Statistical analysis of probabilistic models of software product lines with quantitative constraints

Beek, M.H. ter ; Legay, A. ; Lluch Lafuente, Alberto; Vandin, A.

Published in:
Proceedings of the 19th International Conference on Software Product Line (SPLC '15)

Link to article, DOI:
10.1145/2791060.2791087

Publication date:
2015

Document Version
Peer reviewed version

Statistical Analysis of Probabilistic Models of Software Product Lines with Quantitative Constraints

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy
m.terbeek@isti.cnr.it

Axel Legay
Inria, Rennes, France
axel.legay@inria.fr

Alberto Lluch Lafuente
Technical University of Denmark, Denmark
albl@dtu.dk

Andrea Vandin
University of Southampton, UK and IMT, Lucca, Italy
a.vandin@soton.ac.uk

ABSTRACT

We investigate the suitability of statistical model checking techniques for the analysis of probabilistic models of software product lines with complex quantitative constraints and advanced feature installation options. Such SPL models are defined in the probabilistic feature-oriented language QFLan. QFLan is a rich process algebra whose operational behaviour interacts with a store of constraints and as such it allows to separate product configuration from product behaviour. The resulting probabilistic configurations and behaviour converge seamlessly in a semantics based on discrete-time Markov chains, thus enabling quantitative analysis. To this aim, we combine a Maude implementation of QFLan, integrated with Microsoft's SMT constraint solver Z3, with the distributed statistical model checker MultiVeStA. This enables analyses that range from the likelihood of specific behaviour to the expected average cost of products, in terms of feature attributes. We illustrate our approach by performing quantitative analyses on a bikes product line case study.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking, Statistical methods*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Process models*; G.3 [**Probability and Statistics**]

General Terms

Design, Experimentation, Verification

Keywords

Product lines, Probabilistic models, Quantitative constraints, Statistical model checking

Draft, under review

1. INTRODUCTION

There is a lot of recent research on lifting successful high-level algebraic modelling languages and formal verification techniques known from single (software) system engineering, such as process calculi and model checking, to (software) product line engineering (SPLE), e.g. [2,5,6,14,20,25,31,32]. The challenge is to handle the variability inherent to SPLs, by which the number of possible products of an SPL may be exponential in the number of features. In [8], we contributed with the feature-oriented language FLan and its implementation in Maude [15], allowing analyses ranging from consistency checking (by means of SAT solving) to model checking.

In FLan, a rich set of process-algebraic operators allows one to specify both the configuration and the behaviour of products, while a constraint store allows one to specify all common constraints known from feature models as well as additional action constraints typical of feature-oriented software development. 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 run time).

In [7], we subsequently equipped FLan with the means to specify probabilistic models of SPLs, resulting in PFLan. The main distinguishing modelling feature of FLan is the clean separation between the configuration and run-time aspects of an SPL. PFLan adds to this the possibility to equip each action (including those that install a feature, possibly at run time) with a rate, which can represent uncertainty, a failure rate, randomisation, or preferences. An executable implementation in Maude, together with the distributed statistical model checker MultiVeStA [30], allows us to estimate the likelihood of specific configurations and behaviour of an SPL, and thus to measure non-functional aspects such as quality of service, reliability, or performance.

An emergent fact of our investigations is the urgent need to consider a number of further aspects in the specification and analysis of behavioural models of SPLs, such as the staged configurations known from dynamic software product lines [12,17] (e.g. removal and update of features) and quantitative constraints (e.g. price constraints). Recent surveys on existing approaches on applying formal analysis techniques in SPLE [31] and discussions on model checking SPL behaviour [8] reveal that there are currently no approaches dealing with such aspects in a unifying framework. Indeed,

we are aware of only a few, quite different, approaches on probabilistic model checking of SPLs [19,21,33], whereas, to the best of our knowledge, [7] contains the only other application of statistical model checking in SPLE and [16] is the only approach (in addition to ours) to the model checking of SPL models with quantitative constraints over feature attributes. However, none of those approaches is able to combine dynamic feature configurations, quantitative constraints and quantitative analyses based on statistical model checking. We aim at filling this gap by extending our framework for the formal specification and analysis of SPLs.

For this purpose, in this paper we enrich PFLAN with the possibility to uninstall and replace features at run time and with quantitative constraint modelling options regarding the 'cost' of features, i.e. feature attributes related to non-functional aspects such as price, weight, reliability, etc. In particular, the novel modelling options we introduce are:

1. Arithmetic relations among feature attributes (e.g. the total cost of a set of features must be less than a certain threshold);
2. Propositions relating the absence or presence of a feature to a quantitative 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 certain threshold);
3. Richer action constraints involving quantitative constraints of type 1 (e.g. a certain action can be performed only if the total cost of the set of features constituting the product is less than a certain threshold).

We call the new language presented in this paper QFLAN. The uninstallation and 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 our case study, as well as the use of each of the above type of quantitative constraints over feature attributes, by providing concrete examples. It is important to note that the above type of quantitative constraints are significantly more complex than the ones that are commonly associated to attributed feature models [10].

As feature attributes are typically not Boolean [16], the problem of deciding whether a product satisfies an attributed feature model with quantitative constraints, requires more general satisfiability-checking techniques than SAT solving. This naturally leads to the use of Satisfiability Modulo Theory (SMT) solvers like Microsoft's Z3 [18], which allow one to deal with richer notions of constraints like arithmetic ones. In fact, an important contribution of this paper is the integration of SMT solving into our approach by means of a combination of our Maude QFLAN interpreter and Z3.

