Quantitative Analysis of Probabilistic Models of Software Product Lines with Statistical Model Checking

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy

Axel Legay
Inria, Rennes, France

Alberto Lluch Lafuente
DTU, Lyngby, Denmark

Andrea Vandin
U Southampton, UK

We investigate the suitability of statistical model checking techniques for analysing quantitative properties of software product line models with probabilistic aspects. For this purpose, we enrich the feature-oriented language FLAN with action rates, which specify the likelihood of exhibiting particular behaviour or of installing features at a specific moment or in a specific order. The enriched language (called PFLAN) allows us to specify models of software product lines with probabilistic configurations and behaviour, e.g. by considering a PFLAN semantics based on discrete-time Markov chains. The Maude implementation of PFLAN is combined with the distributed statistical model checker MultiVeStA to perform quantitative analyses of a simple product line case study. The presented analyses include the likelihood of certain behaviour of interest (e.g. product malfunctioning) and the expected average cost of products.

1 Introduction

The modelling and analysis by means of process calculi and formal verification techniques like model checking of the variety of configurations and behaviour that is common to a software product line (SPL) is gaining momentum [5–9,16–18,23,33,38,39]. Compared to the complexity of verifying the behaviour of a single product or a single system, the variability inherent to SPL adds another dimension as the number of possible products of an SPL may be exponential in the number of features [11]. In [6], we introduced the feature-oriented language FLAN as a high-level modelling language for SPLs. A rich set of process-algebraic operators allows one to specify in a procedural, operational way both the configuration and the behaviour of products, while a constraint store allows one to specify in a declarative way all common structural constraints known from feature models and additional action constraints typical of feature-oriented software development. On the one hand, the execution of a process is constrained by the store (e.g. to avoid introducing inconsistencies), while on the other hand a process can query the store (e.g. to resolve configuration options) or update the store (e.g. to add new features, also at run time or by means of a staged configuration process). An implementation of FLAN in the executable modelling language Maude [19] allows one to exploit Maude's rich toolkit for a variety of formal analyses of FLAN models, ranging from consistency checking (by means of SAT solving) to model checking.

In this paper, we introduce a probabilistic extension of FLAN: PFLAN allows to equip actions with rates to specify probabilistic SPL models (e.g. to model uncertainty, failure rates, randomisation). This paves the way for quantitative analyses (e.g. to measure quality of service, reliability, performance). Here we present a proof-of-concept use of an implementation of PFLAN in Maude in combination with the distributed statistical model checker MultiVeStA [36] to estimate the likelihood of specific behaviour. Formally, our approach is to perform a sufficient number of probabilistic simulations of a PFLAN model to obtain statistical evidence (with a desired level of statistical confidence) of quantitative properties under scrutiny. The properties are formulated in MultiVeStA's property specification language MultiQuaTEx, which allows to express and evaluate more than one property over the same simulated path

J. Atlee and S. Gnesi (Eds.): FMSPLE 2015
EPTCS 182, 2015, pp. 56–70, doi:10.4204/EPTCS.182.5

(behaviour) [36]. The advantage over exhaustive (probabilistic) model checking is that there is no need to generate entire state spaces. We argue that this outweighs the main disadvantage of having to give up on obtaining exact results (100% confidence) with exact analysis techniques like probabilistic model checking, in particular when examining an SPL, given their possibly exponential number of products.

We refer to [3] for (probabilistic) model checking and to [28, 29] for statistical model checking. An overview of related work on applying formal analysis techniques in SPLE can be found in [38], while [6] contains an extensive discussion of related work on model-checking SPL behaviour. As far as we know, there are only a few, quite different, approaches on probabilistic model checking of an SPL [22, 24, 40], whereas we present here the first application of statistical model checking in SPL engineering (SPLE).

The paper outline is as follows. Section 2 contains a toy example of a product line of coffee machines, adapted from [5–9]. Section 3 presents PFLAN, followed by a PFLAN model of the example in Section 4. MultiVeStA is introduced in Section 5, followed by experimental quantitative analyses of the example in Section 6. Section 7 summarises the contributions of this paper and discusses future work.

2 An Example Product Line of Coffee Machines

Our toy example is a (simplistic) product line of coffee machines with the following list of requirements:

1. Initially, a coin must be inserted: either a euro, exclusively for products for the European market, or a dollar, exclusively for Canadian products;
2. An optional cancel button allows the user to cancel coin insertion, after which the coin is returned;
3. A machine that contains a coin must offer a choice to add sugar, followed by a choice of beverages;
4. The choice of drinks (coffee, tea, cappuccino) varies, but all products must offer at least one drink, tea may be offered only by European products, and products offering cappuccino must offer coffee;
5. An optional ringtone may be rung after beverage delivery. It must be rung after serving cappuccino;
6. After the drink is taken, the machine returns idle.

These requirements for products combine structural constraints defining valid feature configurations (e.g. *"every product must offer at least one beverage"*) with temporal constraints defining valid product behaviour in terms of valid action sequences (e.g. *"a ringtone must be rung after serving a cappuccino"*).

The de facto standard variability model in SPLE is a *feature model* [27,34]. It provides a compact representation of all valid products of a product line in terms of their features (behaviour is not captured). An (attributed) feature model of our example is depicted in Fig. 1. It has a root (feature) m and a set of non-trivial features, partitioned into the sets $\{b,o\}$ of *compound features* and $Features = \{s,r,x,p,c,t,d,e\}$ of *primitive features*.[1] The only purpose of the former is to group the (primitive) features in the tree, whereas the latter define user observable configuration parameters [4, 34]. We identify a product from the product line with a non-empty subset of *Features*. Deciding whether a product satisfies a feature model can be reduced to Boolean satisfiability (SAT), and efficiently be computed with SAT solvers [4].

By equipping features with (non-functional) attributes (e.g. $cost(Tea) = 3$) we obtain an *attributed feature model*.[2] The cost function $cost : Features \to \mathbb{N}$, associated to the attribute *cost*, straightforwardly extends to products: $cost(product) = \sum \{ cost(feature) \mid feature \in product \}$. Thus, intuitively, *cost* can be seen as a labelling function assigning a non-negative integer to each product defined by a feature model.

[1] In case no confusion can arise, we often simply speak of features when we actually refer to the primitive features.

[2] Additional quantitative constraints on (combinations of) features may be defined (e.g. $cost(Sugar) + cost(Ringtone) \leq cost(Coin)$) but we prefer to neglect them in this paper, as such constraints require the use of SMT solvers like Microsoft's Z3 [30], currently under integration in our framework.

