Probabilistic Model Checking for Feature-oriented Systems*

Clemens Dubslaff, Christel Baier, and Sascha Klüppelholz

Faculty of Computer Science
Technische Universität Dresden
Dresden, Germany
{dubslaff,baier,klueppelholz}@tcs.inf.tu-dresden.de

Abstract. Within *product lines*, collections of several related products are defined through their commonalities in terms of features rather than specifying them individually one-by-one. In this paper we present a compositional framework for modeling dynamic product lines by a state-based formalism with both probabilistic and nondeterministic behaviors. Rules for feature changes in products made during runtime are formalized by a coordination component imposing constraints on possible feature activations and deactivations. Our framework supports large-scaled product lines described through multi-features, i.e., where products may involve multiple instances of a feature.

To establish temporal properties for products in a product line, verification techniques have to face a combinatorial blow-up that arises when reasoning about several feature combinations. This blow-up can be avoided by family-based approaches exploiting common feature behaviors. We adapt such approaches to our framework, allowing for a quantitative analysis in terms of probabilistic model checking to reason, e.g., about energy and memory consumption, monetary costs, or the reliability of products. Our framework can also be used to compute strategies how to trigger feature changes for optimizing quantitative objectives using probabilistic model-checking techniques.

We present a natural and conceptually simple translation of product lines into the input language of the prominent probabilistic model checker Prism and show feasibility of this translation within a case study on an energy-aware server platform product line comprising thousands of products. To cope with the arising complexity, we follow the family-based analysis scheme and apply symbolic methods for a compact state-space representation.

[1] A preliminary version of this paper appeared at Modularity'14 [23].

* The authors are supported by the DFG through the collaborative research centre HAEC (SFB 912), the cluster of excellence cfAED, Deutsche Telekom Stiftung, the ESF young researcher groups IMData (100098198) and SREX (100111037), the Graduiertenkolleg QuantLA (1763), the DFG/NWO-project ROCKS, and the EU-FP-7 grant MEALS (295261).

1 Introduction

The concept of *product lines* is widely used in the development and marketing of modern hardware and software. In a product line, customers can purchase a base system extendible and customizable with functionalities, also called *features*. Following the definition for product lines in the software domain (also called *software product lines)*, a product line can also be understood as the collection of all features itself and rules how the features can be combined into products [14]. The rules for the composition of features are typically provided using *feature diagrams* [35,7], where the features and their hierarchical structure are given by a tree-like structure. For describing large product lines supporting several instances of features, feature diagrams with *multi-features* come into place, where cardinality ranges are annotated to features indicating how many of them can be instantiated towards a valid feature combination [18].

Feature combinations are often assumed to be static, i.e., some realizable feature combination is fixed when the product is purchased and is never changed afterwards. However, this does not faithfully reflect adaptations of modern products during their lifetime. For instance, when in-app purchases are placed or when a free trial version of a software product expires, features are activated or deactivated during runtime of the system. Similarly, components of a hardware system might be upgraded to more powerful or energy-efficient ones or are necessarily replaced due to a hardware failure. In all these situations, the products change but still belong to the same product line. Such product lines capable of modeling adaptations after deployment are called *dynamic product lines* [29], for which the design of specification formalisms is an active and emerging field in product line engineering [31,22,45,19].

Verification of Product Lines

To meet requirements in safety-critical parts of the features or to guarantee overall quality (in particular within features that are used in many or most of the products of the product line) verification is of utter interest. Verification is even more important for dynamic product lines, where side-effects arising from dynamic feature changes are difficult to predict in the development phase of a product. Model checking [11,5] is a fully automatic verification technique for establishing temporal properties of systems (e.g., safety or liveness properties). Indeed, model checking has been already successfully applied to integrate features in components and to detect feature interactions [43]. However, the typical task for reasoning about static product lines is to solve the so-called *featured model-checking problem*:

> Compute the set of all valid feature combinations C such that some given temporal requirement φ holds for the products corresponding to C.

This is in contrast to the classical model-checking problem that amounts to prove that φ holds for some fixed system, e.g., one specific product obtained from a feature combination. The naive approach for solving the featured model-checking problem is to verify the products in the product line one-by-one after their deployment. However, already within static product lines, this approach certainly suffers from an exponential blow-up in the number of different valid feature combinations. To tackle this potential combinatorial blow-up, family-based approaches are very successful, checking all products in a product line at once rather than one-by-one [53]. In [13,12], the concept of *featured transition systems* has been introduced to encode the operational behaviors of all products in a product line into a single model. The transitions in featured transition systems are annotated by feature combinations: a transition can only be fired if it is labeled by the feature combination corresponding to the product deployed. Symbolic techniques [39] describe states and transitions of an operational model as sets with common properties rather than listing them one-by-one. Such techniques can be used for solving the featured model-checking problem for product lines represented by featured transition systems efficiently for both linear-time [13] and branching-time properties [12]. An extension of featured transition systems that introduces guarded transitions for switches between valid feature combinations was presented by Cordy et al. [16] allowing for dynamic adaptions of feature combinations during the lifetime of a product. Besides purely functional temporal requirements, the quality of (software) products crucially depends on quantitative properties. Measurement-based approaches for reasoning about feature-oriented software have been studied intensively, see, e.g., [51,50,41]. In contrast, probabilistic model-checking techniques were studied only recently [28,52], relying on probabilistic operational models based on discrete-time Markov chains and probabilistic computation tree logic. For instance, Ghezzi and Sharifloo analyzed parametric sequence diagrams using the probabilistic model-checking tool PARAM [28].

A Compositional Framework for Feature-oriented Systems

In this paper, we present a *compositional framework* to model dynamic product lines which allows for the automated *quantitative system analysis* using probabilistic model checking [5]. Our approach allows for easily specifying large-scaled product lines with thousands of products described through feature diagrams with multi-features.
Markov chains, the purely probabilistic model used in most approaches of probabilistic product-line analysis [28,52], are less adequate for the compositional design with parallel components than operational models supporting both, nondeterministic and probabilistic choices (see, e.g., [48]). A *Markov decision process (MDP)* is such a formalism, extending labelled transition systems by internal probabilistic choices taken after resolving nondeterminism between actions of the system. Our framework for dynamic product lines presented in this paper relies on MDPs with annotated costs [44]. In particular, it consists of

(1) feature modules: MDP-like models for the feature-dependent operational behavior of the components and their interactions,

(2) a parallel operator: feature-aware composing feature modules to represent the parallel execution of independent actions by interleaving and supporting communication between the feature modules,
(3) a feature controller: an MDP-like model for the potential dynamic switches of feature combinations, and
(4) a join operator: yielding a standard MDP model of the complete dynamic product line represented by feature modules and a feature controller

A product line naturally induces a compositional structure over features, where a feature or a collection thereof corresponds to a component. In our framework, these components are called feature modules (1). Feature modules are composed using a parallel operator (2), which combines the operational behaviors of all features represented by the feature modules into another feature module. We only allow for composing compatible feature modules, i.e., feature modules which represent the operational behavior of different features. Thus, different implementations or versions of the same feature need either to be modeled as distinct features excluding each other or cannot be combined in our framework. Feature activation and deactivation at runtime is described through a feature controller (3), which is a state-based model controlling valid changes in the feature combinations. As within feature modules, choices between feature combinations can be probabilistic (e.g., on the basis of statistical information on feature combinations and their adaptations over time) or nondeterministic (e.g., if feature changes rely on internal choices of the controller or are triggered from outside by an unknown or unpredictable environment) and combinations thereof.

The semantics of a feature module under a given feature controller is defined as a parallel composition synchronizing over common feature annotations (4), providing an elegant formalization of the feature module's behavior within the dynamic product line represented by the feature controller. Note that our approach separates between computation and coordination [27,42,47], which allows for specifying features in the context of various different dynamic product lines. Feature-oriented extensions of programming languages and specialized composition operators such as *superimposition* are an orthogonal approach [36,2,1]. However, the effect of superimposition could also be encoded into our framework, e.g., using techniques proposed by Plath and Ryan [43].

Quantitative Analysis and Strategy Synthesis

Fortunately, the semantics of feature modules under feature controllers yields an MDP. Thus, our approach permits to apply standard but sophisticated probabilistic model-checking techniques to reason about quantitative properties. This is in contrast to existing (also nonprobabilistic) approaches, which require model-checking algorithms specialized for product lines. Within our approach, quantitative queries such as "minimize the energy consumption until reaching a target state" can be answered. Whereas for static product lines, one aims to solve the featured model-checking problem, we introduce the so-called *strategy synthesis problem* for dynamic product lines. This problem amounts to find an optimal strategy to resolve the nondeterminism between feature combination switches in the feature controller [23]. The strategy includes the initial step of the dynamic product line by selecting an initial feature combination, which suffices to solve the featured model-checking problem. Our approach thus additionally provides the possibility to reason over worst-case and best-case scenarios concerning feature changes during runtime.

Implementation and Case Study

Models of product lines have to face a combinatorial blow-up in the number of features. When modeling dynamic product lines, the number of possible feature changes during runtime yield an additional combinatorial blow-up. However, symbolic representations of models including all the behaviors in a product line can avoid these blow-ups [12]. In this paper, we extend our compositional framework for dynamic product lines [23] towards feature modules and controllers with variables, such that it nicely fits with guarded-command languages such as the input language of the symbolic probabilistic model checker Prism [34]. Prism uses multi-terminal binary decision diagrams for the symbolic encoding of the probabilistic model to obtain a compact representation. To demonstrate the usability of Prism within our framework, we carried out a case study based on a real-case server-platform scenario, where several variants of a server can be endowed with different kinds of network interface cards. This product line can be equipped with an energy-aware network driver, similarly as done for the eBond device [30]. Network cards with different performance characteristics are then bonded or switched off according to energy-saving algorithms which, e.g., take usage of the varying bandwidth requirements during day and night time. The arising energy-aware server system product line, which we call eServer, can be subject of several quantitative requirements, e.g., on the energy consumption of the products in the product line. We illustrate how Prism can be used to solve the strategy synthesis problem w.r.t. to such requirements for eServer and can provide strategies how to equip a server used in different environments. In particular, we show that symbolic methods applied to dynamic product lines such as eServer clearly outperform explicit ones.

Outline

The paper starts with a brief summary on the foundations of product lines, feature models, relevant principles of MDPs and their quantitative analysis. The compositional framework for specifying dynamic product lines by means of feature modules and feature controllers is presented in Section 3. Section 4 is devoted to the encoding of our framework into guarded-command languages, such as the input language of the probabilistic model-checking tool Prism. Our case study follows in Section 5, where we use Prism and the encoding of our framework to discuss the scalability of our approach towards product lines with thousands of products and the influence of symbolic representations. The paper ends with some concluding remarks in Section 6.

2 Preliminaries

Before we recall the standard concepts for product lines, probabilistic models and their quantitative analysis, we introduce notations for Boolean and linear expressions to provide intuitive symbolic representations for sets.

Boolean Expressions. The powerset of a set X is denoted by 2^X. For convenience, we sometimes use symbolic notations based on Boolean expressions for the elements of 2^X, i.e., the subsets of X. Let $\mathbb{B}(X)$ denote the set of all Boolean expressions ρ built over Boolean variables $x \in X$ as atoms and the usual connectives of propositional logic (negation $\neg$, conjunction $\wedge$, etc.). The satisfaction relation $\models \subseteq 2^X \times \mathbb{B}(X)$ is defined in the obvious way. For instance, if $X = \{x_1, x_2, x_3\}$ and $\rho = x_1 \wedge \neg x_2$, then $Y \models \rho$ iff $Y = \{x_1\}$ or $Y = \{x_1, x_3\}$. To specify binary relations on 2^X symbolically, we use Boolean expressions $\rho \in \mathbb{B}(X \cup X')$, where X' is the set consisting of pairwise distinct, fresh copies of the elements of X. Then, the relation $R_\rho \subseteq 2^X \times 2^X$ is given by:

$$(Y, Z) \in R_\rho \quad \text{iff} \quad Y \cup \{z' : z \in Z\} \models \rho$$

As an example, the Boolean expression $\rho = (x_1 \vee x_3') \wedge \neg x_2$ represents the relation R_ρ consisting of all pairs $(Y, Z) \in 2^X \times 2^X$, where (1) $x_1 \in Y$ or $x_3 \in Z$ and (2) $x_2 \notin Y$. For $Y \subseteq X$, we use $Y = Y'$ as a shortform notation for the Boolean expression $\bigwedge_{y \in Y} y \leftrightarrow y'$.

Linear Constraints. The symbolic notations for subsets of X using Boolean expressions can be extended towards sets of functions $f\colon X \to \mathbb{N}$, i.e., elements of $\mathbb{N}^X$ which we define through *linear constraints* γ of the form

$$a_1 x_1 \; + \; a_2 x_2 \; + \; \ldots \; + \; a_n x_n \quad \bowtie \quad \theta,$$

where $a_i \in \mathbb{Z}$, $x_i \in X$ for all $i = 1, 2, \ldots, n$, $\bowtie \in \{<, \leq, =, \geq, >\}$ and $\theta \in \mathbb{Z}$. A function $f \in \mathbb{N}^X$ fulfills such a linear constraint γ as above if $a_1 f(x_1) + a_2 f(x_2) + \ldots + a_n f(x_n) \bowtie \theta$. We then write $f \models \gamma$. The set of all linear constraints over X is denoted by $\mathbb{C}(X)$, while the set of Boolean expressions over linear constraints is $\mathbb{BC}(X) = \mathbb{B}(\mathbb{C}(X))$. With these ingredients, the definitions stated above for subsets of variables X take over, e.g., to the satisfaction relation $\models \subseteq \mathbb{N}^X \times \mathbb{BC}(X)$. Note that this is indeed an extension of Boolean expressions over X: for $f \in \mathbb{N}^X$, let $C_f \subseteq X$ denote the *support* of f, i.e., $C_f = \{x \in X : f(x) \geq 1\}$. Then for any Boolean expression $\rho \in \mathbb{B}(X)$, replacing all variables x by the linear constraint $(x \geq 1) \in \mathbb{C}(X)$ yields $\hat{\rho} \in \mathbb{BC}(X)$, where for all $f \in \mathbb{N}^X$

$$f \models \hat{\rho} \;\text{ iff }\; C_f \models \rho$$

Due to this, we also allow for mixed Boolean expressions in $\mathbb{B}(\mathbb{C}(X) \cup X)$, simply also denoted by $\mathbb{BC}(X)$. For instance, with $X = \{x_1, x_2\}$ the mixed expression

$$\rho \;\; = \;\; x_1 \wedge (2 \cdot x_1 + x_2 \leq 4)$$

defines exactly four functions $f \in \mathbb{N}^X$ where $f \models \rho$ is described through the pairs $(x_1, x_2) \in \{(1,0), (1,1), (1,2), (2,0)\}$. Similar as for Boolean expressions without linear constraints, a binary relation on $\mathbb{N}^X$ can be defined via an expression $\rho \in \mathbb{BC}(X \cup X')$, where X' is the set consisting of pairwise distinct, fresh copies of the elements of X. Then, the relation $R_\rho \subseteq \mathbb{N}^X \times \mathbb{N}^X$ is given by:

$$(f, g) \in R_\rho \quad \text{iff} \quad h \models \rho,$$

where $h \in \mathbb{N}^{X \cup X'}$ is defined by $h(x) = f(x)$ and $h(x') = g(x)$ for all $x \in X$. We also use $Y = Y'$ as a shortform notation of the expression $\bigwedge_{y \in Y}(y = y')$.

