2016. LNCS, vol. 9633, pp. 287–304 (2016). https://doi.org/10.1007/978-3-662-49665-7_17
11. 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
12. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with SNIP. Int. J. Softw. Tools Technol. Transf. **14**(5), 589–612 (2012). https://doi.org/10.1007/s10009-012-0234-1
13. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Formal semantics, modular specification, and symbolic verification of product-line behaviour. Sci. Comput. Program. **80**(B), 416–439 (2014). https://doi.org/10.1145/2499777.2499781
14. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: ICSE 2011. pp. 321–330. ACM (2011). https://doi.org/10.1145/1985793.1985838
15. 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). https://doi.org/10.1145/1806799.1806850

16. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude—A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic, LNCS, vol. 4350 (2007). https://doi.org/10.1007/978-3-540-71999-1
17. Cordy, M., Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: ProVeLines: A Product Line of Verifiers for Software Product Lines. In: SPLC 2013. pp. 141–146. ACM (2013). https://doi.org/10.1145/2499777.2499781
18. Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A.: Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-features. In: ICSE 2013. pp. 472–481. IEEE (2013). https://doi.org/10.1109/ICSE.2013.6606593
19. Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wąsowski, A.: Efficient family-based model checking via variability abstractions. Int. J. Softw. Tools Technol. Transf. **19**(5), 585–603 (2017). https://doi.org/10.1007/s10009-016-0425-2
20. Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wąsowski, A.: Family-Based Model Checking Without a Family-Based Model Checker. In: SPIN 2015. LNCS, vol. 9232, pp. 282–299 (2015). https://doi.org/10.1007/978-3-319-23404-5_18
21. Dimovski, A.S., Wąsowski, A.: Variability-Specific Abstraction Refinement for Family-Based Model Checking. In: FASE 2017. LNCS, vol. 10202, pp. 406–423 (2017). https://doi.org/10.1007/978-3-662-54494-5_24
22. Gruler, A., Leucker, M., Scheidemann, K.D.: Modeling and Model Checking Software Product Lines. In: FMOODS 2008. LNCS, vol. 5051, pp. 113–131 (2008). https://doi.org/10.1007/978-3-540-68863-1_8
23. Kowal, M., Schaefer, I., Tribastone, M.: Family-Based Performance Analysis of Variant-Rich Software Systems. In: FASE 2014. LNCS, vol. 8411, pp. 94–108 (2014). https://doi.org/10.1007/978-3-642-54804-8_7
24. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: CAV 2011. LNCS, vol. 6806, pp. 585–591 (2011). https://doi.org/10.1007/978-3-642-22110-1_47
25. Legay, A., Delahaye, B., Bensalem, S.: Statistical Model Checking: An Overview. In: RV 2010. LNCS, vol. 6418, pp. 122–135 (2010). https://doi.org/10.1007/978-3-642-16612-9_11
26. Mauro, J., Nieke, M., Seidl, C., Yu, I.C.: Context Aware Reconfiguration in Software Product Lines. In: VaMoS 2016. pp. 41–48. ACM (2016). https://doi.org/10.1145/2866614.2866620
27. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
28. Muschevici, R., Proença, J., Clarke, D.: Feature Nets: behavioural modelling of software product lines. Softw. Sys. Model. **15**(4), 1181–1206 (2016). https://doi.org/10.1007/s10270-015-0475-z
29. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. **41**(1), 53–84 (2001). https://doi.org/10.1016/S0167-6423(00)00018-6
30. Salay, R., Famelis, M., Rubin, J., Sandro, A.D., Chechik, M.: Lifting Model Transformations to Product Lines. In: ICSE 2014. pp. 117–128. ACM (2014). https://doi.org/10.1145/2568225.2568267
31. Sebastio, S., Vandin, A.: MultiVeStA: Statistical Model Checking for Discrete Event Simulators. In: ValueTools 2013. pp. 310–315. ACM (2013). https://doi.org/10.4108/icst.valuetools.2013.254377
32. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Comput. Surv. **47**(1), 6:1–6:45 (2014). https://doi.org/10.1145/2580950