View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Technische Universität Dresden: Qucosa

Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (postprint):

Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, Christel Baier

Family-Based Modeling and Analysis for Probabilistic Systems – Featuring ProFeat

Erstveröffentlichung in / First published in:

Fundamental Approaches to Software Engineering, Eindhoven, 2016. SpringerLink, S. 287-304. ISBN 978-3-662-49665-7

DOI: https://doi.org/10.1007/978-3-662-49665-7_17

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-707905

SLUB
Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA
Quality Content of Saxony

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

Family-based Modeling and Analysis for Probabilistic Systems – Featuring ProFeat*

Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier

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

Abstract. Feature-based formalisms provide an elegant way to specify families of systems that share a base functionality and differ in certain features. They can also facilitate an all-in-one analysis, where all systems of the family are analyzed at once on a single family model instead of one-by-one. This paper presents the basic concepts of the tool ProFeat, which provides a guarded-command language for modeling families of probabilistic systems and an automatic translation of family models to the input language of the probabilistic model checker Prism. This translational approach enables a family-based quantitative analysis with Prism. Besides modeling families of systems that differ in system parameters such as the number of identical processes or channel sizes, ProFeat also provides special support for the modeling and analysis of (probabilistic) product lines with dynamic feature switches, multi-features and feature attributes. By means of several case studies we show how ProFeat eases family-based modeling and compare the one-by-one and all-in-one analysis approach.

1 Introduction

Feature orientation is a popular paradigm for the development of customizable software systems (see, e.g., [29,3,7]). Formalisms with feature-oriented concepts provide an elegant way to specify families of systems that can be seen as variants, sharing some base functionality but differing in the combinations of features. The most prominent application of feature-oriented formalisms are software product lines [12]. Several techniques for the analysis of feature-oriented models and software product lines using testing, type checking, static analysis, theorem proving

* This is a post-peer-review, pre-copyedit version of an article published in Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol 9633. The final authenticated version is available online at: https://doi.org/10.1007/978-3-662-49665-7_17. The authors are supported by the DFG through the collaborative research centre HAEC (SFB 912), the Excellence Initiative by the German Federal and State Governments (cluster of excellence cfAED and Institutional Strategy), the Research Training Groups QuantLA (GRK 1763) and RoSI (GRK 1907), and the DFG/NWO-project ROCKS, and Deutsche Telekom Stiftung.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

or model checking have been proposed and implemented in tools (see, e.g., [35] for an overview). The focus of the paper is on a feature-oriented formalism for the quantitative analysis of families of probabilistic systems modeled by discrete- or continuous-time Markov chains or Markov decision processes (MDPs). For this purpose, we introduce ProFeat, a feature-oriented extension of the input language of the probabilistic model checker Prism [28] together with an automatic translation of ProFeat models to pure (feature-less) Prism models. To specify valid feature combinations, we rely on a feature-model formalism similar to the Textual Variability Language (TVL) [8]. ProFeat also allows for (numerical) feature attributes and multi-features [16,8,15].

ProFeat follows the approach of [21] for modeling product lines using the parallel composition of (possibly interacting) feature modules and a feature controller that synchronizes with the feature modules when dynamic switches of the feature combinations occur. The dynamics of the feature controller and its interactions with the feature modules are crucial to model dynamic product lines [24,20,17,13]. Probabilistic dynamic product lines as presented in [21] allow, e.g., to model the frequencies of uncontrollable feature switches by stochastic distributions. The potential adaptations are then modeled by non-deterministic feature switches. In ProFeat the operational behavior of the feature modules and the feature controller are represented by an extension of Prism's guarded command language, supporting constraints for the feature combinations and synchronization actions for the activation and deactivation of features. Thus, whereas [21] uses MDP-like models for both the feature modules and the feature controller and handcrafted translations of the feature-oriented concepts into Prism language, the ProFeat framework provides an elegant way to specify the feature modules and feature controller and automatically generates corresponding Prism code.

The quantitative analysis of ProFeat models in terms of the (maximal or minimal) probabilities of path properties or expected costs can be carried out using Prism or other probabilistic model checkers that support Prism's input language. Besides the translation of ProFeat models into Prism models, our implementation also supports the analysis of product lines by providing commands to trigger the Prism model-checking engines either for the family model ("all-in-one") or for each family member separately ("one-by-one"). The one-by-one analysis can be carried out sequentially or in parallel.

Besides static or dynamic product lines, ProFeat can also be used to specify families of probabilistic systems with the same functionality, but different system parameters. Examples for such system parameters that may constitute a family of systems are initial values of discrete variables (and hence the set of starting states), threshold values triggering a certain behavior or reset values, the sizes of a buffer, a data package, an encryption key, the number of redundant components, or of retries and the energy consumption for some send operation.[1] In these cases, ProFeat's family-based modeling approach and support for the one-by-one analysis offers a convenient way to perform analysis benchmarks which till

[1] To ensure the finiteness of the family model (which is necessary to employ standard model-checking techniques) the range of the parameters is required to be finite.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

now are usually done using handcrafted templates. To illustrate the capabilities of ProFeat, we considered a series of examples and compared the performance of all-in-one and one-by-one analyses using the three Prism engines Mtbdd, Hybrid and Sparse. While the Mtbdd engine is fully symbolic and carries out all computations using multi-terminal binary decision diagrams (MTBDD), the numerical computations of the Sparse engine are carried out using sparse matrices, and the Hybrid engine relies on an MTBDD-representation of the model and a sparse representation of probability or expectation vectors. Our experimental results indicate that there is no clear superiority of the all-in-one analysis approach, no matter which of the three Prism engines is used. However, for well-known product line models, where the base functionality contains most of the behaviors and features have comparably less behaviors, all-in-one approaches are feasible (especially within the Mtbdd engine).