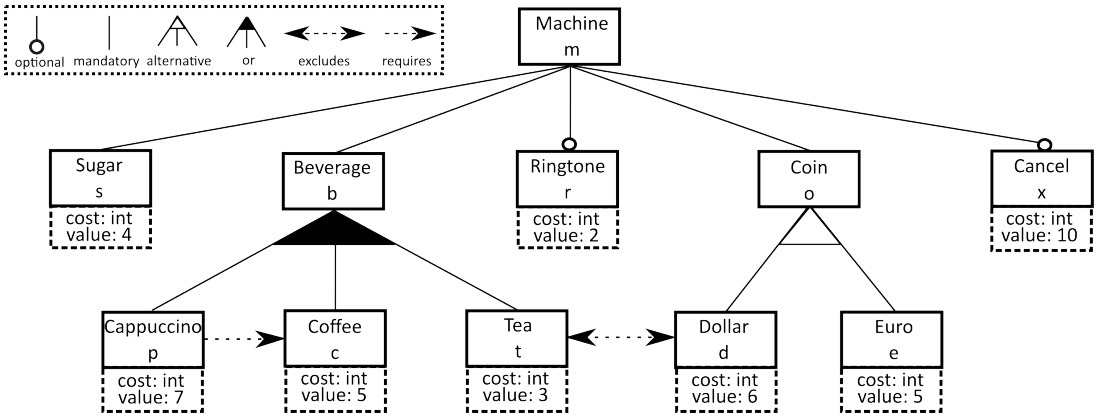

Figure 1: Attributed feature model of Coffee Machine (with shorthand names)

3 PFLAN: Syntax and Semantics

The feature-oriented language PFLAN is a probabilistic extension of FLAN [6], a process algebra that neatly separates declarative (pre-)configuration from procedural run-time aspects. PFLAN is inspired by the concurrent constraint programming paradigm of [32], its adoption in process calculi [14] and its stochastic extension [12]. A constraint store allows one to specify all common constraints known from feature models in a *declarative* way, while a rich set of process-algebraic operators allow to specify the configuration and behaviour of product lines in a *procedural* way. The semantics smoothly unifies *static* (pre-configuration) and *dynamic* (run-time) feature selection.

The core notions of PFLAN are *features*, *constraints*, *processes* (with action *rates*) and *fragments*, all visible in its syntax in Fig. 2. More precisely, f and g range over features while the syntactic categories F, S and P correspond to fragments, a constraint store and processes (with actions from A), respectively.

The universe of (primitive) features is denoted by $\mathscr{F}$. The features of our example are the accepted coin slots (i.e. *euro* and *dollar*), the offered drinks (i.e. *coffee*, *tea* and *cappuccino*) and the additional capabilities *sugar*, *cancel* and *ringtone* (to add sugar, cancel coin insertion and ring a tone, respectively).

The declarative part of PFLAN is represented by a store of constraints on features extracted from the product line requirements plus some additional information (e.g. about the context wherein the product will operate). Two important notions of a constraint store S are the *consistency* of S, denoted by *consistent(S)* (which in our case amounts to logical satisfiability of all constraints constituting S) and the *entailment* $S \vdash c$ of constraint c in S (which in our case amounts to logical entailment). A constraint store contains any term generated by S according to the syntax of PFLAN. The most basic constraint stores are $\top$ (no constraints at all), $\bot$ (inconsistent constraints) and ordinary Boolean propositions (generated by K). Constraints can be combined by juxtaposition (its semantics amounts to logical conjunction).

We assume that constraints on features are expressed using Boolean propositions (cf. [34]). Moreover, we assume that the universe $\mathscr{P}$ of propositions contains a Boolean predicate $has(f)$ that can be used to denote the presence of a feature f in a product. Boolean propositions can also be used to represent additional information such as contextual facts. In our example we use the Boolean propositions $in(Europe)$ and $in(Canada)$ to state the fact that the coffee machine being configured is meant to be used in Europe or in Canada, respectively. Finally, Boolean propositions can state relations between contextual information and features, like $has(euro) \rightarrow in(Europe)$ (i.e. a coffee machine has a coin slot for euro's only if it is intended for the European market).

$$
\begin{array}{rcl}
F & ::= & [S \mid P] \\
S,T & ::= & K \mid f \triangleright g \mid f \otimes g \mid S\ T \mid \top \mid \bot \\
P,Q & ::= & \emptyset \mid X \mid (A,r).P \mid P+Q \mid P;Q \mid P \parallel Q \\
A & ::= & a \mid \mathsf{install}(f) \mid \mathsf{ask}(K) \\
K & ::= & p \mid \neg K \mid K \vee K
\end{array}
$$

Figure 2: Syntax of PFLAN, where $r \in \mathbb{R}^+$, $a \in \mathscr{A}$, $p \in \mathscr{P}$ and $f,g \in \mathscr{F}$

Two common cross-tree constraints are instead handled as first-class citizens in PFLAN. A constraint $f \triangleright g$ expresses that feature f requires the presence of feature g, whereas a constraint $f \otimes g$ expresses that features f and g mutually exclude each other's presence (i.e. they are incompatible). Also these constraints could of course be encoded as Boolean propositions (e.g. $f \otimes g$ and $f \triangleright g$ can equivalently be expressed as $has(f) \leftrightarrow \neg has(g)$ and $has(f) \rightarrow has(g)$, respectively). We in fact use such logical encodings to reduce consistency checking and entailment to logical satisfiability (and hence exploit Maude's SAT solver). However, we prefer to keep this first-class treatment as syntactic sugar. In our example, we extract $dollar \otimes euro$ to formalise that *euro* and *dollar* are mutually exclusive features (requirement 1) and $cappuccino \triangleright coffee$ to formalise that *cappuccino* requires *coffee* (requirement 3).

We also consider a class of *action constraints*, reminiscent of featured transition systems (FTS) [18]. In an FTS, transitions are labelled with actions and with Boolean constraints over the set of features. We associate arbitrary constraints to actions rather than to transitions (and we moreover add a rate to the actions, discussed below). In a coffee machine offering coffee, e.g., we will use *coffee* for the (user) action of choosing coffee and $do(coffee)$ as a proposition stating the execution of that action. The relation between the action *coffee* and the presence of the corresponding feature *coffee* can be formalised as $do(coffee) \rightarrow has(coffee)$, i.e. the choice for coffee requires coffee being offered by the coffee machine. In general, we assume that each action a may have a constraint $do(a) \rightarrow p$, where $p \in \mathscr{P}$ is a proposition. Such constraints act as a kind of guards to allow or forbid the execution of actions (cf. the discussion of the rule ACT below). Note that these action constraints could also be more complex, e.g. we could define an action *café-au-lait* together with the action constraint $do(café\text{-}au\text{-}lait) \rightarrow (has(coffee) \wedge has(milk))$.

The procedural part of PFLAN is represented by *processes* which can be of the following type:

$\emptyset$ the empty process that does nothing;

X a process identifier;[3]

$(A,r).P$ a process that can perform action A with rate r and then behaves as P;

$P+Q$ a process that can non-deterministically choose to behave as either P or Q;

$P;Q$ a process that must progress first as P and then as Q;

$P \parallel Q$ a process formed by the parallel composition of P and Q, which evolve independently.

We distinguish ordinary actions from a universe $\mathscr{A}$ and two special actions $\mathsf{install}(f)$ (which will be used to denote the dynamic installation of a feature f) and $\mathsf{ask}(K)$ (which can used to query the store for the validity of a Boolean proposition from K). As we will see shortly, each action type is treated differently in the operational semantics. Note, moreover, that each action has an associated *rate* (sometimes called *weight*), which is used to determine the probability that this action is executed. As usual, the probability to execute an action in a certain state depends on the rates of all other actions enabled in the same state.

