Statistical Model Checking for Product Lines

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

Published in:
Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques

Link to article, DOI:
10.1007/978-3-319-47166-2_8

Publication date:
2016

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):
ter Beek, M. H., Legay, A., Lluch Lafuente, A., & Vandin, A. (2016). Statistical Model Checking for Product Lines. In T. Margaria, & B. Steffen (Eds.), *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10–14, 2016, Proceedings, Part I* (Vol. 9952, pp. 114-33). Springer. Lecture Notes in Computer Science
https://doi.org/10.1007/978-3-319-47166-2_8

Statistical Model Checking for Product Lines

Maurice H. ter Beek[1], Axel Legay[2],
Alberto Lluch Lafuente[3], and Andrea Vandin[4]

[1] ISTI–CNR, Pisa, Italy
[2] Inria Rennes, France
[3] DTU, Lyngby, Denmark
[4] IMT Lucca, Italy

Abstract. We report on the suitability of statistical model checking for the analysis of quantitative properties of product line models by an extended treatment of earlier work by the authors. The type of analysis that can be performed includes the likelihood of specific product behaviour, the expected average cost of products (in terms of the attributes of the products' features) and the probability of features to be (un)installed at runtime. The product lines must be modelled in QFLan, which extends the probabilistic feature-oriented language PFLan with novel quantitative constraints among features and on behaviour and with advanced feature installation options. QFLan is a rich process-algebraic specification language whose operational behaviour interacts with a store of constraints, neatly separating product configuration from product behaviour. The resulting probabilistic configurations and probabilistic behaviour converge in a discrete-time Markov chain semantics, enabling the analysis of quantitative properties. Technically, a Maude implementation of QFLan, integrated with Microsoft's SMT constraint solver Z3, is combined with the distributed statistical model checker MultiVeStA, developed by one of the authors. We illustrate the feasibility of our framework by applying it to a case study of a product line of bikes.

1 Introduction

Recently, much effort is put into making (process-algebraic) modelling languages and formal analysis techniques amenable to product lines [7,13,23,28,32,36,37]. The challenge is to handle their inherent variability, due to which the number of possible products to be analysed may be exponential in the number of features.

In [10], two of the authors introduced the feature-oriented language FLan implemented in Maude [18], allowing analyses like consistency checking by SAT solving and model checking. In FLan, a rich set of process-algebraic operators allows one to specify the configuration and the behaviour of a product line, while a constraint store allows one to specify all constraints from feature models as well as additional action constraints typical of feature-oriented programming. The execution of a process is constrained by the store (e.g. to avoid introducing inconsistencies), but a process can also query the store (e.g. to resolve configuration options) or update the store (e.g. to add new features, even at runtime).

In [8], we equipped FLan with a means to specify probabilistic product line models, resulting in PFLan. In PFLan, each action (including those installing a feature, possibly at runtime) is equipped with a rate to represent uncertainty, failure rates, randomisation or preferences. An executable Maude implementation, together with the statistical model checker MultiVeStA [34], allows to estimate the likelihood of specific configurations or behaviour of product lines to measure non-functional aspects like quality of service, reliability or performance.

In [9], we enriched PFLan with the possibility to uninstall or replace features at runtime and with advanced quantitative constraint modelling options based on the 'cost' of features, i.e. attributes related to non-functional aspects like reliability, weight or price. The result, QFLan, offers three constraint modelling options:

1. Arithmetic relations among feature attributes (e.g. the total cost of a set of features must be less than a given threshold);
2. Propositions relating the absence or presence of a feature to a constraint of type 1 (e.g. if a certain feature is present, then the total cost of a set of features must be less than a given threshold);
3. Action constraints conditioning the runtime execution of an action by a constraint of type 1 (e.g. a certain action can be executed only if the total cost of the set of features constituting the product is less than a given threshold).

The uninstallation or replacement of features can be the result of malfunctioning or of the need to install a better version of the feature (e.g. a software update). We will illustrate this in a case study, together with examples of each of the above types of constraints. Note that these are significantly more complex constraints than the ones that are commonly associated with attributed feature models [12].

Feature attributes typically are not Boolean [19], meaning that the problem of deciding whether or not a product satisfies an attributed feature model with quantitative constraints requires more general satisfiability-checking techniques than mere SAT solving. This leads to the use of Satisfiability Modulo Theory (SMT) solvers like Microsoft's Z3 [20], which allow one to deal with richer notions of constraints, like arithmetic ones. In fact, an important contribution of [9] is the adoption of SMT solving by integrating Z3 in the Maude QFLan interpreter.

In [9], we combined the Maude/Z3 QFLan interpreter with MultiVeStA to be able to apply SMC to product lines. Formally, our SMC approach is to perform a sufficient number of probabilistic simulations of a QFLan model of a product line to obtain statistical evidence (with a predefined level of statistical confidence) of the quantitative properties being verified. Such properties are formulated in MultiVeStA's property specification language MultiQuaTEx. SMC offers unique advantages over exhaustive (probabilistic) model checking. First, SMC does not need to generate entire state spaces and hence scales better without suffering from the combinatorial state-space explosion problem typical of model checking. In particular in the context of product lines, given their possibly exponential number of products, this outweighs the main disadvantage of having to give up on obtaining exact results (100% confidence) with exact analysis techniques like (probabilistic) model checking. Second, SMC scales better with hardware resources since the set of simulations to be carried out can be trivially parallelised and distributed. MultiVeStA, indeed, can be run on multi-core machines, clusters or distributed computers with almost linear speedup. A unique selling point of MultiVeStA is that it can use the same set of simulations for checking numerous properties at once, thus offering further reductions of computing time. Details on (probabilistic) model checking can be found in [4] and on SMC in [26, 27].

While we know of several, quite different, approaches that apply probabilistic model checking to product lines [16,21,22,24,29,38], to the best of our knowledge, we were the first to apply SMC to product lines in [8, 9]. In this paper, however, we give more details of QFLan and of the case study and report more analyses.

Outline. Section 2 presents QFLan. A case study of a product line of bikes is modelled in QFLan in Section 3. In Section 4, we show how to apply SMC to QFLan models by analyses over the case study. Section 5 concludes the paper.

2 Modelling Product Lines with QFLan

The feature-oriented language QFLan [9] is an evolution of PFLan [8], a probabilistic process algebra that separates declarative configuration from procedural runtime aspects. The FLan family (FLan [10], PFLan [8], QFLan [9]) is inspired by the concurrent constraint programming paradigm of [31], its adoption in process calculi [15], and its stochastic extension [14]. A constraint store allows to specify all common constraints from feature models (and more) in a *declarative* manner, while a rich set of process-algebraic operators allows to specify the configuration and behaviour of product lines in a *procedural* manner. The semantics unifies *static* (configuration) and *dynamic* (runtime) feature selection.

QFLan's core notions are *features*, *constraints*, *processes* and *fragments* (i.e. constrained processes), cf. its syntax in Fig. 1. More precisely, the syntactic categories F, S and P correspond to fragments, constraint stores (with constraints from K, using arithmetic expressions over feature attributes from E) and processes (with actions from A), respectively. The universe of (primitive) features is denoted by $\mathcal{F}$, that of actions by $\mathcal{A}$ and that of propostions by $\mathcal{P}$.