Formally, our statistical model checking approach is to perform a sufficient number of probabilistic simulations of an SPL model 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 [30]. Statistical model checking offers unique advantages over exhaustive (probabilistic) model checking. First, statistical model checking 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 SPLs, 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, statistical model checking 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 advantage of MultiVeStA is that it can use the same set of simulations for checking several properties at the same time, thus offering even further reductions of computing time. Further details on (probabilistic) model checking can be found in [3] and on statistical model checking in [23, 24].

The paper outline is as follows. Section 2 contains a bikes product lines case study. Section 3 presents QFLAN, followed by a QFLAN model of the case study in Section 4. MultiVeStA is introduced in Section 5, followed by experimental quantitative analyses of the case study in Section 6. Section 7 summarises our contributions and future work.

2. BIKES PRODUCT LINE CASE STUDY

We describe in this section a case study that has motivated the extension of our approach to the modelling and analysis of behavioural SPL models and that we have used to validate our novel solutions. We use the case study here as a running example to illustrate the main concepts of our approach and to provide intuitive cases of its possibilities and limitations.

The case study stems from an ongoing collaboration with PisaMo S.p.A., an in-house public mobility company of the Municipality of Pisa, in the context of the European project Quanticol (`www.quanticol.eu`). PisaMo introduced the public bike-sharing system *CicloPi* in the city of Pisa two years ago. This bike-sharing system is supplied by Bicincittà S.r.l. (`www.bicincitta.com`).

To create an attributed feature model of a product line of bikes, we performed requirements elicitation on a set of documents generously shared with us by Bicincittà. This allowed us to extract the main features of the bikes they sell as part of the bike-sharing system, including indicative prices, and to identify their commonalities and variabilities. We then added some features that we found by reading through 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), in part due to the difficulties of analysing systems with high variability to provide guarantees on the deployed products and services. We believe that the progress of SPL analysis techniques (including the contribution of this paper) will help the adoption and hence the provision of richer (bike-sharing) systems with higher variability.

The resulting attributed feature model is depicted in Fig. 1. Without taking the attributes into account this feature model of 21 features gives rise to 1,314 different products. Of course, quantitative constraints over feature attributes can partially reduce the number of products (e.g. some bikes may be too expensive, or too heavy) but not so much as to mitigate the inherent exponential explosion. Such constraints and feature attributes are specified as follows. Each

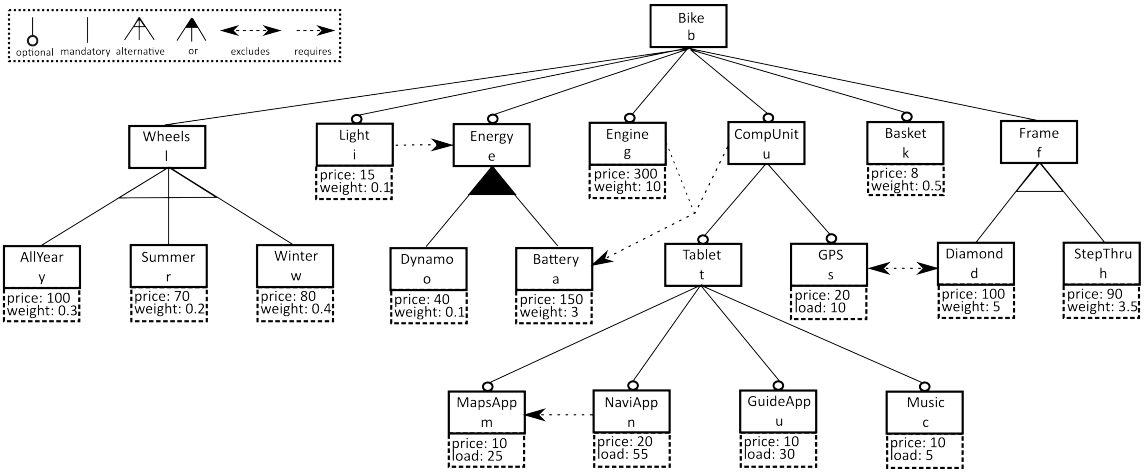


Figure 1: Attributed feature model of bikes product line (with shorthand names)

feature is equipped with a set of 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.[1] The set of all features of the product line is $\mathcal{F} = \{b, l, i, e, g, u, k, f, y, r, w, o, a, t, s, d, h, m, n, u, c\}$. A product $\mathcal{P}$ from the product line is a non-empty subset $\mathcal{P}_\mathcal{F} \subseteq \mathcal{F}$ that moreover fulfills the additional quantitative constraints defined over features and attributes. 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 should 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. if the bike does not have an engine then it cannot weigh more than 10 kilos). Without such constraints, deciding whether a product satisfies a feature model reduces to Boolean satisfiability (SAT), which can efficiently be computed with SAT solvers [4]. However, in this paper we specifically allow such quantitative constraints, which requires the use of SMT solvers like Microsoft's Z3 [18].

For 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 total computational load may not exceed 100%.

Constraints (C1)–(C3) are part of the constraint store of our QFLan model of the case study. As such, they prohibit the execution of any action (e.g. the run-time (un)installation or replacement of features) that would violate these constraints since its execution would result in an inconsistent constraint store. Furthermore, the store also contains two constraints similar to (C1) as constraints on actions, which explicitly specify the precise subset of actions that are affected by them. These constraints are used in the behavioural part of our model, discussed below, to forbid selling bikes that cost less than 250 euros (C4) and to forbid dumping broken (and irreparable) bikes that cost more than 400 euros (C5):

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

[1]We assume b, l, e, u, f, and t to have the sum of the attributes of their respective subfeatures as attribute values.