[3] We assume there is a set of process definitions of the form $X \doteq P$ and recursively defined processes to be finitely branching.

$$(\textsc{Inst})\ \frac{\mathit{consistent}(S\ \mathit{has}(f))}{[S \mid (\mathsf{install}(f), r).P] \xrightarrow{r} [S\ \mathit{has}(f) \mid P]} \qquad (\textsc{Or})\ \frac{[S \mid P] \xrightarrow{r} [S' \mid P']}{[S \mid P + Q] \xrightarrow{r} [S' \mid P']}$$

$$(\textsc{Ask})\ \frac{S \vdash K}{[S \mid (\mathsf{ask}(K), r).P] \xrightarrow{r} [S \mid P]} \qquad (\textsc{Seq})\ \frac{[S \mid P] \xrightarrow{r} [S' \mid P']}{[S \mid P; Q] \xrightarrow{r} [S' \mid P'; Q]}$$

$$(\textsc{Act})\ \frac{S = (do(a) \to K) \qquad S \vdash K}{[S \mid (a, r).P] \xrightarrow{r} [S \mid P]} \qquad (\textsc{Par})\ \frac{[S \mid P] \xrightarrow{r} [S' \mid P']}{[S \mid P \parallel Q] \xrightarrow{r} [S' \mid P' \parallel Q]}$$

Figure 3: Reduction semantics of PFLAN

$$\begin{array}{rclrclrcl}
P+(Q+R) & \equiv & (P+Q)+R & P+\emptyset & \equiv & P & P+Q & \equiv & Q+P \\
P \parallel (Q \parallel R) & \equiv & (P \parallel Q) \parallel R & P \parallel \emptyset & \equiv & P & P \parallel Q & \equiv & Q \parallel P \\
P;(Q;R) & \equiv & (P;Q);R & P;\emptyset & \equiv & P \equiv \emptyset;P & P & \equiv & P[^{Q}/_{X}] \text{ if } X \doteq Q
\end{array}$$

Figure 4: Structural congruence in PFLAN

We will illustrate this in our example in Section 4. We consider the actions *euro*, *dollar* (respective coin insertion), *cancel* (cancellation of coin insertion), *sugar* (sugar selection), *ringtone* (ringtone emission), *coffee*, *tea* and *cappuccino* (beverage selection) in our example. Their associated rates are discussed below. For simplicity we consider only constant rates, but our framework can be easily extended to allow store-dependent rates (e.g. to be able to reflect a higher probability to order cappuccino in Europe).

Finally, a *fragment* F is a term $[S \mid P]$, composed by a constraint store S and a process P. These two components may influence each other according to the concurrent constraint programming paradigm [32]: a process may update its store which, in turn, may condition the execution of the process' actions.

The operational semantics is formalised in terms of the state transition relation $\to \subseteq \mathbb{N}^{\mathbb{F}\times\mathbb{R}^+\times\mathbb{F}}$ defined in Fig. 3, where $\mathbb{F}$ denotes the set of all terms generated by F in the grammar of Fig. 2. Note that multisets of transitions are needed to deal with the possibility of having multiple instances of a transition $F \xrightarrow{r} G$. Technically, such a reduction relation is defined in structural operational semantics (SOS) style (i.e. by induction on the structure of the terms denoting a fragment) modulo the structural congruence relation $\equiv \subseteq \mathbb{F} \times \mathbb{F}$ defined in Fig. 4. The reduction relation implicitly defines a labeled transition system LTS, whose labels are rates. It is straightforward to obtain a discrete time Markov chain (DTMC) from such LTSs by normalising the rates into $[0..1]$ such that in each state, the sum of the rates of its outgoing transitions equals one. As usual, in the resulting DTMC the label of a transition corresponds to the probability that such a transition is executed starting from its source state. Recall that we advocate the use of statistical model checking because in general the DTMC is too large to generate.

The rules INST and ACT of the semantics are very similar, both allowing a process to execute an action if certain constraints are satisfied. Rule INST forbids inconsistencies caused by the introduction of new features. It can be seen as a particular instance of the rule for the tell operation of concurrent constraint programming [32] instantiated as $\mathsf{tell}(\mathit{has}(f))$. Rule ACT forbids inconsistencies with respect to action constraints. A typical action constraint is $do(a) \to \mathit{has}(f)$, i.e. action a is subject to the presence of feature f. Rule ASK formalises the semantics of the $\mathsf{ask}(\cdot)$ operation from concurrent constraint programming [32]. It allows a process to be blocked until a proposition can be derived from the store. Rules PAR, SEQ and OR formalise interleaving parallel composition, sequential composition and non-deterministic choice, respectively. Note that the non-determinism introduced by choices and parallel composition is probabilistically resolved in the aforementioned DTMC semantics.

Summarising, we note a variety of ways in which a feature f can be included in a configuration. First, an *explicit* and *declarative* way is to include the proposition $has(f)$ in the initial store; this is the way to include core features. Second, an *implicit* and *declarative* way is to derive f from other constraints (e.g. if a store contains $g \triangleright f$ and $has(g)$, then f's presence follows). Third, a *procedural* way is to dynamically install f at run time; this key aspect originating from FLAN enables staged configuration as known from dynamic software product lines [13, 21]. Building on FLAN, PFLAN combines these three ways in an elegant and consistent manner. The introduction of action rates in PFLAN moreover allows one to specify probabilistic aspects of SPL models such as the behaviour of the user of a product and the likelihood of installing a certain feature at a specific moment with respect to that of other features.

4 A PFLAN Model of the Example Product Line

Fig. 5 shows a specification of the family of coffee machines in PFLAN. Fragment F is composed of store S and a process Q. The latter specifies an initial configuration phase D, during which all primitive features except *ringtone* can be installed (the order of installation is influenced by the relative weight of the features, more on this below). This phase ends at a certain moment when a specific product (coffee machine) is said to be pre-configured, modeled by the installation of an ad-hoc defined feature *pre-conf*, thus initiating the execution of process R, which specifies the product's run-time behaviour. Note that it is specifically allowed to install (or bind) a feature at run time (cf. *ringtone* in our toy example).

The store, instead, is made up of two parts: constraints derived from the requirements (S_1) plus contextual information (S_2). The current action constraints are quite simple (all are of the form $do(f) \rightarrow has(g)$) but, as said before, they could be more sophisticated upon need (e.g. the constraint on action *cappuccino* could be specified as $do(cappuccino) \rightarrow has(cappuccino) \wedge has(ringtone)$ to require not only the presence of its corresponding feature but also that of the *ringtone* feature). In Fig. 5, a product line of European coffee machines is instantiated by the explicit context information $in(Europe)$.

The configuration process D is a simple rated choice among the installation of some of the features a coffee machine may exhibit. This specifies a sort of race between features and may be thought of as independent designers competing to install the features for which they are responsible. The semantics of PFLAN ensures that all executions will result in a consistent configuration if the process begins with a consistent store, i.e. the semantics forbids the installation of features that are mutually exclusive or prohibited by (a combination of) the constraints. Formally, multiple installations of the same feature does not have any effect, as installed features are organised in a set. The rates of the actions influence this race by determining a higher (or lower) probability for the installation of one feature with respect to another (or prior to another). In our example, to reflect the fact that *coin* and *sugar* are core features, we assign higher rates to them than to the optional features to raise their chances of being installed first. Moreover, since we are modelling a coffee machine and since coffee is a necessary ingredient for cappuccino, we assign a higher rate to the feature *coffee* than to those of other drinks. As a result, the probability to install *sugar* in the first step, given that also *pre-conf*, *euro*, *cancel*, *coffee*, *tea* and *cappuccino* can be installed, thus becomes $\frac{10}{10+10+10+7+9+6+3} = 2/11$.