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

Fig. 1. QFLan syntax ($f, g \in \mathcal{F}$, $r \in \mathbb{R}^{+}$, $a \in \mathcal{A}$, $p \in \mathcal{P}$, $\bowtie \in \{\leq, <, =, \neq, >, \geq\}$, $\pm \in \{+, -, \times, \div\}$)

The declarative part of QFLan is represented by a constraint store on features extracted from the product line requirements with additional information (e.g. about the context wherein the product will be operated). Two important notions of a constraint store S are the *consistency* of S, denoted by *consistent(S)* (which amounts to logical satisfiability of all constraints constituting S) and the *entailment* $S \vdash c$ of constraint c in S (which amounts to logical entailment).

A constraint store contains any term generated by S according to QFLan's syntax. The basic constraint stores are $\top$ (true, i.e. no constraint at all), $\bot$ (false, i.e. an inconsistent constraint) and arbitrary Boolean constraints over propositions generated by K, exploiting the well-known fact that feature constraints can be expressed using Boolean propositions. Constraints can be combined by juxtaposition (its semantics amounts to logical conjunction) of basic constraints.

While Boolean encodings of feature constraints allow to handle all common constraints, we provide syntactic sugar for two common cross-tree constraints: $f \triangleright g$ expresses that feature f requires feature g, whereas $f \otimes g$ expresses that features f and g mutually exclude each other (i.e. they are alternative). We in fact use such logical encodings to reduce consistency checking and entailment to logical satisfiability (and hence exploit Z3's SAT/SMT solving capabilities).

We assume $\mathcal{P}$ to contain a Boolean predicate $has(f)$ to denote the presence of feature f in a product. Let $\mathcal{P}_{\mathcal{F}}$ denote a product of the product line. In our case study, $\neg has(g)$ then models $g \notin \mathcal{P}_{\mathcal{F}}$, i.e. a bike without an engine. A QFLan novelty is that we also consider quantitative constraints based on arithmetic relations among feature attributes. In our case study, we could use a constraint $\neg has(g) \rightarrow \sum_{f \in \mathcal{P}_{\mathcal{F}}} weight(f) \leq 10$ to impose a weight bound on non-electric bikes.

QFLan moreover admits a class of *action constraints*, reminiscent of featured transition systems (FTS) [17]. In an FTS, transitions are labelled with actions and with feature expressions, i.e. Boolean constraints over the set of features. We associate arbitrary constraints to actions rather than to transitions (and we also equip actions with rates, discussed below). In general, we assume that each action a may have a constraint $do(a) \rightarrow p$, where $p \in \mathcal{P}$ is a proposition. Such constraints act as a kind of guards to allow or forbid the execution of actions.

The procedural part of QFLan is represented by *processes* which can be combined by non-deterministic choice, in sequence or in parallel, and which can consist of the empty process or of a single (rated) action followed by a process. We distinguish ordinary actions from $\mathcal{A}$ and special actions $\mathsf{install}(f)$ (dynamic installation of a feature f), $\mathsf{uninstall}(f)$ (dynamic uninstallation of a feature f), $\mathsf{replace}(f, g)$ (dynamic replacement of feature f by g) and $\mathsf{ask}(K)$ (query the store for the validity of constraint K). We will see below that each action type is treated differently in the operational semantics. As anticipated, each action moreover has an associated *rate*, 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. These action rates, originating from PFLan, allow one to specify probabilistic aspects of product line models (e.g. the behaviour of the user of a product, failure rates of the components of a product or the likelihood of installing a certain feature at a specific moment). We will illustrate all this in our example in Section 3.

Finally, a *fragment* F is a term $[S \mid P]$ composed of a constraint store S and a process P. These components may influence each other according to the concurrent constraint programming paradigm [31]: a process may update its store which, in turn, may condition the execution of the process' actions. For the sake of simplicity, initial fragments are such that S uniquely characterises a product of a product line (i.e. for each feature f, S contains either $has(f)$ or $\neg has(f)$).

The operational semantics of fragments is formalised in terms of the state transition relation $\rightarrow \subseteq \mathbb{N}^{\mathbb{F}\times\mathbb{R}^{+}\times\mathbb{F}}$ defined in Fig. 2, where $\mathbb{F}$ denotes the set of all terms generated by F in the grammar of Fig. 1. Note that we use multisets of transitions to deal with the possibility of multiple instances of a transition $F \xrightarrow{r} G$. Technically, such a reduction relation is defined in structural operational semantics (SOS), 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. 3.

$$(\textsc{Inst})\ \frac{consistent(S\ has(f))}{[S\ \neg has(f) \mid (\mathsf{install}(f), r).P] \xrightarrow{r} [S\ has(f) \mid P]}$$

$$(\textsc{Unst})\ \frac{consistent(S\ \neg has(f))}{[S\ has(f) \mid (\mathsf{uninstall}(f), r).P] \xrightarrow{r} [S\ \neg has(f)\ \mid P]}$$

$$(\textsc{Rpl})\ \frac{consistent(S\ \neg has(f)\ has(g))}{[S\ has(f)\ \neg has(g)\ \mid (\mathsf{replace}(f,g), r).P] \xrightarrow{r} [S\ \neg has(f)\ has(g) \mid P]}$$

$$(\textsc{Act})\ \frac{S = (do(a) \rightarrow K) \qquad S \vdash K}{[S \mid (a,r).P] \xrightarrow{r} [S \mid P]} \qquad (\textsc{Ask})\ \frac{S \vdash K}{[S \mid (\mathsf{ask}(K), r).P] \xrightarrow{r} [S \mid P]}$$

$$(\textsc{Or})\ \frac{[S \mid P] \xrightarrow{r} [S' \mid P']}{[S \mid P + Q] \xrightarrow{r} [S' \mid P']} \qquad (\textsc{Seq/Par})\ \frac{[S \mid P] \xrightarrow{r} [S' \mid P']}{[S \mid P\,?\,Q] \xrightarrow{r} [S' \mid P'\,?\,Q]}\ ? \in \{;\,,\,\|\}$$

Fig. 2. Reduction semantics of QFLan

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

Fig. 3. Structural congruence in QFLan

As usual, the reduction rules in Fig. 2 are expressed as a set of premises (above the line) and a conclusion (below the line). The reduction relation implicitly defines a labeled transition system (LTS), with rates as labels. It is straightforward to obtain a discrete-time Markov chain (DTMC) from such an LTS by normalising the rates into [0..1] such that in each state, the sum of the rates of its outgoing transitions equals one. In the resulting DTMC, a transition label corresponds to the probability that the transition is taken from its source state. Recall that we advocate the use of SMC since it uses on-the-fly generated simulations of the DTMC, which in general is too large to be generated explicitly.

The rules Inst, Unst, Rpl and Act are very similar, all allowing a process to execute an action if certain constraints are satisfied. Rules Inst, Unst and Rpl deal with the installation, removal and replacement of features, respectively, and are applicable as long as they do not introduce inconsistencies. Rule Act forbids inconsistencies with respect to action constraints. A typical action constraint is $do(a) \rightarrow has(f)$, i.e. action a is subject to the presence of feature f. Rule Ask formalises the $\mathsf{ask}(\cdot)$ operation semantics from concurrent constraint programming [31], blocking a process until a proposition can be derived from the store. Rules Or, Seq and Par, finally, are standard, formalising non-deterministic choice, sequential composition and interleaving parallel composition, respectively. Note that non-determinism introduced by choice and parallel composition is probabilistically resolved in the aforementioned DTMC semantics.

