# Probabilistic Software Product Line Model Checking

Clemens Dubslaff, Sascha Klüppeholz, Christel Baier*
Institute for Theoretical Computer Science
Technische Universität Dresden, Germany

{dubslaff,klueppel,baier}@tcs.inf.tu-dresden.de

In a *software product line (SPL)*, a collection of software products is defined by their commonalities in terms of features rather than explicitly specifying all products one-by-one.

Several verification techniques were adapted to establish temporal properties of SPLs. Symbolic and family-based model checking have been proven to be successful for tackling the combinatorial blow-up arising when reasoning about several feature combinations. However, most formal verification approaches for SPLs presented in the literature focus on the *static* SPLs, where the features of a product are fixed and cannot be changed during runtime. This is in contrast to *dynamic* SPLs, allowing to adapt feature combinations of a product dynamically after deployment.

The presentation deals with a compositional modeling framework for dynamic SPLs, which supports probabilistic and nondeterministic behaviors and allows for *quantitative analysis*. Feature changes during runtime are modeled within an automata-based coordination component, enabling to reason over strategies how to trigger dynamic feature changes for optimizing various quantitative objectives, e.g., energy or monetary costs and reliability. There is a natural and conceptually simple translation from the presented framework into the input language of the prominent probabilistic model checker PRISM. This facilitates the application of PRISM to verify the operational behavior of dynamic probabilistic SPLs against various quantitative queries. Feasibility of this approach is demonstrated by means of a case study issuing an energy-aware bonding network device.

## 1 Introduction

In order to meet economic requirements and to provide customers individualized solutions, the development and marketing of modern hardware and software products often follows the concept of *product lines*. Within this concept, customers purchase a base system extendible and customizable with additional functionalities, called *features*. Although product lines are commonly established in both, hardware and software development, they have been first and foremost considered in the area of software engineering. A *software product line (SPL)* (see, e.g., [4]) specifies a collection of software systems built from features according to rules describing realizable feature combinations. It is rather natural to consider features as basic modules for a compositional structure of SPLs. Such rules for the composition of features are typically provided using *feature diagrams* [11]. Feature combinations are often assumed to be static, i.e., some realizable feature combination is fixed when the product is purchased by a customer and is never changed afterwards. However, this do not faithfully reflect adaptations of modern software during its lifetime. For instance, when a software is updated or when a free trial version expires, features are activated or deactivated during runtime of the system. SPLs which model such adaptations are called *dynamic SPLs* [7], for which the design of specification formalisms is an active and emerging field in SPL engineering.

*This work is partly supported by Deutsche Telekom Stiftung, the German Research Foundation (the SFB 912 HAEC, DFG/NWO-project ROCKS, cluster of excellence cfAED), and the EU 7th Framework Programme grant no. 295261 (MEALS).

Submitted to:
QAPL 2014

Besides extending SPLs by dynamic feature adaptations, also quantitative aspects of the operational behavior in SPLs are of increasing interest [12, 6, 16]. However, existing models are limited to static SPLs and are not compositional. This limits their application in terms of family-based quantitative analysis, i.e., reason about quantitative properties for all systems in an SPL by using the commonalities between them rather than checking them one-by-one [14].

Based on a recent publication [5], the goal of this presentation is to provide a *compositional framework* for modeling *dynamic SPLs* which allows for a *quantitative analysis* in order to reason, e.g., about system's resource requirements.

## 2 The Compositional Framework

For the compositional design of software products with parallel components, Markov chains are known to be less adequate than operational models supporting both, nondeterministic and probabilistic choices (see, e.g., [15]). A *Markov decision process (MDP)* is such a formalism, extending labeled transition systems by internal probabilistic choices taken after resolving nondeterminism between actions of the system. The presented compositional framework for dynamic SPLs relies on MDPs with annotated costs, used, e.g., to reason about resource requirements, energy consumption or monetary costs. In particular:

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

(2) a parallel operator for feature modules that represents the parallel execution of independent actions by interleaving, supporting communication according to the handshaking principle and over shared variables, and

(3) a feature controller: an MDP-like model for the dynamic switches of feature combinations.

An SPL naturally induces a compositional structure over features, where features or collections thereof correspond to components. In the framework, these components are called feature modules (1), which can contain both, nondeterministic and probabilistic choices. The former might be useful in early design stages, whereas probabilistic choices can be used to model the likelihood of exceptional behaviors (e.g., if some failure appears) or to represent randomized activities (e.g., coin tossing actions to break symmetry). Both kinds of choices may depend on other features of the SPL – for instance, whether another feature is activated during runtime or not.

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. This composition is defined upon compatible feature interfaces of the feature modules, which keep track of the features owned by the feature modules and those which the behavior of the feature modules depends on.

Feature activation and deactivation 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 probabilistically (e.g., on the basis of statistical information on feature combinations and their adaptations over time) or nondeterministically (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.

## 3 Verification of SPLs

In order to meet requirements in safety-critical parts of SPLs or to guarantee overall quality, verification is highly desirable. This is especially the case for dynamic SPLs, where side-effects arising from dynamic feature changes are difficult to predict in development phases. Model checking [10, 1] is a fully automatic verification technique for establishing temporal properties of systems (e.g., safety or liveness properties). Indeed, it has been successively applied to integrate features in components and to detect feature interactions [13]. However, as observed by Classen et al. [3, 2], the typical task for reasoning about static SPLs is to solve the so-called *featured model-checking problem*:

> Compute the set of all feature combinations such that the considered temporal property $\varphi$ holds for the corresponding software products.