Process R, finally, describes the run-time execution of a coffee machine. The machine may either accept a euro or a dollar, depending on the market it is meant for. After that, the user may cancel coin insertion, upon which the machine returns to its initial state and (usually) returns the coin. With a probability of $1/11$, however, the machine does not return the coin (viz. $\frac{1}{10+1}$). If coin insertion is not canceled, the user may (P_2) or may not (P_3) push a button for sugar. In case sugar is selected, it is also poured, after which the user can select a beverage. But, with a probability of $\frac{2}{10+2} = 1/6$ the machine is out of sugar, after which the user may either cancel the coin insertion or go for an unsugared drink.

$$
\begin{array}{rcl}
F & \doteq & [S \mid Q] \\
S & \doteq & S_1\ S_2 \\
S_1 & \doteq & has(euro) \vee has(dollar) \quad has(euro) \to in(Europe) \quad has(dollar) \to in(Canada) \\
 & & has(coffee) \vee has(tea) \vee has(cappuccino) \qquad has(tea) \to in(Europe) \\
 & & dollar \otimes euro \qquad cappuccino \triangleright coffee \\
 & & do(euro) \to has(euro) \qquad do(dollar) \to has(dollar) \\
 & & do(sugar) \to has(sugar) \qquad do(ringtone) \to has(ringtone) \qquad do(cancel) \to has(cancel) \\
 & & do(pour\_sugar) \to has(sugar) \qquad do(out\_of\_sugar) \to has(sugar) \\
 & & do(return\_coin) \to has(cancel) \qquad do(no\_return) \to has(cancel) \\
 & & do(coffee) \to has(coffee) \qquad do(tea) \to has(tea) \qquad do(cappuccino) \to has(cappuccino) \\
 & & do(pour\_coffee) \to has(coffee) \qquad do(out\_of\_coffee) \to has(coffee) \\
 & & do(pour\_tea) \to has(tea) \qquad do(out\_of\_tea) \to has(tea) \\
 & & do(pour\_milk) \to has(cappuccino) \qquad do(out\_of\_milk) \to has(cappuccino) \\
S_2 & \doteq & in(Europe) \\
Q & \doteq & D + (\mathsf{install}(pre\text{-}conf), 10).R \\
D & \doteq & (\mathsf{install}(euro), 10).Q + (\mathsf{install}(dollar), 10).Q + (\mathsf{install}(sugar), 10).Q + (\mathsf{install}(cancel), 7).Q \\
 & & + (\mathsf{install}(coffee), 9).Q + (\mathsf{install}(tea), 6).Q + (\mathsf{install}(cappuccino), 3).Q \\
R & \doteq & ((euro, 25).\emptyset + (dollar, 25).\emptyset); P_1 \\
P_0 & \doteq & (return\_coin, 10).R + (no\_return, 1).R \\
P_1 & \doteq & (cancel, 5).P_0 + P_2 + P_3 \\
P_2 & \doteq & (sugar, 15).\emptyset; ((pour\_sugar, 10).P_3 + (out\_of\_sugar, 2).P_1) \\
P_3 & \doteq & (coffee, 20).P_4 + (tea, 12).P_5 + (cappuccino, 8).P_6 \\
P_4 & \doteq & (pour\_coffee, 10).P_8 + (out\_of\_coffee, 2).P_3 \\
P_5 & \doteq & (pour\_tea, 10).P_8 + (out\_of\_tea, 2).P_3 \\
P_6 & \doteq & (pour\_milk, 10).\emptyset; ((pour\_coffee, 10).P_8 + (out\_of\_coffee, 2).R) + (out\_of\_milk, 2).P_3 \\
P_8 & \doteq & P_9 + (\mathsf{install}(ringtone), 8).(ringtone, 18).P_9 \\
P_9 & \doteq & (take\_drink, 10).R + (no\_cup, 1).R
\end{array}
$$

Figure 5: PFLAN specification of the family of coffee machines (instantiated for Europe)

Beverage selection (more likely coffee than tea or cappuccino) is followed by the drink being poured (again with a probability that the chosen drink is unavailable), which in case of cappuccino concerns both milk and coffee. In case coffee or tea was chosen but unavailable, the user can again choose a beverage (and the machine may have been refilled). In the specific case that milk was poured but coffee is not available, the user has bad luck as the machine returns to its idle state before completing the chosen beverage. In case a drink was poured successfully, a ringtone may follow (in which case it first needs to be installed). The user then either takes the drink or, with a $1/11$ probability, realizes that sadly enough there was no cup available. Either way, the machine returns to its initial state.

Note how the rates 'influence' the behavior, in the sense that the choice operator is no longer purely non-deterministic, but probabilistic, i.e. the rates provide a probabilistic model of the behavior of the coffee machine and its environment (the users). Consider, e.g., the choice of a beverage: $(coffee, 20).P_4 + (tea, 12).P_5 + (cappuccino, 8).P_6$. The probability to choose coffee is ½ (viz. $\frac{20}{20+12+8}$), compared to 0.3 for tea and ⅕ for cappuccino. Similarly, the probability to cancel coin insertion is $\frac{5}{5+15+20+12+8} = 1/11$ (i.e. rather low). Note that we need to expand processes P_2 and P_3 to calculate this probability.

The rates that we assigned in this example merely serve to illustrate the proof-of-concept that we present in this paper. In practice, those rates may be obtained from a statistical analysis of the actual product configuration processes and product behaviours, possibly contained in historical logs.

Note that D and R are not purely distinct (pre-)configuration and run-time processes, respectively: feature *ringtone* may be installed dynamically at run time (i.e. possibly by R but never by D) and it can be thought of as, e.g., a software module. This is an example of a staged configuration process, in which some optional features are bound at run time rather than at (pre-)configuration time.

5 Quantitative Analysis with MultiVeStA

MultiVeStA [36] is a statistical analysis tool developed and maintained by S. Sebastio and A. Vandin. It extends the (distributed) statistical model-checking tools PVeStA [2] and VeStA [37], developed at the Department of Computer Science of the University of Illinois at Urbana-Champaign. Differently from its predecessors, MultiVeStA can easily be integrated with any formalism which allows for probabilistic simulations. It has so far been used to analyse transportation systems [25], volunteer clouds [35], crowd-steering [31] and swarm robotic [10] scenarios.

In this paper, we use MultiVeStA to analyse PFLAN specifications in order to obtain statistical estimations of quantitative properties expressed in MultiVeStA's query language MultiQuaTEx (an extension of QuaTEx [1]). MultiVeStA provides such estimations by means of distributed statistical analysis techniques known from statistical model checking [28, 29]. A prototypical tool integrating MultiVeStA and PFLAN is available at `https://code.google.com/p/multivesta/wiki/PFLan` together with all files necessary to reproduce the experiments discussed in this section.