We note three ways to include a feature f in a product configuration. First, an *explicit*, *declarative* way is to include the proposition $has(f)$ in the initial store; this is the way to include core features. Second, an *implicit*, *declarative* way is to derive f from other constraints; this is the way to include features that are not known as core features, but that turn out to be enforced by the 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 runtime, possibly by replacement.

3 A Product Line of Bikes

In this section, we briefly describe a product line of bikes that we have used as a case study to validate our approach. It stems from an ongoing collaboration with PisaMo S.p.A., a public mobility company of the Municipality of Pisa, in the context of the European project QUANTICOL (`www.quanticol.eu`). PisaMo introduced the bike-sharing system *CicloPi* in the city of Pisa in 2013. It is supplied and maintained by Bicincittà S.r.l. (`www.bicincitta.com`).

We performed requirements elicitation on documents given to us by PisaMo and Bicincittà to distill a product line of bikes. We identified the common and variable features of the bikes they sell as part of their bike-sharing systems, including indicative prices, to which we added some features after consulting a number of documents on the technical characteristics and prices of bikes and their components as currently being sold by major bike vendors. The resulting model has thus more variability than typical in bike-sharing systems. Indeed, vendors of such systems traditionally allow little variation to their customers (e.g. most vendors only sell bikes with a so-called step-thru frame, a.k.a. open frame or low-step frame, typical of utility bikes instead of considering other kind of frames as we do). This is partly due to the difficulties of analysing systems with high variability to provide guarantees on the deployed products and services.

The resulting *attributed feature model* [12], depicted in Fig. 4, is an and/or-tree of features of a product line, regulating their presence in products: a trivial root feature is always present, *optional* features may be present provided their parent is, *mandatory* features must be present provided their parent is, exactly one *alternative* feature must be present provided their parent is and at least one *or* feature must be present whenever their parent is. A *cross-tree constraint* either *requires* the presence of another feature for a feature to be present or *excludes* two features to both be present. Ignoring the attributes, this model of 20 non-trivial features yields $1,314$ different products. This number can be reduced by quantitative constraints over feature attributes (e.g. limiting the price or weight of a bike) but not so much as to mitigate the inherent exponential explosion.

In Fig. 4, the primitive features (leaves of the tree) are equipped with non-functional attributes, like *price* and *weight* or *load*, which represent the specific feature's price in euros, weight in kilos and computational load, respectively.

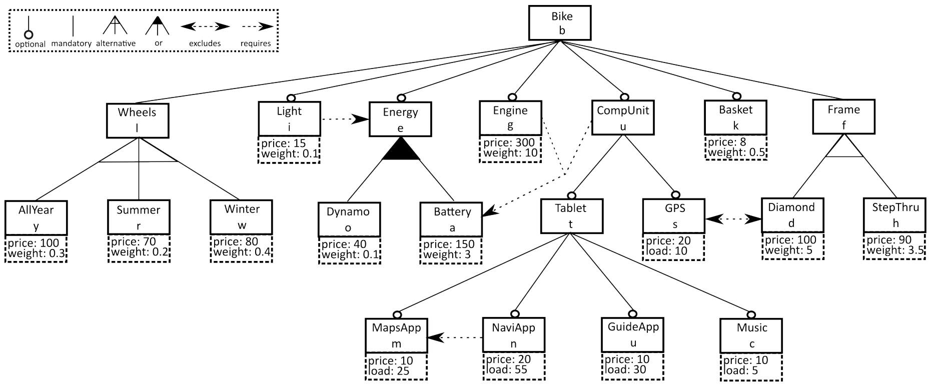


Fig. 4. Attributed feature model of bikes product line (shorthand names for features)

Given the set $\mathcal{F}$ of all features, a product of the product line is identified by a non-empty subset $\mathcal{P}_{\mathcal{F}} \subseteq \mathcal{F}$ that moreover fulfills the additional quantitative constraints over features and attributes.[5] As we have seen in the Introduction, these can range from rather simple constraints (e.g. $price(u) \leq 20$, i.e. the price of the computational unit must be less than 20 euros) to quite more complex ones (e.g. $g \notin \mathcal{P}_{\mathcal{F}} \rightarrow \sum_{f \in \mathcal{P}_{\mathcal{F}}} weight(f) \leq 10$, i.e. a bike without engine cannot weigh more than 10 kilos). Without such constraints, deciding whether or not a product satisfies a feature model reduces to Boolean satisfiability (SAT), which can efficiently be computed with SAT solvers [6]. However, we specifically allow quantitative constraints, requiring the use of SMT solvers like Microsoft's Z3 [20].

In our case study, we consider the following constraints:

(C1) $\sum_{f \in \mathcal{P}_{\mathcal{F}}} price(f) \leq 600$: a bike may cost at most 600 euros;
(C2) $\sum_{f \in \mathcal{P}_{\mathcal{F}}} weight(f) \leq 15$: a bike may weigh up to 15 kilos;
(C3) $\sum_{f \in \mathcal{P}_{\mathcal{F}}} load(f) \leq 100\%$: a bike's computational load may not exceed 100%.

Constraints (C1)–(C3) will be part of the constraint store of the QFLan model of the case study presented below. As such, they prohibit the execution of any action (e.g. the runtime (un)installation or replacement of features) that would violate these constraints since its execution would make the store inconsistent. Furthermore, the store contains two constraints similar to (C1) as explicit constraints on actions, specifying the precise subset of actions affected by them. These constraints will be used in the behavioural part of the QFLan model, presented below, to forbid selling bikes cheaper than 250 euros (C4) and to forbid dumping broken (irreparable) bikes that cost more than 400 euros (C5):

(C4) $do(sell) \rightarrow \sum_{f \in \mathcal{P}_{\mathcal{F}}} price(f) \geq 250$;
(C5) $do(irreparable) \rightarrow \sum_{f \in \mathcal{P}_{\mathcal{F}}} price(f) \leq 400$.

[5] The attribute functions extend to non-primitive features and products in a straightforward manner (e.g. the function $load \colon \mathcal{F} \rightarrow \mathbb{N}$, associated to the attribute *load*, extends to $load(\mathcal{P}_{\mathcal{F}}) = \sum \{ load(f) \mid f \in \mathcal{P}_{\mathcal{F}} \}$.

The behaviour of the bikes product line is based on a bike-sharing scenario that we abstracted from *CicloPi*, with some additional futuristic behaviour concerning still to be realised features such as the use of electric bikes and the possible runtime installation of apps. A rough sketch of it is depicted in Fig. 5.