Related work. Various authors have presented model-checking techniques for families of non-probabilistic systems. For the automatic detection of feature interactions, Plath and Ryan [36] introduced a feature-oriented extension of the input language of the model checker SMV and Apel et al. [5] presented the tool SPLVerifier. FeatureIDE [40] is a tool set supporting all phases of the software-product-line development with connections to the theorem prover KeY and the model checker JPF-BDD. Gruler et al. [25] introduced a feature-based extension of the process algebra CCS and presented model-checking algorithms to verify requirements expressed in the μ-calculus. We are not aware of any implementation of this approach. Lauenroth et al. [34] deal with family models based on I/O automata with may ("variable") and must ("common") transitions and a model checker for a CTL-like temporal logic that has been adapted for reasoning about the variability of product lines. Featured transition systems (FTS) are labeled transition systems with annotations for the feature combinations of static product lines [11] or a variant of dynamic product lines [13]. The SNIP tool [9,11,15] relies on FTS specified using a feature-based extension of the modeling language Promela and allows for checking FTS against LTL properties one-by-one or using a symbolic all-in-one verification algorithm. Its re-engineered version ProVeLines [14] provides several extensions, including verification techniques for reachability properties with real-time constraints. For branching-time temporal-logic specifications, [13,10] proposed a symbolic model-checking approach for (adaptive) FTS. We are not aware of an implementation of the approach of [13]. In [10], an all-in-one analysis based on the feature-oriented extension of the SMV input language by [36] has been proposed, which allows verifying static product lines using the (non-probabilistic) symbolic model checker NuSMV. This extension of SMV follows the compositional feature-oriented software design paradigm (as we do) but puts the emphasis on superimposition [30,2,1], rather than parallel composition of feature behaviors [21].

None of the approaches mentioned above deals with probabilistic behaviors. To the best of our knowledge, there is no other tool that provides support for family-based probabilistic model checking of dynamic product lines. The benefits of probabilistic model checking for the analysis of adaptive software has

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

been already drawn by Filieri et al. [22]. The work on model-checking algorithms for parametric Markov chains [18,27] and tool support in the model checkers PARAM [26] (which has been reimplemented and integrated in PRISM) and PROPhESY [19] is orthogonal. By computing rational functions for the probabilities of reachability conditions or expected accumulated costs, these techniques can be seen as an all-in-one analysis of families of probabilistic systems with the same state space, but different transition probabilities. Ghezzi and Sharifloo [23] and the recent work by Rodrigues et al. [37] illustrate the potential of parametric probabilistic model-checking techniques for the analysis of product lines. The ProFeat language can handle probability parameters as well and translate them to PRISM code. However, there is no direct connection between ProFeat and the parametric probabilistic model checkers as they do not support multiple initial states. The recent work by Beek et al. [39] presents a framework for the analysis of software product lines using statistical model checking. An approach towards a family-based performance analysis of dynamic probabilistic product lines arising from UML activity diagrams has been presented by [32].

Outline. Section 2 presents the main principles of the ProFeat language. Details on the automatic generation of PRISM code from ProFeat models as well as explanations on ProFeat's support for the all-in-one and one-by-one analysis will be given in Section 3. Section 4 reports on experimental studies. A brief conclusion is provided in Section 5. The source code of ProFeat can be obtained at `https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/FASE16`.

2 Modeling Families of Systems: The ProFeat Language

A ProFeat model might represent a family of randomized protocols or other probabilistic systems that can be modeled by finite-state Markovian models, such as discrete- or continuous-time Markov chains (DTMCs, or CTMCs, respectively) or Markov decision processes (MDPs). Therefore, the model consists of two parts: the declaration of the family, and a compact feature-oriented representation of the operational behavior of all family members. Inspired by the application of feature-oriented formalisms for software product lines, a family member is specified by some combination of features, each either *active* or *inactive*. The language constructs for the declaration of feature models, i.e., the valid feature combinations, are inspired by the *Textual Variability Language* (TVL) [8]. For the definition of the operational behaviors, we adopt the guarded-command input language of the model checker PRISM [28] and extend it by feature-specific concepts presented in [21]. Guards in commands of a PRISM module can contain constraints for feature combinations. To model dynamic product lines, dynamic feature switches may occur by interactions with a feature controller, which is represented by a separate PRISM module with synchronization actions for activating and deactivating features. Apart from feature models, other families of probabilistic systems that differ, e.g., in the number of processes, the queue size or other system parameters can easily be modeled. For instance, existing

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

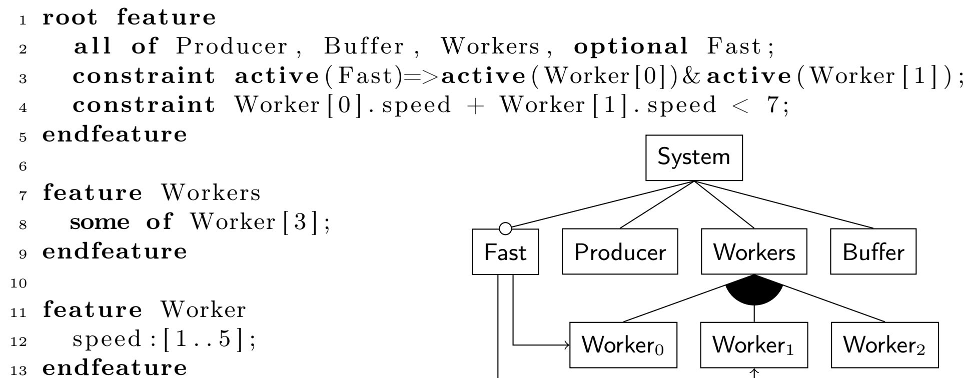
parametrized PRISM models are usually checked within PRISM's experiment environment in a one-by-one fashion. Within PROFEAT only minor modifications are necessary to represent the whole family of systems that differ in the system parameters.