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

The behaviour associated to our bikes product line is based on a bike-sharing scenario that we abstracted from the bike-sharing system *CicloPi* with some additional behaviour concerning not yet realised features such as the use of electric bikes and the possible run-time installation of apps. A rough sketch of it is depicted in Fig. 2.

Initially, we assume that a pre-configured bike, containing precisely one of the alternative subfeatures from each of the core features *Wheels* and *Frame*, arrives at the initial state FACTORY (a process). In our case study, we assume such an initial product from the bikes product line to contain the feature set $\{y, d\}$. At this point it is important to underline that all actions that we are to describe next actually have an associated rate (omitted in Fig. 2) in the QFLan model of our case study (described in Section 4).

In FACTORY (e.g. of Bicincittà), further features may be installed or replaced (e.g. different wheels or a different frame). At a certain point, the configured bike may be sold (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 in the DEPOT (e.g. of PisaMo). It may then be ready to be deployed as part of the bike-sharing system run from this depot, or it may first need to be further fine-tuned by (un)installing or replacing factory-installed features. Once it is deployed, it actually results PARKED in one of the docking stations of the bike-sharing system (e.g. *CicloPi*).

A user may book a PARKED bike, resulting in a MOVING bike. While biking, a user may decide to listen to music or switch on the light, in case the corresponding features have been installed. If a user wants to consult 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 to bike again or park the bike in a docking station. Unfortunately, the 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. 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. At regular intervals, assistance from the bike-sharing system exploiter takes a PARKED bike to the DEPOT for maintenance.

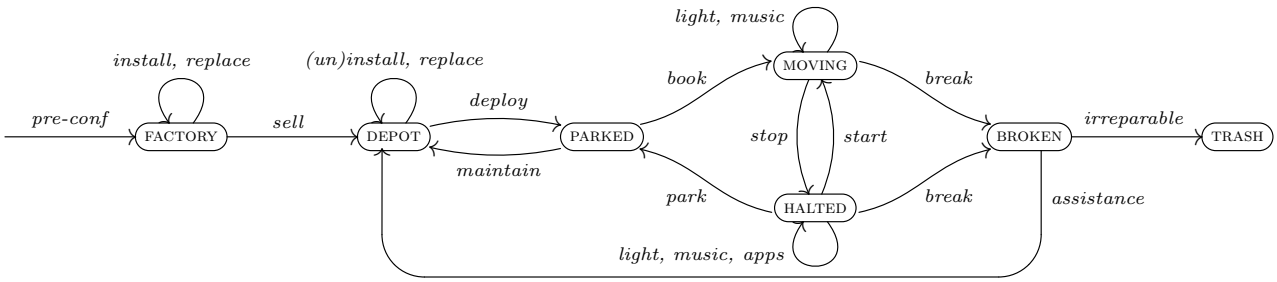


Figure 2: Sketch of bike-sharing behaviour

The above described behaviour is probabilistic, in the sense that in the presence 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.).

Some typical properties of interest on the case study are:

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

(P_2) For each of the 15 primitive features that appear as leaves in the feature model of Fig. 1, the probability to have it installed when a bike is deployed for the first time, or as time progresses;

(P_3) The probability for a bike to be disposed;

(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.

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 worth, e.g., to reconsider the installation of a certain feature if there is a high probability of uninstalling it shortly after.

In the sequel we show how we can specify the case study in QFLan and analyse above properties with its tool support.

3. SYNTAX AND SEMANTICS OF QFLAN

The feature-oriented language QFLan is an evolution of probabilistic PFLan [7], a process algebra that separates declarative (pre-)configuration from procedural run-time aspects. The FLan family (FLan [8], PFLan [7], QFLan) is inspired by the concurrent constraint programming paradigm of [27], its adoption in process calculi [13], and its stochastic extension [11]. A constraint store allows to specify all common constraints from feature models (and more) in *declaratively*, while a rich set of process-algebraic operators allows to specify the configuration and behaviour of product lines in *procedurally*. The semantics unifies *static* (pre-configuration) and *dynamic* (run-time) feature selection.

QFLan's core notions are *features*, *constraints*, *processes* and *fragments* (i.e. constrained processes), cf. its syntax in Fig. 3. More precisely, the syntactic categories F, S, and P correspond to fragments, constraint stores (with constraints from K, using arithmetic expressions of feature attributes from E) and processes (with actions from A), respectively. The universe of (primitive) features is denoted by $\mathcal{F}$.

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

Figure 3: QFLan syntax (with $\mathbf{f}, \mathbf{g} \in \mathcal{F}$, $\mathbf{r} \in \mathbb{R}^+$, $\mathbf{a} \in \mathcal{A}$, $\mathbf{p} \in \mathcal{P}$, $\bowtie \in \{\leq, <, =, \neq, >, \geq\}$, and $\pm \in \{+, -, \div, \times\}$)

The declarative part of QFLan 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 QFLan. The most basic constraint stores are $\top$ (true, i.e. no constraint at all), $\bot$ (false, i.e. an inconsistent constraint), and arbitrary Boolean constraints over a universe $\mathcal{P}$ of propositions (generated by K), exploiting the fact that constraints on features can be expressed using Boolean propositions (cf. [28]). Boolean propositions can also be used to represent additional information such as contextual facts, which however we do not use in this paper. Constraints can be combined by juxtaposition (its semantics amounts to logical conjunction) of basic constraints.