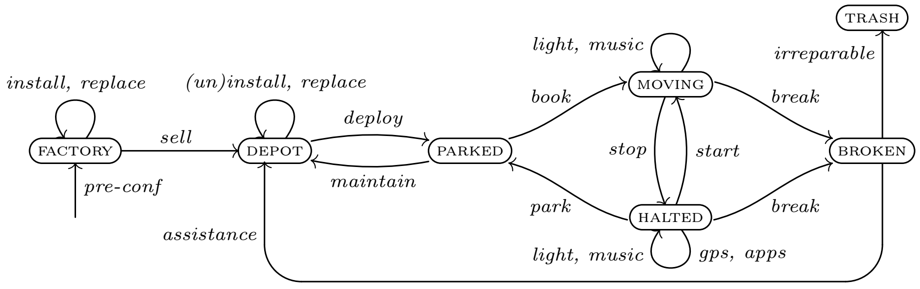


Fig. 5. Sketch of bike-sharing behaviour

Initially, we assume a *pre-conf* igured bike, containing precisely one of the alternative subfeatures from each of the core features Wheels (l) and Frame (f), to arrive in the (initial state) FACTORY (a process). In the QFLan model of this product line, described below, we will assume an initial product to contain the features AllYear (y) and Diamond (d). Moreover, all actions that we will describe next actually have an associated rate (omitted in Fig. 5 to avoid clutter).

In FACTORY (e.g. of Bicincittà), further features may be *install*ed or *replace*d (e.g. different wheels or a different frame). At a certain point, one may *sell* the configured bike (as part of a bike-sharing system), but only if it costs at least 250 euro (to satisfy constraint (C4) on action *sell*), after which it arrives at the DEPOT (e.g. of PisaMo). It may then be ready to be *deploy*ed as part of the bike-sharing system run from the DEPOT, or it may first need to be further fine-tuned (i.e. *(un)install* or *replace* factory-installed features). Once *deploy*ed, it results PARKED in one of the docking stations of the bike-sharing system (e.g. *CicloPi*).

A user may *book* a PARKED bike and *start* biking (MOVING). While biking, a user may decide to listen to *music* or switch on the *light*, in case the corresponding features are installed. If a user wants to consult a *gps* or one of the *apps* (a map, a navigator or a guide), then (s)he first needs to *stop* biking, resulting in a HALTED bike, from where (s)he may *start* biking again or *park* the bike in a docking station. Unfortunately, a bike may also *break*, resulting in a BROKEN bike. Hence, *assistance* from the bike-sharing system exploiter arrives. If the bike can be fixed, it is brought to the DEPOT (and bikes are *maintain*ed by regularly taking PARKED bikes into the DEPOT). If the damage is too severe, and the bike has a price of at most 400 euros (to satisfy constraint (C5) on action *irreparable*), then we dump the bike in the TRASH.

This behaviour is probabilistic, in the sense that in case of several enabled actions some may occur with a higher likelihood than others. Such a probabilistic specification models the uncertainty of the behaviour of the bike, its components and its interacting environment (the users, the exploiters, road conditions, etc.).

The following are some typical properties of interest for the case study:

P_1 Average price, weight and load of a bike when it is first deployed, or as time progresses;

P_2 For each of the 15 primitive features, the probability to have it installed when a bike is first deployed, or as time progresses;

P_3 The probability for a bike to be disposed of;

P_4 The probability to uninstall a factory-installed feature of a bike during a given time interval after it was sold.

When analysed at the first deployment of a bike, P_1 and P_2 are useful for studying a sort of *initial scenario*, in order to estimate the required initial investments and infrastructures. For instance, bikes with a high price and a high load (i.e. with a high technological footprint) or equipped with a battery might require docking stations with specific characteristics or they might have to be collected for the night to be stored safely. Instead, analysing P_1 and P_2 as time progresses provides an indication of how those values evolve, e.g. to estimate the average value in euros of a deployed bike and the monetary consequences of its loss.

From a more general perspective, properties like P_2 measure how often (on average) a feature is actually installed in a product from a product line, which is important information for those responsible for the production or programming of a specific feature or software module. Property P_3 is similar to P_2, but it allows to estimate how often, on average, a bike is dumped in the trash.

Property P_4, finally, is useful for analysing the effect of the factory's pre-configuration choices, and to adapt them to better fit specific scenarios. It might be worthwhile, e.g., to reconsider the installation of a certain feature if there is a high probability of uninstalling it shortly after.

In the remainder of this section, we show how we can specify the case study in QFLan, after which the rest of this paper is devoted to showing how to analyse the above properties with QFLan's tool support.

In Fig. 6, we provide a QFLan model of the bikes product line. Fragment FR is composed of a constraints store S and a process F. The former has five subsets:

DS Constraints from the feature diagram of Fig. 4, like $d \otimes h$, requiring precisely one feature among Diamond and StepThru to be installed, and $g \triangleright a$, requiring the Battery feature to be installed whenever the Engine feature is;

PS Predicates for the attributes of the concrete features in the feature diagram of Fig. 4, like $\mathsf{price}(y) = 100$ and $\mathsf{weight}(y) = 0.3$, indicating that the AllYear feature costs 100 euros and weighs 0.3 kilos;

QS Quantitative constraints affecting all actions, i.e. (C1)–(C3);

AS Action constraints discussed above, like (C4), (C5) or $do(c) \rightarrow has(c)$, requiring Music to be installed in order to play music;

IS The initially installed feature set $has(y) \quad has(d)$, implying that the AllYear and Diamond features are pre-installed.

The process F specifies the behaviour of the bikes product line. In particular, it has one process for each node in Fig. 5. F corresponds to FACTORY, implemented as a choice, weighted by the rates, among three main activities:

(1) With rate 7 the bike is sold and sent to the depot (D corresponds to DEPOT). This action can only be executed if (C4) is respected;
(2) Install optional features and iterate on F. The installations are performed only if FS and QS are respected;
(3) Replace pre-installed mandatory exclusive features IS, i.e. Wheels or Frame. Again, FS and QS must be preserved.

Note that in (2) we assume that Music (c) is the feature installed with higher probability, followed by MapsApp (m), Dynamo (o) and Light (i). Recall that the semantics of QFLAN (Fig. 2) forbids the re-installation of installed features. In (3), we favour the replacement of Winter or Summer wheels by AllYear ones. A frame may be changed as well, but with lower probability. The actual probability to replace Winter or Summer wheels by AllYear ones is $\frac{10+10}{10+10+5+5+5+5+3+3} = {}^{20}/_{46}$, whereas the probability to change a frame is $\frac{3+3}{46} = {}^{6}/_{46}$. Ideally, the rates in an product line model are inferred from statistical analyses of its historical records.

D corresponds to DEPOT, and is similar to F. One obvious difference is the possibility to perform an action *deploy* leading to P (P corresponds to PARKED). In addition, features may be uninstalled in the depot to allow for customisation. Optional features can be installed and uninstalled with the same rates, except for Engine(g), Battery (a) and Dynamo (o), which can be uninstalled with a lower rate to penalise their occurrences. This modelling choice is justified by the fact that it is reasonable to assume that uninstalling such features might cost more than installing them. We further assume that the frame identifies the bike that was sold, and thus it cannot be modified in the depot. The final action that can occur in the depot is an interesting one: a Battery (a) can be replaced with a much cheaper Dynamo (o). According to the semantics of QFLAN, this action is performed only if no subfeatures of CompUnit nor the Engine are currently installed (cf. Fig. 4). This is useful to reduce costs and weight, in case some previously installed feature requiring the Battery has by now been uninstalled.

The remaining processes P, M, H, B and T correspond to PARKED, MOVING, HALTED, BROKEN and TRASH, respectively. These processes are rather simple and are faithful to their description above. The process T installs a fictitious feature *trashed* to express the bike's disposal, after which it evolves into the idle process.