This is in contrast to the classical model-checking problem that amounts to prove that $\varphi$ holds for some fixed system, such as one software product obtained from a feature combination. The standard approach solving the featured model-checking problem is to verify the products in the SPL one-by-one (see, e.g., [14]). However, already within static SPLs this approach certainly suffers from an exponential blow-up, since the number of different software products may rise exponentially in the number of features. To tackle this potential combinatorial blow-up, family-based [14] and symbolic approaches are very successful. Within *family-based analysis*, all products in an SPL are checked at once rather than one-by-one. This requires a model which represents all behaviors of all the products of the SPL.

**Quantitative Analysis.** Fortunately, the semantics of SPLs according to the proposed framework rise a standard MDP, which permits the application of standard but sophisticated probabilistic model-checking techniques to reason about quantitative properties. This is in contrast to existing (nonprobabilistic) approaches, which require model-checking algorithms specialized for SPLs. Within our approach, temporal or quantitative queries such as "minimize the energy consumption until reaching a target state" or "maximize the utility value to reach a target state for a given initial energy budget" can be answered. Corresponding to the nonprobabilistic case, the solution of the featured model-checking problem would then provide answers of these queries for all initial feature combinations. In the setting of dynamic SPLs, we go a step further and define the *strategy synthesis problem* aiming to find an optimal strategy of resolving the nondeterminism between feature combination switches in the feature controller. This strategy includes the initial step of the dynamic SPL by selecting an initial feature combination, which suffices to solve the featured model-checking problem. However, this approach additionally provides the possibility to reason over worst-case scenarios concerning feature changes during runtime. Note that solving the strategy synthesis problem imposes a family-based analysis approach of the dynamic SPL, which is also novel in the nonprobabilistic setting.

As in the nonprobabilistic case, symbolic techniques can help to avoid the exponential blow-up when analyzing probabilistic SPLs. The presented compositional framework nicely fits with guarded-command languages such as the input language of the symbolic probabilistic model checker PRISM [9]. We carried out a case study based on a real-case scenario from the hardware domain according to our framework to demonstrate applicability of PRISM. This case study details the energy-aware network device EBOND+, an extension of the recently presented EBOND device [8]. It is explained how PRISM can be used to solve the aforementioned strategy synthesis problem w.r.t. to several quantitative queries formalizing requirements, e.g., on the energy consumption of the EBOND+ device.

# References


[1] Christel Baier & Joost-Pieter Katoen (2008): *Principles of model checking*. The MIT Press.

[2] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens & Axel Legay (2011): *Symbolic Model Checking of Software Product Lines*. In: *ICSE'2011*, ACM, pp. 321–330.

[3] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay & Jean-François Raskin (2010): *Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines*. In: *ICSE'2010*, ACM, pp. 335–344.

[4] Paul Clements & Linda Northrop (2001): *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional.

[5] Clemens Dubslaff, Sascha Klüppelholz & Christel Baier (2014): *Probabilistic Model Checking for Energy Analysis in Software Product Lines*. In: *Proc. 13th International Conference on Modularity (MODULARITY'14)*. To appear.

[6] Carlo Ghezzi & Amir Molzam Sharifloo (2013): *Model-based verification of quantitative non-functional properties for software product lines*. *Information & Software Technology* 55(3), pp. 508–524.

[7] Hassan Gomaa & Mohamed Hussein (2003): *Dynamic Software Reconfiguration in Software Product Families*. In: *PFE*, pp. 435–444.

[8] Marcus Hähnel, Björn Döbel, Marcus Völp & Hermann Härtig (2013): *eBond: Energy Saving in Heterogeneous R.A.I.N.* In: *Proceedings of the Fourth International Conference on Future Energy Systems*, e-Energy '13, ACM, New York, NY, USA, pp. 193–202.

[9] Andrew Hinton, Marta Kwiatkowska, Gethin Norman & David Parker (2006): *PRISM: A Tool for Automatic Verification of Probabilistic Systems*. In Holger Hermanns & Jens Palsberg, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 3920, Springer Berlin Heidelberg, pp. 441–444.

[10] Edmund M. Clarke Jr., Orna Grumberg & Doron A. Peled (1999): *Model Checking*. The MIT Press.

[11] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak & A. Spencer Peterson (1990): *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report, Carnegie-Mellon University Software Engineering Institute.

[12] Mahdi Noorian, Ebrahim Bagheri & Weichang Du (2012): *Non-functional Properties in Software Product Lines: A Taxonomy for Classification*. In: *SEKE'12*, Knowledge Systems Institute Graduate School, pp. 663–667.

[13] Malte Plath & Mark Ryan (2001): *Feature integration using a feature construct*. *Science of Computer Programming* 41(1), pp. 53 – 84.

[14] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm & Ina Schaefer (2013): *The PLA Model: On the Combination of Product-line Analyses*. In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, ACM, New York, NY, USA, pp. 14:1–8.

[15] Roberto Segala (1995): *Modeling and Verification of Randomized Distributed Real-Time Systems*. Ph.D. thesis, Massachusetts Institute of Technology.

[16] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel & Sergiy S. Kolesnikov (2013): *Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption*. *Information & Software Technology* 55(3), pp. 491–507.