2.1 Feature Models

A *product line* is a collection of products, which have commonalities w.r.t. assets called *features* [14]. We discuss here a variant of product lines which allows for *multi-features*, i.e., a feature can appear in a product within multiple instances [18]. Let F denote the finite set of all such (multi-)features of a product line. A *feature combination* is a function f assigning to each feature $x \in F$ the cardinality $f(x)$. We say that $f \in \mathbb{N}^F$ is *valid* if there is a corresponding product in the product line consisting of exactly $f(x)$ instances of the features $x \in F$. A product line can hence be formalized in terms of a *feature signature* $(F, \mathcal{V})$, where $\mathcal{V} \subseteq \mathbb{N}^F$ is the set of valid feature combinations. A feature signature $(F, \mathcal{V})$ is *Boolean*, if $\mathcal{V} \subseteq \{0, 1\}^F$, i.e., there is at most one instance of each feature in a valid feature combination. *Feature diagrams* [35] provide a compact representation of feature signatures via a tree-like hierarchical structure (see, e.g., Figure 1). Nodes in feature diagrams correspond to features of F. The nodes are annotated with integer ranges that restrict the number of instances built for the given feature [18,17]. The integer ranges are of the form $[l..u]$, where $l, u \in \mathbb{N}$ with $l \leq u$ standing for the lower and upper cardinality bound on the feature instances. Usually, range annotations [1..1] are omitted in the feature diagrams, and instead of range annotations [0..1] (corresponding to optional features) a circle above the respective feature node is drawn. If the node for feature x' is a son of the node for feature x, then every instance of feature x' requires its corresponding instance of x. Several types of branchings from a node for feature x towards its sons $x'_1, \ldots, x'_n$ are possible. Standard branchings denote that all sons of x are instantiated according to their cardinality range (AND connective) and connected branchings indicate that exactly one son is required (XOR connective). Boolean expressions of linear constraints over F may be further annotated to describe, e.g., numerical dependencies between the number of instances of features. In this paper, we stick to this informal and rather intuitive description of multi-feature diagrams as it suffices for obtaining the feature signature and the hierarchical structure of features. We refer to [18,17] for a detailed discussion on the semantics of multi-feature diagrams.

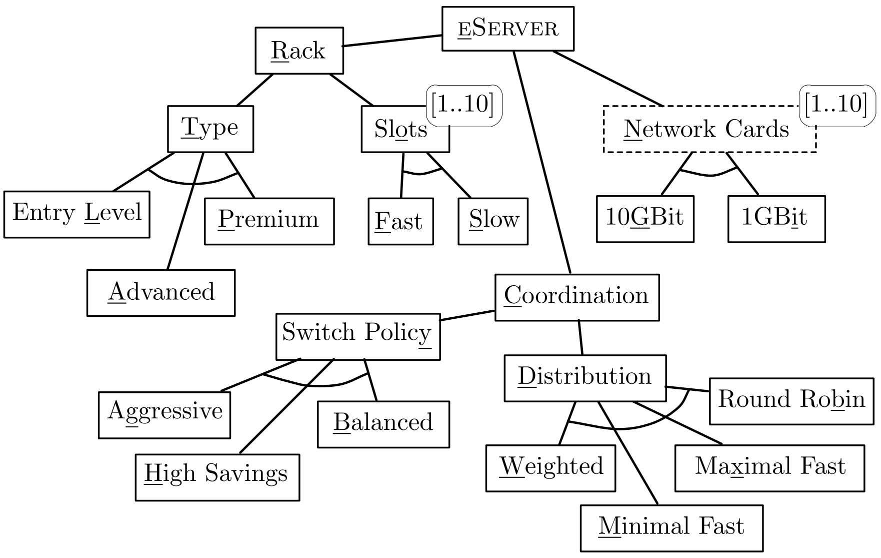
Dynamic Product Lines. Usually, product lines are static in the sense that a valid feature combination is fixed prior of launching the product. Product lines allowing for activation and deactivation of features during runtime of the system are called *dynamic product lines*. The common approach towards dynamic product lines is to indicate disjoint sets of *dynamic features* D and *environment features* E, which respectively include features that can be activated or deactivated at runtime either by the system itself (features of D) or by the environment (features of E). Intuitively, an activation and deactivation of an environment feature may impose (de-)activations of dynamic features [16]. In [22] dynamic product lines are formalized using a generalization of feature diagrams where dashed nodes represent elements of $D \cup E$. When not restricted by further annotated constraints, each instance of such a dynamic feature can be activated and deactivated at any time. In the approach by [19], the possible (de)-activations of each feature are defined explicitly by a switching relation over (Boolean) feature combinations. We choose a similar approach towards our compositional framework, which is also capable of supporting multi-features and explained in Section 3.

Further Extensions. Costs for feature activations in dynamic product lines have been considered in [54]. Besides assigning ranges to features describing their number of instances, ranges can also be annotated to branchings in the feature diagram, generalizing Boolean connectives for the branchings. In our formalization using linear constraints, such *group cardinalities* allow for a compact representation of lower and upper bounds on the number of instances in sons of the feature diagram [17].

Example 1. As the running example of this paper, let us consider an energy-aware server product line eServer, which is inspired by the server-rack product line of a famous computer vendor and incorporates an energy-aware driver for bonding network cards as presented in [30]. This product line can be represented by a feature diagram as shown in Figure 1.
Each node is identified with the underlined letter, i.e., the set of features is

$$F = \{\mathrm{e, R, T, L, A, P, o, F, S, C, y, g, H, B, D, W, M, x, b, N, G, i}\}.$$

The eServer product line consists of a server rack (R), which has at most ten slots (o) where up to ten network cards (N) can be plugged in. Each slot supports either a high-speed data transfer (F) or only a slow-speed transfer (S). Clearly, a fast 10 GBit network card (G) can only be used when plugged into a fast slot. Depending on the type of the rack, the number of slots and their kind is restricted according to the linear constraints over F provided at the bottom of the feature diagram. E.g., an advanced (A) eServer has at least 2 but at most 7 slots, where at most 2 of them are fast ones. Besides these hardware features, the eServer product line consists also of a software feature in terms of a driver coordinating the interplay of the heterogeneous network cards (C). The eServer incorporates eBond [30]: depending on the selected switch policy (y), network cards can be activated for serving more bandwidth or deactivated for saving energy. Furthermore, the method how the requested bandwidth is distributed along the active cards is coordinated by the feature D. For instance, the round robin feature (b) may stand for the standard uniform distribution of bandwidth, whereas the weighted feature (W) refers to distributing bandwidth such that every card has the same workload.



$L \Rightarrow (o \leq 2) \quad \wedge \quad A \Rightarrow (o \leq 7 \wedge o \geq 2 \wedge F \leq 2) \quad \wedge$
$P \Rightarrow (o \leq 10 \wedge o \geq 6 \wedge F \leq 8) \quad \wedge \quad (N \leq o) \wedge (G \leq F)$

Fig. 1. Feature diagram of the dynamic eServer product line

Note that the network card feature (N) is a dynamic multi-feature, i.e., its cardinality may vary from 1 to 10 during runtime of the eServer system. However, the annotated constraints still need to be fulfilled, e.g., $N \leq o$ for ensuring that at most as many cards as slots available are plugged.

Obviously, this dynamic product line is large-scaled, dominated by the possible combinations of the multi-features representing slots and network cards. Taking the linear constraints on the card combinations into account, the eServer product line amounts to 17,544 valid feature combinations.

2.2 Probabilistic Systems and Their Quantitative Analysis

The operational model used in this paper for modeling and analyzing the behavior of products in a dynamic product line is given in terms of *Markov decision processes (MDPs)* [44]. We deal here with MDPs where transitions are labeled with a cost value. MDPs with multiple cost functions of different types (e.g., for reasoning about energy and memory requirements and utility values) can be defined accordingly.

Distributions. Let S be a countable nonempty set. A *distribution over* S is a function $\sigma\colon S \to [0,1]$ with $\sum_{s \in S} \sigma(s) = 1$. The set $\{s \in S : \sigma(s) > 0\}$ is called the *support of* σ and is denoted by *supp*(σ). *Distr*(S) denotes the set of all distributions over S. Given $t \in S$, the *Dirac distribution $Dirac[t]$ of t over S* is defined by

$$Dirac[t](t) = 1 \quad \text{and} \quad Dirac[t](s) = 0 \text{ for all } s \in S \backslash \{t\}.$$

The *product* of two distributions $\sigma_1 \in Distr(S_1)$ and $\sigma_2 \in Distr(S_2)$ is defined as the distribution $\sigma_1 * \sigma_2 \in Distr(S_1 \times S_2)$, where $(\sigma_1 * \sigma_2)(s_1, s_2) = \sigma_1(s_1) \cdot \sigma_2(s_2)$ for all $s_1 \in S_1$ and $s_2 \in S_2$.

Markov Decision Processes. An MDP is a tuple

$$\mathcal{M} \;=\; (S, S^{init}, \mathsf{Moves}),$$

where S is a finite set of states, $S^{init} \subseteq S$ is the set of initial states and $\mathsf{Moves} \subseteq S \times \mathbb{N} \times Distr(S)$ specifies the possible moves of $\mathcal{M}$ and their costs. We require Moves to be finite and often write $s \xrightarrow{c} \sigma$ iff $(s, c, \sigma) \in \mathsf{Moves}$. Intuitively, the operational behavior of $\mathcal{M}$ is as follows. The computations of $\mathcal{M}$ start in some nondeterministically chosen initial state of S^{init}. If during $\mathcal{M}$'s computation the current state is s, one of the moves $s \xrightarrow{c} \sigma$ is selected nondeterministically first, before there is an internal probabilistic choice, selecting a successor state s' with probability $\sigma(s') > 0$. Value c specifies the cost for taking the move $s \xrightarrow{c} \sigma$.

Steps of $\mathcal{M}$, written in the form $s \overset{c}{\hookrightarrow}_p s'$, arise from moves $s \xrightarrow{c} \sigma$ resolving the probabilistic choice by plugging in some state s' with positive probability, i.e., $p = \sigma(s') > 0$. *Paths* in $\mathcal{M}$ are sequences of consecutive steps. In the following, we assume a finite path π having the form

$$\pi \;=\; s_0 \overset{c_1}{\hookrightarrow}_{p_1} s_1 \overset{c_2}{\hookrightarrow}_{p_2} s_2 \overset{c_3}{\hookrightarrow}_{p_3} \ldots \overset{c_n}{\hookrightarrow}_{p_n} s_n. \qquad (*)$$

We refer to the number n of steps as the length of π. If $0 \leq k \leq n$, we write $\pi \downarrow k$ for the prefix of π consisting of the first k steps (then, $\pi \downarrow k$ ends in state s_k). Given a finite path π, the probability $\Pr(\pi)$ is defined as the product of the probabilities in the steps of π and the accumulated costs $\mathrm{cost}(\pi)$ are defined as the sum of the costs of π's steps. Formally,

$$\Pr(\pi) = p_1 \cdot p_2 \cdot \ldots \cdot p_n \quad \text{and} \quad \mathrm{cost}(\pi) = c_1 + c_2 + \ldots + c_n.$$

State $s \in S$ is called *terminal* if there is no move $s \xrightarrow{c} \sigma$. A path is *maximal*, if it is either infinite or ends in a terminal state. The set of finite paths starting in some state of S^{init} is denoted by $FPaths$.

Schedulers and Probability Measure. Within MDPs, reasoning about probabilities requires the selection of an initial state and resolution of the nondeterministic choices between possible moves. The latter is formalized via *schedulers*, which take as input a finite path and decide which move to take next. For the purposes of this paper it suffices to consider deterministic, possibly history-dependent schedulers, i.e., partial functions

$$\mathfrak{S}\colon\ FPaths \to \mathbb{N} \times Distr(S),$$

where for all finite paths π as in $(*)$, $\mathfrak{S}(\pi)$ is undefined if π is maximal and otherwise $\mathfrak{S}(\pi) = (c, \sigma)$ for some $s_n \xrightarrow{c} \sigma$. An $\mathfrak{S}$-*path* is any path that arises when the nondeterministic choices in $\mathcal{M}$ are resolved by $\mathfrak{S}$. Thus, a finite path π is a $\mathfrak{S}$-path iff there are distributions $\sigma_1, \ldots, \sigma_n \in \mathit{Distr}(S)$ such that $\mathfrak{S}\big(\pi \downarrow k{-}1\big) = (c_k, \sigma_k)$ and $p_k = \sigma_k(s_k)$ for all $1 \leq k \leq n$. Infinite $\mathfrak{S}$-paths are defined accordingly.

Given a scheduler $\mathfrak{S}$ and some initial state $s \in S^{init}$, the behavior of $\mathcal{M}$ under $\mathfrak{S}$ and s is purely probabilistic and can be formalized by a tree-like infinite-state Markov chain $\mathcal{M}_s^{\mathfrak{S}}$ over the finite $\mathfrak{S}$-paths of $\mathcal{M}$ starting in s. Markov chains are MDPs that do not have any nondeterministic choices, i.e, where S^{init} is a singleton and $|\mathsf{Moves}(s)| \leq 1$ for all states $s \in S$. Using standard concepts, a probability measure $\mathbb{P}_s^{\mathfrak{S}}$ for measurable sets of maximal branches in the Markov chain $\mathcal{M}_s^{\mathfrak{S}}$ is defined and can be transferred to maximal $\mathfrak{S}$-paths in $\mathcal{M}$ starting in s. For further details we refer to standard text books such as [32,37,44].

Quantitative Properties. The concept of schedulers permits to talk about the probability of a measurable path property φ for paths starting in a fixed state s and respecting a given scheduler $\mathfrak{S}$. Typical examples for such a property φ are reachability conditions of the following type, where T and V are sets of states:

- reachability: $\varphi = \Diamond T$ denotes that eventually some state in T will be visited
- constrained reachability: $\varphi = V \,\mathcal{U}\, T$ imposes the same constraint as $\Diamond T$ with the side-condition that all states visited before reaching T belong to V

For a worst-case analysis of a system modeled by an MDP $\mathcal{M}$, one ranges over all initial states and all schedulers (i.e., all possible resolutions of the nondeterminism) and considers the maximal or minimal probabilities for φ. If φ represents a desired path property, then $\mathbb{P}_s^{\min}(\varphi) = \inf_{\mathfrak{S}} \mathbb{P}_s^{\mathfrak{S}}(\varphi)$ is the probability for $\mathcal{M}$ satisfying φ that can be guaranteed even for the worst-case scenarios. Similarly, $\mathbb{P}_s^{\max}(\varphi) = \sup_{\mathfrak{S}} \mathbb{P}_s^{\mathfrak{S}}(\varphi)$ is the least upper bound that can be guaranteed for the likelihood of $\mathcal{M}$ to satisfy φ (best-case scenario).