Note that F is a pure configuration process, while D is not. In fact, once parked a bike can be returned to D so features can be (un)installed or replaced at runtime. This is an example of a staged configuration process, in which some optional features are bound at runtime rather than at configuration time.

The interested reader can find the full specification of the case study at `http://sysma.imtlucca.it/tools/multivesta/qflan/`

4 Statistical Model Checking of QFLan Models

In this section, we first briefly explain MultiVeStA's SMC capabilities and then set some parameters for the analyses described in the second part of this section.

MultiVeStA [34] is a distributed statistical model checker co-developed and maintained by one of the authors. It extends statistical analysis tools VeStA [35]

$$
\begin{aligned}
FR &\doteq [\, S \mid F \,] \\
S &\doteq DS \quad PS \quad QS \quad AS \quad IS \\
DS &\doteq y \otimes r \otimes w \quad d \otimes h \; \ldots \; g \triangleright a \quad n \triangleright m \; \ldots \\
PS &\doteq \mathsf{price}(y) = 100 \quad \mathsf{weight}(y) = 0.3 \; \ldots \; \mathsf{price}(c) = 100 \quad \mathsf{load}(c) = 5 \; \ldots \\
QS &\doteq \mathsf{price}(b) < 800 \quad \mathsf{weight}(b) < 20 \quad \mathsf{load}(b) < 100 \\
AS &\doteq do(sell) \rightarrow (\mathsf{price}(b) > 250) \; \ldots \; do(c) \rightarrow has(c) \; \ldots \\
IS &\doteq has(y) \quad has(d) \\
F &\doteq (sell, 7).D \\
&\quad \textit{// Installing optional features} \\
&+ (\mathsf{install}(s), 6).F + (\mathsf{install}(i), 10).F + (\mathsf{install}(n), 6).F + (\mathsf{install}(o), 10).F + (\mathsf{install}(c), 20).F \\
&+ (\mathsf{install}(g), 4).F + (\mathsf{install}(a), 5).F + (\mathsf{install}(u), 3).F + (\mathsf{install}(m), 10).F + (\mathsf{install}(k), 8).F \\
&\quad \textit{// Replacing mandatory and exclusive features} \\
&+ (\mathsf{replace}(y, r), 5).F + (\mathsf{replace}(y, w), 5).F + (\mathsf{replace}(r, y), 10).F + (\mathsf{replace}(r, w), 5).F \\
&+ (\mathsf{replace}(w, y), 10).F + (\mathsf{replace}(w, r), 5).F + (\mathsf{replace}(d, h), 3).F + (\mathsf{replace}(h, d), 3).F \\
D &\doteq (deploy, 10).P \\
&\quad \textit{// Installing optional features; same as F} \\
&+ (\mathsf{install}(s), 6).F + (\mathsf{install}(i), 10).F + (\mathsf{install}(n), 6).F + (\mathsf{install}(o), 10).F + (\mathsf{install}(c), 20).F \\
&+ (\mathsf{install}(g), 4).F + (\mathsf{install}(a), 5).F + (\mathsf{install}(u), 3).F + (\mathsf{install}(m), 10).F + (\mathsf{install}(k), 8).F \\
&\quad \textit{// Uninstalling optional features; same features and rates as installing \ldots} \\
&+ (\mathsf{uninstall}(s), 6).F + (\mathsf{uninstall}(i), 10).F + (\mathsf{uninstall}(n), 6).F + (\mathsf{uninstall}(c), 20).F \\
&+ (\mathsf{uninstall}(u), 3).F + (\mathsf{uninstall}(m), 10).F + (\mathsf{uninstall}(k), 8).F \\
&+ \ldots \textit{except for} \\
&+ (\mathsf{uninstall}(g)), 1).D + (\mathsf{uninstall}(a), 2).D + (\mathsf{uninstall}(o), 3).D \\
&\quad \textit{// Replacing mandatory and exclusive features; like F, but Frame cannot be changed} \\
&+ (\mathsf{replace}(y, r), 5).F + (\mathsf{replace}(y, w), 5).F + (\mathsf{replace}(r, y), 10).F + (\mathsf{replace}(r, w), 5).F \\
&+ (\mathsf{replace}(w, y), 10).F + (\mathsf{replace}(w, r), 5).F \\
&\quad \textit{// Replacing battery by dynamo in case no features requiring a Battery are installed} \\
&+ (\mathsf{replace}(a, o), 1).D \\
P &\doteq (book, 10).M + (maintain, 1).D \\
M &\doteq (stop, 5).H + (break, 1).B + (i, 20).M + (c, 20).M \\
H &\doteq (i, 10).H + (c, 20).H + (s, 10).H + (u, 10).H + (m, 10).H + (n, 10).H \\
&+ (park, 5).P + (start, 5).M + (break, 1).B \\
B &\doteq (assistance, 10).D + (irreparable, 1).T \\
T &\doteq (\mathsf{install}(trashed), 1).\emptyset
\end{aligned}
$$

Fig. 6. QFLan specification of bikes product line

and PVeStA [2] with distributed statistical analysis capabilities. It allows easy integration with any existing discrete event simulator or formalism catering for probabilistic simulations. It has already been used successfully in the analysis of a broad variety of scenarios, including public transportation systems [25], volunteer clouds [33], crowd-steering [30], swarm robotics [11], opportunistic network protocols [3], contract-oriented middleware [5] and software product lines [8, 9].

Below we will describe the tool's usage for obtaining statistical estimations of quantitative properties of QFLan specifications, repeating and extending the analysis results reported in [9]. MultiVeStA provides such estimations by means of efficient distributed statistical analysis techniques known from statistical model checking (SMC) [26,27]. The integration of MultiVeStA and QFLan is available at `http://sysma.imtlucca.it/tools/multivesta/qflan/` together with all files necessary to reproduce the experiments discussed in this paper.

MultiVeStA's property specification language MultiQuaTEx is a highly flexible extension of QuaTEx [1] with the following features: real-valued observations on the system states (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 then 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).

We can obtain probabilistic simulations of a QFLan model by executing it step-by-step applying the rules of Fig. 2, each time selecting one of the computed one-step next-states according to the probability distribution resulting from normalising the rates of the generated transitions (cf. Section 2).

Classical SMC techniques allow one to perform analyses like "is the probability that a property holds greater than a given threshold?" or "what is the probability that a property is satisfied?". Next to performing such classical SMC analyses over products, MultiVeStA can estimate the expected values of properties that can take on any value from $\mathbb{R}$, like "what is the average cost, weight or load of products configured according to a product line specification?".

MultiVeStA estimations are computed as the mean value of n samples obtained from n independent simulations, with n large enough to grant that the size of the $(1-\alpha)\times 100\%$ *confidence interval* is bounded by δ, i.e. if a MultiQuaTEx expression is estimated as $\overline{x} \in \mathbb{R}$, then with probability $(1-\alpha)$ its actual expected value belongs to the interval $[\overline{x}-\delta/2, \overline{x}+\delta/2]$. A confidence interval is thus specified in terms of two parameters: α and δ. For all the experiments discussed below, we fix $\alpha = 0.1$ and we set $\delta = 20.0$ for costs, $\delta = 1.0$ for weights, $\delta = 5.0$ for loads, $\delta = 1.0$ for steps and $\delta = 0.1$ for probabilities. The experiments were performed on a laptop equipped with a 2.4 GHz Intel Core i5 processor and 4 GB of RAM, distributing the simulations among its 4 cores.