We illustrate the modeling approach of PROFEAT using a simple producer-consumer model. The system consists of a single producer that enqueues jobs with probabilistic workload sizes into a FIFO buffer handed to one or more workers. One worker can only process one package at a time and the duration of the processing is determined by the work-package size. Varying the buffer size, the number of workers, the processing speed of individual workers or the load caused by the producer yields different variants, i.e., families of systems.

2.1 Feature Modeling

A product line comprises a set of feature combinations, which are defined through a *feature model*. For the producer-consumer product line, the following figure

```
1  root feature
2    all of Producer, Buffer, Workers, optional Fast;
3    constraint active(Fast)=>active(Worker[0])&active(Worker[1]);
4    constraint Worker[0].speed + Worker[1].speed < 7;
5  endfeature
6
7  feature Workers
8    some of Worker[3];
9  endfeature
10
11 feature Worker
12   speed:[1..5];
13 endfeature
```



depicts a feature diagram, the standard formalism to define feature models in the software-engineering domain, and a part of its declaration in PROFEAT. Similar to feature diagrams, where each feature is represented by a rectangular node, PROFEAT uses textual feature-blocks to declare features and also has a tree-like structure. The `root feature` (denoted by System in the diagram) is a special feature, representing the base functionality on which the product line is built upon. Each feature can be decomposed into one or more sub-features. Here, the System is decomposed into four sub-features. An `all of` decomposition indicates that all sub-features are required in every feature combination whenever their parent feature is active. As used by the Workers feature, the `some of` operator implies that at least one of the sub-features has to be active if the parent is active. In addition to the `one of` operator (which requires exactly one sub-feature), the decomposition can also be given by a cardinality. Optional features are preceded by the `optional` keyword, indicating that the feature may or may not be part of a valid feature combination, regardless of the decomposition operator. PROFEAT has built-in support for *multi-features* [15], i.e., features that can appear more than once in a feature combination. The number of instances is given

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

in brackets behind the feature name. In the producer-consumer example, the Workers feature is decomposed into three distinct copies of the Worker feature. It is important to note that the decomposition operator ranges over the feature instances. Thus, the `some` operator could be replaced by cardinality `[1..3]` in the above listing. Multi-features can be marked `optional` as well. Then, each individual copy of the multi-feature is an optional feature. Besides multi-features, the ProFeat language supports non-Boolean features in the form of numeric *feature attributes* [8]. In the example shown above, the Worker has the attribute `speed` which can take any integer value from 1 to 5. Access to feature attributes is possible regardless of whether the corresponding feature is active or not. The combination of multi-features and feature attributes enables a compact representation of complex product lines.

The introduction of multi-features necessitates the distinction between features and feature instances. In ProFeat, each feature instance is uniquely identified by its *fully qualified name*. Sub-feature instances as well as feature attributes are addressed using the familiar dot-notation. Instances of multi-features are referred to by an array-like syntax. For example, the fully qualified name of the second worker's speed attribute is `root.Workers.Worker[1].speed`. As long as the qualified name is unambiguous, the prefix can be omitted. For instance, the name `Worker[1].speed` is valid as well.

A `feature` block may also contain cross-tree constraints over feature instances and feature attributes. In our example, the first constraint given in the root feature expresses that the first two Worker instances must be active whenever the Fast feature is active. The second constraint limits the accumulated speed of the first two workers. A constraint can be preceded by the `initial` keyword, which only affects the initial set of valid feature combinations. Obviously, this distinction is only relevant for dynamic product lines.

Behavior of Features. In a ProFeat model, the declarative feature model is strictly separated from the operational behavior of features. A feature may be "implemented" by one or more *feature modules*, which are listed after the `modules` keyword inside the `feature` block. In our running example, the Worker feature is implemented by the `Worker_impl` module. The listing on the left shows the feature module and the extended feature declaration of the Worker feature. For the definition of feature modules, we use an extension of Prism's guarded command language. Besides Boolean or integer variables defined in feature modules as in Prism, ProFeat supports (one dimensional) arrays. A set of *commands* defines the behavior of the feature module, having the form *guard* → *stochastic-update*. If the guard evaluates to *true*, the module can transi-

```
1  feature Worker
2    speed:[1..5];
3    block dequeue[id];
4    modules Worker_impl;
5  endfeature
6
7  module Worker_impl
8    t:[0..max_work_size] init 0;
9    [working[id]] t>0 ->
10           (t'=max(0, t-speed));
11   [dequeue[id]] t=0 ->
12           (t'=Buffer.cell[0]);
13 endmodule
```


Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

tion (with some probability) into a successor state defined through the updates of the variables. Consider the following command of the producer feature module.

```
[enqueue] !buffer_full -> 0.1:(size'=2) + 0.9:(size'=1);
```

Here, the producer enqueues a work package of size 2 with a probability 0.1 if the buffer is not full. The guard expression may reference the local variables of other feature modules. Furthermore, the built-in `active` function can be used in guards, evaluating to *true* if applied to a feature which is currently active.