The Boolean encoding of feature constraints allows us as to handle all common constraints, including two common

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

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

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

$$(\textsc{Or})\ \frac{[S\mid P]\xrightarrow{r}[S'\mid P']}{[S\mid P+Q]\xrightarrow{r}[S'\mid P']}\qquad (\textsc{Seq})\ \frac{[S\mid P]\xrightarrow{r}[S'\mid P']}{[S\mid P;Q]\xrightarrow{r}[S'\mid P';Q]}\qquad (\textsc{Par})\ \frac{[S\mid P]\xrightarrow{r}[S'\mid P']}{[S\mid P\parallel Q]\xrightarrow{r}[S'\mid P'\parallel Q]}$$

Figure 4: Reduction semantics of QFLan

cross-tree constraints for which we sometimes use the following ad-hoc syntax: $f\triangleright g$ expresses that feature f requires the feature g, while $f\otimes g$ 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 moreover assume that the universe $\mathcal{P}$ of propositions contains a Boolean predicate $\mathit{has}(f)$ that can be used to denote the presence of a feature f in a product. In our case study, e.g., $\neg\mathit{has}(g)$ models $g\notin\mathcal{P}_{\mathcal{F}}$, i.e. a bike without an engine.

Finally, a novelty of QFLan is that we also consider quantitative constraints based on arithmetic relations among feature attributes. In our case study, e.g., it would be possible to define the constraint $\neg\mathit{has}(g)\to\sum_{f\in\mathcal{P}_{\mathcal{F}}}\mathit{weight}(f)\leq 10$, which imposes a weight bound on non-electric bikes.

As mentioned, QFLan admits a class of *action constraints*, reminiscent of featured transition systems (FTS) [14]. 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 general, we assume that each action a may have a constraint $\mathit{do}(a)\to p$, where $p\in\mathcal{P}$ is a proposition. Such constraints act as a kind of guards to allow or forbid the execution of actions (e.g. the constraints (C4) and (C5) of Section 2).

The procedural part of QFLan is represented by *processes* which can be combined in sequence, in parallel, or with non-deterministic choices, and which can consist of the empty process or of a single (rated) action followed by a process. We distinguish ordinary actions from a universe $\mathcal{A}$ 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 $\mathsf{ask}(K)$ (to query the store for the validity of constraint K). As we will see shortly, each action type is treated differently in the operational semantics. 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 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. We will illustrate all this in our example in Section 4.

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 [27]: a process may update its store which, in turn, may condition the execution of the process' actions. For the sake of simplicity, we consider in this paper initial fragments where S uniquely characterises a product of a product line (i.e. for each feature f, S contains either $\mathit{has}(f)$ or $\neg\mathit{has}(f)$).

The operational semantics of fragments is formalised in terms of the state transition relation $\to\subseteq\mathbb{N}^{\mathbb{F}\times\mathbb{R}^+\times\mathbb{F}}$ defined in Fig. 4, where $\mathbb{F}$ denotes the set of all terms generated by F in the grammar of Fig. 3. 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 a structural congruence relation $\equiv\subseteq\mathbb{F}\times\mathbb{F}$ that axiomatises the structure of processes (e.g. parallel composition and non-deterministic choice are associative and commutative and have the empty process $\emptyset$ as identity, etc.).

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 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. As usual, the reduction rules in Fig. 4 are expressed in terms of a set of premises (above the line) and a conclusion (below the line).

The rules Inst, Unst, Rpl, and Act of the semantics are very similar, all allowing a process to execute an action if certain constraints are satisfied. Rules Inst, Unst, and Rpl deal with 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 $\mathit{do}(a)\to\mathit{has}(f)$, i.e. action a is subject to the presence of feature f. Other examples are (C4) and (C5) of Section 2. Rule Ask formalises the semantics of the $\mathsf{ask}(\cdot)$ operation from concurrent constraint programming [27]. It allows a process to be blocked until a proposition can be derived from the store.

Rules PAR, SEQ, and OR, finally, are standard, formalising 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.

4. BIKES PRODUCT LINE IN QFLAN

Fig. 5 sketches a QFLAN model of our bikes product line. Fragment FR is composed of store S and a process F. The former consists of four sets of constraints:

- FS Constraints from the feature diagram of Fig. 1, like $d \otimes h$, requiring precisely one feature among Diamond and StepThru to be installed;
- AS Constraints on actions discussed in Sections 2 and 3, like (C4) or $do(c) \rightarrow has(c)$, requiring Music to be installed in order to play music;
- QS Quantitative constraints affecting all actions, like (C2);
- IS The initially installed feature set $has(y)$ $has(d)$, implying that AllYear and Diamond are pre-installed.

The process F specifies the behaviour of the bikes product line from Section 2. In particular, it has one process for each node in Fig. 2. 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. 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 preserved;

(3) Replace pre-installed mandatory exclusive features IS, i.e. Wheels or Frame. Again, FS and QS are preserved.

Note that in (2) we assume that Music is the feature installed with higher probability, followed by MapsApp, Dynamo, and Light. Recall that the semantics of QFLAN (Fig. 4) 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.