We now apply MultiVeStA to analyse properties P_1–P_4 from Section 3 on the bikes product line case study. We start with P_1 and P_2, which we study both at a precise point in time (when a bike is first deployed) and as time progresses.

Listing 1.1 depicts a MultiQuaTEx expression for evaluating P_1 and P_2 at a bike's first deployment. Lines 1-4 define a parametric recursive temporal operator `ObsAtFD` which is evaluated against a simulation. It takes as input a string `obs` representing a *state observation* of interest. Then, if the bike has completed its first deployment (Line 1), the value in the current simulation state of the provided observation is returned (Line 2). Otherwise, the operator is recursively evaluated in the next simulation state (Line 3). Intuitively, `#` is the one-step temporal operator, while real-valued observations on the current state are evaluated resorting to the keyword `s.rval`. A number of predefined observations is supported. For instance, we can query whether a given feature is currently installed, obtaining 1 if it is installed and 0 otherwise. An example can be found in Line 1 for `first-deploy`, a fictitious feature installed when terminating the first phase of deployment (for ease of presentation, we did not show this in Section 3). In addition, we can query for price, weight and load of the current product, obtained by summing the corresponding values for all installed features. Finally, Lines 5-6 specify the properties to be studied: the expected `price`, `weight` and `load` of bikes (Line 5), as well as the probabilities of installing each of the 15 primitive features (Line 6), all measured at first deployment.

```
1 ObsAtFD(obs) = if {s.rval("first-deploy") == 1.0}
2 then s.rval(obs)
3 else #ObsAtFD(obs)
4 fi;
5 eval E[ObsAtFD("price")]; eval E[ObsAtFD("weight")]; eval E[ObsAtFD("load")];
6 eval E[ObsAtFD("y")]; eval E[ObsAtFD("r")]; ... eval E[ObsAtFD("c")];
```

Listing 1.1. P_1 and P_2 at first deployment

Listing 1.1 shows how MultiQuaTEx allows one to express more properties at once (18 in this case) which are estimated by MultiVeStA reusing the same simulations. A procedure considering that each property might require a different number of simulations is adopted to satisfy the given confidence interval.

We evaluated the MultiQuaTEx expression of Listing 1.1 against the QFLan model of Section 3. The results are shown in the first row of Table 1. Notably, the probability of installing an Engine (g) is very low, estimated at 0 (i.e. with probability 0.9 it belongs to $[0, 0.05]$, according to the given confidence interval). This is presumably due to constraints (C1) and (C2), imposing bikes to cost less than 600 euros, and weighing less than 15 kilos. In fact, the estimated average price and weight of bikes at first deployment is 391.91 euros and 7.8 kilos, respectively, while an Engine costs 300 euros and weighs 10 kilos. In order to confirm this hypothesis, we analysed the same property in a new model where (C1) and (C2) allow bikes to cost at most 800 euros and weigh at most 20 kilos. The results, shown in the second row of Table 1, confirm our hypothesis. This reveals that the constraints are sort of in disagreement with the quantitative attributes of the features. The estimation of the average price required $1,200$ simulations, as opposed to 120 in the aforementioned case. This is because the looser constraints of the latter analysis induce a higher variability of bike prices. In fact, the installation of an Engine, the most expensive among the considered features, results in a steep increase of bike prices.

Table 1. Properties P_1 and P_2 evaluated when a bike is first deployed

		Attributes (P_1)			Features (P_2)														
C1	C2	price	weight	load	y	r	w	i	o	a	g	m	n	u	c	s	k	d	h
600	15	391.91	7.80	33.50	0.57	0.24	0.18	0.59	0.84	0.92	0.0	0.50	0.20	0.24	0.47	0.17	0.60	0.61	0.39
800	20	509.83	11.98	34.45	0.54	0.23	0.19	0.57	0.88	0.92	0.40	0.52	0.21	0.25	0.47	0.20	0.63	0.60	0.40

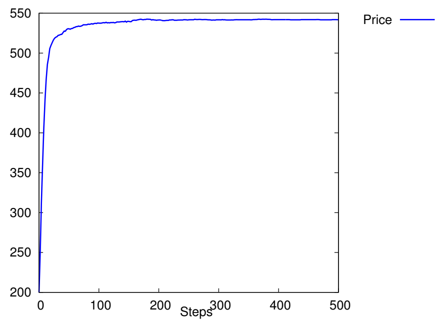
We now discuss the variants of P_1 and P_2 measured as time progresses, demonstrating how MultiVeStA can be used to analyse properties upon varying a parameter, in this case the number of performed simulation steps. Listing 1.2 shows how the MultiQuaTEx expression of Listing 1.1 can be made parametric with respect to a given set of simulation steps. First, the temporal operator was modified so that it is evaluated with respect to a specific step given as parameter (Lines 1-4). Second, it was necessary to specify a range of values for the parameter. Lines 5-7 specify that we are interested in measuring the properties for steps going from 0 to 500, with an increment of 2. Recall from Section 3 that dumping a bike is modelled by the installation of a fictitious feature *trash*. Hence, we can use the expression of Listing 1.2 to measure also P_3 (the probability of a bike being dumped) by simply adding `E[ObsAtStep("trashed",st)]` (Line 7).

```
1  ObsAtStep(obs,st) = if {s.rval("steps") == st}
2  then s.rval(obs)
3  else #ObsAtStep(obs,st)
4  fi;
5  eval parametric(E[ObsAtStep("price",st)], E[ObsAtStep("weight",st)],
6  E[ObsAtStep("load",st)], E[ObsAtStep("y",st)], E[ObsAtStep("r",st)], ...,
7  eval E[ObsAtStep("c",st)]; E[ObsAtStep("trashed",st)], st, 0, 2, 500);
```

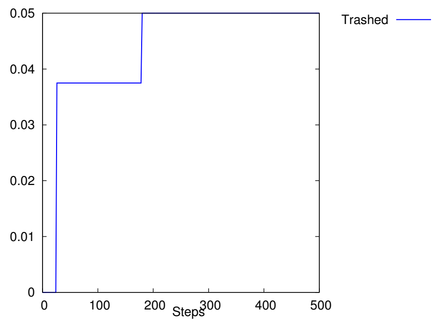
Listing 1.2. P_1–P_3 for varying simulation steps

Next, we evaluated the parametric property of Listing 1.2 against the model. We report the results obtained for the model in which (C1) and (C2) bound the price and weight of the bike to 800 and 20, respectively. All such analyses (19×251 different properties) were evaluated using the same simulations. Overall, $1,200$ simulations were necessary. The results are presented in six plots in Fig. 7: one each for average prices (7a), weights (7c) and loads (7e), one for the probability of dumping a bike (7b), one for the probability of installing features (7d) and one for the probability of uninstalling pre-installed features (7f).