One can also reason about bounds for expected costs of paths in $\mathcal{M}$. We consider here accumulated costs to reach a set $T \subseteq S$ of target states from a state $s \in S$. Formally, if $\mathfrak{S}$ is a scheduler such that $\mathbb{P}_s^{\mathfrak{S}}(\Diamond T) = 1$, then the *expected accumulated costs* for reaching T from s under $\mathfrak{S}$ are defined by

$$\mathbb{E}_s^{\mathfrak{S}}(\Diamond T) \quad = \quad \textstyle\sum_{\pi} \mathrm{cost}(\pi) \cdot \Pr(\pi),$$

where π as in $(*)$ ranges over all finite $\mathfrak{S}$-paths with $s_n \in T$, $s_0 = s$ and $\{s_0, \ldots, s_{n-1}\} \cap T = \emptyset$. If $\mathbb{P}_s^{\mathfrak{S}}(\Diamond T) < 1$, i.e., with positive probability T will never be visited, then $\mathbb{E}_s^{\mathfrak{S}}(\Diamond T) = \infty$. Furthermore,

$$\mathbb{E}_s^{\min}(\Diamond T) = \inf_{\mathfrak{S}} \mathbb{E}_s^{\mathfrak{S}}(\Diamond T) \quad \text{and} \quad \mathbb{E}_s^{\max}(\Diamond T) = \sup_{\mathfrak{S}} \mathbb{E}_s^{\mathfrak{S}}(\Diamond T)$$

specify the greatest lower bound (least upper bound, respectively) for the expected accumulated costs reaching T from s in $\mathcal{M}$.

Quantitative Analysis. Several powerful probabilistic model-checking tools support the algorithmic quantitative analysis of MDPs against temporal specifications, such as the reachability properties stated above. But also for temporal properties such as formulas of linear temporal logic (LTL) or probabilistic computation-tree logic (PCTL) [8,6], there is a broad tool support. PCTL provides an elegant formalism to specify various temporal properties, reliability and resource conditions. In our case study, we will use the prominent probabilistic model checker Prism [34] that offers a symbolic MDP-engine for PCTL, dealing with a compact internal representation of the MDP using multi-terminal binary decision diagrams [26]. For the purpose of the paper, the precise syntax and semantics of PCTL over MDPs is not relevant. It suffices to know that in PCTL, the (constrained) reachability properties above can be described and encapsulated with a probability or expectation operator. Probabilistic model-checking algorithms for PCTL then allow for computing minimizing and maximizing schedulers for probabilities (e.g., $\mathbb{P}_s^{\max}(\varphi)$) and expectations (e.g., $\mathbb{E}_s^{\min}(\Diamond T)$) up to an arbitrary precision [8,6,20,25]. For the computation of the latter we assume that $\mathbb{P}_s^{\min}(\Diamond T) = 1$.

3 Compositional Framework

A product line naturally induces a compositional structure where features correspond to modules composed, e.g., along the hierarchy provided by feature diagrams. Thus, it is rather natural to choose a compositional approach towards a modeling framework for dynamic product lines. We formalize feature implementations by so-called *feature modules* that might interact with each other and can depend on the presence of other features and their current own configurations. Dependencies between feature modules are represented in form of guarded transitions in the feature modules, which may impose constraints on the current feature combination and perform synchronized actions. The interplay of the feature modules can also be described by a single feature module, which arises from the feature implementations via parallel composition and hence only depends on the dynamic feature changes. Unlike other models for dynamic product lines, there is no explicit representation of the dynamic feature combination changes inside the feature modules. Instead, we implement a clear separation between computation and coordination as it is central for exogenous coordination languages [27,42,47]. In our approach, the dynamic activation and deactivation of features is represented in a separate module, called *feature controller*. This separation yields some advantages: feature modules can be replaced and reused for many scenarios that vary in constraints for switching feature combinations and that might even rely on different feature signatures.

We model both, feature modules and feature controllers, as MDP-like automata models with annotations for (possibly feature-dependent) interactions between modules and the controller. To reason about resource constraints, cost functions are attached to the transitions of both, the feature modules and the feature controller. Through parallel composition, the operational behavior of the complete dynamic product line has a standard MDP semantics. We show also that our approach towards dynamic product lines is more expressive than existing approaches by providing embeddings into our framework. The compositional framework we present here aims also to provide a link between abstract models for feature implementations and the guarded command languages supported by state-of-the art model checkers. This approach is orthogonal to the compositional approaches for product lines that have been proposed in the literature, presenting an algebra for the nonprobabilistic feature-oriented composition of modules that covers subtle implementation details (see, e.g., [33,43,2,40]).

3.1 Feature Modules

To keep the mathematical model simple, we put the emphasis on the compositional treatment of features and therefore present first a data-abstract lightweight formalism for the feature modules. In this setting, feature modules can be seen as labeled transition systems, where the transitions have guards that formalize feature-dependent behaviors and are annotated with probabilities and costs to model stochastic phenomena and resource constraints.

We start with the definition of a feature interface that declares which features are "implemented" by the given feature module (called *own features*) and on which *external features* the behavior of the module depends. In the following, we assume a given feature signature $(F, \mathcal{V})$, e.g., provided by a feature diagram, where $\mathcal{V} \subseteq \mathbb{N}^F$ is finite.

Definition 1 (Feature interface). *A* feature interface F *is a pair* $\langle \mathsf{OwnF}, \mathsf{ExtF} \rangle$ *consisting of two subsets* OwnF *and* ExtF *of* F *such that* $\mathsf{OwnF} \cap \mathsf{ExtF} = \varnothing$.

With abuse of notations, we often write F to also denote the set $\mathsf{OwnF} \cup \mathsf{ExtF}$ of features affected by the feature interface F. We now define feature modules as an MDP-like formalism according to a feature interface, where moves may depend on features of the feature interface and the change of own features can be triggered, e.g., by the environment. Note that we assume a feature module to incorporate all behaviors of the instances of the own features, i.e., its behavior depends on the cardinality of the instances of own features and its types, but cannot depend on the implementation of single instances.

Definition 2 (Feature module). *A tuple* $\mathsf{Mod} = (\mathsf{Loc}, \mathsf{Loc}^{init}, \mathsf{F}, \mathsf{Act}, \mathsf{Trans})$ *is called* feature module *when*

- Loc *is a countable set of locations,*
- $\mathsf{Loc}^{init} \subseteq \mathsf{Loc}$ *is the set of initial locations,*
- $\mathsf{F} = \langle \mathsf{OwnF}, \mathsf{ExtF} \rangle$ *is a feature interface,*
- Act *is a finite set of actions, and*
- $\mathsf{Trans} = \mathsf{TrAct} \cup \mathsf{TrSw}$ *is a finite transition relation.*

The operational behavior of Mod *specified by* Trans *is given by feature-guarded transitions that are either labeled by an action* (TrAct) *or by a switch event describing own feature changes* (TrSw)*. Formally:*

$$\mathsf{TrAct} \subseteq \mathsf{Loc} \times \mathbb{BC}(\mathsf{F}) \times \mathsf{Act} \times \mathbb{N} \times \mathit{Distr}(\mathsf{Loc})$$

$$\mathsf{TrSw} \subseteq \mathsf{Loc} \times \mathbb{BC}(\mathsf{F}) \times \mathbb{BC}(\mathsf{OwnF} \cup \mathsf{OwnF}') \times \mathbb{N} \times \mathit{Distr}(\mathsf{Loc})$$

Recall that $\mathbb{BC}(\cdot)$ stands for the set of Boolean expressions over linear constraints on feature combinations.

Let us go more into detail concerning the operational behavior of feature modules. Both types of transitions in Mod, action-labeled transitions and switch transitions, have the form $\theta = (\ell, \phi, \_\,, c, \lambda)$, where

- ℓ is a location, called *source location* of θ,
- $c \in \mathbb{N}$ specifies the cost[1] caused by executing θ,
- $\phi \in \mathbb{BC}(\mathsf{F})$ is a Boolean expression of linear constraints on feature combinations, called *feature guard*, and
- λ is a distribution over Loc specifying an internal choice that determines the probabilities for the successor locations.

For action-labeled transitions, the third component $\_$ is an action $\alpha \in \mathsf{Act}$ representing some computation of Mod, which will be enabled if the current feature combination fulfills the feature guard ϕ and not avoided by the interaction with other feature modules. For switch transitions, $\_$ is a Boolean expression $\rho \in \mathbb{BC}(\mathsf{OwnF} \cup \mathsf{OwnF}')$, enabling Mod to react or impose constraints on dynamic changes of features owned by Mod.

Note that we defined feature modules in a generic way, such that feature modules need not to be aware of the feature signature and realizable feature switches. This makes them reusable for different dynamic product lines.

3.2 Parallel Composition

We formalize the interactions of feature modules by introducing a parallel operator on feature modules. Thus, starting with separate feature modules for all features $f \in F$ one might generate feature modules that "implement" several features, and eventually obtain a feature model that describes the behavior of all "controllable" features of the product line. Additionally, there might be some features in the set of features F provided by an unknown environment, where no feature modules are given.

The parallel operator for two composable feature modules follows the style of parallel composition of probabilistic automata [49,48] using *synchronization* over shared actions (handshaking) and interleaving for all other actions. Let

$$\begin{aligned}
\mathsf{Mod}_1 &= (\mathsf{Loc}_1, \mathsf{Loc}_1^{init}, \mathsf{F}_1, \mathsf{Act}_1, \mathsf{Trans}_1) \\
\mathsf{Mod}_2 &= (\mathsf{Loc}_2, \mathsf{Loc}_2^{init}, \mathsf{F}_2, \mathsf{Act}_2, \mathsf{Trans}_2),
\end{aligned}$$

where $\mathsf{F}_i = \langle \mathsf{OwnF}_i, \mathsf{ExtF}_i \rangle$ and $\mathsf{Trans}_i = \mathsf{TrAct}_i \cup \mathsf{TrSw}_i$ for $i = 1, 2$. Composability of Mod_1 and Mod_2 means that $\mathsf{OwnF}_1 \cap \mathsf{OwnF}_2 = \varnothing$. Own features of Mod_1 might be external for Mod_2 and vice versa, influencing each others behavior.

[1] For simplicity, we deal here with a single cost value for each guarded transition. Feature modules with multiple cost values will be considered in the case study of Section 5 and can be defined accordingly.

$$\frac{\alpha \in \mathsf{Act}_1 \setminus \mathsf{Act}_2, \quad (\ell_1, \phi, \alpha, c, \lambda_1) \in \mathsf{TrAct}_1}{(\langle \ell_1, \ell_2 \rangle, \phi, \alpha, c, \lambda_1 * \mathit{Dirac}[\ell_2]) \in \mathsf{TrAct}} \qquad \frac{\alpha \in \mathsf{Act}_2 \setminus \mathsf{Act}_1, \quad (\ell_2, \phi, \alpha, c, \lambda_2) \in \mathsf{TrAct}_2}{(\langle \ell_1, \ell_2 \rangle, \phi, \alpha, c, \mathit{Dirac}[\ell_1] * \lambda_2) \in \mathsf{TrAct}}$$

$$\frac{\alpha \in \mathsf{Act}_1 \cap \mathsf{Act}_2, \quad (\ell_1, \phi_1, \alpha, c_1, \lambda_1) \in \mathsf{TrAct}_1, \quad (\ell_2, \phi_2, \alpha, c_2, \lambda_2) \in \mathsf{TrAct}_2}{(\langle \ell_1, \ell_2 \rangle, \phi_1 \wedge \phi_2, \alpha, c_1 + c_2, \lambda_1 * \lambda_2) \in \mathsf{TrAct}}$$

$$\frac{(\ell_1, \phi, \rho, c, \lambda_1) \in \mathsf{TrSw}_1}{(\langle \ell_1, \ell_2 \rangle, \phi, \rho \wedge \mathsf{OwnF}_2 = \mathsf{OwnF}_2', c, \lambda_1 * \mathit{Dirac}[\ell_2]) \in \mathsf{TrSw}}$$

$$\frac{(\ell_2, \phi, \rho, c, \lambda_2) \in \mathsf{TrSw}_2}{(\langle \ell_1, \ell_2 \rangle, \phi, \rho \wedge \mathsf{OwnF}_1 = \mathsf{OwnF}_1', c, \mathit{Dirac}[\ell_1] * \lambda_2) \in \mathsf{TrSw}}$$

$$\frac{(\ell_1, \phi_1, \rho_1, c_1, \lambda_1) \in \mathsf{TrSw}_1, \quad (\ell_2, \phi_2, \rho_2, c_2, \lambda_2) \in \mathsf{TrSw}_2}{(\langle \ell_1, \ell_2 \rangle, \phi_1 \wedge \phi_2, \rho_1 \wedge \rho_2, c_1 + c_2, \lambda_1 * \lambda_2) \in \mathsf{TrSw}}$$

Fig. 2. Rules for the parallel composition of feature modules

Definition 3 (Parallel composition). *The* parallel composition *of two composable feature modules* Mod_1 *and* Mod_2 *is defined as the feature module*

$$\mathsf{Mod}_1 \parallel \mathsf{Mod}_2 \ = \ (\mathsf{Loc}, \mathsf{Loc}^{init}, \mathsf{F}, \mathsf{Act}, \mathsf{Trans}),$$

where the feature interface $\mathsf{F} = \langle \mathsf{OwnF}, \mathsf{ExtF} \rangle$,

$$\begin{array}{rclrcl} \mathsf{Loc} & = & \mathsf{Loc}_1 \times \mathsf{Loc}_2 & \mathsf{Loc}^{init} & = & \mathsf{Loc}_1^{init} \times \mathsf{Loc}_2^{init} \\ \mathsf{OwnF} & = & \mathsf{OwnF}_1 \cup \mathsf{OwnF}_2 & \mathsf{ExtF} & = & (\mathsf{ExtF}_1 \cup \mathsf{ExtF}_2) \setminus \mathsf{OwnF} \\ \mathsf{Act} & = & \mathsf{Act}_1 \cup \mathsf{Act}_2 & \mathsf{Trans} & = & \mathsf{TrAct} \cup \mathsf{TrSw} \end{array}$$

and TrAct *and* TrSw *are defined by the rules shown in Figure 2.*

Obviously, $\mathsf{Mod}_1 \parallel \mathsf{Mod}_2$ is again a feature module. In contrast to the (non-probabilistic) superimposition approach for composing modules [36,43], the parallel operator $\parallel$ is commutative and associative. More precisely, if Mod_i for $i \in \{1, 2, 3\}$ are pairwise composable feature modules, then:

$$\begin{array}{rcl} \mathsf{Mod}_1 \parallel \mathsf{Mod}_2 & = & \mathsf{Mod}_2 \parallel \mathsf{Mod}_1 \\ (\mathsf{Mod}_1 \parallel \mathsf{Mod}_2) \parallel \mathsf{Mod}_3 & = & \mathsf{Mod}_1 \parallel (\mathsf{Mod}_2 \parallel \mathsf{Mod}_3) \end{array}$$

For the parallel composition of feature modules with multiple cost functions, one has to declare which cost functions are combined. This can be achieved by dealing with types of cost functions (e.g., energy, money, memory requirements) and accumulating costs of the same type.