D corresponds to DEPOT, and is similar to F. Clearly, D differs from F by the possibility to perform an action *deploy* leading to P (i.e. PARKED). In addition, D may also uninstall features, so as to allow for customisation. Optional features can be installed and uninstalled with the same rate by D, except for Engine, Battery, and Dynamo, uninstalled with a lower rate to penalize their occurrences. This modeling choice is justified by the fact that it is reasonable to assume that uninstalling such features might cost more than installing them. In addition, we assume that the frame identifies the bike that was sold, and thus it cannot be modified in D. The final action that D can perform is an interesting one: Battery can be replaced with the much cheaper Dynamo. According to the semantics of QFLAN, this action is performed only if no subfeature of CompUnit or the Engine are currently installed (cf. Fig. 1). 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 in Section 2. The process T installs a fictitious feature *trashed* to express the fact that the bike has been disposed, and then evolve in the idle process.

$$
\begin{aligned}
FR &\doteq [\, S \mid F \,] \\
S &\doteq FS\ AS\ QS\ IS \\
FS &\doteq \ldots \quad AS \doteq \ldots \quad QS \doteq \ldots \quad IS \doteq \ldots \\
F &\doteq (sell, 7).D \\
&\quad //\textit{Installing optional features} \\
&\quad + (\mathsf{install}(s), 6).F + (\mathsf{install}(m), 10).F + (\mathsf{install}(n), 6).F \\
&\quad + (\mathsf{install}(u), 3).F + (\mathsf{install}(c), 20).F + (\mathsf{install}(g), 4).F \\
&\quad + (\mathsf{install}(a), 5).F + (\mathsf{install}(o), 10).F + (\mathsf{install}(i), 10).F \\
&\quad + (\mathsf{install}(k), 8).F \\
&\quad //\textit{Replacing mandatory and exclusive features} \\
&\quad + (\mathsf{replace}(y, r), 5).F + (\mathsf{replace}(y, w), 5).F \\
&\quad + (\mathsf{replace}(r, y), 10).F + (\mathsf{replace}(r, w), 5).F \\
&\quad + (\mathsf{replace}(w, y), 10).F + (\mathsf{replace}(w, r), 5).F \\
&\quad + (\mathsf{replace}(d, h), 3).F + (\mathsf{replace}(h, d), 3).F \\
D &\doteq (deploy, 10).P \\
&\quad //\textit{Installing optional features} \\
&\quad + \ldots \textit{same as } F \\
&\quad //\textit{Uninstalling optional features} \\
&\quad + \ldots \textit{same features and rates as installing, except for} \\
&\quad + (\mathsf{uninstall}(g),1).D + (\mathsf{uninstall}(a),2).D + (\mathsf{uninstall}(o),3).D \\
&\quad //\textit{Replacing mandatory and exclusive features} \\
&\quad + \ldots \textit{same as F, but replacubg just wheels} \\
&\quad //\textit{Replacing battery by dynamo} \\
&\quad + (\mathsf{replace}(a, o), 1).D \\
P &\doteq (book, 10).M + (maintain, 1).D \\
M &\doteq (stop, 5).H + (break, 1).B + (c, 20).M + (i, 20).M \\
H &\doteq (start, 5).M + (break, 1).B + (c, 20).H + (i, 10).H \\
&\quad + (s, 10).H + (u, 10).H + (m, 10).H + (n, 10).H \\
B &\doteq (assistance, 10).D + (irreparable, 1).T \\
T &\doteq (\mathsf{install}(trashed), 1).\emptyset
\end{aligned}
$$

Figure 5: QFLan specification of bikes product line

Note that F is a pure (pre-)configuration process, while D is not. In fact, parked bikes can be brought back into the depot, and thus features can be (un)installed or replaced at run time. 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.

The QFLAN specification is completed with the definition of the attributes of the features as depicted in Fig. 1, not shown here due to reasons of space. The interested reader can find the full specification of the case study at `https://code.google.com/p/multivesta/wiki/QFLan`

5. MODEL CHECKING WITH MULTIVESTA

In this section, we briefly explain the statistical model checking capabilities of MultiVeStA and set the parameters for the actual analyses described in the subsequent section.

MultiVeStA [30] is a distributed statistical model checker that was co-developed and is being maintained by the fourth author. MultiVeStA can easily be integrated with any formalism that allows probabilistic simulations and it has already be used to analyse a wide variety of systems, including transportation systems [22], volunteer clouds [29], crowd-steering [26] and swarm robotic [9] scenarios.

Recently, we investigated the suitability of MultiVeStA for the quantitative analysis of SPL behaviour modelled in PFLAN [7]. In this paper, we use the tool to obtain statisti-

			Attributes (P_1)			Features (P_2)														
C1	C2	steps to deploy	price	weight	load	y	r	w	i	o	a	g	m	n	u	c	s	k	d	h
600	15	17.86	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	18.28	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

Table 1: Properties P_1 and P_2 evaluated at a bike's first deployment.

cal estimations of quantitative properties of QFLan specifications. MultiVeStA provides such estimations by means of distributed analysis techniques known from statistical model checking (SMC) [23, 24]. The integration of MultiVeStA and QFLan is available at `https://code.google.com/p/multivesta/wiki/QFLan` together with all files necessary to reproduce the experiments discussed in this paper.

MultiVeStA's property specification language MultiQuaTEx (an extension of QuaTEx [1]) is very flexible, based on the following ingredients: 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 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).

We obtain probabilistic simulations of a QFLan model by executing it step-by-step applying the rules of Fig. 4, 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 3).

Classical SMC allows 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?". In addition, MultiVeStA also allows one to estimate the expected values of properties that can take on any value from $\mathbb{R}$, like "what is the average cost/weight/load of products configured according to an SPL specification?". Estimations are computed as the mean 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* (CI) is bounded by δ. In other words, 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 CI is thus specified in terms of two parameters: α and δ. In all the experiments discussed in the next section, we fixed $\alpha = 0.1$. Also, 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. 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.

6. ANALYSIS OF BIKES PRODUCT LINE

In this section, we show how MultiVeStA can be used to analyse our bikes product line case study, focusing in particular on properties P_1–P_4 from Section 2. We start with P_1 and P_2, which we study both at a precise point in time (at the first deployment of a bike) and as time progresses.