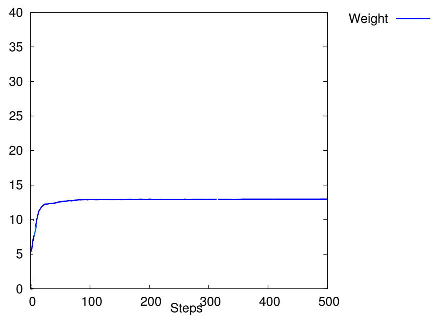
Figure 7a show that the average price (on the y-axis) of the intermediate bikes generated from the product line starts at 200 euros, in line with the initial configuration (*IS* in Fig. 6, i.e. with AllYear and Diamond installed). Then the price grows with respect to the number of performed simulation steps. In particular, it is possible to see an initial fast growth until reaching an average price of about 510 euros, after which the growth slows down, reaching about 537 euros at step 100 and 542 at step 500. This is consistent with the QFLan specification, which has a pre-configuration phase (FACTORY) during which a number of features can be installed, followed by a customisation phase (DEPOT), where features can be (un)installed and replaced. We recall that features cannot be uninstalled in the FACTORY and we note that uninstalling features in the DEPOT



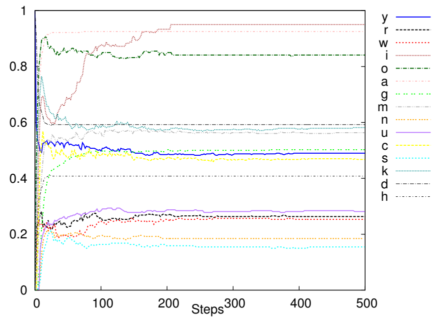
(a) P_1 and P_2 (average price)



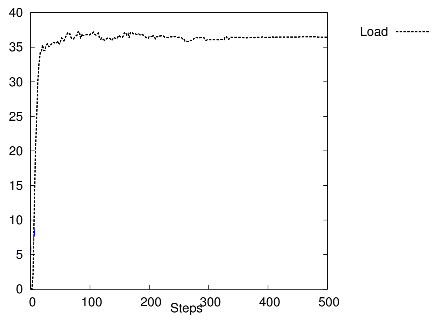
(b) P_3 (probability to dump a bike)



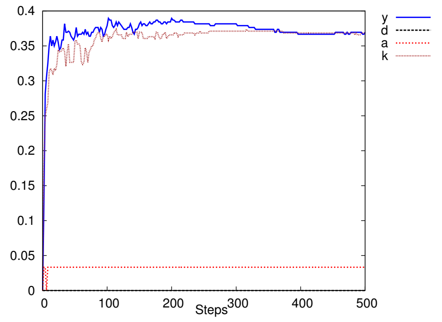
(c) P_1 and P_2 (average weight)



(d) P_1 and P_2 (installation probabilities)



(e) P_1 and P_2 (average load)



(f) P_4 (uninstallation probabilities)

Fig. 7. Results of measuring P_1–P_4 with MultiVeStA

does not introduce decrements of the price, on average. A manual inspection of the data revealed that the phase of fast growth terminates after about 19 steps. This is consistent with the analysis described in the second row of Table 1, where the average number of steps to complete the first DEPOT phase is estimated as being close to 19 (not shown in the table). In addition, the average price at the end of such a phase is estimated to be around 510 euros, exactly as in Table 1. Note, finally, that the probability of a bike to return to the DEPOT after its first deployment is quite low, viz. $^1/_{11}$, based on a transition from PARKED with rate 10 towards MOVING and with rate 1 towards DEPOT (cf. Fig. 6). Thus, on average, the price of bikes is only slightly affected by (un)installations and replacements performed by successive phases spent in the DEPOT.

Figures 7c and 7e shows that the average weight and load, respectively, of a bike evolve similarly to its average price: an initial phase of fast growth of 19 steps is followed by one of slower growth.

Figure 7b shows that bikes are dumped with a very low probability. The reason is twofold. First, the transition from BROKEN to TRASH has a much lower rate than the one to DEPOT, i.e. often a bike is not *irreparable*, and likewise for those from MOVING and HALTED to BROKEN, i.e. bikes do not *break* all the time (cf. Fig. 6). Second, the average price of a bike quickly exceeds 400 euros (cf. Fig. 7a) and action constraint (C5) prohibits dumping such bikes.

Figure 7d confirms that also the probabilities (on the y-axis) for each of the features to be installed evolve similarly to the average price, weight and load of the generated products, although, clearly, with different scales. Note that the pre-installed features AllYear (y) and Diamond (d) both have probability 1 of being installed at step 0, after which their probabilities decrease.

We conclude this section by considering P_4. This property was analysed against a slight variant of the behavioural scenario, viz. without the FACTORY phase but with the following set of four features pre-installed: AllYear (y), Diamond (d), Battery (a) and Basket (k). Subsequently, we studied how the probability of *not* having each of these features installed at a certain simulation step changes upon varying the considered simulation step. The corresponding MultiQuaTEx expression, adapted from Listing 1.2, is given in Listing 1.3.

```
ObsAtStep(obs,st) = if {s.rval("steps") == st}
then 1 - s.rval(obs)
else #ObsAtStep(obs,st)
fi;
eval parametric(E[ObsAtStep("y",st)], E[ObsAtStep("d",st)],
E[ObsAtStep("a",st)], E[ObsAtStep("k",st)], st, 0, 2, 500);
```

Listing 1.3. P_4 for varying simulation steps

We again focus on the case in which (C1) and (C2) bound the cost and weight of bikes to 800 and 20, respectively. The analysis required 380 simulations. The results are presented in Fig. 7f, where we can once more appreciate the two distinct phases of faster and slower growth. A manual inspection of the data revealed that the turning point of these two phases again lies around step 19. Diamond (d) has 0 probability of being uninstalled. This is coherent with the QFLAN model, as the Frame can be replaced only during the FACTORY phase, which was however removed for this particular experiment. As regards the three remaining features, Fig. 7f highlights the effect of constraints on the behaviour of QFLan specifications. In fact, we can clearly see that the feature set can be partitioned in two, based on the probability of being uninstalled: Battery (a) has almost no probability of being uninstalled, while AllYear (y) and Basket (k) have a higher probability to be uninstalled. The lower uninstall probability manifested by Battery (a) is justified by the fact that it is required by the Engine as well as by all subfeatures of CompUnit, thus the presence of even one of these features in the store (i.e. installed on the bike) blocks the uninstallation of Battery (a). Finally, the remaining two features, AllYear (y) and Basket (k), uninstalled with higher probability, produce a similar graph. This is consistent with process D for DEPOT given in Fig. 6, as AllYear (y) can be replaced with rate 10 (due to the two replace actions) and Basket (k) can be uninstalled with rate 8.

5 Conclusions and Future Work

We have presented the probabilistic feature-oriented language QFLan and its tool support: a prototypical Maude interpreter integrated with Z3 and MultiVeStA, originally introduced in a short paper at SPLC'15 [9]. In this paper, we provide more explanations of QFLan, more details of the case study and more analyses. The bikes product line case study was developed from interactions with companies we work with in the context of the European project QUANTICOL.

Our analyses have revealed a number of interesting properties of the product line specification, such as the existence of an apparent disagreement between the constraints imposed on the total price and weight of bikes with respect to the price and weight of some of its features, as well as the high probability of replacing certain features that tend to appear in initial factory configurations, which suggest to prioritise their installation in the earliest stages of configuration.

To improve the performance of our analyses, which currently take minutes, we developed a Java implementation of QFLan integrated with Z3 and MultiVeStA, reducing analysis time to seconds. We now work on completing this tool with a user-friendly interface for the specification and SMC of QFLan models.