3.3 Feature Controller

We now turn to feature controllers, which specify the rules for the possible changes of feature combinations during runtime of the system. We start with purely nondeterministic controllers (Definition 4) switching feature combinations similar to [19]. Then, we extend the purely nondeterministic controllers by assigning probabilities to the feature switch events (Definition 5).

Definition 4. *A* simple feature controller *is a tuple*

$$\mathsf{Con} \;=\; (\mathcal{V}, \mathcal{V}^{init}, \mathsf{SwRel}),$$

where $\mathcal{V}^{init} \subseteq \mathcal{V}$ *is the set of* initial feature combinations *and* $\mathsf{SwRel} \subseteq \mathcal{V}\times\mathbb{N}\times\mathcal{V}$ *is a relation, called* (feature) switch relation*, that formalizes the possible dynamic changes of the feature combinations and their cost. We refer to elements in* SwRel *as* switch events *and require that* $(f, d_1, f'), (f, d_2, f') \in \mathsf{SwRel}$ *implies* $d_1 = d_2$*.*

If there are several switch events $(f, d_1, f_1), (f, d_2, f_2), \ldots$ that are enabled for the feature combination f, then the choice which switch event fires is made nondeterministically. This is adequate, e.g., to represent user activities such as upgrades or downgrades of a software product or to express environmental influences.

Although our focus is on dynamic product lines, static product lines can easily be modeled using the simple feature controller $\mathsf{Con}_{\mathsf{static}} = (\mathcal{V}, \mathcal{V}, \varnothing)$. The concept of simple feature controllers also covers the approach of [16,22], where dynamic product lines are represented by Boolean feature signatures $(F, \mathcal{V})$ extended with disjoint sets of dynamic features $D \subseteq F$ and environment features $E \subseteq F$. The features in $D \cup E$ can be activated or deactivated at any time, while the modes of all other features remain unchanged. This dynamic behavior of the feature combinations is formalized using the controller

$$\mathsf{Con}_{D,E} \;=\; (\mathcal{V}, \mathcal{V}, \mathsf{SwRel}_{D,E}),$$

where $\mathsf{SwRel}_{D,E}$ is defined for all $f, g \in \mathcal{V}$, omitting cost values of switch events for better readability:

$$(f, g) \in \mathsf{SwRel}_{D,E} \quad \text{iff} \quad \varnothing \neq \{x \in F : f(x) + g(x) = 1\} \subseteq D \cup E.$$

There might also be switch events where statistical data on the frequency of uncontrollable feature switch events is at hand. For instance, the deactivation of features that are damaged due to rare environmental events (electrical power outage, extreme hotness, etc.) might be better modeled probabilistically instead of nondeterministically. This leads to the more general concept of *probabilistic feature controllers*, where switch events are pairs (f, d, γ) consisting of a feature combination f, a cost value $d \in \mathbb{N}$ and a distribution γ over $\mathcal{V}$. Thus, probabilistic feature controllers can be seen as MDPs with switch events as moves.

Definition 5 (Controller). *A* probabilistic feature controller*, briefly called* controller*, is a tuple* $\mathsf{Con} = (\mathcal{V}, \mathcal{V}^{init}, \mathsf{SwRel})$ *as in Definition 4, but where*

$$\mathsf{SwRel} \ \subseteq \ \mathcal{V} \times \mathbb{N} \times \mathit{Distr}(\mathcal{V})$$

is finite and (f, d_1, γ), $(f, d_2, \gamma) \in \mathsf{SwRel}$ *implies* $d_1 = d_2$.

Clearly, each simple feature controller Con can be seen as a (probabilistic feature) controller. For this, we just have to identify each switch event (f, d, g) with $(f, d, \mathit{Dirac}[g])$.

3.4 MDP-semantics

The semantics of a feature module Mod under some controller Con is given in terms of an MDP. If Mod stands for the parallel composition of all modules that implement the features of a given product line and the controller Con specifies the dynamic adaptions of the feature combinations, then the arising MDP formalizes the operational behaviors of the product line where the feature switches are resolved according to the rules specified by the controller. In what follows, we fix a feature module and a controller

$$\begin{aligned} \mathsf{Mod} &= (\mathsf{Loc}, \mathsf{Loc}^{init}, \mathsf{F}, \mathsf{Act}, \mathsf{Trans}) \\ \mathsf{Con} &= (\mathcal{V}, \mathcal{V}^{init}, \mathsf{SwRel}) \end{aligned}$$

as in Definition 2 and Definition 5, where $\mathsf{F} \subseteq F$. Intuitively, an action-labeled transition $(\ell, \phi, \alpha, c, \lambda)$ of Mod is a possible behavior of Mod in location ℓ, provided that the current state f of the controller Con (which is simply the current feature combination) meets the guard ϕ. Switch events of the controller can be performed independently from Mod if they do not affect the own features of Mod, whereas if they affect at least one feature in OwnF, the changes of the mode have to be executed synchronously. Thus, feature modules can trigger or prevent switch events by offering or refusing the required interactions with the feature controller. This allows, e.g., to model that system upgrades may be only permitted when all internal actions of the feature modules are completed.

Definition 6 (Semantics of feature modules). *Let* Mod *and* Con *be as before. The behavior of* Mod *under the controller* Con *is formalized by the MDP*

$$\mathsf{Mod} \bowtie \mathsf{Con} \ = \ (S, S^{init}, \mathsf{Moves}),$$

where $S = \mathsf{Loc} \times \mathcal{V}$, $S^{init} = \mathsf{Loc}^{init} \times \mathcal{V}^{init}$ *and where* Moves *is defined by the rules in Figure 3. Recall that* $\rho \in \mathbb{BC}(\mathsf{OwnF} \cup \mathsf{OwnF}')$ *is regarded as a Boolean expression on linear constraints over* $F \cup F'$ *specifying a binary relation* $R_\rho \subseteq \mathbb{N}^F \times \mathbb{N}^F$.

Due to the MDP semantics of feature modules under a controller, standard probabilistic model-checking techniques for the quantitative analysis can be directly applied. This includes properties involving feature combinations, since these are encoded into the states of the arising MDP.

$$\frac{(\ell,\phi,\alpha,c,\lambda)\in\mathsf{TrAct},\quad f\models\phi}{(\langle\ell,f\rangle,c,\lambda * \mathit{Dirac}[f])\in\mathsf{Moves}}
\qquad
\frac{\begin{array}{c}(\ell,\phi,\rho,c,\lambda)\in\mathsf{TrSw},\quad f\models\phi,\quad f\xrightarrow{d}\gamma,\\ \exists g\in\mathit{supp}(\gamma),x\in\mathsf{OwnF}.f(x)\neq g(x),\\ \forall g\in\mathit{supp}(\gamma).(f,g)\in R_\rho\end{array}}{(\langle\ell,f\rangle,c+d,\lambda*\gamma)\in\mathsf{Moves}}$$

$$\frac{f\xrightarrow{d}\gamma,\quad\forall g\in\mathit{supp}(\gamma),x\in\mathsf{OwnF}.f(x)=g(x)}{(\langle\ell,f\rangle,d,\mathit{Dirac}[\ell]*\gamma)\in\mathsf{Moves}}$$

Fig. 3. Rules for the moves in the MDP $\mathsf{Mod}\bowtie\mathsf{Con}$

3.5 Remarks on our Framework

In this section, we briefly discuss how the basic formalisms of our framework can be refined for more specific applications.

Handling Switch Events. Within the presented formalism the switch events appear as nondeterministic choices and require interactions between the controller and all modules that provide implementations for the affected features. Employing the standard semantics of MDPs, where one of the enabled moves is selected nondeterministically, this rules out the possibility to express that certain switch events might be unpreventable. However, such unpreventable switch events can be included into our framework, refining the concept of feature controllers by explicitly specifying which switch events must be taken whenever they are enabled in the controller. This could modeled by adding an extra transition relation for *urgent* switch events or prioritizing switches.

Instead of urgency or priorities, one might also keep the presented syntax of feature modules and controllers, but refine the MDP-semantics by adding *fairness conditions* that rule out computations where enabled switch events are postponed ad infinitum. Also here, we can benefit from standard techniques to treat fairness assumptions within PCTL properties developed for MDPs [6].

Another option for refining the nondeterministic choices in the controller is the distinction between switch events that are indeed controllable by the controller and those that are triggered by the environment. This naturally leads to a game-based view of the MDP for the composite system.

Feature Controller as Feature Module. To emphasize the feature-oriented aspects of our framework, we used a different syntax for controllers and feature modules. Nevertheless, controllers can be viewed as special feature modules when we discard the concept of switch events and switch transitions and rephrase them as action-labeled transitions. To transform controllers syntactically to feature modules, we have to add the trivial guard "true" and introduce names for all switch events. When turning the switch transitions of the feature modules into action-labeled transitions, matching names must be introduced to align the parallel operators $\|$ and $\bowtie$. Note that in the constructed feature modules, all features are external and the controller locations coincide with feature combinations. However, the framework can easily be extended supporting also own operational behavior of the controllers by adding locations to the feature combinations. Furthermore, since controllers are then a special kind of feature modules, different feature controllers may be composed, enabling to specify the rules for switching features provided by different stakeholder perspectives, e.g., restrictions on feature combination switches imposed by the vendor of the product line, the operator of the system, or the user.

Multi-features as Multiple Features. We assumed that a multi-feature includes all the behaviors of the instances of the feature, i.e., the instances do not have a distinguishable characteristics. However, annotating each feature and its actions with the number of its instantiation makes multi-features explicit, breaking the symmetry between the multi-features. One consequence for feature models is that multi-feature diagrams then have the same expressiveness as simple feature diagrams. Concerning our framework, every instance of each multi-feature then requires its own implementation in terms of a feature module.

Superimposition. Feature modules and feature controllers might serve as a starting point for a low-level implementation of features in a top-down design process. Vice versa, feature modules may also be extracted from "real" implementations using appropriate abstraction techniques. Prominent composition operators for feature-oriented software such as superimposition [36,43,2] are only supported implicitly in our framework by representing the effect of superimposition by means of feature guards and synchronization actions.

4 Variables and Guarded-command Languages

So far, we presented a lightweight data-abstract formalism for feature modules with abstract action and location names. This simplified the presentation of the mathematical model. From the theoretical point of view, feature modules in the sense of Definition 2 are powerful enough to encode systems where the modules operate on variables with finite domains. Even communication over shared variables can be mimicked by dealing with handshaking and local copies of shared variables. In practice, however, the explicit use of assignments for variables and guards for the transitions that impose constraints for local and shared variables is desirable; not only to avoid unreadable encodings, but also for performance reasons of the algorithmic analysis. The concept of variables can also help to generate more compact representations of the MDP-semantics for product lines according to our compositional framework, using, e.g., symbolic representations with linear constraints over variables. Furthermore, feature modules with variables could also provide operators that mimic the concept of superimposition [43]. The formal definition of an extension of *feature modules by variables* is rather technical, but fairly standard. However, such extended feature modules directly yield a translation into guarded-command languages, which makes our framework useful for the application of model-checking tools, such as PRISM [34].

4.1 Feature Modules with Variables

Let use suppose that Var is a finite set of typed variables, where the types are assumed to be finite as well (e.g., Boolean variables or integers with some fixed number of digits). We denote furthermore by $\mathcal{VAL}$ the set of valuation functions for the variables, i.e., type-consistent mappings that assign to each variable $x \in \mathit{Var}$ a value. In analogy to the symbolic representation of sets of integer-valued functions by Boolean expressions over linear constraints we introduced in the preliminaries, we can represent subsets of $\mathcal{VAL}$ by Boolean expressions, where the atoms are assertions on the values of the variables. Let $\mathbb{BC}(\mathit{Var})$ denote the set of these Boolean expressions. Then, e.g., if x and y are variables with domain $\{0,1,2,3\}$ and z a variable with domain $\{\text{red}, \text{green}, \text{blue}\}$, the Boolean expression $\phi = (x < y) \wedge (y > 2) \wedge (z \neq \text{green})$ represents all valuations $v \in \mathcal{VAL}$ with $v(x) < v(y) = 3$ and $v(z) \in \{\text{red}, \text{blue}\}$.

Interface. The interface of a feature module Mod now consists of a feature interface $\mathsf{F} = \langle \mathsf{OwnF}, \mathsf{ExtF} \rangle$ as in Definition 1 and a declaration which variables from Var are local and which ones are external. The local variables can appear in guards and can be modified by Mod, while the external variables can only appear in guards, but cannot be written by Mod. Instead, the external variables of Mod are supposed to be local for some other module. We denote these sets by OwnV and ExtV, write V for $\mathsf{OwnV} \cup \mathsf{ExtV}$ and extend the notion of composability of two feature modules by the natural requirement that there are no shared local variables.

Locations and Initial Condition. One can think of the variable valuations for the local variables to serve as locations in the module Mod. However, there is no need for an explicit reference to locations since all transitions will be described symbolically (see below). Instead of initial locations, we deal with an initial condition for the local variables.

Updates and Symbolic Transitions. Transitions in Mod might update the values of the local variables. The updates are given by sequences of assignments $x_1 := \mathit{expr}_1; \ldots; x_n := \mathit{expr}_n$, where $x_1, \ldots, x_n$ are pairwise distinct variables in OwnV and expr_i are type-consistent expressions that refer to variables in V. We formalize the effect of the updates that might appear in Mod by functions $\mathsf{upd} : \mathcal{VAL} \to \mathcal{VAL}$ with $\mathsf{upd}(v)(y) = v(y)$ for all non-local variables $y \in \mathit{Var} \setminus \mathsf{OwnV}$.