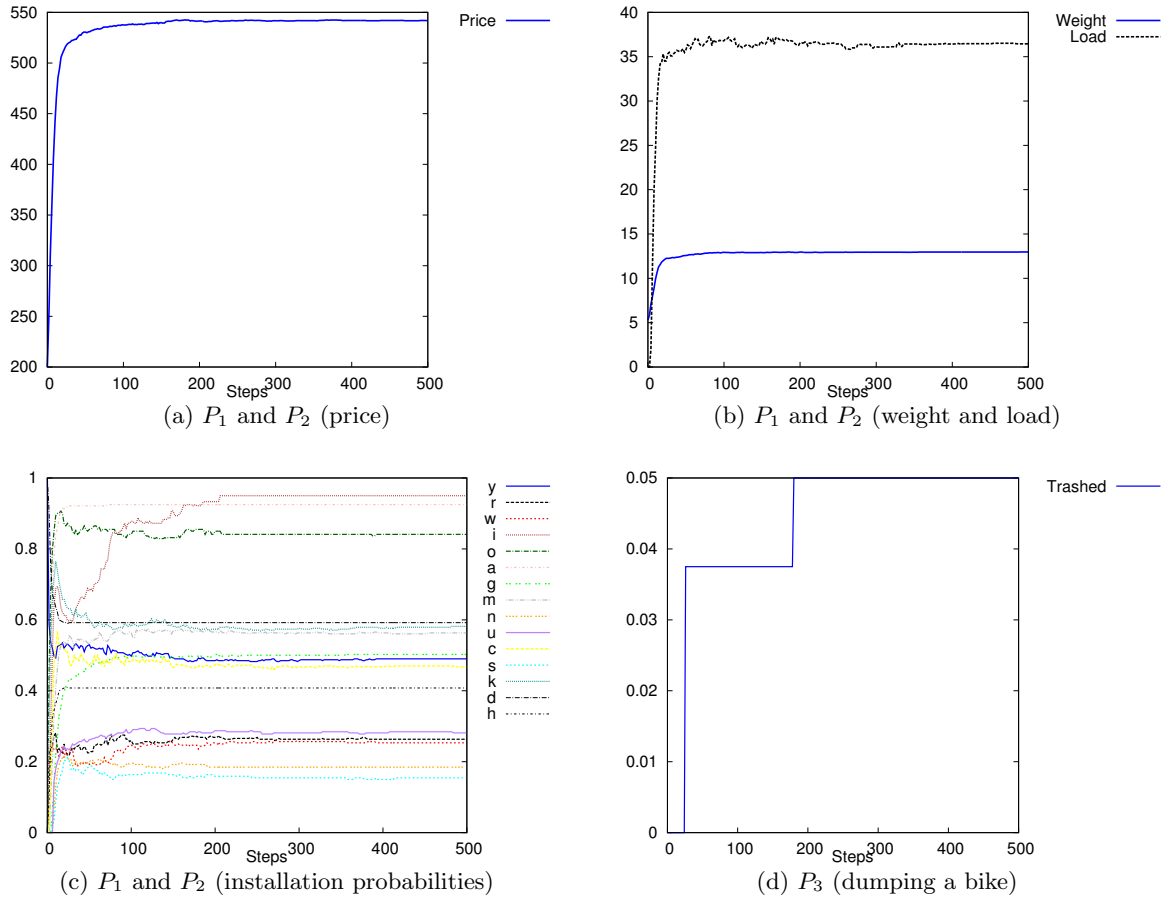
Listing 1 depicts a MultiQuaTEx expression to evaluate P_1 and P_2 at a bike's first deployment. Lines 1-4 define a parametric recursive temporal operator `ObsAtFD` which is evaluated against a simulation. The operator takes in input a string `obs` representing a *state observation* of interest. Then, 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 currently supported, e.g., we can query whether a given feature is currently installed, obtaining 1 if the feature is installed and 0 otherwise. An example is in Line 1 for `first-deploy`, a fictitious feature installed when terminating the first phase of deployment (to ease presentation, we did not show this in Section 4). In addition, we can query for price, weight, and load of the current product, obtained by summing the corresponding values for all installed features, or the number of simulation steps done to obtain the current state. Finally, Lines 5-7 specify the properties to be studied: the expected `price`, `weight`, and `load` of bikes (Lines 5-6), as well as the probabilities of installing each of the 15 primitive features (Line 7), all measured at first deployment. In addition, we also query the expected number of simulation `steps` to perform the first deployment (Line 6).

```
ObsAtFD(obs) = if {s.rval("first-deploy") == 1.0}                    1
                    then s.rval(obs)                                 2
                    else #ObsAtFD(obs)                               3
               fi;                                                   4
eval E[ObsAtFD("price")]; eval E[ObsAtFD("weight")];                 5
eval E[ObsAtFD("load")]; eval E[ObsAtFD("steps")];                   6
eval E[ObsAtFD("y")]; eval E[ObsAtFD("r")];  ...                     7
```

Listing 1: P_1 and P_2 at first deployment

Notably, Listing 1 shows how MultiQuaTEx allows one to express more properties at once (in this case 19) which are estimated by MultiVeStA reusing the same simulations. We remark that a procedure taking into account that each property might require a different number of simulations is adopted to satisfy the given confidence interval CI.