Another means for communication besides shared variables is *synchronization* between feature modules. A command can be labeled with an *action* that is placed between the square brackets preceding the guard. If two or more modules share an action, they are forced to take the labeled transitions simultaneously. However, if any of those modules cannot take the transition (because its guard is not fulfilled), then the action is *blocked*, so that none of the modules can take the transition. In our running example, a worker synchronizes with the feature module implementing the FIFO buffer over the `dequeue` action to obtain a new work package (line 11). In ProFeat, action labels can be indexed using an array-like syntax. In case of multi-features, the implicit `id` parameter evaluates to the index of the feature instance. Thus, there exist three distinct `dequeue` actions in our model. By default, feature modules of inactive features do not block actions. Thus, with regard to synchronization, deactivating a feature has the same effect as removing it entirely from the model. This is useful if the model is fully synchronous, i.e., if there is a global action that synchronizes over all transitions. However, in some cases it is crucial that an inactive feature hinders active features to synchronize with its actions. In the producer-consumer example, an inactive worker should not take a work package out of the queue (line 11). Therefore, its `dequeue` action is modeled as blocking using the `block` keyword inside the `feature` module (line 3).

Specification of Costs and Rewards. As in Prism, states and transitions in ProFeat can be augmented with costs and rewards. This allows reasoning about quantitative measures, such as energy consumption, performance and throughput. Costs and rewards are defined as a part of feature declarations using the keyword `rewards`. In the listing on the left, the energy consumption of a worker is specified depending on its processing speed.

```
1 feature Worker
2   speed:[1..5];
3   rewards "energy"
4     [working[id]]
5         true : pow(2,speed);
6   endrewards
7 endfeature
```

Feature Controller. In a ProFeat model, the feature combination is not necessarily static, but may also change over time. The *feature controller* is a special module that defines the rules for the dynamic activation and deactivation of features, whose declaration we exemplify within our producer-consumer model:

```
controller
  [] buffer_full & !active(Worker[2]) -> activate(Worker[2]);
  [] buffer_low & active(Worker[2]) -> deactivate(Worker[2]);
```


Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17


```
endcontroller
```

Essentially, a `controller` is a module, which can modify feature combinations using the `activate` and `deactivate` updates. In the controller shown above, the third worker is activated to speed up processing whenever the buffer is full. Once the buffer is nearly empty, the worker is deactivated. The definition of a feature controller is optional. If no controller is given, the defined product line is assumed to be static. Feature modules can synchronize with the controller over the `activate` and `deactivate` actions, which enables them to react or even block the activation or deactivation of their corresponding feature. For instance, by adding the following line to the `Worker_impl` module, the deactivation of the worker is blocked as long as it is still processing a work package:

```
[deactivate] t=0 -> true;
```

Templates and Metaprogramming. ProFeat also provides constructs commonly found in template engines but not included in Prism's input language. Commands can be generated at translation time by using `for` loops. Furthermore, `feature` blocks as well as feature modules can be parametrized, which, in turn, allows for parametrization of guards, probabilities and costs/rewards. These feature and module templates are instantiated by referencing them in a decomposition or using the `modules` keyword, respectively. Consider the following excerpt of the feature module implementing the FIFO buffer:

```
1 module fifo(capacity)
2   cell : array [0..capacity-1] of [-1..max_work_size] init -1;
3   for w in [0..2]
4     [dequeue[w]] cell[0] != -1 ->
5       (cell[capacity-1]'=-1) &
6       for i in [0..capacity-2] (cell[i]'=cell[i+1]) endfor;
7   endfor
8   ...
9 endmodule
```

The module is parametrized over the capacity of the buffer (line 1). The `for` loop stretching from line 3 to 7 generates a `dequeue` labeled command for each worker. The inner loop (lines 6) shifts the buffer entries to remove the first element from the buffer.

2.2 Parametrization

While ProFeat provides special support for feature-oriented modeling, families can also be formed by ranging over system parameters. In our running example, such a parameter might be the FIFO buffer size. Parameters are declared in a `family` block, as shown on the left. Similar to feature attributes, system parameters can be constrained as well. Furthermore, a family declaration can be combined with a feature

```
1 family
2   buffer_size:[1..8];
3   initial constraint
4         buffer_size != 5;
5 endfamily
```


Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

model, resulting in a family that is both defined by system parameters and all valid initial feature combinations. To declare subsets of valid feature combinations as initial ones, ProFeat provides the `initial constraint` keyword (see line 3 of the listing). Valid feature combinations not fulfilling the listed constraints are still possible during runtime by dynamic feature switches.

System parameters can be used anywhere in the model description, including guards, probabilities and costs/rewards. In contrast to feature attributes, system parameters are constant for each instance of a family. This has an important consequence: parameters can be used to specify the range of variables, the size of arrays, the range of `for` loops and even the number of multi-feature instances. Thus, system parameters can directly influence the structure of the system.