Instead of explicit references to the variable valuations in the transitions, we use a symbolic approach based on symbolic transitions. Symbolic transitions represent sets of guarded transitions, possibly originating from multiple locations, and are of the following form

$$\theta \quad = \quad (\mathit{guard}, \phi, \_, c, \mathit{prob\_upd}),$$

where $\mathit{guard} \in \mathbb{BC}(\mathsf{V})$ is a variable guard imposing conditions on the local and external variables, and $\phi \in \mathbb{BC}(\mathsf{F})$ is a feature guard as before. The third and fourth component $\_$ and c are as in the data-abstract setting. That is, $\_$ stands for an action label $\alpha \in \mathsf{Act}$ or a Boolean expression $\rho \in \mathbb{BC}(\mathsf{OwnF} \cup \mathsf{OwnF}')$ for the switch events, while $c \in \mathbb{N}$ stands for the cost caused by taking transition θ.

The last component *prob_upd* is a *probabilistic update*, i.e., a distribution over finitely many updates for variables in OwnV. These are written in the form

$$p_1 : \mathsf{upd}_1 + p_2 : \mathsf{upd}_2 + \ldots + p_k : \mathsf{upd}_k,$$

where p_i are positive rational numbers with $p_1 + \ldots + p_k = 1$ and the upd_i's are updates for the local variables. That is, p_i is the probability for update upd_i.

4.2 Data-aware Parallel Composition

The extension of the parallel operator $\|$ for composable feature modules with variables is rather tedious, but straightforward. As stated above, composability requires that there are no common own features and no common local variables. The local variables of the composite module $\mathsf{Mod}_1 \| \mathsf{Mod}_2$ are the variables that are local for one module Mod_i, i.e., $\mathsf{OwnV} = \mathsf{OwnV}_1 \cup \mathsf{OwnV}_2$ and $\mathsf{ExtV} = (\mathsf{ExtV}_1 \cup \mathsf{ExtV}_2) \setminus \mathsf{OwnV}$. The feature interface of $\mathsf{Mod}_1 \| \mathsf{Mod}_2$ is defined as in the data-abstract setting. The initial variable condition of $\mathsf{Mod}_1 \| \mathsf{Mod}_2$ arises by the conjunction of the initial conditions for Mod_1 and Mod_2. Let us now turn to the transitions in $\mathsf{Mod}_1 \| \mathsf{Mod}_2$.

- All action-labeled symbolic transitions in Mod_1 or Mod_2 with some non-shared action α are also transitions in $\mathsf{Mod}_1 \| \mathsf{Mod}_2$.
- Action-labeled symbolic transitions

  $$\begin{aligned} \theta_1 &= (guard_1, \phi_1, \alpha, c_1, prob\_upd_1) \in \mathsf{TrAct}_1 \\ \theta_2 &= (guard_2, \phi_2, \alpha, c_2, prob\_upd_2) \in \mathsf{TrAct}_2 \end{aligned}$$

  with a shared action $\alpha \in \mathsf{Act}_1 \cap \mathsf{Act}_2$ are combined into a symbolic transition of $\mathsf{Mod}_1 \| \mathsf{Mod}_2$:

  $$\theta_1 \| \theta_2 = (guard, \phi, \alpha, c_1 + c_2, prob\_upd),$$

  where $guard = guard_1 \wedge guard_2$, $\phi = \phi_1 \wedge \phi_2$ and *prob_upd* combines the probabilistic update functions $prob\_upd_1$ and $prob\_upd_2$. That is, if upd_i has probability p_i under distribution $prob\_upd_i$ for $i = 1, 2$, then the combined update that performs the assignments in upd_1 and upd_2 simultaneously has probability $p_1 \cdot p_2$ under *prob_upd*.
- The adaption of the rules for switch transitions in $\mathsf{Mod}_1 \| \mathsf{Mod}_2$ can be obtained analogously as for action transitions and is omitted here.

4.3 Data-aware MDP-semantics

In the data-abstract setting, a reasonable MDP-semantics of a feature module Mod under controller $\mathsf{Con} = (\mathcal{V}, \mathcal{V}^{init}, \mathsf{SwRel})$ has been defined, no matter whether Mod is just a fragment of the product line and may interact with other modules or not. An analogous definition for the data-aware setting can be provided either for modules without external variables or by modelling the changes of the values of the external variables by nondeterministic choices.

Let us here consider the first case where we are given a module $\mathsf{Mod} = \mathsf{Mod}_1 \,\|\, \ldots \,\|\, \mathsf{Mod}_n$ that arises through the parallel composition of several modules such that all variables $x \in \mathit{Var}$ are local for some module Mod_i. Then, Mod has no external variables and $\mathit{Var} = \mathsf{OwnV} = \mathsf{V}$. Furthermore, OwnF is the set of all features of the given product line for which implementations are given, while ExtF stands for the set of features controlled by the environment. The MDP $\mathsf{Mod} \bowtie \mathsf{Con}$ has the state space $S = \mathcal{VAL} \times \mathcal{V}$. The initial states are the pairs $\langle v, f \rangle$ where v satisfies the initial variable condition of Mod and $f \in \mathcal{V}^{init}$. The moves in $\mathsf{Mod} \bowtie \mathsf{Con}$ arise through rules that are analogous to the rules shown in Figure 3 on page 18. More precisely, Moves is the smallest set of moves according to the following three cases, where $\langle v, f \rangle$ is an arbitrary state in $\mathsf{Mod} \bowtie \mathsf{Con}$:

- An action-labeled transition $(\mathit{guard}, \phi, \alpha, c, \mathit{prob\_upd})$ in Mod is enabled in state $\langle v, f \rangle$ if $f \models \phi$ and $v \models \mathit{guard}$. If $\mathsf{upd}_i(v) \neq \mathsf{upd}_j(v)$ for $i \neq j$, then:

$$(\langle v, f \rangle, c, \lambda * \mathit{Dirac}[f]) \in \mathsf{Moves},$$

  where $\lambda(\mathsf{upd}_i(v)) = p_i$ for $i = 1, \ldots, k$ and $\lambda(\hat{v}) = 0$ for all other valuation functions $\hat{v}$.
- If $f \xrightarrow{d} \gamma$ is a switch transition in Con that does affect at most the features of the environment, i.e., $f(x) = g(x)$ for all $g \in \mathit{supp}(\gamma), x \in \mathsf{OwnF}$, then:

$$(\langle v, f \rangle, d, \mathit{Dirac}[\ell] * \gamma) \in \mathsf{Moves}$$

- Suppose that $(\mathit{guard}, \phi, \rho, c, \mathit{prob\_upd})$ is a switch transition in Mod enabled in $\langle v, f \rangle$ and affecting own features, i.e., $f \models \phi$ and $v \models \mathit{guard}$ and there are $g \in \mathit{supp}(\gamma)$, $x \in \mathsf{OwnF}$ with $f(x) \neq g(x)$. Again, $\rho \in \mathbb{BC}(\mathsf{OwnF} \cup \mathsf{OwnF}')$ specifies a binary relation $R_\rho \subseteq \mathbb{N}^F \times \mathbb{N}^F$. If $(f, g) \in R_\rho$ for all $g \in \mathit{supp}(\gamma)$ then:

$$(\langle v, f \rangle, c + d, \lambda * \gamma) \in \mathsf{Moves},$$

  where λ is defined as in the first (action-labeled) case.

5 Quantitative Feature Analysis

Within the compositional framework presented in the previous sections, let us assume that we are given feature modules $\mathsf{Mod}_1, \ldots, \mathsf{Mod}_n$ which stand for abstract models of certain features in F and a feature controller Con specifying the rules for feature combination changes. The feature set F might still contain other features where no implementations are given, which are external features controlled by the environment. Alternatively, one of the feature modules can formalize the interference of the feature implementations with a partially known environment, e.g., in form of stochastic assumptions on the workload, the frequency of user interactions, or reliability of components. Applying the compositional construction by putting feature modules in parallel and joining them with the feature controller, we obtain an MDP of the form

$$\mathcal{M} \;=\; (\mathsf{Mod}_1 \,\|\, \ldots \,\|\, \mathsf{Mod}_n) \bowtie \mathsf{Con}.$$

This MDP $\mathcal{M}$ formalizes the operational behavior of a dynamic product line and can now be used for a quantitative analysis. Whereas other family-based model-checking approaches for product lines require feature-adapted algorithms [13,12], the task of a quantitative analysis of dynamic product lines is thus reduced to standard algorithmic problems for MDPs and permits the use of generic probabilistic model-checking techniques.

5.1 The Strategy Synthesis Problem

A *quantitative worst-case analysis* in the MDP $\mathcal{M}$ that establishes least upper or greatest lower bounds for the probabilities of certain properties or for the expected accumulated costs as introduced in Section 2.2 can be carried out with standard probabilistic model-checking tools. These values provide guarantees on the probabilities under all potential resolutions of the nondeterministic choices in $\mathcal{M}$, possibly imposing some fairness constraints to ensure that continuously enabled dynamic adaptions of the feature combinations (switch events) cannot be superseded forever by action-labeled transitions of the feature modules.

In our framework, we separated the specifications of the potential dynamic adaptions of feature combinations (the controller) and the implementations of the features (the feature modules). Hence, although a worst-case analysis can give important insights in the correctness and quality of a product line, it appears natural to go one step further by asking for *optimal strategies* triggering switch events. Optimality can be understood with respect to queries like maximizing the probability for desired behaviors or minimizing the expected energy consumption while meeting given deadlines.

Several variants of this problem can be considered. The basic and most natural variant that we address here relies on the assumption that the nondeterminism in the MDP $\mathcal{M}$ for the composite system stands for decisions to be made by the controller, i.e., only the switch events appear nondeterministically, whereas the feature modules behave purely probabilistically (or deterministically) after joining them with the controller. More formally, we suppose that in each state s of $\mathcal{M}$, either there is a single enabled move representing some action-labeled transition of one or more feature modules or all enabled moves stand for switch events. Furthermore, we assume that features which are implemented as software or hardware components (usually the features not modeling the environment) act deterministically. In this case, an optimal strategy for the controller is just a scheduler for $\mathcal{M}$ that optimizes the quantitative measure of interest. The task that we address is the *strategy synthesis problem*, i.e., given $\mathcal{M}$ and some PCTL-query Φ as in Section 2.2, construct a scheduler $\mathfrak{S}$ for $\mathcal{M}$ that optimizes the solution of the query Φ. Indeed, the standard probabilistic model-checking algorithms for PCTL are applicable to solve the strategy synthesis problem. Note that if the feature controller represents a static behavior (see $\mathsf{Con}_{\mathsf{static}}$ in Section 3.3), the strategy synthesis problem coincides with the probabilistic version of the featured model-checking problem mentioned in the introduction, where the task amounts of computing all initial feature combinations such that the corresponding product satisfies Φ.

5.2 The eServer Product Line

In this section, we describe the eServer product line for which the feature model has been already introduced in the preliminaries (see Example 1). We modeled this dynamic product line following our framework, i.e., implementing feature modules and a feature controller.

Feature Modules. The feature diagram shown in Figure 1 depicts the features of the eServer product line, including their hierarchical dependencies and cardinalities which restrict the valid feature combinations. We implemented each feature in a single feature module, where three basic feature modules arise through parallel composition: the rack (R), network cards (N), and coordination feature (C). The rack feature is the basic server hardware, where depending on its type multiple slots (o) for network cards can be chosen. Slots are either supporting a high or low bandwidth. The initial hardware configuration cannot be changed after deployment, except for the network cards feature, where during runtime the quantities of cards may increase (until the number of slots in the basic system is reached) or the type of the card can be changed upgrading from a slow 1 GBit to a fast 10 GBit network card. Clearly, a fast network card can only be used as such in a slot supporting high bandwidths. The rules for network card switches are formalized by the feature controller and will be described in the next section.

Besides these hardware features of the product line, the coordination feature stands for the software features. More precisely, it stands for the drivers which control the interplay between the hardware and the higher-level software layers. The distribution feature (D) manages how the requested bandwidth is distributed amongst the network cards in the system:

Round Robin stands for the standard distribution scheme, where a data package is served by the next network card having free capacities
Weighted is like the round robin scheme, but weighs the fast cards according to their maximal bandwidth with a factor of 10 compared to the lower ones
Maximal Fast first lets all fast network cards serve packages before a round robin distribution over all slow cards is performed
Minimal Fast is as Maximal Fast with switched roles for fast and slow cards

The switch policy feature (y) implements an energy-aware bonding of (heterogenous) network cards according to the eBond principle [30] and exploits the different energy characteristics of the network cards to save energy. Individual network cards can be switched on at any time whenever more bandwidth is required and switched off otherwise. In [30], simulation-based techniques were used to show that within eBond, energy savings up to 75% can be achieved when the demand for bandwidth varies over time, e.g., between day and night time. In the eServer product line, we follow the energy-savings algorithms presented for eBond, providing a switch policy how to activate and deactivate network cards during runtime:[2]

[2] Activation and deactivation of network cards should not be confused with changing the network cards feature by plugging or unplugging cards.

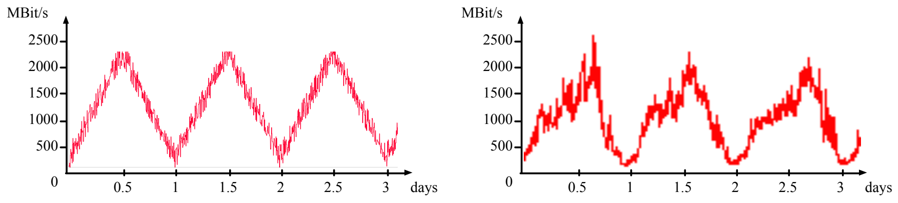


Fig. 4. Bandwidth feature (left) and real-world bandwidth behavior (right)

Aggressive stands for a policy where all those cards are switched off which have not been used within the last five minutes

High Savings assumes 10% higher bandwidth before switching off cards

Balanced behaves as the high savings policy, but with an additional cool-down phase of 30 minutes after the activation of a network card in which network cards can only be activated but not deactivated

Note that both, the distribution and switch policy feature, are chosen initially when the eServer is deployed and cannot be changed any further during runtime. Furthermore, all the features described by now behave deterministically, but depend on the environment modeled probabilistically.

Environment Features. For a quantitative analysis of eServer, we further incorporate environment features, which implement deterministic environment behavior such as time and statistical assumptions on the environment, e.g., the feature switch behavior or the requested bandwidth the server platform has to face. Feature switches are influenced by the environment, since replacing hardware clearly depends on the reliability of the technical staff of the server operator. We exemplified this influence by assuming that the technical staff requires at least five minutes after the need for a new network card has been discovered, and arrives with a probability of 90% in each time interval of five minutes. The bandwidth is modeled via a noised zick-zag curve parameterized over a maximal bandwidth value the server has to expect. This curve follows the behavior of real-world server systems, where the same characteristics can be observed: during night time, bandwidth requirements are almost vanished, whereas in the mid day, the requested bandwidth peaks at a value which is almost constant over the days. In Figure 4, a plot of our bandwidth model over three days is shown on the left, whereas a real-world example taken from [30] is shown on the right. In both cases, the peak for the requested bandwidth is at about 2.4 GBit/s. Thanks to enhancing our framework by variables, these environmental parameters can easily be encoded as variables `time` and `bandwidth`.