We evaluated the MultiQuaTEx expression of Listing 1 against the model discussed in Section 4. The analysis required 1,340 simulations, performed in about 20 minutes. In particular, `steps` is the property that required more simulations, viz. 1,340, while `price` required only `120` simulations. 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 specified confidence interval). We guess that this is due to the 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 engine costs 300 euros and weighs 10 kilos. In order to confirm this hypothesis, we analysed the same property



(a) P_1 and P_2 (price)

(b) P_1 and P_2 (weight and load)

(c) P_1 and P_2 (installation probabilities)

(d) P_3 (dumping a bike)

Figure 6: Results of measuring P_1–P_3 with MultiVeStA

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 analysis thus revealed that the constraints were in disagreement with the quantitative attributes of the features. The latter analysis required 1,360 simulations, performed in about 20 minutes. In this case the estimation of the average price required 1,200 simulations rather than 120 as in the first 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.

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

Listing 2: P_1–P_3 for varying simulation steps

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 2 shows how the expression of Listing 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 5-8 specify that we are interested in measuring the properties for steps going from 0 to 500, with an increment of 2. Recall from Section 4 that dumping a bike is modelled by the installation of a fictitious feature *trash*. Hence, we can use the expression of Listing 2 to measure also P_3 (the probability of a bike being dumped) by simply adding `E[ObsAtStep("trashed",st)]` (Line 8).

We evaluated the parametric property of Listing 2 against our case study. 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, performed in about 75 minutes. The results are presented in four plots in Fig. 6: one for prices (a), one for weights and loads (b), one for the probabilities of installing features (c), and one for the probability of dumping the bike (d).

Fig. 6(a) shows 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*, 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 our 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 FACTORY does not perform any uninstalling, while we note that the uninstalling actions of DEPOT do 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. In addition, the average price at the end of such a phase is estimated to be around 510 euros, as in Table 1. Note, finally, that the probability of a bike to return to the DEPOT after its first deployment is quite low. In fact, as specified in Fig. 5, PARKED has a transition with rate 10 towards MOVING and one with rate 1 towards DEPOSIT. Thus, in average, the price of bikes is only slightly affected by (un)installations and replacements performed by successive DEPOT phases.

Fig. 6(b) shows that the weight and load of a bike evolve similarly to the price: there is a first phase of growth during the first 19 steps, followed by a slower growth.

As confirmed by Fig. 6(c), the probabilities (on the y-axis) for each of the features that can be installed evolve similarly to the average price, weight, and load of the generated products, although, clearly, with different scales. It is interesting to note that the pre-installed features AllYear (y) and Diamond (d) have probability 1 of being installed at step 0, after which the probability decreases during the first 19 steps.

Fig. 6(d) shows that bikes are dumped with very low probability. The reason is twofold. First, the transition from BROKEN to TRASH has a much lower rate than the one to DEPOT, and similarly for those from MOVING and HALTED to BROKEN (cf. Fig. 5). Second, the average price of bikes quickly rises above 400 euros (Fig. 6(a)), and constraint (C5) prohibits dumping bikes costing more than 400 euros.

We conclude this section by considering P_4. This property was analysed against a slight variant of our scenario, viz. without the FACTORY phase but with the following set of features pre-installed: AllYear (y), Diamond (d), Battery (a), and Basket (k). In particular, we studied how the probability of having each of these 4 features not installed in a certain simulation step changes upon varying the considered simulation step. The corresponding MultiQuaTEx expression can easily be obtained from Listing 2 by changing Line 2 in "`then 1 - s.rval(obs)`", and writing in Lines 5-8 only the "`E`" corresponding to the 4 features. 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 performed in about 15 minutes. The results are presented in Fig. 7, where we can again appreciate the two distinct phases with faster and slower growth, respectively. A manual inspection of the data revealed that the two phases change again around step 19. Diamond (d) has 0 probability of being uninstalled. This is coherent with the considered model, as the frame can be replaced only during the FACTORY phase, removed for this experiment. As regards the 3 remaining features, Fig. 7 highlights the effect of constraints to the behaviour of QFLan

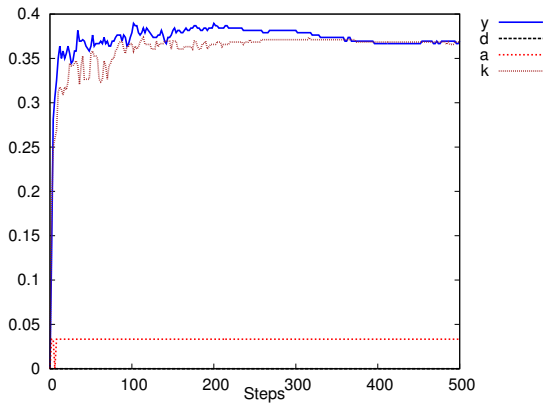


Figure 7: P_4 (uninstalling factory-installed features)

specifications. In fact, we can clearly see that the features can be partitioned in two, based on the probability of being uninstalled: a has almost no probability of being uninstalled, while y and k are uninstalled with higher probability. The lower uninstall probability manifested by a is justified by the fact that the Engine and all CompUnit subfeatures require it, thus the presence of one of these features in the store prevents the uninstallation of a. Finally, the other two features, y and k, uninstalled with higher probability, have a similar graph. This is consistent with process D for DEPOT given in Fig. 5, as AllYear is replaced with rate 10 (due to the two replace actions), while Bike is uninstalled with rate 8.

7. CONCLUSIONS AND FUTURE WORK

In a recent workshop, we have presented the probabilistic feature-oriented language PFLan [7]. In this paper, we have introduced QFLan, which extends PFLan with dynamic uninstallation and replacement of features and with advanced quantitative constraint modelling options, thus allowing for more involved quantitative analyses (now requiring SMT solving). We have achieved this by integrating an efficiently executable Maude implementation of QFLan with Z3 and with the distributed statistical model checker MultiVeStA. We have applied the resulting modelling and analysis framework to a bikes product line case study taken from companies with whom we cooperate in the context of the European project Quanticol. Our analysis has revealed some interesting properties of the model, like the existence of a disagreement among constraints imposed on the price and weight of bikes, and prices and weights of bike components, as well as the high probability of replacing some features that tend to appear in initial configurations, which suggest to prioritise their installation in the early stages of the configuration. All in all, our detailed analysis has served to validate our methodology and its tool support. We believe that our work will hence provide a further contribution towards the adoption of formal specification and analysis techniques in SPLE.