3 Implementation

We have implemented a software tool[2] that translates a ProFeat model into the input language of the model checker Prism. This translation-based approach enables the use of existing machinery for the verification and quantitative analysis of ProFeat models. The ProFeat tool furthermore supports the translation of queries into Prism's properties file format. Thus, queries can be formulated in the extended syntax of ProFeat and allow reasoning about feature-specific properties. In this section, we provide a semantics for the ProFeat language and highlight notable steps of the translation process. The compositional modeling framework for probabilistic dynamic product lines by [21] provides a translation of feature modules under a feature controller into the input language of Prism, naturally mapping feature composition to the parallel composition of Prism. Thus, the semantics of the behavioral model of ProFeat is defined in terms of the Prism language semantics. The semantics of ProFeat's feature modeling formalism is given by the semantics of TVL [8] extended with multi-features as described in [15].

3.1 Translation of Feature-specific Constructs

In ProFeat, the access to the feature combination is provided through the use of the `active` function and the `activate` and `deactivate` updates of the feature controller. We encode feature combinations by a set of integer variables with range `[0..1]`, which simplifies the handling of feature cardinalities (compared to a Boolean encoding). Instead of creating one variable per feature instance, the tool generates one variable per *atomic set* to reduce the number of variables: An atomic set is a set of features that can be treated as a unit as they never appear separately in a feature combination [38]. Given this representation, the translation of the `active` function is simple: The call to `active` is replaced by a check testing whether the atomic-set variable evaluates to 1. Analogously,

[2] For the Haskell source code of the tool, we refer to `https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/FASE16`

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

the `activate` and `deactivate` updates assign a 0 or a 1 to the corresponding variable, respectively. However, the feature controller cannot change the feature combination arbitrarily: As an update has to yield a valid feature combination, the translation has to add a guard to commands containing atomic-set variable updates. This guard is synthesized from the feature model and evaluates to *false* if the transition described by the command would result in an invalid feature combination.

Another aspect of the translation concerns the synchronization between the feature controller and the feature modules in case of feature activation and deactivation. Implicitly, an $activate_f$ action and a $deactivate_f$ action is created for each feature instance f. In PRISM, commands can only be labeled with a single action. However, an update may activate or deactivate multiple feature instances at once, thus requiring multiple action labels per command for synchronization. To circumvent this restriction of the PRISM language, the set of action labels is merged into a single action label. This solution requires special care in the translation of feature modules. Let us assume a command C labeled with the action $activate_f$. Then, we collect the action labels of all feature-controller commands that activate the feature instance f. Finally, we create a copy of the command C for each collected action label. This translation realizes the intended synchronization between the feature controller and the feature modules, even in the case of multiple simultaneous feature activations and deactivations.

Lastly, the translation must ensure that feature modules of inactive features do not block actions, i.e., deactivating a feature should have the same effect as removing the corresponding feature modules from the model. To achieve this behavior, we take the following approach. Suppose the feature module M implements the feature instance f. Then, for each command in M that has the form $[\alpha]$ *guard* $\rightarrow$ *update*, a command $[\alpha]\ \neg\texttt{active}(f) \rightarrow$ *true* is generated. Thus, if the feature instance f is not active, the translated module does not block the action α. However, this command is not generated if the user explicitly requests the blocking of action α by using the `block` keyword in the feature declaration.

3.2 All-in-One and One-by-One Translation

Essentially, there are two different approaches for the analysis of a family of systems described by some PROFEAT model: The *one-by-one* and the *all-in-one* approach. Within a one-by-one approach, each member of the family is analyzed separately. Differently, within an all-in-one approach, the whole family is encoded into a single PRISM model and analyzed in a single run. The result of the all-in-one analysis is then interpreted for each member of the family, providing results as the members would have been analyzed separately. An all-in-one approach can potentially exploit the similarities between the family instances and speed up the analysis, but may require additional memory. However, a big advantage of the one-by-one approach is that it can be easily parallelized. As we illustrate in our case studies (see Section 4), it depends on the model as well as the time and memory constraints which approach is appropriate. For this reason, PROFEAT supports both, an all-in-one and a one-by-one translation of the family model.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

Switching from one analysis approach to the other requires no adjustments to the model.

In case of one-by-one translation, ProFeat generates a Prism model for every instance of the family. That is, for each valid valuation of the system parameters and for each initial feature combination, the system parameters are replaced by constants. If no `family` block is given, then one model for each valid initial feature combination is generated. The all-in-one translation generates a single Prism model with multiple initial states, one for each instance of the family. However, there is a technical difficulty in the translation into an all-in-one model: Array sizes, numbers of multi-features and variable bounds can be defined in terms of system parameters. Hence, these parameters might depend on the initial state and thus are not known at translation time. Therefore, ProFeat instantiates these parametrized structures with their maximal size.

4 Experimental Studies

As ProFeat follows a translational approach, all-in-one and one-in-one analyses can be carried out using the same model-checking tool Prism, allowing for a conceptual comparison of both approaches. Besides a sequential one-by-one analysis as usually performed within product-line verification (see, e.g., [4]), we also provide results for analyzing the models generated by the one-by-one translation in parallel. Clearly, under the quite unrealistic assumption that lots of CPU cores (which allow for parallelization) and enough memory is provided, a parallel execution is likely to outperform an all-in-one approach. For our experiments we used a Linux machine with two 8-core Intel Xeon E5-2680 CPUs running at 2.7 GHz and equipped with 384 GBytes of RAM, hyper-threading enabled. Thus, we restricted ourselves to an execution of 32 analyses in parallel.

4.1 The Producer-Consumer Example

In the base model of the producer-consumer example, as considered already in previous sections, the controller can activate or deactivate workers in the workers pool, increase or decrease the size of the buffer, and increase or decrease the processing speed of individual workers. For realizing fairness among regular actions and controller actions, we introduced an additional progress module. When considering expected costs, the goal will be to finish a certain number of jobs. For this we enriched the model with a counter. In this section, we consider three variants of the base model and corresponding analysis queries:

Best Buffer. A static product line which parametrizes over the buffer size. Here, we ask for the buffer size for which minimal expected storage costs arise until a certain number of jobs are processed.