Acknowledgements. Maurice ter Beek and Andrea Vandin are supported by the EU project QUANTICOL, 600708. We thank Bicincittà and M. Bertini of PisaMo for the case study and D. Lucanu, G. Rosu, A. Stefanescu and A. Arusoaie for sharing their Maude/Z3 integration, which we adapted for our purposes.

References

1. G. A. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based Specification Language for Probabilistic Object Systems. *ENTCS*, 153:213–239, 2005.
2. M. AlTurki and J. Meseguer. PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In A. Corradini, B. Klin, and C. Cîrstea, editors, *CALCO*, volume 6859 of *LNCS*, pages 386–392. Springer, 2011.

3. S. Arora, A. Rathor, and M. V. P. Rao. Statistical Model Checking of Opportunistic Network Protocols. In *Proceedings 11th Asian Internet Engineering Conference (AINTEC'15)*, pages 62–68. ACM, 2015.
4. C. Baier and J. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
5. M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, and L. Pompianu. A Contract-Oriented Middleware. In C. Braga and P. C. Ölveczky, editors, *FACS*, volume 9539 of *LNCS*, pages 86–104. Springer, 2015.
6. D. Batory. Feature Models, Grammars, and Propositional Formulas. In J. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
7. M. H. ter Beek, D. Clarke, and I. Schaefer. *Special Issue on Formal Methods in Software Product Line Engineering. J. Log. Algebr. Meth. Program.*, 85(1), 2016.
8. M. H. ter Beek, A. Legay, A. Lluch Lafuente, and A. Vandin. Quantitative Analysis of Probabilistic Models of Software Product Lines with Statistical Model Checking. *EPTCS*, 182:56–70, 2015.
9. M. H. ter Beek, A. Legay, A. Lluch Lafuente, and A. Vandin. Statistical analysis of probabilistic models of software product lines with quantitative constraints. In *Proceedings 19th International Software Product Line Conference (SPLC'15)*, pages 11–15. ACM, 2015.
10. M. H. ter Beek, A. Lluch Lafuente, and M. Petrocchi. Combining Declarative and Procedural Views in the Specification and Analysis of Product Families. In *Proceedings 17th International Software Product Line Conference (SPLC'13)*, volume 2, pages 10–17. ACM, 2013.
11. L. Belzner, R. De Nicola, A. Vandin, and M. Wirsing. Reasoning (on) Service Component Ensembles in Rewriting Logic. In S. Iida, J. Meseguer, and K. Ogata, editors, *Futatsugi Festschrift*, volume 8373 of *LNCS*, pages 188–211. Springer, 2014.
12. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.*, 35(6), 2010.
13. P. Borba, M. B. Cohen, A. Legay, and A. Wąsowski. *Analysis, Test and Verification in The Presence of Variability. Dagstuhl Reports*, 3(2):144–170, 2013.
14. L. Bortolussi. Stochastic Concurrent Constraint Programming. *ENTCS*, 164:65–80, 2006.
15. M. G. Buscemi and U. Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In R. De Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.
16. P. Chrszon, C. Dubslaff, S. Klüppelholz, and C. Baier. Family-Based Modeling and Analysis for Probabilistic Systems - Featuring ProFeat. In P. Stevens and A. Wąsowski, editors, *FASE*, volume 9633 of *LNCS*, pages 287–304. Springer, 2016.
17. A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Softw. Eng.*, 39(8):1069–1089, 2013.
18. M. Clavel *et al.*, editor. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.
19. M. Cordy, P. Schobbens, P. Heymans, and A. Legay. Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-features. In *Proceedings 35th International Conference on Software Engineering (ICSE'13)*, pages 472–481. IEEE, 2013.
20. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
21. C. Dubslaff, C. Baier, and S. Klüppelholz. Probabilistic Model Checking for Feature-Oriented Systems. In S. Chiba, É. Tanter, E. Ernst, and R. Hirschfeld, editors, *Transactions on AOSD XII*, volume 8989 of *LNCS*, pages 180–220. Springer, 2015.

22. C. Dubslaff, S. Klüppelholz, and C. Baier. Probabilistic Model Checking for Energy Analysis in Software Product Lines. In *Proceedings 13th International Conference on Modularity (MODULARITY'14)*, pages 169–180. ACM, 2014.
23. M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1), 2011.
24. C. Ghezzi and A. Sharifloo. Model-based verification of quantitative non-functional properties for software product lines. *Inform. Softw. Technol.*, 55(3):508–524, 2013.
25. S. Gilmore, M. Tribastone, and A. Vandin. An Analysis Pathway for the Quantitative Evaluation of Public Transport Systems. In E. Albert and E. Sekerinski, editors, *IFM*, volume 8739 of *LNCS*, pages 71–86. Springer, 2014.
26. K. G. Larsen and A. Legay. Statistical model checking: Past, present, and future. In T. Margaria and B. Steffen, editors, *ISoLA*, volume 8802 of *LNCS*, pages 135–142. Springer, 2014.
27. A. Legay, B. Delahaye, and S. Bensalem. Statistical Model Checking: An Overview. In H. Barringer *et al.*, editor, *RV*, volume 6418 of *LNCS*, pages 122–135. Springer, 2010.
28. M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. DeltaCCS: A Core Calculus for Behavioral Change. In T. Margaria and B. Steffen, editors, *ISoLA*, volume 8802 of *LNCS*, pages 320–335. Springer, 2014.
29. G. Nunes Rodrigues *et al.* Modeling and Verification for Probabilistic Properties in Software Product Lines. In *Proceedings 16th International Symposium on High Assurance Systems Engineering (HASE'15)*, pages 173–180. IEEE, 2015.
30. D. Pianini, S. Sebastio, and A. Vandin. Distributed Statistical Analysis of Complex Systems Modeled Through a Chemical Metaphor. In *Proceedings International Conference on High Performance Computing & Simulation (HPCS'14)*, pages 416–423. IEEE, 2014.
31. V. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Conference Record 17th Annual Symposium on Principles of Programming Languages (POPL'90)*, pages 232–245. ACM, 1990.
32. I. Schaefer and R. Hähnle. Formal methods in software product line engineering. *IEEE Comp.*, 44(2):82–85, 2011.
33. S. Sebastio, M. Amoretti, and A. Lluch Lafuente. A Computational Field Framework for Collaborative Task Execution in Volunteer Clouds. In *Proceedings 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*, pages 105–114. ACM, 2014.
34. S. Sebastio and A. Vandin. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. In *Proceedings 7th International Conference on Performance Evaluation Methodologies and Tools (ValueTools'13)*, pages 310–315. ACM, 2013.
35. K. Sen, M. Viswanathan, and G. A. Agha. VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems. In *Proceedings 2nd International Conference on Quantitative Evaluation of Systems (QEST'05)*, pages 251–252. IEEE, 2005.
36. T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1), 2014.
37. M. Tribastone. Behavioral Relations in a Process Algebra for Variants. In *Proceedings 18th International Software Product Line Conference (SPLC'14)*, pages 82–91. ACM, 2014.
38. M. Varshosaz and R. Khosravi. Discrete Time Markov Chain Families: Modeling and Verification of Probabilistic Software Product Lines. In *Proceedings 17th International Software Product Line Conference (SPLC'13)*, volume 2, pages 34–41. ACM, 2013.