Feature Controller. Rules for plugging new network cards or upgrading a slow network card to a fast one are implemented into a feature controller. These rules are a combination of restrictions provided by the vendor of the product line or the server operator. Whereas the vendor restricts feature switches only in the sense that the aimed product should not leave the product line, the server operator may require that money for new network cards should only be spent if the network card is needed for the eServer to operate faithfully, i.e., when the workload of the system is almost at the maximum of the available bandwidth. Furthermore, we assume that network cards that are inactive do not consume any energy and the system operator does not allow for downgrading, i.e., unplugging a network card from the system. Figure 5 shows the fragment of the feature controller we implemented for eServer, where it is assumed that the initial product is an advanced server with two fast and one slow slot (similar to the professional eServer device presented in [23]) initially equipped with one slow network card only. The figure shows the quantity of the features for the network cards only,

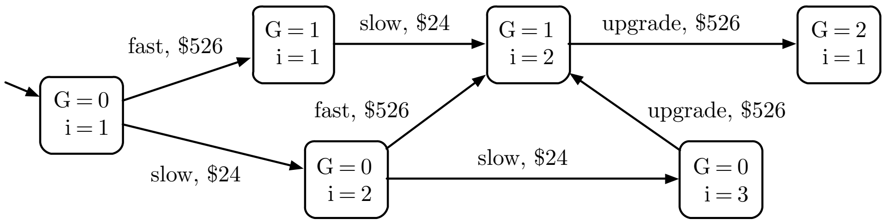


Fig. 5. Fragment of the eServer feature controller

captured by pairs [G $= n$, i $= k$], which stand for n active 10$\underline{\text{G}}$Bit features and k active 1GB$\underline{\text{i}}$t network card features. There are three possible actions which can be performed by the feature controller: plugging a fresh 10 GBit card into the system (fast), replacing a 1 GBit card by a 10 GBit one (upgrade), and plugging a fresh 1 GBit card into the system (slow). New cards go along with monetary costs, i.e., a 10 GBit card costs \$526, whereas a 1 GBit card sells at \$24. These prices are taken from the vendors product line which inspired the eServer example. Not drawn in the figure are the constraints on the transitions, requiring, e.g., that the technical staff is present and that the current requested bandwidth justifies the need of changing the feature combination, both influenced by the environment feature. Expressing this more formally with variables and linear constraints on feature combinations, and assuming that the environment variable bandwidth is measured in GBit/s, each transition is in fact equipped with a guard

$$80\% \cdot (10 \cdot G + i) \ < \ \mathsf{bandwidth},$$

meaning that the workload of the network cards is higher than 80% and hence, the system is under stress. This may lead to a point where the server may not be able to serve the bandwidth requested. The latter corresponds to a *service-level agreement (SLA) violation* in the terms by [30].

Energy Consumption and Monetary Costs. Quantitative properties of the eServer product line are incorporated through the annotation of costs, where we consider in particular the energy consumption of the network cards and monetary costs. For the latter, we annotated costs to the feature controller describing the money to be spent for plugging new cards. Furthermore, we annotate the initial costs for the system purchased, where the entry systems range from \$629 to \$1494, the advanced systems from \$1279 to \$1699 and the premium systems from \$2139 to \$9399. For requesting technical staff we assume costs of \$39.

The energy consumption of the network cards (we refer to an Intel Ethernet Server Adapter X520-T featuring an E76983 CPU as 10 GBit card and an Intel EXPI9301CTBLK network interface card with an E25869 CPU as 1 GBit card) highly depends on the workload. Detailed measurements for the cards mentioned above have been undertaken in [30] in the scope of eBond. We approximate their results by linear functions, as suggested by the authors of [30]. The 10 GBit card consumes 7.88 Watts in the idle state and 8.10 Watts under full load. For the 1 GBit card, the power consumption rises linearly from 1.35 Watts until reaching a throughput of 540 MBit/s at 1.92 Watts, staying constantly at this energy consumption until the full load is reached. Thus, as one expects, the energy consumption of the fast card is higher than of the slower one. This yields potential for energy saving when controlling the utilization of the network cards, e.g., through different coordination features.

MDP-semantics. Via parallel composition of the feature modules described above, including the environment features containing stochastic assumptions and joining them with the feature controller, we obtain the MDP semantics of the eServer product line:

$$\mathcal{M} \;=\; (\;\underbrace{\mathsf{eServer} \,\|\, \mathsf{Hardware} \,\|\, \mathsf{Coord}}_{\text{eServer}} \;\|\; \underbrace{\mathsf{Env}}_{\text{environment}}\;) \bowtie \mathsf{Con}$$

Here, eServer stands for the basic server functionality, incorporating the interplay between software, hardware and environment, which are in turn implemented through the feature modules Hardware, Coord and Env. This interplay is managed in a cyclic manner through three phases: first, the hardware is allowed to be changed according to the rules by the feature controller Con, then the control is handed to Coord, activating and deactivating network cards according to its switching policy, before the environment takes over for five minutes, providing the model of the requested bandwidth from the users the server has to compete with. Each phase corresponds to a step in $\mathcal{M}$. Note that the feature modules given above are in fact feature modules which arise by parallel composition of feature modules standing for features in the feature diagram of the eServer product line. Coord arises by parallel composition of the modules belonging to the coordination feature, whereas Hardware arises by parallel composition of all the other feature modules except the environment features, which are implemented in the feature module Env.

5.3 Quantitative Analysis of the eServer Product Line

Besides solving the strategy synthesis problem for the eServer product line under certain assumptions on the environment regarding energy consumption and

monetary costs (those characteristics rely on the eServer product line itself), we also consider the amount of time the server could not deliver the bandwidth requested by the users. This situation is called a *service-level agreement (SLA) violation* (according to [30]) and may happen either when the eServer is not appropriately equipped (the feature combination does not suffice) or when the requested bandwidth peaks and the coordination feature deactivated too many cards for saving energy. SLA violations also influence the money spent for the system during runtime. Besides the costs for purchasing the eServer, the costs for the technical staff and reconfiguration of features, we modeled costs for SLA violations that are rather expensive. Five minutes not serving the bandwidth requested costs \$200. It is clear that a customer then tries to avoid SLA violations by purchasing a device whose reliability guarantees the desired throughput functionality. On the other hand, a customer also tries to save initial costs when buying the device and to save energy during runtime using the advantages of the energy-saving switch policies.
This trade-off directly leads to the question how to choose the initial feature combination and when to reconfigure the system by feature switches. That is, solving the strategy synthesis problem for $\mathcal{M}$ regarding various quantitative objectives concerning, e.g., energy, money and SLA violations. Although our framework directly permits to consider arbitrary quantitative objectives which can be stated for standard MDPs, e.g., expressed within PCTL, we restrict ourselves to (constrained) reachability objectives in this case study. In particular, we consider here four different strategy synthesis problems for $\mathcal{M}$: maximizing the probability of not raising an SLA violation (i.e., reliability of the device), minimizing the expected energy consumption, money spent or time with SLA violations, respectively, all within a fixed time horizon:

$$\text{pmax} = \mathbb{P}^{\max}((\neg\, \mathit{Violation})\ \mathcal{U}\ T) \qquad \text{emin} = \mathbb{E}^{\min}[\text{energy}](\Diamond T)$$
$$\text{mmin} = \mathbb{E}^{\min}[\text{money}](\Diamond T) \qquad \text{vmin} = \mathbb{E}^{\min}[\text{violation}](\Diamond T)$$

Here, the type of the expected minimal costs is annotated to the query (i.e., energy, money and violation for SLA violations). Furthermore, *Violation* stands for the set of states in $\mathcal{M}$ where an SLA violation occurred and T for the set of states in $\mathcal{M}$ where some fixed time horizon is reached. Using the compositional framework presented in Section 3 and its extension with variables (Section 4), we modeled a parameterized version of the eServer product line in the guarded-command input language of Prism. Our model is parameterized in terms of the peak bandwidth during a day/night-cycle. Depending on this maximal bandwidth, different initial feature combinations and strategies for feature switches may provide different optimal solutions for pmax, emin, mmin, and vmin.

General Facts. For our case study we fixed certain model parameters. We chose a time horizon of the first day the deployed system is in operation ($T = 24$ hours) and solved the strategy synthesis problem for maximal bandwidths ranging from 100 MBit/s to 16 GBit/s in steps of 100 MBit/s. For each of the quantitative objectives, we present four graphs, each showing one chart for each product configuration at the deployment of the eServer. The first three show the results for all entry level, advanced and premium eServer products, respectively. Charts with similar colors are representing similar multi-features, i.e., a similar number and types of slots and network cards. In all these graphs, the difference between the coordination features chosen can hardly be figured out, due to the large-scaled product line, which yields many overlapping charts. Hence, we spot on those advanced eServer products in the lower right graph which have one fast and two slow slots and are purchased with one 10 GBit card only. This gives rise to 12 possible charts, representing the feature combinations for the coordination feature: colors encode the distribution feature (black, red, green, blue for Round Robin, Weighted, Maximal Fast, and Minimal Fast features, respectively) and the line type stands for different switching policies (solid, dotted, dashed for Aggressive, High Savings, and Balanced, respectively).

Utility Analysis. We first look at pmax, i.e., the maximum probability of avoiding an SLA violation within the first day of usage, corresponding to a measure of reliability for an eServer product. In Figure 6 it can be seen that when the maximal required bandwidth is below 1 GBit/s, SLA violations can almost surely be avoided within all kinds of servers. This is clear, since at least one card needs to be active in the server, such that at least a 1 GBit/s can be served at any time. When the initial feature combination is not sufficient to serve the maximal requested bandwidth, the maximal probability avoiding an SLA violation during one day drops significantly. This can be seen especially at bandwidths with 1, 2 or 10 GBit/s. In general, given the maximal bandwidth assumed to be requested by users, the best choice for an initial feature combination is the one corresponding to the topmost chart. The advanced products detailed in the last graph show that the chosen switching policy has a very similar influence on the results as determined in the original eBond case studies [30,23]. An aggressive strategy almost surely raises an SLA violation when turning 10 GBit/s, whereas plugging a new slow card and choosing a strategy with a higher bandwidth assumption still retains a possibility to circumvent an SLA violation until 11 GBit/s are reached. The distribution algorithms do not influence significantly this probability property and are almost indistinguishable.

Energy Analysis. When turning to the minimization of the expected energy consumption, i.e., computing emin for $\mathcal{M}$, it is clear that the best strategy is to never upgrade or buy new cards, keeping the energy costs as small as possible. Hence, the smallest configuration with only one slow card initially activated performs best with only 1.88 Watts of energy consumption for maximal bandwidths greater than 1 GBit/s (cf. Figure 7). Configurations activating a fast card only in situations when the bandwidth is above 1 GBit/s range between the energy consumption of slow and fast cards until reaching 10 GBit/s. In between, the charts in Figure 7 show mixed configurations, where mainly the switching policies influence the energy consumption. The aggressive policy requires least energy, followed by the high savings and the balanced one. This can also be seen in our example shown in the last graph, where until reaching 80% workload of the initial fast card, the energy consumption equals the energy characteristics of the 10 GBit card. For higher bandwidths than 8 GBit/s, a new slow card can

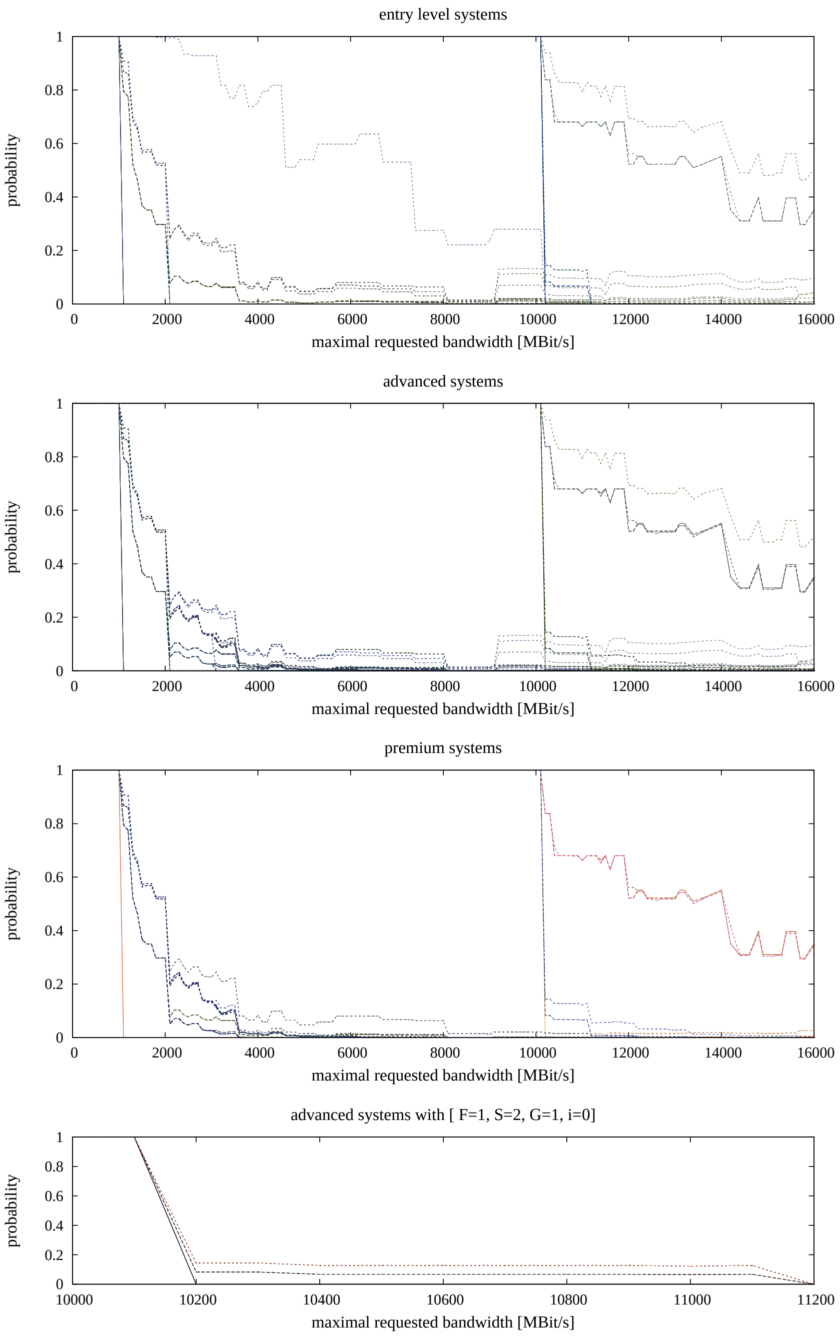


Fig. 6. Evaluation of pmax for the different eServer variants

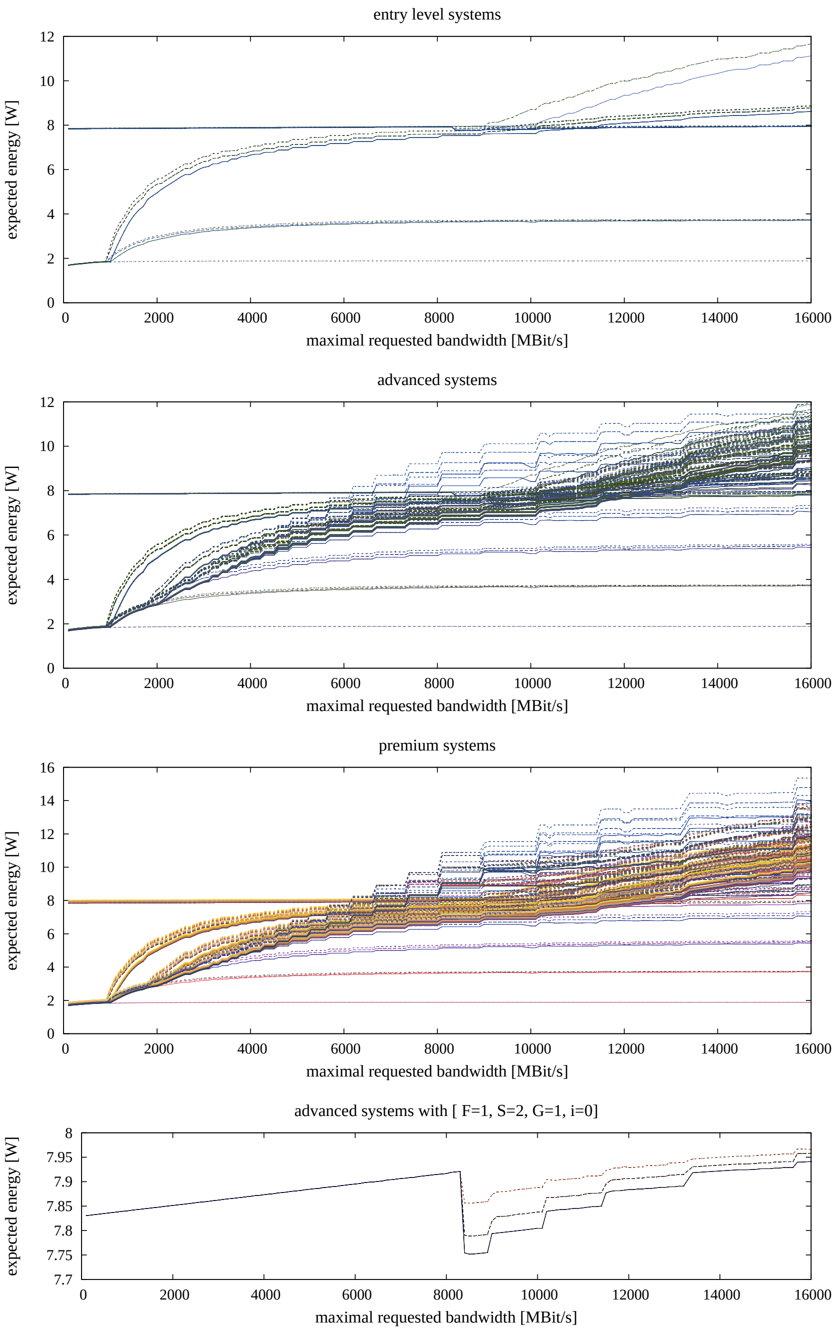


Fig. 7. Evaluation of emin for the different eSERVER variants

then be plugged and thus, the energy consumption can be reduced relying on the switching policy.

SLA Violation Analysis. When minimizing the expected number of SLA violations, i.e., computing vmin for $\mathcal{M}$, similar phenomena can be observed as within our utility analysis. The solution of the strategy synthesis problem yields a scheduler upgrading and plugging new and fast cards as soon as the feature controller permits it. However, as one can see in Figure 8, choosing initial configurations with only slow slots, the expected percentage of time within an SLA violation increases significantly when the maximal required bandwidth exceeds the supported bandwidth of the server with the maximally equipped network cards. Especially for the entry level systems, one can easily distinguish between the systems having only one slot (raising SLA violations when the bandwidth exceeds 1 GBit/s or 10 GBit/s) and having two slots from which at least one is a slow slot (raising SLA violations at 2 GBit/s or 11 GBit/s). In the lower left, premium systems stay below 12% of the time within an SLA violation if the bandwidth is below 6 GBit/s, which then may grow very fast. This is mainly due to the fact that a premium server system has at least six slots where cards can be plugged. When choosing the example configuration (see the last graph), the minimal expected percentage of time run with SLA violations with a maximal bandwidth of 11 GBit/s is quite low with at most 3%. Note that as in eBond case study, the balanced switching policy minimizes SLA violations always best, followed by the high savings and aggressive policy.

Monetary Analysis. Closely related to the SLA violation time analysis is the solution of the strategy synthesis problem which minimizes the money to be spent for the eServer system. Figure 9 shows the results for computing mmin for $\mathcal{M}$. Choosing a system with a fast 10 GBit network interface card does not yield additional costs after purchase, since SLA violations are very unlikely (see utility analysis for pmax). However, when purchasing only small configurations, expenses may exceed the costs for high equipped server products when facing higher bandwidths due to SLA violation fees to be paid. Thus, the customer may purchase a better performing but more expensive system if the maximal required bandwidth is high. However, as the first graph shows, it is a good strategy to buy an entry-level system with fast slots and upgrade cards on demand, facing only a few of SLA violations and resulting into low monetary costs.

5.4 Scalability and Statistical Evaluation

As the case study in the last section already illustrated, it is a challenging task to verify large-scaled product lines with thousands of feature combinations. However, using symbolic encodings for the model and information about the structure of the feature diagram, we managed to apply probabilistic model checking for a quantitative analysis. But even after we could reduce the size of the model encoding, we had to carefully choose the numerical methods to guarantee convergence of the approximation algorithms. In this section, we deal with the model and runtime characteristics of the eServer case study to show scalability of

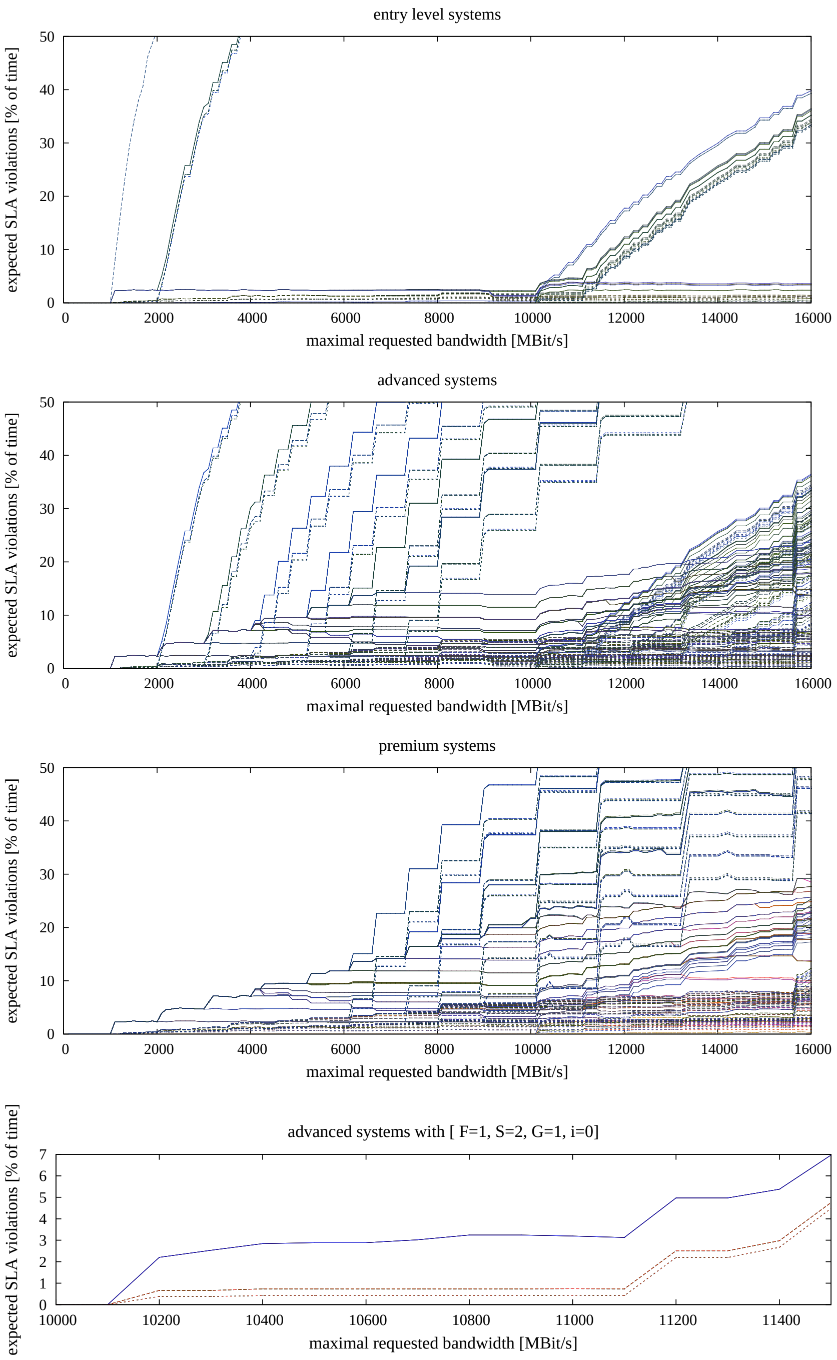


Fig. 8. Evaluation of vmin for the different eServer variants

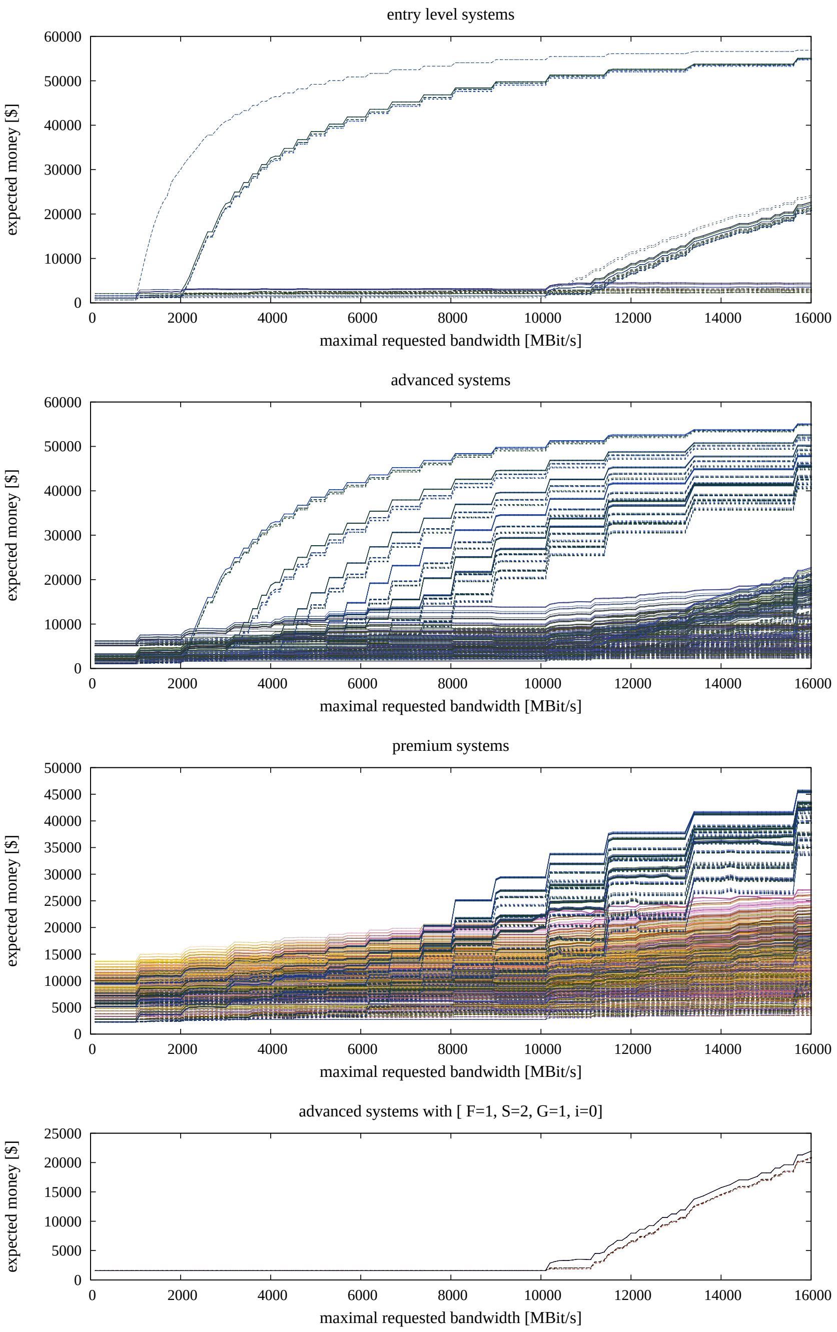


Fig. 9. Evaluation of mmin for the different eSERVER variants

our approach towards model checking dynamic product lines which incorporate multi-features and hence are large-scaled.

Runtime Characteristics. The case study was carried out on an Intel Xeon X5650 @ 2.67 GHz with 384 GBytes of RAM and using the symbolic MTBDD engine of Prism 4.1 with a precision of 10^{-5}. The logarithmically scaled Figure 10 shows the time needed for model checking, the number of states in the model, and the memory consumption – each depending on the maximal bandwidth assumed in the environment feature. Note that the behaviors of all feature combinations and feature switches are encoded into one single model, such that the model-checking time includes the computation time of all the four properties of the case study and for all 17,544 initial feature combinations. The symbolic representation of the model allowed for a memory consumption of only a few MBytes in all cases. At 16 GBit/s we had to construct a system model of 465,950,960 states and 1,072,736,675 transitions.