Best Worker. This family parametrizes over all possible combinations of workers. Within this family model, we ask for the combination of workers where the minimal expected energy is required to finish a given number of jobs.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

Distributions. Here, we consider different workload distributions as parameter space of the model. The goal is to compute the distribution where the expected energy required to finish a certain number of jobs is minimal.

Figure 1a shows the number of MTBDD nodes for representing the three model variants depending on the family parameter. Within all variants, the number of nodes in the all-in-one model is significantly smaller that the sum of the MTBDD nodes for the separate models, indicating shared behaviors between the family members. We evaluated the quantitative queries stated above using both, the MTBDD and the SPARSE engine of PRISM. In general, the SPARSE engine turned out to perform slightly better than the MTBDD engine, especially within expectation queries. The results are illustrated in Figure 1b–d.

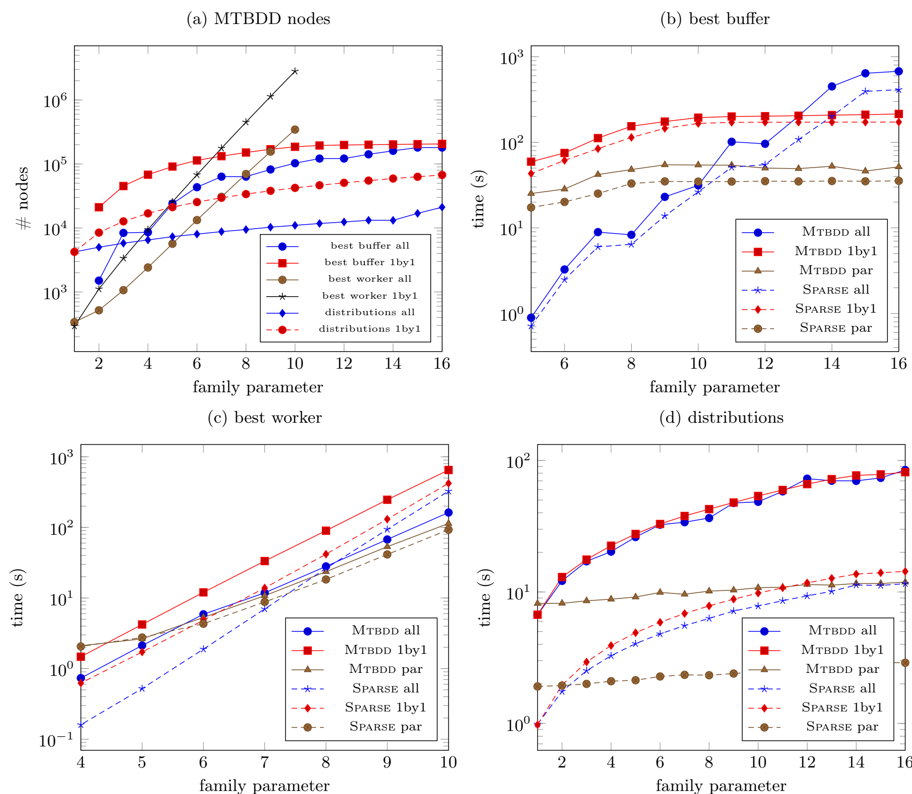


Fig. 1. Number of MTBDD nodes for the producer-consumer models (a), Analysis times of the variants best buffer (b), best worker (c) and distributions (d)

In some cases, where the number of instances is exponential in the family parameter (cf. Figure 1c – Best Worker), the all-in-one analysis approach outperforms the one-by-one approach and can even keep up with the parallel computation. In other cases (cf. Figure 1b – Best Buffer), the all-in-one approach

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

was only superior up to a system size of 14. For the third model variant (cf. Figure 1d – Distributions), the all-in-one and one-by-one approaches asymptotically displayed similar performance. Overall, there is a no clear trend on which approach is favorable, the one-by-one or the all-in-one analysis.

4.2 Feature-Aware Case Studies

The development of ProFeat has been first and foremost motivated by several studies from the domain of feature-oriented systems such as product lines, where all-in-one analysis approaches turned out to outperform the traditional one-by-one analysis approach. In this section, we demonstrate how (probabilistic) versions of classical product lines can be modeled and analyzed with ProFeat.

Body Sensor Network Product Line. A Body Sensor Network (BSN) system is a network of connected sensors sending measurements to a central entity which evaluates the data and identifies health critical situations. In [37], a BSN product line with features for several sensors has been introduced. The approach presented in [37] follows the ideas by [23] towards parameterized DTMC models: For each feature, a Boolean parameter f is 1 if the feature is active and 0 otherwise. A factor p is multiplied to the probability of every transition, where $p{=}f$ in case the feature enables the transition and $p = 1{-}f$ otherwise. Parametric model checkers are then used to compute a single formula which for each feature combination evaluates to the probability of reaching a successful configuration, i.e., the reliability of the BSN. The authors of [37] report that the parametric approach using Param can be seven times faster, a novel symbolic bounded-search approach can be eleven times faster, and a handcrafted (model dependent) compositional parametric approach can even be 100 times faster than a Prism-based one-by-one analysis. For obtaining the results, three different model-checking tools have been used. Furthermore, special tailored scripts were required to perform the one-by-one analysis and to evaluate the formulas returned by the parametric model checkers. With ProFeat the feature model of the BSN product line can be directly incorporated into the parametric model specified by [37], as ProFeat's representation of features as Boolean parameters is compatible with the approach by [23]. Thus, ProFeat allows for an all-in-one approach on the same model as of [37] and simplifies the comparison to one-by-one analysis also concerning different model-checking engines such as the explicit or symbolic engines of Prism.