In future work we plan to further develop the integration of Z3 with MultiVeStA, e.g. to equip our toolset with optimisation capabilities, so that users can not only validate configuration choices but also automatically obtain configuration options optimising their objective functions (possibly combining behavioural and non-functional aspects).

8. ACKNOWLEDGEMENTS

Research supported by EU project QUANTICOL (600708) and Italian MIUR project CINA (PRIN 2010LHT4KM).

We thank Bicincittà S.r.l. and Marco Bertini from PisaMo S.p.A. for generously sharing with us relevant information concerning the bikes in Pisa's *CicloPi* bike-sharing system.

We thank Dorel Lucanu, Grigore Rosu, Andrei Stefanescu, and Andrei Arusoaie for sharing with us their own integration of Maude and Z3, which we adapted for our purposes.

9. REFERENCES

[1] G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based Specification Language for Probabilistic Object Systems. *ENTCS*, 153:213–239, 2005.

[2] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. Formal Description of Variability in Product Families. In *SPLC*, pages 130–139. IEEE, 2011.

[3] C. Baier and J. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[4] D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[5] M. H. ter Beek and E. P. de Vink. Software Product Line Analysis with mCRL2. In *SPLC workshop SPLat*, pages 78–85. ACM, 2014.

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

[7] 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. In *FMSPLE*, volume 182 of *EPTCS*, pages 56–70, 2015.

[8] M. H. ter Beek, A. Lluch Lafuente, and M. Petrocchi. Combining Declarative and Procedural Views in the Specification and Analysis of Product Families. In *SPLC workshop FMSPLE*, pages 10–17. ACM, 2013.

[9] L. Belzner, R. De Nicola, A. Vandin, and M. Wirsing. Reasoning (on) Service Component Ensembles in Rewriting Logic. In *Specification, Algebra, and Software*, volume 8373 of *LNCS*, pages 188–211. Springer, 2014.

[10] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.*, 35(6), 2010.

[11] L. Bortolussi. Stochastic Concurrent Constraint Programming. *ENTCS*, 164:65–80, 2006.

[12] J. Bürdek, S. Lity, M. Lochau, M. Berens, U. Goltz, and A. Schürr. Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints. In *VaMoS*. ACM, 2014.

[13] M. G. Buscemi and U. Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In *ESOP*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.

[14] 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 TSE*, 39(8):1069–1089, 2013.

[15] M. Clavel *et al.*, editor. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[16] M. Cordy, P. Schobbens, P. Heymans, and A. Legay. Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-features. In *ICSE*, pages 472–481. IEEE, 2013.

[17] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Using Feature Models. In *SPLC*, volume 3154 of *LNCS*, pages 266–283. Springer, 2004.

[18] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[19] C. Dubslaff, S. Klüppelholz, and C. Baier. Probabilistic Model Checking for Energy Analysis in Software Product Lines. In *MODULARITY*, pages 169–180. ACM, 2014.

[20] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM TOSEM*, 21(1), 2011.

[21] C. Ghezzi and A. M. Sharifloo. Model-based verification of quantitative non-functional properties for software product lines. *Inform. Softw. Technol.*, 55(3):508–524, 2013.

[22] S. Gilmore, M. Tribastone, and A. Vandin. An Analysis Pathway for the Quantitative Evaluation of Public Transport Systems. In *IFM*, volume 8739 of *LNCS*, pages 71–86. Springer, 2014.

[23] K. G. Larsen and A. Legay. Statistical model checking: Past, present, and future. In *ISoLA*, volume 8802 of *LNCS*, pages 135–142. Springer, 2014.

[24] A. Legay, B. Delahaye, and S. Bensalem. Statistical Model Checking: An Overview. In *RV*, volume 6418 of *LNCS*, pages 122–135. Springer, 2010.

[25] M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. DeltaCCS: A Core Calculus for Behavioral Change. In *ISoLA*, volume 8802 of *LNCS*, pages 320–335. Springer, 2014.

[26] D. Pianini, S. Sebastio, and A. Vandin. Distributed Statistical Analysis of Complex Systems Modeled Through a Chemical Metaphor. In *HPCS*, pages 416–423. IEEE, 2014.

[27] V. Saraswat and M. Rinard. Concurrent Constraint Programming. In *POPL*, pages 232–245. ACM, 1990.

[28] P. Schobbens, P. Heymans, and J. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *RE*, pages 136–145. IEEE, 2006.

[29] S. Sebastio, M. Amoretti, and A. Lluch Lafuente. A Computational Field Framework for Collaborative Task Execution in Volunteer Clouds. In *ICSE workshop SEAMS*, pages 105–114. ACM, 2014.

[30] S. Sebastio and A. Vandin. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. In *ValueTools*, pages 310–315. ACM, 2013.

[31] 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.

[32] M. Tribastone. Behavioral Relations in a Process Algebra for Variants. In *SPLC*, pages 82–91. ACM, 2014.

[33] M. Varshosaz and R. Khosravi. Discrete Time Markov Chain Families: Modeling and Verification of Probabilistic Software Product Lines. In *SPLC workshop FMSPLE*, pages 34–41. ACM, 2013.