Probabilistic simulations of a PFLAN specification can easily be obtained by executing the model step-by-step by applying the rules of Fig. 3, each time selecting one of the computed one-step next-states according to the probability distribution obtained after normalising the rates of the generated transitions. Classical statistical model checking techniques allow one to perform analyses like "is the probability that a property holds greater than 0.3?" or "what is the probability that a property is satisfied?" over a given specification. Next to performing such kinds of analyses over products, MultiVeStA also allows to estimate the expected values of properties that can take on any value from $\mathbb{R}$, like "what is the average cost of products generated from a software product line specification?". Estimations are computed as the mean value of n samples obtained from n simulations, with n large enough to grant that the size of the $(1-\alpha)\times 100\%$ *confidence interval* (CI) is bounded by δ. In other words, if a MultiQuaTEx expression is estimated as $\bar{x}$, then with probability $(1-\alpha)$ its actual expected value belongs to the interval $[\bar{x}-\delta/2, \bar{x}+\delta/2]$. A CI is thus specified in terms of two parameters: α and δ. In all experiments discussed in this section, we fixed $\alpha = 0.1$, and $\delta = 0.1$ and $\delta = 0.5$ for probabilities and costs of products, respectively.

MultiVeStA's property specification language MultiQuaTEx is very flexible, based on the following ingredients: real-valued observations on the current 'state' (e.g. the total cost of installed features), arithmetic expressions and comparison operators, if-then-else statements, a one-step next operator (which triggers the execution of one step of a simulation) and recursion. Intuitively, we can use MultiQuaTEx to associate a value from $\mathbb{R}$ to each simulation and subsequently use MultiVeStA to estimate the expected value of such number (in case this number is 0 or 1 upon the occurrence of a certain event, we thus estimate the probability of such an event to happen).

6 Quantitative Analyses of the Example Product Line

Some properties that we can verify over our toy example are as follows:

P_1 The probability to run into a deadlock before completing the pre-configuration phase;

P_2 For each of the 8 primitive features (sugar, ringtone, cancel, cappuccino, coffee, tea, dollar, euro), the probability to have it installed after the pre-configuration phase or at a given simulation step x;

P_3 The average cost of products obtained from the pre-configuration phase, or of the 'intermediate' ones obtained at a given simulation step x.

Note that we consider any configuration obtained by intermediate steps to be a (possibly intermediate) product. This may thus refer to an unfinished product or to underspecified software, or concern a not yet fully developed product. When no more features can be installed, we speak of a final product.

While not explicitly stated, all experiments discussed in this section refer to versions (defined below) of the PFLAN specification of Fig. 5 without the contextual information $S_2 \doteq in(Europe)$, so as to study properties of our example without restrictions to a specific context (thus implicitly allowing deadlocks).

Property P_1 is useful for studying the correctness of the PFLAN specification of a product line, in this case by verifying the probability to successfully complete the pre-configuration phase of a product from the product line. Property P_2 is useful for studying how often (on average) a feature is actually installed in a product from the product line, which is important information for those designers or programmers responsible for the production or programming of a specific feature or software module. Property P_3, finally, is useful for studying the average cost of assembling a product from the product line, based on the costs of the features constituting a product defined by the attributed feature model depicted in Fig. 1.

Listing 1 depicts a MultiQuaTEx expression to evaluate P_1. Lines 1-4 define a recursive temporal operator which is evaluated against a simulation: it gives 0.0 if the feature *pre-conf* is installed in the current simulation state (Line 2); it gives 1.0 if the current state is a deadlock (Line 3); or it is recursively evaluated in the next simulation state (Line 4). Intuitively, # is the one-step temporal operator, while real-valued observations on the current state are evaluate resorting to the keyword `s.rval`. A number of predefined observations is currently supported, e.g. we can query whether a given feature is currently installed (as in Line 2 for *pre-conf*) or whether the current process has no more actions that are allowed by the constraints, in which case we say that it is in a deadlock state (Line 3). Finally, Line 5 specifies the property to be studied: the expected value of the defined recursive temporal operator.

```
1 DeadlockInPreconf() =
2  if {s.rval("pre-conf") == 1.0} then 0.0
3  else if {s.rval("deadlock") == 1.0} then 1.0
4  else #DeadlockInPreconf() fi fi ;
5  eval E[ DeadlockBeforePreconf() ] ;
```

Listing 1: The MultiQuaTEx expression corresponding to property P_1

We evaluated P_1 against our PFLAN model, obtaining probability 0.0, i.e. the pre-configuration phase (almost surely) always terminates.

Now consider our model to be modified according to Fig. 6, i.e. by replacing F with F', and both Q and D with D'. This version still contains a pre-configuration phase (D') followed by the same run-time phase (R) of the original model. Essentially, D' tries to install all features, possibly in different orders.

$$
\begin{aligned}
F' &\doteq [S \mid D'; (\mathsf{install}(\mathit{pre\text{-}conf}), 10).R] \\
D' &\doteq (\mathsf{install}(\mathit{euro}), 10).\emptyset \parallel (\mathsf{install}(\mathit{dollar}), 10).\emptyset \parallel (\mathsf{install}(\mathit{sugar}), 10).\emptyset \parallel (\mathsf{install}(\mathit{cancel}), 7).\emptyset \\
&\quad \parallel (\mathsf{install}(\mathit{coffee}), 9).\emptyset \parallel (\mathsf{install}(\mathit{tea}), 6).\emptyset \parallel (\mathsf{install}(\mathit{cappuccino}), 3).\emptyset
\end{aligned}
$$

Figure 6: A modified version of the PFLAN specification of Fig. 5

By evaluating P_1 against the modified version of our model we obtain probability 1.0, i.e. the pre-configuration phase (almost surely) never terminates. In fact, we can install only one among *dollar* or *euro* (cf. the first constraint of S_1 in Fig. 5), and consequently one of the two installations will never succeed. P_1 can thus indeed be used to check liveness properties of PFLAN specifications, e.g. to individuate specifications leading, with a certain probability, to deadlocks.

Listing 2 depicts a MultiQuaTEx expression to evaluate P_2 and P_3 when considering the products obtained after the pre-configuration phase. Such an expression shows how MultiQuaTEx allows one to express more properties at once, which can be estimated by MultiVeStA reusing the same simulations. Lines 1-3 define the recursive temporal operator `ProductCostAfterPreconf`. It is evaluated against a simulation as the cost of the product obtained from the pre-configuration phase. As shown in Line 2, a further predefined observation is supported, viz. *cost*, which provides the cost of the current product. Lines 4-6 define a parametric recursive temporal operator which evaluates to 1.0 if the feature provided as parameter is installed during the pre-configuration phase, and to 0.0 otherwise. Finally, Lines 7-11 specify the properties to be analysed: the average cost of products generated by the pre-configuration phase (Line 7) and for each of the 8 primitive features the probability to have it installed (Lines 8-11). We remark that MultiVeStA adopts a procedure which takes into account that each property might require a different number of simulations to satisfy the required confidence interval CI.

```
1  ProductCostAfterPreconf() =
2   if {s.rval("pre-conf") == 1.0} then s.rval("cost")
3                                  else #ProductCostAfterPreconf() fi ;
4  IsInstalledAfterPreconf(feature) =
5   if {s.rval("pre-conf") == 1.0} then s.rval(feature)
6                                  else #IsInstalledAfterPreconf({feature}) fi ;
7  eval E[ ProductCostAfterPreconf() ];
8  eval E[ IsInstalledAfterPreconf("sugar") ]; eval E[ IsInstalledAfterPreconf("ringtone") ];
9  eval E[ IsInstalledAfterPreconf("cancel") ]; eval E[ IsInstalledAfterPreconf("cappuccino") ];
10 eval E[ IsInstalledAfterPreconf("coffee") ]; eval E[ IsInstalledAfterPreconf("tea") ];
11 eval E[ IsInstalledAfterPreconf("dollar") ]; eval E[ IsInstalledAfterPreconf("euro") ];
```

Listing 2: The MultiQuaTEx expression corresponding to properties P_2 and P_3

We evaluated the MultiQuaTEx expression of Listing 2 against the original model of Fig. 5. The obtained average cost is 14.07, while the probabilities of installing the primitive features are given in the first row of Table 1. Clearly, the probability with which features are installed (as well as the average cost of the obtained products) is highly affected by the rate at which *pre-conf* is installed (10 in Fig. 5): a lower or higher rate rate leads to more or less iterations of D, respectively. In order to quantify the influence of this rate, we further evaluated the expression of Listing 2 against the model obtained changing the aforementioned rate to 50. The obtained average cost of products is 4.46, while the probabilities of installing the features are provided in the second row of Table 1. As expected, the higher installation rate of *pre-conf* has the effect of decreasing the average number of iterations of the pre-configuration phase, leading to a lower probability of installation of the features and to a lower average cost of products.

	Features							
Rate of *install*(*pre-conf*)	*sugar*	*ringtone*	*cancel*	*cappuccino*	*coffee*	*tea*	*dollar*	*euro*
10	0.49	0.0	0.45	0.13	0.50	0.40	0.33	0.38
50	0.17	0.0	0.11	0.0	0.14	0.10	0.12	0.13

Table 1: Probability of installing a feature during the pre-configuration phase of model in Fig. 5 (P_2).

We conclude this section by showing how MultiVeStA can be used to analyse properties of a PFLAN specification upon varying the number of performed simulation steps. Listing 3 sketches how the MultiQuaTEx expression of Listing 2 can be made parametric with respect to a given set of simulation steps. First of all, the temporal operators were modified so that they are evaluated with respect to a specific step given as parameter. We actually provide only the updated temporal operator regarding the costs of products (Lines 1-4), as the other has been modified similarly. Subsequently, it is necessary to specify a range of values for the parameter. Lines 6-7 specify that we are interested in studying the properties for steps going from 0.0 to 40, with an increment of 1.0.

```
1 ProductCostAtStep(step) =
2  if {s.rval("steps") == step} then s.rval("cost")
3                               else #ProductCostAfterPreconf(step)
4  fi ;
5 IsInstalledAtStep(step,feature) =  ...
6 eval parametric(E[ ProductCostAtStep(step) ],E[ IsInstalledAtStep(step,"sugar") ],...,
7                 E[ IsInstalledAtStep(step,"euro") ],step,0.0,1.0,40.0) ;
```

Listing 3: The MultiQuaTEx expression corresponding to P_2 and P_3 upon varying the simulation steps

We evaluated also the parametric property of Listing 3 against the original model of Fig. 5. All such analyses (41×9 different properties) were evaluated using the same simulations. The results are presented in two plots: one for costs (`ProductCostAfterPreconf`) in Fig. 7, and one for probabilities (`IsInstalledAfterPreconf`) in Fig. 8.

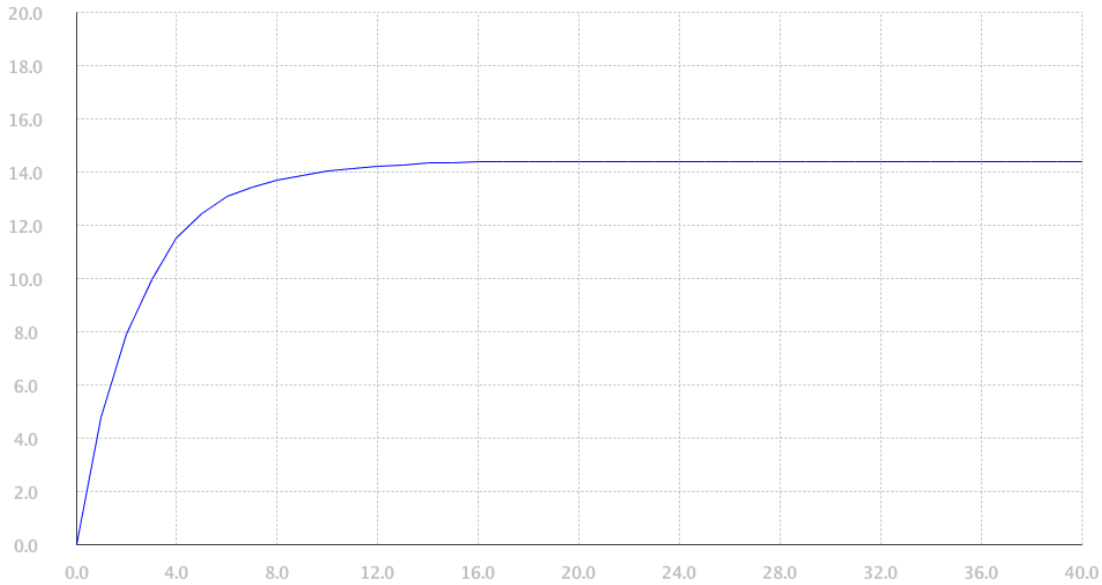

Figure 7: Average product cost during the first 40 steps

As expected, Fig. 7 shows that the average cost (on the y-axis) of the intermediate products generated from the software product line grows with respect to the number of performed simulation steps. In particular, it shows a fast growth during the first 6 steps, reaching an average cost of 13, and then it essentially stabilises, eventually reaching its maximum (14.38) from step 26 onwards. This is consistent with our PFLAN specification, consisting of a pre-configuration phase during which the majority of the features are installed, followed by a run-time phase modelling the behaviour of the generated product (and possibly installing *ringtone*).

Fig. 8 shows that the probabilities (on the y-axis) for each of the features to be installed evolve similarly to the average cost of the generated products, although, clearly, with different scales: they show a fast growth during the first 6 steps, after which they essentially stabilise while approaching their maximum. The maximum probabilities of installing the various primitive features are as follows: 0.0 for *ringtone*, 0.07 for *cappuccino*, 0.32 for *euro*, 0.35 for *dollar*, 0.38 for *tea*, 0.43 for *cancel*, 0.49 for *coffee* and 0.51 for *sugar*. Note that the probability of installing *ringtone* is really very low (0.0 or, to be precise, its actual expected value belongs to the interval $[0,0+\delta/2] = [0,0+\frac{0.1}{2}] = [0,0.05]$). Indeed, while the installation of *ringtone* is allowed by our specification, it is optional (except when serving cappuccino). Note, however, that we considered simulations consisting of only 40 steps (the x-axes in the two figures). For longer simulations, we would of course have obtained a higher probability to install *ringtone*.