In the first line of Table 1, we show the results of our experiments for computing the same reliability probability as in [37]. The all-in-one approach turns out to be $\approx$100 times faster than the one-by-one approach, independent of the chosen engine. Hence, ProFeat directly enables a speed up of the analysis time in the same magnitude as handcrafted decomposition optimizations by [37].

Elevator Product Line. A classical (non-probabilistic) product line considers an elevator system, introduced by [36] for checking feature interactions. It has been then considered in several case studies issuing family-based product-line verification (see, e.g., [4,15]). An elevator system is modeled by a cabin which

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

can transport persons to floors of a building. The persons first have to push a button at the floor and then in the cabin for calling the elevator and defining a direction where to ride, respectively. In its basic version [36], the product line comprises 32 products built by five features, not changeable after deployment. We extend this product line in various aspects. First, we resolve some non-deterministic choices by probabilities when appropriate, e.g., modeling the request rate of a person and introducing a probability of failure. Second, we add a service feature, which enables to call technical staff repairing the elevator or change feature combinations. As a consequence, our elevator system is a dynamic product line where features can be changed during runtime. Third, we modeled dynamic feature changes as non-deterministic choices in the feature controller. This yields an MDP model for which a strategy-synthesis problem can be considered: Compute best- and worst-case strategies on how to activate or deactivate features to reach certain goals [21]. We deal with a simple instance of the elevator which can transport one person and where at most two persons act in the system. Our product lines have 64 feature combinations each, parametrized over the number of floors (2-4) in the building. We finally consider the family of the three product lines, containing 192 single instances of the elevator system. We asked for the minimal probability that if the cabin is on the ground floor and the top floor is requested, the probability to serve the top floor within the next three steps is greater than 0.99. Our analysis results are depicted in Table 1, where especially for larger instances the MTBDD all-in-one analysis outperforms other approaches and engines. Notice that the number of MTBDD nodes of the family model containing all three elevator product lines (cf. the row above the double rule) is greater than the sum of nodes of the family models for each product line. Possibly, other MTBDD variable orderings, e.g., provided by methods presented in [31], could yield smaller model representations and faster all-in-one analyses.

4.3 Benchmark Suite Examples

We used ProFeat also to model and analyze some examples taken from the Prism benchmark suite [33] and the probabilistic locking protocol PWCS [6] to investigate whether also standard parametrized models can profit from an all-in-one analysis. In the PWCS model, we consider two family parameters: The number of writers that intend to access a shared object (1) and the number of replicas for a given object (2). When providing ProFeat code for the examples based on the existing models, the scripting and parameterization of ProFeat yield a more compact model representation and required only mild modifications. Each row in the lower part of Table 1 stands for the evaluation of a query, which cover minimal and maximal expected values as well as probabilities for bounded and unbounded reachability. The Hybrid engine of Prism does not yet support the computation of expectations. A reduction of the MTBDD size was only achieved for the self-stabilization protocol. In all other cases, the size of the family model was in the order of the sum of the separate models. The one-by-one approaches outperform the all-in-one approaches in almost all cases, even for the self-stabilization protocol.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

Table 1. Analysis times (in seconds) of feature and benchmark suite models

Model	MTBDD nodes		MTBDD			HYBRID			SPARSE		
	family	separate	all	1by1	par	all	1by1	par	all	1by1	par
BSN	5 651	111 507	1	129	25	1	128	25	1	128	25
Elevator (2 floors)	42 254	1 329 204	1	65	7	2	49	7	1	45	7
Elevator (3 floors)	151 274	4 924 349	4	223	11	98	2 531	96	7	286	18
Elevator (4 floors)	420 448	13 519 274	15	910	32	2 601	54 262	1 952	56	2 008	83
Elevator (2-4 floors)	779 569	19 772 827	29	1 199	49	5 089	56 843	2 052	74	2 339	106
CSMA	633 997	634 076		timeout			not supported		1 236	1 251	1 220
(2–4 processes)	“	“		timeout		3 660	3 577	3 384	1 078	1 013	954
Stabilization	4 340	10 662	2 036	1 643	932	251	37	22	129	33	20
(3–21 processes)	“	“	$\ll 1$	1	2		not supported		122	24	15
	“	“		timeout			not supported		2 629	476	269
	“	“	13	10	7	12	10	6	12	10	7
	“	“	13	10	7	13	10	7	13	10	6
Philosophers (3–12)	82 995	82 689	9 056	6 212	3 945	9 722	5 949	4 009		out of memory	
PWCS (3 replicas,	134 236	134 190	49	26	15	232	165	130	314	271	220
1–9 writers)	“	“	6 564	2 247	960		not supported		5 473	1 544	1 230
PWCS (3 writers,	955 505	958 033	752	2 279	1 628	968	348	306	738	2 209	1 265
1–7 replicas)	“	“		timeout			not supported		1 221	3 857	2 735

5 Conclusions

We presented the language ProFeat for family-based modeling and analysis of probabilistic systems. To the best of our knowledge, ProFeat is the first modeling language for probabilistic dynamic product lines with tool support for an all-in-one and one-by-one analysis without employing templates, scripting or different model descriptions. Whereas for experiments on product-line inspired case studies an all-in-one approach turns out to be usually faster than a one-by-one approach, this cannot be generalized to arbitrary families, e.g., when only a few common behaviors exist within the family members. There are various directions for further work, e.g., establishing an all-in-many approach clustering families and thus mixing both approaches. Symmetry reductions on the model could also speed up an all-in-one analysis, especially within multi-features.