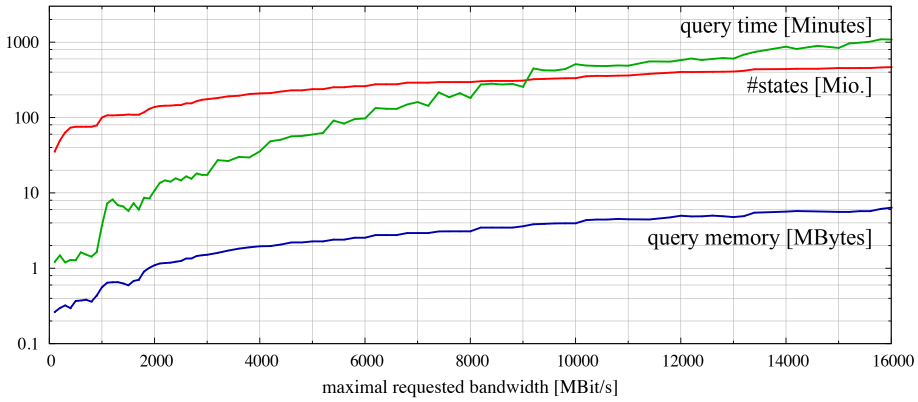


Fig. 10. Statistical evaluation of the experiments

Compact State-space Representation. Within models symbolically represented using multi-terminal binary decision diagrams (MTBDDs), states and transitions are encoded by a binary tree-like structure where branchings stand for decisions on variable valuations [26]. Traversing this tree-like diagram, the decisions are not made in an arbitrary fashion, but follow a given variable ordering. It is well-known that the size of such diagrams crucially depends on that variable ordering [9]. In our setting, the variables which appear in the MTBDD representation exactly correspond to the variables *Var* of the feature modules presented in Section 4, where we assume that also the locations of feature modules are encoded by variables. More formally, a variable ordering on *Var* is a partial order $\pi = (Var, \leq)$, where if $x < y$, the variable x is decided prior to y in the MTBDD of the model. Given two sets of variables $A, B \in Var$, we write $A \leq B$ iff for all $x \in A$, $y \in B$ we have $x \leq y$.

To optimize the size of the model representation of the large-scaled eServer product line $\mathcal{M}$, we investigated several variable orderings on Var. A good heuristic for variable orderings in MTBDDs is to first decide variables which are "most-influential", i.e., changed only at the beginning of an execution of the modeled system but influence the systems behavior significantly [38]. This directly fits to the product-line setting by placing variables of static features before the variables of dynamic features and to order variables of the same feature module close together. Also environment features and the base scheduling of interplay between hardware, software and environment can be assumed to change their behavior quite often and should be the greatest elements of an ordering. Keeping these facts in mind, we hence started with an intuitive variable ordering π_{start} defined using the modular structure of the product line:

$$Con \;<\; \underbrace{Hardware}_{\mathrm{T<o<N}} \;<\; \underbrace{Coordination}_{\mathrm{D<y}} \;<\; \underbrace{eServer}_{\mathsf{phase}} \;<\; \underbrace{Env}_{\mathsf{bandwidth<time}} \;,$$

where the sets stand for variables in Var contained in the respective feature modules, e.g., $Hardware$ contains all variables of the feature modules T, o and N incorporated in Hardware. phase is a variable encoding the three phases of eServer, i.e., whether the system is in a reconfiguration, coordination or environment phase. bandwidth and time are environment variables encoding the requested bandwidth and the time passed. Then, we applied sifting methods [46] for dynamically optimizing variable orderings, which revealed a variable ordering π_{opt}:

$$eServer \;<\; Con \;<\; Hardware \;<\; Coordination \;<\; Env.$$

For comparison reasons, we also defined variable orderings ρ_{start} and ρ_{opt}, which denote the reverse variable orderings of π_{start} and π_{opt}, respectively.

Table 1. Statistics of various variable orderings (maximal bandwidth = 2.4 GBit/s)

variable order	#states	#nodes	memory [MBytes]	query time [min]
π_{start}	145,984,112	116,381	2,337	17
ρ_{start}	"	168,043	3,275	72
π_{opt}	"	63,990	1,207	11
ρ_{opt}	"	175,467	9,253	237

Table 1 depicts the influence of these variable orderings on the performance of solving the strategy synthesis problem for $\mathcal{M}$ and the four queries pmax, emin, vmin and mmin under the assumption that the maximal requested bandwidth is 2.4 GBit/s. As it can be seen, optimizing the variable ordering has a strong impact on the nodes of the MTBDD required to encode the model and the time needed for the query computation. The complete case study presented in the last section has been carried out using the variable ordering π_{opt}. The computations would have taken more than one day each for maximal bandwidths greater than 5.4 GBit/s if we would have chosen the variable ordering ρ_{opt}.

Symbolic vs. Explicit Model Checking. It is well-known that an explicit engine is usually faster than a symbolic one when the model contains lots of different numeric values or available memory is not the restricting factor of the system setup. However, the operational model for product lines designed through multi-features contain lots of symmetric behaviors due to the several instances of multi-features and hence, symbolic methods outperform the explicit ones in our case study. Table 2 compares the characteristics solving the strategy synthesis problems for an eServer (again assuming 2.4 GBit/s maximal bandwidth) and the four queries of our case study using various engines. Besides the MTBDD engine used in the whole case study, we run the sparse and explicit engine of Prism. Whereas the sparse engine constructs the model symbolically and then uses an explicit sparse matrix representation for solving queries, the explicit engine also constructs the model explicitly. This has a strong impact especially on memory consumption, peaking at over 240 GBytes within the explicit engine.

Table 2. Statistics of various Prism engines (maximal bandwidth = 2.4 GBit/s)

engine	all-in-one			one-by-one
	#states	memory [MBytes]	query time [min]	query time [min]
MTBDD	145,984,112	1,207	11	3,112
sparse	”	11,167	224	3,156
explicit	”	241,991	432	802

All-in-one vs. One-by-one. Within our approach, all behaviors of the products in the dynamic product line are encoded into a single model, similar to the family-based approaches for product line analysis [53]. This allows to exploit the commonalities between the products, especially in combination with a symbolic representation of the model. However, we have shown in the last paragraph that explicit engines for probabilistic model checking do not perform well on large models due to memory constraints, such that checking every product in isolation and hence dividing the model into smaller parts might still yield a faster analysis method. Table 2 also depicts a comparison between the explicit and symbolic engines of Prism used to analyze the 17,544 products of the eServer product line one-by-one when assuming a maximal bandwidth of 2.4 GBit/s. The largest model of a single product in the product line contains 120,575 states. Both, the MTBDD and sparse engine computations took more than two days. Although the explicit engine turned out to be the fastest engine for the one-by-one approach, it took more than 70 times longer than the all-in-one MTBDD-approach.

6 Conclusions

We presented a compositional modeling framework for dynamic product lines that relies on annotated versions of probabilistic automata. The implementation of features and the behavior of possibly unknown or only partially known implementations of external features are represented by feature modules, which are probabilistic automata with guards and special switch transitions for the feature changes. Constraints on the activation and deactivation of features during runtime of the system are imposed by feature controllers, probabilistic automata synchronizing with switch transitions of feature modules. Most of the family-based verification approaches for static and nonprobabilistic product lines use monolithic models including all behaviors of the products in the product line. Our approach with feature modules and controllers allows to generate such operational models in a compositional way.

Dynamic product lines modeled within our framework yield an MDP semantics, such that many problems for feature-oriented systems can be solved using standard algorithms. This includes model-checking problems for properties referring to feature combinations, which till now required specialized algorithms even in the nonprobabilistic setting [12]. We also presented a translation from our framework into guarded-command languages used, e.g., by the prominent probabilistic model checker Prism. For a case study concerning an energy-aware server product line (called eServer), we used Prism to solve the strategy synthesis problem that asks for strategies to trigger feature combination changes according to various quantitative properties. We also placed the focus on large-scaled product lines which contain thousands of valid feature combinations and can be described elegantly through multi-feature diagrams. For large-scaled eServer models, we compared different model-checking engines and showed that symbolic approaches clearly outperform explicit ones.

There are many other interesting variants of the strategy synthesis problem that are also solvable by known algorithms applicable to the MDP semantics of our framework. One might distinguish between switch events that are indeed controllable and those that cannot be enforced or prevented, but are triggered by the environment. In this case, the arising MDP can be seen as a stochastic game structure, where the controller and the environment are opponents and the task to generate an optimal strategy for the controller reduces to well-known game-based problems [15,24,21,10]. Similarly, one might take into account that also the feature modules can behave nondeterministically.

A challenge remaining for further work is to integrate our feature-oriented formalisms into model-checking tools to ease their use for software developers, enabling to integrate quantitative analyses into the workflow of product-line development. This includes the interpretation and compact output of the strategies solving the strategy synthesis problem, till now only internally computed by existing model-checking tools. Also investigations on feature-dependent multi-objectives are important in this context [4]. Such requirements would, e.g., enable to check whether the trade-off between energy consumption and the time without SLA violations is better for premium or advanced eServer variants [3].

References

1. S. Apel and D. Hutchins. A calculus for uniform feature composition. *ACM Transactions on Programming Languages and Systems*, 32(5), 2010.
2. S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model superimposition in software product lines. In *Proc. of the 2nd Conference on Theory and Practice of Model Transformations (ICMT)*, volume 5563 of *LNCS*, pages 4–19. Springer, 2009.
3. C. Baier, C. Dubslaff, J. Klein, S. Klüppelholz, and S. Wunderlich. Probabilistic model checking for energy-utility analysis. In *Horizons of the Mind. A Tribute to Prakash Panangaden*, volume 8464 of *LNCS*, pages 96–123. Springer, 2014.
4. C. Baier, C. Dubslaff, S. Klüppelholz, M. Daum, J. Klein, S. Märcker, and S. Wunderlich. Probabilistic model checking and non-standard multi-objective reasoning. In *Proc. of the 17th Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 8411 of *LNCS*, pages 1–16. Springer, 2014.
5. C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.
6. C. Baier and M. Kwiatkoswka. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.
7. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.
8. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc. of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 1026 of *LNCS*, pages 499–513, Berlin, 1995. Springer.
9. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
10. K. Chatterjee, M. Jurdzinski, and T. Henzinger. Quantitative simple stochastic parity games. In *Proc. of the 15th ACM-SIAM Symposium on Discrete algorithms (SODA)*, pages 121–130. SIAM, 2004.
11. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.
12. A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *Proc. of the 33rd Conference on Software Engineering (ICSE)*, pages 321–330. ACM, 2011.
13. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. of the 32rd Conference on Software Engineering (ICSE)*, pages 335–344. ACM, 2010.
14. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2001.
15. A. Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992.
16. M. Cordy, A. Classen, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking adaptive software with featured transition systems. In *Assurances for Self-Adaptive Systems*, volume 7740 of *LNCS*, pages 1–29. Springer, 2013.
17. M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *Proc. of the 35rd Conference on Software Engineering (ICSE)*, pages 472–481. IEEE Press, 2013.

18. K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
19. F. Damiani and I. Schaefer. Dynamic delta-oriented programming. In *Proc. of the 15th Software Product Line Conference (SPLC), Volume 2*, pages 34:1–34:8. ACM, 2011.
20. L. de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In *Proc. of the 10th Conference on Concurrency Theory (CONCUR)*, volume 1664 of *LNCS*, pages 66–81. Springer, 1999.
21. L. de Alfaro and R. Majumdar. Quantitative solution of omega-regular games. *Journal of Computer and System Sciences*, 68(2):374–397, 2004.
22. T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A dynamic software product line approach using aspect models at runtime. In *Proc. of the 1st Workshop on Composition and Variability*, 2010.
23. C. Dubslaff, S. Klüppelholz, and C. Baier. Probabilistic model checking for energy analysis in software product lines. In *Proc. of the 13th Conference on Modularity (MODULARITY)*, pages 169–180. ACM, 2014.
24. J. Filar and K. Vrieze. *Competitive Markov Decision Processes.* Springer, 1997.
25. V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In *Proc. of the 11th School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM)*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.
26. M. Fujita, P. McGeer, and J.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2-3):149–169, 1997.
27. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, 1992.
28. C. Ghezzi and A. M. Sharifloo. Model-based verification of quantitative non-functional properties for software product lines. *Information & Software Technology*, 55(3):508–524, 2013.
29. H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. In *Proc. of the 5th Workshop on Software Product-Family Engineering (PFE)*, volume 3014 of *LNCS*, pages 435–444. Springer, 2003.
30. M. Hähnel, B. Döbel, M. Völp, and H. Härtig. eBond: Energy saving in heterogeneous R.A.I.N. In *Proc. of the 4th Conference on Future Energy Systems (e-Energy)*, pages 193–202, New York, NY, USA, 2013. ACM.
31. S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *IEEE Computer*, 41(4):93–95, 2008.
32. B. Haverkort. *Performance of Computer Communication Systems: A Model-Based Approach.* John Wiley & Sons, Inc., 1998.
33. J. D. Hay and J. M. Atlee. Composing features and resolving interactions. In *Proc. of the 8th Symposium on Foundations of Software Engineering (SIGSOFT)*, pages 110–119. ACM, 2000.
34. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Proc. of the 12th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.
35. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University, November 1990.

36. S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, 1993.
37. V. Kulkarni. *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, 1995.
38. S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. of the IEEE Conference on Computer-Aided Design (ICCAD)*, pages 6–9, 1988.
39. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
40. J.-V. Millo, S. Ramesh, S. N. Krishna, and G. K. Narwane. Compositional verification of software product lines. In *Proc. of the 10th Conference on Integrated Formal Methods (IFM)*, volume 7940 of *LNCS*, pages 109–123. Springer, 2013.
41. M. Noorian, E. Bagheri, and W. Du. Non-functional properties in software product lines: A taxonomy for classification. In *Proc. of the 24th Conference on Software Engineering & Knowledge Engineering (SEKE)*, pages 663–667. Knowledge Systems Institute Graduate School, 2012.
42. G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:329–400, 1998.
43. M. Plath and M. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84, 2001.
44. M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.
45. M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Flexible feature binding in software product lines. *Automated Software Engineering*, 18(2):163–197, 2011.
46. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. of the IEEE/ACM Conference on Computer-Aided Design (ICCAD)*, pages 42–47. IEEE Computer Society, 1993.
47. J.-G. Schneider, M. Lumpe, and O. Nierstrasz. Agent coordination via scripting languages. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 153–175. Springer, 2001.
48. R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.
49. R. Segala and N. A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.
50. N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information & Software Technology*, 55(3):491–507, 2013.
51. N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake. Measuring non-functional properties in software product line for product derivation. In *Proc. of the 15th Asia-Pacific Software Engineering Conference (APSEC)*, pages 187–194. IEEE, 2008.
52. M. Varshosaz and R. Khosravi. Discrete time Markov chain families: modeling and verification of probabilistic software product lines. In *Proc. of the 17th Software Product Line Conference Co-located Workshops*, pages 34–41. ACM, 2013.
53. A. von Rhein, S. Apel, C. Kästner, T. Thüm, and I. Schaefer. The PLA model: On the combination of product-line analyses. In *Proc. of the 7th Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 14:1–14:8. ACM, 2013.
54. J. White, B. Dougherty, D. C. Schmidt, and D. Benavides. Automated reasoning for multi-step feature model configuration problems. In *Proc. of the 13th Software Product Line Conference (SPLC)*, pages 11–20. ACM, 2009.