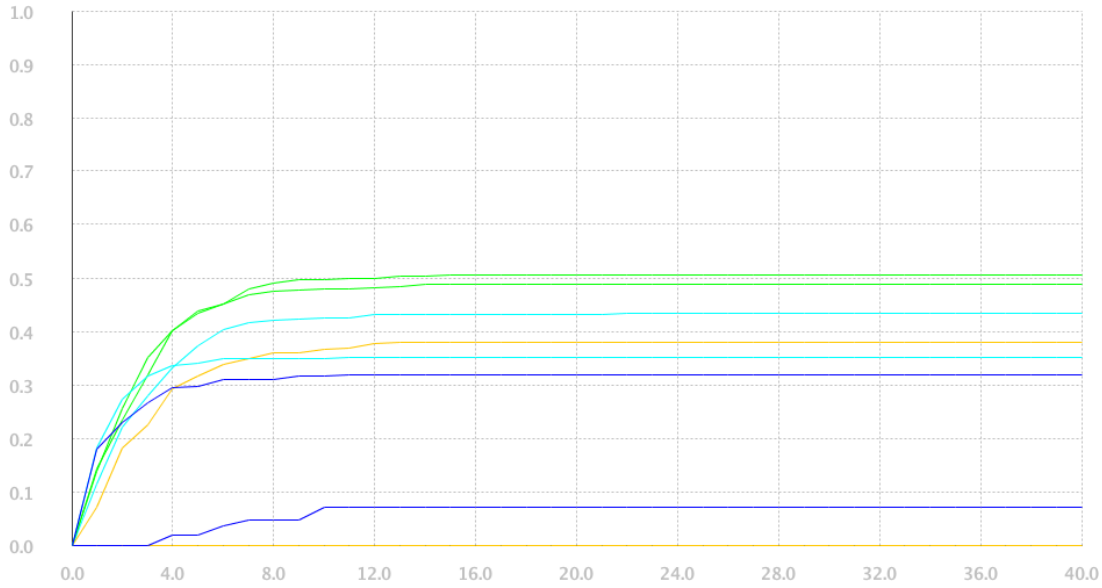

Figure 8: Average feature installation probability during the first 40 steps

7 Conclusion

In this paper, we have continued a line of research presented at earlier editions of FMSPLE [6, 26] by enriching FLAN, a high-level feature-based modelling language for software product lines, with quantitative information. The result, PFLAN, allows one to model and analyse the likelihood of installing features, the probabilistic behaviour of users of products of the product line, and the costs of products, next to probabilistic quantifications of ordinary temporal properties (e.g. "what is the probability that coffee is poured while no cup was available?"). In addition, we extended the qualitative analysis framework for software product lines implemented in Maude with statistical techniques for quantitative analysis.

The modelling and analysis capabilities of PFLAN were illustrated on a simple product line of coffee machines. In the future, we plan to investigate the scalability of our (tool) framework by considering more realistic and complex scenarios. We also intend to add the possibility to define quantitative constraints to PFLAN, possibly by adopting further operations from extensions of the concurrent constraint paradigm that can deal with quality of service and mobility [14] and its stochastic extension [12].

Both the check operation of concurrent constraint programming, to prevent inconsistencies, and its retract operation, to remove (syntactically present) constraints from the store, might be useful to enable the dynamic (un)installation of features in the presence of (*soft*) quantitative constraints (i.e. not only Boolean [20]). In particular, we would like to investigate the consequences of allowing the explicit uninstallation of a feature, e.g. due to its malfunctioning or due to the need of replacing it by a better (version of the) feature. Such features were shown successful in services computing for the specification of service-level agreements and negotiation processes [15]. Finally, we would like to allow behaviour that is explicitly influenced by the constraint store, as in [12]. In our example, this would allow us to model, e.g., the probability of a user choosing a coffee to depend on the location of the coffee machine (i.e. Europe or Canada), thus allowing us to assign, e.g., a higher weight to ordering cappuccino in Europe.

Acknowledgements

This research was supported by the EU FP7-ICT FET-Proactive project QUANTICOL (600708) and the Italian MIUR project CINA (PRIN 2010LHT4KM).

Moreover, we thank the reviewers for their detailed comments, which helped us to improve the paper.

References

[1] G.A. Agha, J. Meseguer & K. Sen (2005): *PMaude: Rewrite-based Specification Language for Probabilistic Object Systems*. In: *QAPL*, *ENTCS* 153, pp. 213–239, doi:10.1016/j.entcs.2005.10.040.

[2] M. AlTurki & J. Meseguer (2011): *PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool*. In A. Corradini, B. Klin & C. Cîrstea, editors: *CALCO*, *LNCS* 6859, Springer, pp. 386–392, doi:10.1007/978-3-642-22944-2_28.

[3] C. Baier & J.-P. Katoen (2008): *Principles of Model Checking*. The MIT Press. Available at `http://mitpress.mit.edu/books/principles-model-checking`.

[4] D.S. Batory (2005): *Feature Models, Grammars, and Propositional Formulas*. In J.H. Obbink & K. Pohl, editors: *SPLC*, *LNCS* 3714, Springer, pp. 7–20, doi:10.1007/11554844_3.

[5] M.H. ter Beek, A. Fantechi, S. Gnesi & F. Mazzanti (2015): *Modelling and Analysing the Variability in Product Families: Model Checking of Modal Transition Systems*. Submitted.

[6] M.H. ter Beek, A. Lluch Lafuente & M. Petrocchi (2013): *Combining Declarative and Procedural Views in the Specification and Analysis of Product Families*. In: *FMSPLE workshop at SPLC*, ACM, pp. 10–17, doi:10.1145/2499777.2500722.

[7] M.H. ter Beek, F. Mazzanti & A. Sulova (2012): *VMC: A Tool for Product Variability Analysis*. In D. Giannakopoulou & D. Méry, editors: *FM*, *LNCS* 7436, Springer, pp. 450–454, doi:10.1007/978-3-642-32759-9_36.

[8] M.H. ter Beek & E.P. de Vink (2014): *Software Product Line Analysis with mCRL2*. In: *SPLat workshop at SPLC*, ACM, pp. 78–85, doi:10.1145/2647908.2655970.

[9] M.H. ter Beek & E.P. de Vink (2014): *Using mCRL2 for the Analysis of Software Product Lines*. In S. Gnesi & N. Plat, editors: *FormaliSE workshop at ICSE*, IEEE, pp. 31–37, doi:10.1145/2593489.2593493.

[10] L. Belzner, R. De Nicola, A. Vandin & M. Wirsing (2014): *Reasoning (on) Service Component Ensembles in Rewriting Logic*. In S. Iida, J. Meseguer & K. Ogata, editors: *Specification, Algebra, and Software*, *LNCS* 8373, Springer, pp. 188–211, doi:10.1007/978-3-642-54624-2.

[11] P. Borba, M.B. Cohen, A. Legay & A. Wąsowski (2013): *Analysis, Test and Verification in The Presence of Variability (Dagstuhl Seminar 13091)*. *Dagstuhl Reports* 3(2), pp. 144–170, doi:10.4230/DagRep.3.2.144.

[12] L. Bortolussi (2006): *Stochastic Concurrent Constraint Programming*. In: *QAPL*, *ENTCS* 164, pp. 65–80, doi:10.1016/j.entcs.2006.07.012.

[13] J. Bürdek, S. Lity, M. Lochau, M. Berens, U. Goltz & A. Schürr (2014): *Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints*. In P. Collet, A. Wąsowski & T. Weyer, editors: *VaMoS*, ACM, doi:10.1145/2556624.2556627.

[14] M.G. Buscemi & U. Montanari (2007): *CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements*. In R. De Nicola, editor: *ESOP*, *LNCS* 4421, Springer, pp. 18–32, doi:10.1007/978-3-540-71316-6_3.

[15] M.G. Buscemi & U. Montanari (2011): *QoS negotiation in service composition*. *J. Log. Algebr. Program.* 80(1), pp. 13–24, doi:10.1016/j.jlap.2010.04.001.

[16] A. Classen, M. Cordy, P. Heymans, A. Legay & P.-Y. Schobbens (2012): *Model checking software product lines with SNIP*. *STTT* 14(5), pp. 589–612, doi:10.1007/s10009-012-0234-1.

[17] A. Classen, M. Cordy, P. Heymans, A. Legay & P.-Y. Schobbens (2014): *Formal semantics, modular specification, and symbolic verification of product-line behaviour*. *Sci. Comput. Program.* 80(B), pp. 416–439, doi:10.1145/2499777.2499781.

[18] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay & J.-F. Raskin (2013): *Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking*. *IEEE TSE* 39(8), pp. 1069–1089, doi:10.1109/TSE.2012.86.

[19] M. Clavel *et al.*, editor (2007): *All About Maude — A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. *LNCS* 4350, Springer, doi:10.1007/978-3-540-71999-1.

[20] M. Cordy, P.-Y. Schobbens, P. Heymans & A. Legay (2013): *Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-features*. In: *ICSE*, IEEE, pp. 472–481, doi:10.1109/ICSE.2013.6606593.

[21] K. Czarnecki, S. Helsen & U.W. Eisenecker (2004): *Staged Configuration Using Feature Models*. In R.L. Nord, editor: *SPLC*, *LNCS* 3154, Springer, pp. 266–283, doi:10.1007/978-3-540-28630-1_17.

[22] C. Dubslaff, S. Klüppelholz & C. Baier (2014): *Probabilistic Model Checking for Energy Analysis in Software Product Lines*. In W. Binder, E. Ernst, A. Peternier & R. Hirschfeld, editors: *MODULARITY*, ACM, pp. 169–180, doi:10.1145/2577080.2577095.

[23] M. Erwig & E. Walkingshaw (2011): *The Choice Calculus: A Representation for Software Variation*. *ACM Trans. Softw. Eng. Methodol.* 21(1):6, doi:10.1145/2063239.2063245.

[24] C. Ghezzi & A.M. Sharifloo (2013): *Model-based verification of quantitative non-functional properties for software product lines*. *Inform. Softw. Technol.* 55(3), pp. 508–524, doi:10.1016/j.infsof.2012.07.017.

[25] S. Gilmore, M. Tribastone & A. Vandin (2014): *An Analysis Pathway for the Quantitative Evaluation of Public Transport Systems*. In E. Albert & E. Sekerinski, editors: *IFM*, *LNCS* 8739, Springer, pp. 71–86, doi:10.1007/978-3-319-10181-1_5.

[26] S. Gnesi & M. Petrocchi (2012): *Towards an executable algebra for product lines*. In: *FMSPLE workshop at SPLC*, ACM, pp. 66–73, doi:10.1145/2364412.2364424.

[27] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak & A.S. Peterson (1990): *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University. Available at `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231`.

[28] K.G. Larsen & A. Legay (2014): *Statistical Model Checking: Past, Present, and Future*. In T. Margaria & B. Steffen, editors: *ISoLA*, *LNCS* 8802, Springer, pp. 135–142, doi:10.1007/978-3-662-45231-8_10.

[29] A. Legay, B. Delahaye & S. Bensalem (2010): *Statistical Model Checking: An Overview*. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G.J. Pace, G. Rosu, O. Sokolsky & N. Tillmann, editors: *RV*, *LNCS* 6418, Springer, pp. 122–135, doi:10.1007/978-3-642-16612-9_11.

[30] L. Mendonça de Moura & N. Bjørner (2008): *Z3: An Efficient SMT Solver*. In C.R. Ramakrishnan & J. Rehof, editors: *TACAS*, *LNCS* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[31] D. Pianini, S. Sebastio & A. Vandin (2014): *Distributed Statistical Analysis of Complex Systems Modeled Through a Chemical Metaphor*. In: *HPCS*, IEEE, pp. 416–423, doi:10.1109/HPCSim.2014.6903715.

[32] V.A. Saraswat & M.C. Rinard (1990): *Concurrent Constraint Programming*. In F.E. Allen, editor: *POPL*, ACM, pp. 232–245, doi:10.1145/96709.96733.

[33] I. Schaefer & M.H. ter Beek (2014): *Fomal Methods and Analyses in Software Product Line Engineering*. In T. Margaria & B. Steffen, editors: *ISoLA*, *LNCS* 8802, Springer, pp. 253–256, doi:10.1007/978-3-662-45234-9_18.

[34] P.-Y. Schobbens, P. Heymans & J.-C. Trigaux (2006): *Feature Diagrams: A Survey and a Formal Semantics*. In: *RE*, IEEE, pp. 136–145, doi:10.1109/RE.2006.23.

[35] S. Sebastio, M. Amoretti & A. Lluch Lafuente (2014): *A Computational Field Framework for Collaborative Task Execution in Volunteer Clouds*. In G. Engels & N. Bencomo, editors: *SEAMS workshop at ICSE*, ACM, pp. 105–114, doi:10.1145/2593929.2593943.

[36] S. Sebastio & A. Vandin (2013): *MultiVeStA: Statistical Model Checking for Discrete Event Simulators*. In A. Horvath, P. Buchholz, V. Cortellessa, L. Muscariello & M.S. Squillante, editors: *ValueTools*, ACM, pp. 310–315, doi:10.4108/icst.valuetools.2013.254377.

[37] K. Sen, M. Viswanathan & G.A. Agha (2005): *VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems*. In: *QEST*, IEEE, pp. 251–252, doi:10.1109/QEST.2005.42.

[38] T. Thüm, S. Apel, C. Kästner, I. Schaefer & G. Saake (2014): *A Classification and Survey of Analysis Strategies for Software Product Lines*. *ACM Comput. Surv.* 47(1):6, doi:10.1145/2580950.

[39] M. Tribastone (2014): *Behavioral Relations in a Process Algebra for Variants*. In S. Gnesi, A. Fantechi, P. Heymans, J. Rubin & K. Czarnecki, editors: *SPLC*, ACM, pp. 82–91, doi:10.1145/2648511.2648520.

[40] M. Varshosaz & R. Khosravi (2013): *Discrete Time Markov Chain Families: Modeling and Verification of Probabilistic Software Product Lines*. In: *FMSPLE workshop at SPLC*, ACM, pp. 34–41, doi:10.1145/2499777.2500725.