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 *ICMT'09*, volume 5563 of *LNCS*, pages 4–19. Springer, 2009.
3. S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.
4. S. Apel, A. von Rhein, P. Wendler, A. Groesslinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proc. of the 2013 Int. Conference on Software Engineering*, ICSE '13, pages 482–491. IEEE, 2013.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

5. S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Int. Conference on Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011.
6. C. Baier, B. Engel, S. Klüppelholz, S. Märcker, H. Tews, and M. Völp. A probabilistic quantitative analysis of probabilistic-write/copy-select. In *Proc. of the 5th NASA Formal Methods Symposium (NFM)*, LNCS, pages 307–321. Springer, 2013.
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. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76(12):1130–1143, 2011.
9. A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with SNIP. *STTT*, 14(5):589–612, 2012.
10. A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming*, 80:416–439, 2014.
11. A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, 2013.
12. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2001.
13. M. Cordy, A. Classen, P. Heymans, A. Legay, and P.-Y. Schobbens. *Model Checking Adaptive Software with Featured Transition Systems*, pages 1–29. LNCS. Springer, 2013.
14. M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. ProVeLines: a product line of verifiers for software product lines. In *17th Int. Software Product Line Conference (SPLC)*, pages 141–146. ACM, 2013.
15. 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 2013 Int. Conference on Software Engineering*, ICSE '13, pages 472–481. IEEE Press, 2013.
16. 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.
17. F. Damiani and I. Schaefer. Dynamic delta-oriented programming. In *Proc. of the 15th Int. Software Product Line Conference*, SPLC '11. ACM, 2011.
18. C. Daws. Symbolic and parametric model checking of discrete-time Markov chains. In *Theoretical Aspects of Computing - ICTAC 2004*, volume 3407 of *LNCS*, pages 280–294, 2004.
19. C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, H. Bruintjes, J.-P. Katoen, and E. Abraham. PROPhESY: a probabilistic parameter synthesis tool. In *27th Int. Conference on Computer Aided Verification (CAV)*, volume 9206 of *LNCS*, pages 214–231, 2015.
20. 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.
21. C. Dubslaff, C. Baier, and S. Klüppelholz. Probabilistic model checking for feature-oriented systems. *Transactions on Aspect-Oriented Software Development XII*, 8989:180–220, 2015.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

22. A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Aspects of Computing*, 24(2):163–186, 2012.
23. 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.
24. H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. In *PFE*, pages 435–444, 2003.
25. A. Gruler, M. Leucker, and K. D. Scheidemann. Modeling and model checking software product lines. In *10th Int. Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, LNCS, pages 113–131, 2008.
26. E. M. Hahn, H. H., B. Wachter, and L. Zhang. PARAM: A model checker for parametric Markov models. In *22nd Int. Conference on Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 660–664, 2010.
27. E. M. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric Markov models. *Software Tools and Technology Transfer*, 13(1):3–19, 2011.
28. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *TACAS'06*, volume 3920 of *LNCS*, pages 441–444, 2006.
29. 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, 1990.
30. S. Katz. A superimposition control construct for distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(2):337–356, 1993.
31. J. Klein, C. Baier, P. Chrszon, M. Daum, C. Dubslaff, S. Klüppelholz, S. Märcker, and D. Müller. Advances in symbolic probabilistic model checking with PRISM. In *Proc. of the 22th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS. Springer, 2016. to appear.
32. M. Kowal, I. Schaefer, and M. Tribastone. Family-based performance analysis of variant-rich software systems. In *Fundamental Approaches to Software Engineering*, volume 8411 of *LNCS*, pages 94–108, 2014.
33. M. Z. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *Proc. Quantitative Evaluation of Systems (QEST'12)*, pages 203–204. IEEE, 2012. `https://github.com/prismmodelchecker/prism-benchmarks/`.
34. K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *24th IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 269–280. IEEE, 2009.
35. J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, and G. Saake. An overview on analysis tools for software product lines. In *18th Int. Software Product Lines Conference (SPLC)*, pages 94–101. ACM, 2014.
36. M. Plath and M. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84, 2001.
37. G. N. Rodrigues, V. Alves, V. Nunes, A. Lanna, M. Cordy, P.-Y. Schobbens, A. M. Sharifloo, and A. Legay. Modeling and verification for probabilistic properties in software product lines. In *High Assurance Systems Engineering (HASE)*, pages 173–180. IEEE, 2015.
38. S. Segura. Automated analysis of feature models using atomic sets. In *SPLC (2)*, pages 201–207, 2008.
39. M. H. ter Beek, A. Legay, A. Lluch-Lafuente, and A. Vandin. Statistical analysis of probabilistic models of software product lines with quantitative constraints. In *19th Int. Conference on Software Product Line (SPLC)*, pages 11–15. ACM, 2015.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "FASE 2016. Eindhoven 2016", S. 287-304. ISBN: 978-3-662-49665-7
https://doi.org/10.1007/978-3-662-49665-7_17

40. T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden