




ProFeat: feature-oriented engineering for family-based probabilistic model checking

Philipp Chrszon¹ , Clemens Dubslaff¹, Sascha Klüppelholz¹ and Christel Baier¹

¹ Faculty of Computer Science, Technische Universität Dresden, Dresden, Germany

Abstract. The concept of features provides an elegant way to specify families of systems. Given a base system, features encapsulate additional functionalities that can be activated or deactivated to enhance or restrict the base system's behaviors. Features can also facilitate the analysis of families of systems by exploiting commonalities of the family members and performing 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. Most prominent, the concept of features has been successfully applied to describe and analyze (software) product lines. We present the tool PROFEAT that supports the feature-oriented engineering process for stochastic systems by probabilistic model checking. To describe families of stochastic systems, PROFEAT extends models for the prominent probabilistic model checker PRISM by feature-oriented concepts, including support for probabilistic product lines with dynamic feature switches, multi-features and feature attributes. PROFEAT provides a compact symbolic representation of the analysis results for each family member obtained by PRISM to support, e.g., model repair or refinement during feature-oriented development. By means of several case studies we show how PROFEAT eases family-based quantitative analysis and compare one-by-one and all-in-one analysis approaches.

Keywords: Feature-oriented systems, Probabilistic model checking, Software product line analysis

1. Introduction

Feature orientation is a popular paradigm for the development of customizable software systems (see, e.g., [KCH⁺90, AK09, BSRC10]). In general, features can be understood as functionalities changing the behaviors of a core software system and thus provide an elegant way to specify families of systems: every member of the family comprises the core system and a combination of features. The concept of features is widely used, e.g., for optimizing software depending on the platform the software is rolled out for evaluated during compile time depending on the system the software is compiled for. Another example for a feature-oriented system is the support of component-based software development, where parts of the developed software are encapsulated into easily replaceable components, e.g., components with similar functionalities but different characteristics concerning performance or energy consumption.

The most prominent application of feature-oriented formalisms are software product lines [CN01]. A software product line is a family of software systems, where each member is a variant of the software developed to satisfy the needs of different customers, e.g., by providing a free but ad-financed version or a professional version of the software including functionalities for enterprises. Usually, software product lines are developed using the following approach: First, the application domain is analyzed by identifying and isolating features those combinations yield a software variant. The so-called *valid feature combinations* are then modeled, e.g., through *feature diagrams* [KCH⁺90], which are tree-like structures. The number of valid feature combinations might be exponential in the number of features and thus, feature interactions are difficult to predict by the developers. Hence, prototype implementations or abstract versions of features should be subject of formal analysis to avoid costly redesign steps. This second step of the development process checks whether basic requirements on the software products are met and if not, triggers adaptations to the feature model. When all requirements analyzed are satisfied by each of the valid feature combinations, the actual feature-aware implementation of the software takes place, and software variants are released by selecting a particular combination. Clearly, although first and foremost applied in software product line engineering, this development process is applicable to any feature-oriented design of systems.

This paper considers the formal-analysis aspect of the described engineering process with *non-functional requirements*, i.e., requirements that involve quantitative properties such as energy consumption, monetary costs or the probability of failures. In the area of software-product-line verification, mainly *functional requirements* have been examined so far, i.e., requirements not involving quantitative aspects (see, e.g., [CHS⁺10, TAK⁺14]). The literature focused on establishing algorithms that enable all-in-one analysis approaches as they usually outperform the naive one-by-one analysis approach. Within one-by-one approach, each member of the family described by some valid feature combination is analyzed separately. Differently, an all-in-one approach analyzes a single model for the whole family from which the results for each family member is extracted from. All-in-one approaches are usually superior to the one-by-one approach when family members share lots of behaviors and a symbolic representation of the family model is chosen.

In [DBK15], a theoretical framework for the quantitative analysis of families specified with feature-oriented concepts using probabilistic model checking (see, e.g., [BK08]) has been provided. To exemplify an application of the framework, [DBK15] considered also a handcrafted model of an energy-aware server product line specified in the input language of the prominent symbolic probabilistic model checker PRISM [KNP11]. An analysis was then performed using a symbolic model-checking engine of PRISM that is based on multi-terminal binary decision diagrams (MTBDD) [CFM⁺93] for the symbolic system representation. It is well-known that the size of an MTBDD represented model is sensitive to the order of variables in the model description. Recently, automated variable reordering techniques to optimize the memory consumption and speedup the analysis time when performing probabilistic model checking has been included into PRISM [KBC⁺16].

Contribution Based on the formal framework by [DBK15], we introduce the tool PROFEAT, which gives tool support for the analysis of families of stochastic systems using probabilistic-model-checking techniques. In particular, the contribution of PROFEAT is

- (1) extending PRISM's input language by feature-oriented concepts to describe families of stochastic systems
- (2) providing an automated translation of such families to standard PRISM input language
- (3) returning a compact symbolic feature-oriented representation of PRISM's analysis results

We illustrate the benefits of using PROFEAT in several case studies, where we provide

- (4) a comparison between all-in-one and one-by-one approaches
- (5) the impact of symbolic MTBDD-based analysis including reordering techniques
- (6) the advantages of using PROFEAT not only for analyzing families of stochastic systems but also for the analysis of single systems with mode switches modeled through dynamic feature switches

Let us describe the PROFEAT contributions in more detail. (1) Operational behaviors of system families are described with a feature-aware extension of the input language of the probabilistic model checker PRISM [KNP11], implementing the formal framework of [DBK15]. To specify valid feature combinations and to describe the structure of the family of systems, we rely on a feature-model formalism similar to the Textual Variability Language (TVL) [CBH11]. PROFEAT also allows for (numerical) feature attributes and multi-features [CHE05, CBH11, CSHL13], i.e., features that can appear more than once in a feature combination.

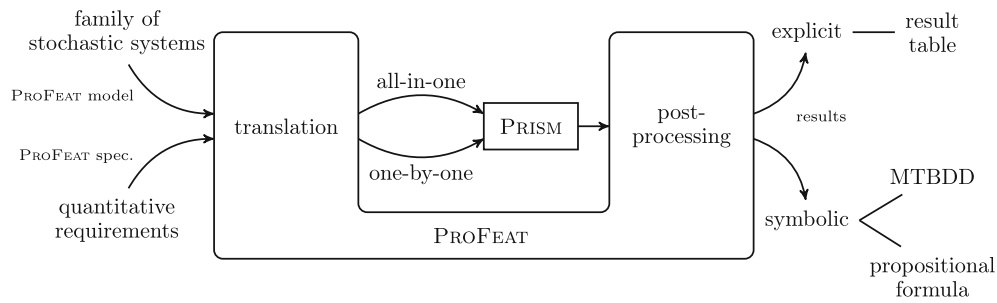


Fig. 1. Workflow overview

Following [DBK15], the behaviors of each feature are specified in *feature modules* described by discrete- or continuous-time Markov chains or Markov decision processes. The support for dynamic feature switches during runtime is maintained by a module called *feature controller*, which synchronizes with the feature modules when activating and deactivating features. The dynamics of the feature controller and its interactions with the feature modules are crucial to model dynamic product lines [GH03, DMFM10, DS11, CCH⁺13a].

(2) PROFEAT follows a translational approach towards a quantitative analysis of system families, illustrated in Fig. 1. In a preprocessing step, PROFEAT family models and quantitative requirements are translated into the pure input language of PRISM. Then, the actual analysis is performed by invoking PRISM on the translated models and requirements, returning results through PRISM logs. The returned logs are post-processed and PROFEAT represents the results for each family member in the feature-aware context. Note that using this approach, PROFEAT can benefit from the power of the state-of-the-art probabilistic model checker PRISM and its (possibly future) extensions. (3) Since the number of family members can be exponential in the number of features, just providing enumerated results for each member can be cumbersome in the feature-aware engineering process. Thus, PROFEAT provides compact symbolic representations of the results using MTBDDs or propositional formulas over features.

We evaluate the approach of using PROFEAT in a feature-oriented engineering process by several case studies. (4) PROFEAT seamlessly integrates both one-by-one and all-in-one analysis without requiring adaptations to the family model and specification (see (1)). This is obtained through different translation and post-processing mechanisms and fits into the feature-oriented analysis approach: When there are many commonalities within family members, a symbolic model-checking engine might be chosen in PRISM to exploit the commonalities within an all-in-one approach and speed up the analysis. (5) Since the symbolic MTBDD representation of the family is sensitive to the chosen variable order, we investigate the application of automated reordering techniques recently introduced in PRISM [KBC⁺16]. PROFEAT fully supports the maintenance of the variable reordering, such that no adaptations to the family model is required to enable or disable reordering techniques.

(6) Besides useful for the quantitative analysis of families of stochastic systems, PROFEAT can also be used to describe single dynamic stochastic systems where features model operational modes of the system. Feature switches then model the dynamic mode switches during runtime of the system. To underpin this statement, we carry out a case study issuing the feature-oriented engineering process of an energy-aware adaptive network system.

The source code of PROFEAT can be obtained at <https://www.tcs.inf.tu-dresden.de/ALGI/PUB/ProFeat>. This paper is an extension of the conference version [CDKB16], where PROFEAT has been presented first. There, the symbolic representation of analysis results was not included in PROFEAT yet. Furthermore, the seamless integration of automated variable reordering techniques for symbolic analysis engines [KBC⁺16] has been newly integrated. Additional case studies and more detail on the feature-oriented engineering approach with PROFEAT is provided. In contrast to our previous work [DBK15], where we introduced the formal framework PROFEAT is relying on, the PROFEAT approach provides an elegant way to specify feature modules and the feature controller and to automatically generate corresponding PRISM code, rather than requiring a handcrafted translation from formal descriptions of Markov decision processes to PRISM as done in [DBK15].

```

module Producer
  work_size : [1..2];

  [enqueue] !buffer_full -> 0.7: (work_size' = 1) + 0.3: (work_size' = 2);
endmodule

```

Lst. 1. A module for a simple producer in the PRISM language

```

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

```

Lst. 2. Excerpt of the feature model for the producer–consumer product line

Related work Several techniques for the analysis of (non-probabilistic) feature-oriented models and software product lines using testing, type checking, static analysis, theorem proving or model checking have been already proposed and implemented in tools (see, e.g., [TAK⁺14] for an overview). Commonly, feature-aware analysis techniques try to avoid the combinatorial blowup in the number of features arising when analyzing all members of a feature-oriented system family separately. Symbolic representations of the family and appropriate algorithms to reason about the whole family at once turned out to be very successful. PROFEAT complements most of the existing approaches by not tailoring feature-aware analysis algorithms but fully relying on standard model-checking tools and exploiting their symbolic representations of the systems. Here, we briefly summarize tool support for the analysis of feature-oriented systems using model checking. An overview about theoretical considerations towards these tools and prototype implementations in the literature can be found in [DBK15], where the formal framework PROFEAT is based on has been developed. For the automatic detection of feature interactions, Plath and Ryan [PR01] introduced a feature-oriented extension of the input language of the model checker SMV and Apel et al. [ASW⁺11] presented the tool SPLVerifier. FeatureIDE [TKB⁺14] 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. Lauenroth et al. [LPT09] 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 [CCS⁺13] or a variant of dynamic product lines [CCH⁺13a]. The SNIP tool [CCH⁺12, CCS⁺13, CSHL13] 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 [CCH⁺13b] provides several extensions, including verification techniques for reachability properties with real-time constraints.

Similar to the approach by [DKB14] for probabilistic product lines, [DABW15] considers (non-probabilistic) product-line verification of linear-time requirements and presents an approach that avoids specialized feature-aware model-checking algorithms. For branching-time temporal-logic specifications, [CCH⁺13a, CCH⁺14] proposed a symbolic model-checking approach for (adaptive) FTS. We are not aware of an implementation of the approach of [CCH⁺13a]. Relying on the formal framework of [AtBGF11, tBFGM16], where product line families are modeled through modal transition systems, the tool VMC has been presented in [tBMS12] that allows for the verification of branching-time requirements.

In [CCH⁺14], an all-in-one analysis based on the feature-oriented extension of the SMV input language by [PR01] 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 [Kat93, AJTK09, AH10], rather than parallel composition of feature behaviors [DBK15].

None of the approaches mentioned above deals with probabilistic behaviors. To the best of our knowledge, there is no other tool that provides family-based probabilistic model checking for families using a compositional framework based on Markov decision processes. An overview about quantitative analysis for probabilistic product lines can be found in [LP17]. The benefits of probabilistic model checking for the analysis of adaptive software has been also drawn by Filieri et al. [FGT12], where adaptive Markov chains models were investigated within a continuous verification approach. There, the focus was not on a family-based analysis but on a holistic approach towards supporting required adaptations to environmental changes. The work on model-checking algorithms for parametric Markov chains [Daw04, HHZ11] and tool support in the model checkers PARAM [HHWZ10] (which has been reimplemented and integrated in PRISM) and PROPHECY [DJJ⁺15] 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 [GS13] and the recent work by Rodrigues et al. [RAN⁺15] illustrate the potential of parametric probabilistic model-checking techniques for the analysis of product lines. PROFEAT can handle probability parameters as well and translate them to PRISM code, such that models used in [GS13, RAN⁺15] can also be subject to a family-based analysis using PROFEAT. However, the analysis approach with PROFEAT is different in the sense that the probability parameters are encoded as model variables, using the standard engines of PRISM rather than its parametric one. An approach towards a family-based performance analysis of dynamic probabilistic product lines modeled by a variant of UML activity diagrams has been presented by [KST14]. Dynamic changes on the performance-annotated activity diagrams (PAADs) are expressed using the delta-modeling approach [Sch10], which complements the compositional approach by [DBK15]. Our approach covers the class of models considered, as the semantics of PAADs is given as a very restricted class of CTMCs. Thus, PROFEAT could also be applied to analyze PAADs. However, since [KST14] uses specialized algorithms to avoid explicit encodings into the state space that PROFEAT does not provide, only small instances are feasible to analyze with PROFEAT. The recent work by Beek et al. [tBLLV15] presents a framework for the analysis of software product lines using statistical model checking. They focus on Markov chain models rather than Markov decision processes and present a process-algebraic characterization of feature-oriented concepts.

Outline Section 2 presents the main principles of the PROFEAT language to describe families of stochastic systems. The standard feature-oriented engineering process with PROFEAT is described in Sect. 3. Details on the implementation of PROFEAT will be given in Sect. 4. Section 5 reports on experimental studies within PROFEAT. A brief conclusion is provided in Sect. 6.

2. Describing families of stochastic systems: the PROFEAT language

The PROFEAT language can be seen as an extension of the input language of the probabilistic model checker PRISM [KNP11] that is used for modeling and analysis of parallel probabilistic systems. PRISM and hence PROFEAT supports various types of finite system models such as discrete- or continuous-time Markov chains (DTMCs, or CTMCs, respectively) and Markov decision processes (MDPs).

In this section, we present the additional language concepts introduced for the analysis of families of stochastic systems. In Sect. 2.1 we will first give a brief summary of the core PRISM language. Sect. 2.2 introduces the meta-programming language constructs as a first class of extensions provided by PROFEAT. These meta-programming constructs, such as parameters, arrays, and loops facilitate the feature-based modeling of families of stochastic systems, but are useful in general as well. The main language concepts for the compositional description of system families (cf. [DBK15], Section 4) are introduced in Sect. 2.3.

In what follows, we describe the PROFEAT language using a simple producer–consumer example. The system consists of a single producer that enqueues jobs with probabilistic workload sizes into a FIFO buffer handed to one or more workers. The workers can only process one package at a time each. The time it takes for a worker to process a work package is determined by the package size and the processing speed of the individual worker. 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. The PRISM language

A model in the input language of PRISM [KNP11] consists of one or more reactive *modules* [AH99] that can interact with each other. A set of variables defines the local state space of a module and the local variables of all modules constitute the global state space of the model. PRISM supports two types of variables: bounded integers and Boolean variables. The behavior of a module, i.e., the possible transitions between its states, is given by a set of *guarded commands* [Dij75]. A command has the form:

$$[\text{action-name}] \text{ guard} \rightarrow p_1 : \text{update}_1 + p_2 : \text{update}_2 + \dots + p_n : \text{update}_n$$

The guard is an expression over the variables of the model (including local variables of other modules). If the guard evaluates to *true*, the module can transition into a successor state by updating its local variables. One of the updates is chosen according to the probability distribution given by expressions p_1, p_2, \dots, p_n . In every state (fulfilling the guard) the evaluations of these expressions must sum up to 1. A command can be labeled with an action name. They stand for *actions* used for synchronization between modules. 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. Listing 1 shows an example of a simple producer that enqueues a work package into the buffer whenever the enqueue action is not blocked and the buffer is not full. The work package size is determined probabilistically, where a bigger work package is produced with a probability of 0.3.

Constants can be defined using the *const* keyword. A constant definition has the form *const type = expression*. Here, the *type* is either *bool*, *int* or *double*. Additionally, the *formula* keyword can be used to introduce a shorthand name for an expression to reduce code duplication. In contrast to constants, a formula may be defined in terms of model variables.

A model can be annotated with costs and rewards¹ using reward structures in order to reason about quantitative properties, such as energy consumption, throughput and performance. Costs and rewards are real values attached to certain states or transitions of the model. A state reward definition has the form `guard : reward`, indicating that every state fulfilling the guard has the specified reward associated with it. Similarly, the definition of a transition reward is of the form `[action-name] guard : reward`. Here, all transitions originating from a state fulfilling the guard and labeled with the given action have the specified reward attached to it.

2.2. Metaprogramming language extensions

PROFEAT provides several extensions to the PRISM language that allow the user to parametrize parts of the system. This is especially important for modeling system families. Besides Boolean and integer variables, PROFEAT supports (one dimensional) arrays. Additionally, the PROFEAT language offers metaprogramming constructs commonly found in template languages. PROFEAT allows the parametrization of PRISM’s formula definitions to define function-like macros. Furthermore, PROFEAT supports for loops. Loops can be used to generate sequences of commands, probability distributions, variable updates and expressions. This is especially useful for the definition of system families where the instances differ in their structure, e.g., buffer sizes or the number of multi-feature instances. In the following example, a parametrized formula that stands for the expression summing up the first n elements of an array is defined:

```
formula sum(arr, n) = for i in [0..n-1] { arr[i] + ... };
```

If a for loop is used in an expression (as is the case in the example above), the body of the loop must contain the placeholder `...` exactly once. Intuitively, in iteration step i this placeholder is replaced with the resulting expression of the iteration step $i + 1$.

Prior to the translation of a PROFEAT model into a PRISM model, macros defined via the `formula` keyword are expanded at all call sites. Similarly, for loops are expanded generating PROFEAT code. For example, the call `sum(buffer, 4)` of the macro defined above is expanded to:

```
buffer[0] + buffer[1] + buffer[2] + buffer[3]
```

In PROFEAT, actions can be used and indexed like arrays. The size of an action array does not need to be declared. Arrays of actions are useful in conjunction with for loops, for example:

```
for w in [0..2]
  [dequeue[w]] cell != -1 -> (cell' = -1);
endfor
```

Here, three distinct actions are generated: `dequeue[0]`, `dequeue[1]` and `dequeue[2]`.

2.3. Feature-oriented language extensions

A PROFEAT model usually comprises two distinct parts: the declaration of a family parameters and/or feature combinations, and a modular (feature-oriented) representation of the operational behavior for all building blocks of the family model. For the definition of the operational behaviors, we adopt the guarded-command input language of PRISM and extend it with the feature-specific concepts as presented in [DBK15]. First, the guards used in transition definitions within a PRISM module can now contain constraints on the activity of the declared features. Second, to specify dynamic product lines in which features can be activated and deactivated “at runtime”, an additional PRISM module called feature controller can be added that is responsible for performing the feature switches.

¹ In the following, we use the notions of costs and rewards synonymously. Conceptually they are the same, it is only the interpretation that differs (costs are regarded as negative, while rewards represent “something good”).

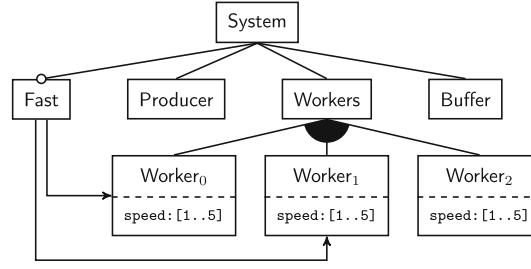


Fig. 2. Feature diagram of the producer–consumer product line

The syntax for specifying the feature controller is the same as for other PRISM modules, but allows for feature activation and deactivation in the update part of transition definitions. Additional action names that can be used for synchronization with other modules are inserted automatically when a feature controller is specified.

Feature models in PROFEAT. PROFEAT provides additional support to describe families of systems that correspond to (software) product lines. Within product lines, the standard formalism to define feature models in the software-engineering domain is provided by *feature diagrams* [KCH⁺90]. Figure 2 depicts the feature diagram for the producer–consumer example. Nodes in the diagram denote features and the root node stands for the so-called root feature. Arcs connect an inner node with its children, namely the sub-features of the respective feature. They can be of different type imposing certain constraints on the set of valid feature combinations. In the example, the root feature **System** has four sub-features. Three of them are mandatory in valid feature combinations, only the **Fast** feature is optional (indicated by the circle on the corresponding arc). The **Workers** feature has three sub-features for the workers. The arcs imply that at least one of the workers must be active in all valid feature combinations. Moreover, there can be arrows indicating additional dependencies possibly across the entire diagram. For instance, the arcs between **Fast** and **Worker₀**/**Worker₁** stand for the constraint that fast systems need to have at least two active workers. There are many other possible annotations to feature diagrams, e.g., feature attributes (as the *speed* in Fig. 2) and cardinalities indicating the minimum or maximum number of instances of the same feature that could be activated, etc.

Feature diagrams are usually conveniently described using the Textual Variability Language (TVL) [CBH11] and we adapt the concepts of TVL for describing feature models also in PROFEAT. To provide an example, Listing 2 is an excerpt of the producer–consumer feature model. Features are declared within feature blocks indicated by the keywords *feature* and *endfeature*. The root feature is a designated feature that represents the base functionality on which the product line is built. In the given example the root feature is the **System** feature, which is decomposed into the 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 whenever 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 also provides support for *multi-features* [CSHL13], i.e., features that can appear more than once in feature combinations. The number of instances is given 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 *feature attributes* [CBH11]. In the example shown above, the **Worker** features carry the attribute `speed` which can take any integer value from 1 to 5. Access to feature attributes, e.g., in guards of transition definitions within other features, is possible regardless of whether the corresponding feature is active or not. Multi-features and feature attributes in combination as supported by PROFEAT allow for compact textual descriptions of large and rather complex product lines. The support for multi-features requires the distinction between features and feature instances. In PROFEAT, each feature instance has to be uniquely identified by a *fully qualified name*. Sub-feature instances as well as feature attributes are accessible 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. Within the local scope of a feature one can refer to its attributes without a qualifier. For example, within the feature module of **Worker** (Listing 3, line 9), `speed` refers to the local attribute of the feature.

As seen already in the feature diagram, features may also contain cross-tree constraints and dependencies on feature instances and values of feature attributes. In our example, the first constraint (line 3) 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.

Operational behavior of features In a PROFEAT model, the declarative feature model is strictly separated from the operational behavior of features. A feature can be “implemented” by one or more *feature modules*, which are listed after the `modules` keyword inside the `feature` block. In the producer–consumer example, the **Worker** feature is implemented by the `Worker_impl` module. Listing 3 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 modules (as described in Sect. 2.1). Local variables of other features can be accessed using the dot-notation as described above. Access to local variables of feature modules is possible regardless of whether the corresponding feature is part of the current feature combination or not. If a feature is deactivated, its state (i.e., the local variables) remains unchanged and can still be read. To the standard language constructs already present in PRISM, PROFEAT adds the predicate `active` that can be used in any expression including guards, evaluating to `true` if the corresponding feature is active in the current system state. Furthermore, actions can either be declared as blocking or non-blocking. By default, feature modules of inactive features do not block on synchronous actions. Thus, with regard to synchronization, deactivating a feature has the same effect as removing it entirely from the model.

```

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 -> (t' = max(0, t - speed));
10  [dequeue[id]] t = 0 -> (t' = Buffer.cell[0]);
11 endmodule

```

Lst. 3. Declaration and implementation of the **Worker** feature

```

1 controller
2 [] buffer_full & !active(Worker[2]) -> activate(Worker[2]);
3 [] buffer_low & active(Worker[2]) -> deactivate(Worker[2]);
4 endcontroller

```

Lst. 4. The feature controller of the producer–consumer 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 10 in Listing 3). Therefore, its dequeue action is modeled as blocking using the `block` keyword inside the `feature` module (line 3). For the dequeue action, we use an indexed action label with the `id` as index. In case of multi-features, the implicit `id` parameter evaluates to the index of the feature instance.

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. In Listing 4, one can see an example of a feature controller suited for the producer–consumer model.

Essentially, a controller is a module (possibly with its own internal memory), 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;
```

Feature templates and module templates Both feature blocks as well as feature modules can be instantiated multiple times. Every time a feature is referenced in a decomposition, a new instance of that feature is created. Additionally, all of its associated feature modules are instantiated as well. Thus, both feature blocks and feature modules can be regarded as reusable templates. Furthermore, PROFEAT allows the parametrization of these templates, which in turn enables parametrization of guards, probabilities and costs.

```

1 feature Buffer
2 modules fifo(buffer_size);
3 endfeature
4
5 module fifo(capacity)
6 cell : array [0..capacity - 1] of [-1..max_work_size] init -1;
7 for w in [0..2]
8   [dequeue[w]] cell[0] != -1 ->
9     (cell[capacity-1] == -1) &
10    for i in [0..capacity-2] (cell[i] = cell[i+1]) endfor;
11 endfor
12 ...
13 endmodule

```

Lst. 5. A FIFO buffer implementation parameterized over the capacity

```

1 const double dist = binom(0.3, max_work_size-1);
2
3 module Producer_impl
4   work_size : [1..max_work_size] init 1;
5   [enqueue] true ->
6     for i in [1..max_work_size]
7       dist[i-1]: (work_size'=i)
8     endfor;
9 endmodule

```

Lst. 6. A producer implementation with parameterized probability distribution over the work package size

Consider the Buffer feature and its accompanying feature module shown in Listing 5. The module is parameterized over the capacity of the buffer (line 5). The actual buffer size will be determined on instantiation. In the given example, the buffer size will even be declared as system parameter ranging over a finite set of possible values. The for loop stretching from line 7 to 11 generates a dequeue labeled command for each worker. The inner loop (line 10) shifts the buffer entries to remove the first element from the buffer.

The feature module of the Producer feature (Listing 6) is parameterized over the maximal size of a work package. Because of that, the support of the probability distribution over the size of the next work package is not fixed (lines 5–7). For cases like that, PROFEAT can generate probability distributions of a given size. In the example, the work package size is binomially distributed (line 1 and line 7). Currently, PROFEAT only supports binomial distributions, however, other discrete distributions may be added in the future.

2.4. Parametrization

Families can also be formed by ranging over system parameters. In our running example, such a parameter might be the FIFO buffer size. System parameters are declared in a family block, as shown below. Similar to feature attributes, system parameters can be constrained as well.

```

1 feature Worker
2   rewards "energy"
3     active(this) & t > 0 : 1;
4     [activate] true : 5;
5   endrewards
6 endfeature

```

Lst. 7. Specification of the energy consumption of a Worker

Furthermore, a family declaration can be combined with a feature model, resulting in a family that is both defined by system parameters and all valid initial feature combinations. This is of particular importance for dynamic product lines where there is an initial feature combination that can evolve over time. To declare subsets of valid feature combinations as initial ones, PROFEAT provides the `initial constraint` keyword (see line 3 of Listing 8). 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. By an instance of a family we mean a particular system or product in the product line. This has an important consequence: parameters can, e.g., 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 state space of the system.

2.5. Property specification in PROFEAT

PROFEAT uses an extension of PRISM’s property specification language, which is based on the probabilistic computation tree logic (PCTL) [BdA95, BK98]. Reasoning about probabilities in an MDP requires the selection of an initial state and resolution of the non-deterministic choices between actions. The latter is formalized by a *scheduler*. In the following, we write $\Pr^{\min}(\varphi)$ for the minimal probability of that φ is fulfilled, ranging over all schedulers. For the specification of path properties, we use the usual temporal operators G (globally) and F (finally). We consider two example properties of the producer–consumer model. The following property states that the filling level of the buffer is almost surely below 75%, even in the worst case:

$$\Pr^{\min}(G(\text{level} < 0.75)) = 1$$

One can also ask for computing, e.g., the minimal probability that at some point **Worker**₂ is active, which could be expressed by

$$\Pr^{\min}(F(\text{“Worker}_3 \text{ is active”}))$$

As in the PRISM language, a PROFEAT model can be annotated with costs and rewards. However, reward structures are not global, instead they are defined within feature declarations. This modularizes the specification of rewards and furthermore enables the parametrization of rewards inside of parametrized feature declarations. PROFEAT extends the reward construct of PRISM by allowing the use of the *active* function to specify rewards in terms of the current feature combination. Additionally, rewards can be attached to feature switches by using the predefined *activate* and *deactivate* actions. This enables quantitative reasoning about dynamic software product lines. Listing 7 shows again the declaration of the **Worker** feature and its reward structure. In line 3, energy costs of 1 are specified for all states where the **Worker** is active and not idle. The activation of the feature (line 4) has a cost of 5.

The second property specifies that even in the worst case, the maximal expected energy consumption of the producer–consumer system does not exceed a given threshold. Here, *goal* denotes the state where all work packages have been processed and each active **Worker** is idle.

$$Ex^{\max}(\text{“accumulated energy until reaching goal”}) \leq \text{threshold}$$

```

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

```

Lst. 8. Parametrization of a model using the `family` block

Listing 9 shows the properties in the specification language of PROFEAT. In the definition of the atomic proposition *goal* (lines 8–11) we use a *for* loop to iterate over all **Worker** instances. The *active* function can be used in the same way as in a PROFEAT model as well as the access of qualified variables (such as the *t* variable of a **Worker** in line 10).

2.6. Semantics of PROFEAT models

As in the PRISM input language, every model in PROFEAT has to begin with identifying the kind of model specified, i.e., either a discrete Markov chain (DTMC, keyword *dtmc*), continuous-time Markov chain (CTMC, keyword *ctmc*), or Markov decision process (MDP, keyword *mdp*). The semantics of the PROFEAT model then is then a family of formal models of DTMCs, CTMCs or MDPs, respectively. We depart from explicitly providing a formal semantics for PROFEAT models since it can be obtained in a straight-forward way by the following three ingredients: First, PROFEAT fully relies on the framework of [DBK15], where a formal semantics for feature modules and feature controllers has been provided. Second, the semantics of PROFEAT’s feature modeling formalism is given by the semantics of TVL [CBH11] extended with multi-features as described in [CSHL13]. Third and last, PROFEAT uses a translational approach towards PRISM models, which have a formally defined semantics [KNP11].

```

1 formula level = (for i in [0..buffer_size-1]
2                 (cell[i] > 0 ? 1 : 0) + ...
3                 endfor) / buffer_size;
4 Pmin>=1 [ G (level < 0.75) ];
5 Pmin=? [ F (active(Worker[2])) ];
6
7 const threshold = 20;
8 label "goal" = (counter = 0) &
9               for w in [0..2]
10                  (active(Worker[w])) => Worker[w].t = 0 & ...
11               endfor;
12
13 R{"energy"}max<=threshold [ F "goal" ];

```

Lst. 9. Property specifications using the language extensions of PROFEAT

We refer to Sect. 4, where the details on the implementation of how PROFEAT models are translated to pure PRISM models are provided.

3. Family-based analysis with PROFEAT

In this section, we describe the general workflow for analyzing families of systems specified in the PROFEAT language using our implementation². As depicted in Fig. 1, the PROFEAT approach comprises a preprocessing, analysis, and post-processing step for the analysis of families of stochastic systems. The standard of the PROFEAT approach is that all three steps are executed in a row. However, the designer can also decide to perform these steps separately, which is useful, e.g., when the translated model(s) should further be processed, be simulated within the PRISM simulator for debugging, or analyzed using some alternative tool or different analysis engine.

Let us now consider the standard case where the family-based analysis using PROFEAT is performed in one run. First, the designer decides which analysis engine provided by PRISM will be used for the analysis. This decision mainly depends on how the members of the family of stochastic systems share common behaviors. In case there are lots of commonalities, e.g., when the main functionalities are encapsulated into one feature module included in all family members, a symbolic engine is usually the best choice. Also if the model of each family member is as huge that it would possibly not fit into the memory, an analysis still might be successful when using a symbolic engine.

² For the Haskell source code of the tool, we refer to <https://www.tcs.inf.tu-dresden.de/ALGI/PUB/ProFeat>

```

1 root feature
2   all of optional x, optional y;
3   constraint active(x) => active(y);
4   modules base_impl;
5 endfeature
6
7 feature x modules x_impl; endfeature
8 feature y modules y_impl; endfeature
9
10 module base_impl
11   state : [0..1] init 0;
12   [tick] !(x.state = 1) & !(y.state = 1) -> 1/3:(state'=0) + 2/3:(state'=1);
13 endmodule
14
15 module x_impl
16   state : [0..1] init 0;
17   [] state = 0 -> 1/2:(state'=0) + 1/2:(state'=1);
18 endmodule
19
20 module y_impl
21   state : [0..1] init 0;
22   [tick] !(x.state = 1) -> 1/4:(state'=0) + 3/4:(state'=1);
23 endmodule

```

Lst. 10. Simple product line example in PROFEAT

In this spirit, also the decision whether an all-in-one or one-by-one analysis is used by PROFEAT is made: when a symbolic engine is chosen, commonalities in the members of the described family can be further exploited by choosing an all-in-one analysis approach. Within an all-in-one analysis, PROFEAT generates a single PRISM model of the whole family which could be compactly represented within a symbolic representation of the family model. As the symbolic engine of PRISM relies on *multi-terminal binary decision diagrams (MTBDDs)*, the performance and memory consumption of the analysis crucially depends on the variable order inside the MTBDD. The designer has the choice of using automated reordering techniques [KBC⁺16], which could provide a further speed up of the analysis. On the other hand, when an explicit analysis engine is chosen, one-by-one approaches are usually favorable. In this case, PROFEAT generates one PRISM model for each family member, not exploiting commonalities but enabling the parallelization of the analysis, e.g., using multiple cores or distributed computation.

After the decisions about the analysis methods are made, the designer chooses how the analysis results are returned, i.e., post-processed. For families described by only a few features, an explicit representation of the results in terms of a table is usually sufficient. However, as the number of analysis results can be exponential in the number of features, an explicit representation could be difficult to be interpreted by the designer during the feature-oriented engineering process if the family described involves many features. For this, PROFEAT provides symbolic representation of the results, where the designer can choose between propositional formulas or representations based on binary decision diagrams (BDDs) for describing the solution space.

In the following, we provide details on the PROFEAT approach by first describing the concept of symbolic representations using BDDs. Then, we illustrate the pre- and post-processing steps using the simple PROFEAT model provided in Listing 10 as an example. The model comprises the declaration of the feature model (lines 1-8) and the implementation of the base feature and the two feature modules *x* and *y* (line 10-23). In this example, no feature controller is provided, i.e., the resulting family is assumed to be static. Each feature module implementation has a local state *state*. Access to the local state of a feature module implementation is performed by qualifying the scope of the state variable, e.g., *x.state* to refer to the local of the feature *x*.

3.1. Symbolic representation by binary decision diagrams

Binary decision diagrams (BDDs) have first been introduced by Lee [Lee59] and Akers [Ake78] as universal data structure for Boolean functions. In the context of model checking, BDDs are, e.g., used to represent the characteristic function of the transition relation for state-based models.

A BDD induces a rooted, directed, and acyclic graph comprising terminal nodes (representing the constant functions zero and one) and decision nodes. Each decision node is labeled by a Boolean variable and has exactly two successor nodes, where for the 0-successor node, the respective Boolean variable is assigned to **false** and the for the 1-successor node to **true**. Figure 3a shows a simple example of a BDD graph structure, where the 0-successors are indicated by a dashes lines and the 1-successor by solid lines. Ordered BDDs [Bry86] rely on a fixed variable ordering that it used consistently along all paths from the root to a sink. The graph structure of an ordered BDD for a Boolean function f arises from a binary decision tree for f using the given variable ordering by merging isomorphic subgraphs and eliminating terminal nodes with the same value and any node whose two children are isomorphic. This yields a reduced ordered BDDs (briefly called BDD from now on), where every two BDD nodes represent different functions. Removing redundancies from the decision tree and hence the sharing of structure in BDDs yields the potential of compactly representing Boolean functions. When using BDDs to represent family models, this means that ideally behavior that is common amongst all family members as, e.g., a base feature included in every family member, has to be represented only once.

In general, the size of an BDD, namely the overall number of BDD nodes, crucially depends on the given variable ordering. There are Boolean functions that have a very compact BDD-representation independent from the chosen order. For instance, for the n -bit parity function the number of nodes is linear in n . For some Boolean functions, however, the size of the BDD varies from linear to exponential depending on the given variable ordering. An example is the Boolean function for the most significant bit of the addition. Finally, there are Boolean functions for which no compact BDD-representation exists, e.g., the $n-1$ -st bit of the multiplication function. See, e.g. [Weg00] for further details on BDDs. Reordering algorithms such as sifting [Rud93, PS95], can be applied to dynamically improve the size BDD for a given Boolean function.

Multi-terminal binary decision diagrams [CFM⁺93] (MTBDDs), also known as algebraic decision diagrams (ADDs) [BFG⁺97], are a variant of BDDs that support terminal nodes with real values rather than only 0 and 1. MTBDDs are, e.g., used as data structure for representing and analyzing probabilistic systems. The symbolic engine of PRISM for instance, relies on MTBDDs for representing the transition probability matrix of the stochastic models. In PROFEAT we use MTBDDs also to encode the computed results for all family members symbolically. Here, the decision nodes correspond to feature activation or deactivation and the terminals represent the model checking results such as probabilities and expectations.

3.2. All-in-one vs. one-by-one

In the example in Listing 10, the feature model describes a family comprising three feature combinations: the base feature alone, with the y feature activated in addition to the base feature and all features activated. The combination that x is solely activated besides the base feature is excluded through the constraint provided in line 3. When a one-by-one approach is chosen, PROFEAT generates three different models, one for each feature combination and invokes PRISM three times (once for each generated model). Within the all-in-one approach, PROFEAT generates one single model with three initial states.

Note that when the automated variable reordering techniques are used, each verification run may yield a different variable order. Thus, whereas in the all-in-one approach the family is represented by an MTBDD only with respect to one particular variable order, the variable orders between the models for a one-by-one analysis may differ.

Conceptually, it is possible to combine the all-in-one approach and the one-by-one approach. First, the set of instances of the system family is partitioned. Then, each element of the partition can be regarded as a sub-family, which is analyzed using the all-in-one approach. The partition can be determined by the user and allows a trade-off between analysis time, memory usage, and the ability to parallelize the analysis. Von Rhein [vR16] reported that indeed the combined approach uses less memory than the all-in-one approach and less time than the one-by-one approach for selected (non-probabilistic) models. PROFEAT provides limited support for the combined approach.

```
Final result: [0.0,0.9876543209876543]
Results for initial configurations:
  (x, y)=0.0
  (y)=0.73
  ()=0.99
```

Lst. 11. Output of analyzing the simple product line with PROFEAT

Within the `family` block, a set of feature instances can be selected. Features selected in the `family` block then only vary between the sub-families, while the other features vary within each sub-family. For an example, we consider again the producer–consumer model.

```
family
  features Fast, Worker[2];
endfamily
```

If we add the `family` block given above, we get a partition consisting of 4 elements because two features are selected using the `features` keyword.

3.3. Post-processing: output of the analysis results

PROFEAT generates an output of the results similar to PRISM, but for readability’s sake, listing only the valid feature combinations. If the analysis is carried out using the one-by-one approach where PRISM is invoked for each instance of the family, PROFEAT automatically collects the results. In case automatic variable reordering [KBC⁺16] was used during the analysis, PROFEAT also converts the variable ordering of the results to a common ordering. Thus, the representation of the final results is independent of the chosen analysis approach. An example result generated by PROFEAT for the example in Listing 10 is given in Listing 11. Here, we asked for the minimal probability such that `state=1` is reached in the base feature (formally, asking for $\Pr^{\min}(F(\text{state} = 1))$). PROFEAT supports *rounding* of the results up to a given precision and we chose a precision of two digits in our example. Rounding is very handy, e.g., if the results are close together and one likes to investigate the features with the most impact on the result. Currently, PROFEAT supports post-processing of analysis results only if the results were produced by PRISM.

The representation shown in Listing 11 lists the results for each initial configuration separately. This is acceptable if just a small number of configurations is of interest. However, since the number of initial feature combinations is usually exponential in the number of features, it is difficult to draw general conclusions about the whole family of systems from this *explicit* representation of results. In order to give the user a better understanding of the analysis results, PROFEAT can additionally provide a *symbolic* result representations. Similarly to the analysis, where the symbolic representation of the model is often more compact than the explicit representation, a symbolic representation of the results is often smaller than the list of feature combinations. PROFEAT can compute two different symbolic representations. First, PROFEAT can generate a *propositional formula* over features representing the feature combinations satisfying some qualitative property (similar to [CHS⁺10, CCH⁺13b]). This also includes quantitative properties where a certain threshold must be reached. For example, asking for the feature combinations where (in the worst case) `state=1` is reached with a probability of at least 0.99 (formally $\Pr^{\min}(F(\text{state} = 1)) \geq 0.99$), yields the result $\neg x \wedge \neg y$. Second, PROFEAT can produce a BDD, where each inner node stands for the decision whether the corresponding feature is included or not, and the terminal nodes indicate whether the property is satisfied or not. However, in case of a quantitative analysis, the result for a given feature combination is not just true or false, but can be any rational number. Therefore, PROFEAT additionally supports the symbolic representation of the results as an MTBDD. Here, each terminal node represents a distinct numerical analysis result. The rounding of the results can reduce the number of distinct results and, in turn, the number of terminal nodes as well as the overall size of the MTBDD. The BDD or MTBDD graph can be exported into the DOT-language (that can then be rendered into graphs using Graphviz³). Details on how MTBDDs are used

³ <http://www.graphviz.org>

for yielding a user-friendly view on the results are presented in the next section. Examples of different outputs possible within PROFEAT for the analysis of the simple product line of Listing 10 are provided in Fig. 3.

4. Implementation details

In this section, we provide further details on our implementation of PROFEAT. Feature model declarations follow the semantics of TVL [CBH11]. Based on the feature model, the translation of a set of feature modules under a feature controller into a PRISM model is based on the compositional modeling framework for probabilistic feature-oriented systems presented in [DBK15], which naturally maps feature composition to the parallel composition of PRISM. The translation of PROFEAT specifications into PRISM specifications is purely syntactical replacing PROFEAT language identifiers by their translated correspondents in the generated PRISM code.

In the following, we highlight the implementation of notable steps in the translation of PROFEAT models into PRISM models, underpinned by examples from our running example of Sect. 2, the producer–consumer product line example. Furthermore, we describe different analysis methodologies supported by PROFEAT and give an overview on PROFEAT’s result post-processing mechanisms. As illustrated in Fig. 1, PROFEAT takes a PROFEAT model, a PROFEAT formula specification, and the choice of an analysis method (all-in-one or one-by-one) as input. The pre-processor translates the given model into a single PRISM model or a collection of PRISM models respectively. Additionally, the formulas given in the PROFEAT specification file are translated into a standard PRISM properties file. This purely syntactical translation is necessary because the formulas of the PROFEAT specification may contain language constructs not understood by PRISM. These include, for example, qualified names for local variables and the built-in `active` function, which is used to refer to the current feature combination. After the translation step, PROFEAT can automatically invoke PRISM and the results are being post-processed and presented such that they are readable in the feature context.

4.1. Translation of feature-specific constructs

In a PROFEAT model, read and write accesses to the feature combination are only possible by the use of the `active` function and the `activate/ deactivate` updates, respectively. The basic idea is to add one Boolean variable per feature to the PRISM model, indicating whether some feature is part of the feature combination. However, because features can be mandatory (must always be included in a feature combination) or can depend on each other, it is sufficient to generate one variable per non-mandatory *atomic set*. An atomic set is a set of features that can be treated as a unit as they always appear together in a feature combination [Seg08]. For example, in case of the producer–consumer system family (Fig. 2), the tool generates a feature variable for the **Fast** feature and one variable for each **Worker**. Instead of Boolean feature variables, PROFEAT generates integer variables with a range of $[0..1]$, which simplifies the handling of cardinality constraints. Given this representation, the translation of the `active` function is simple. If the atomic set of the feature is mandatory, the call is replaced by `true`. Otherwise, it is replaced by a check of the feature variable. Analogously, the `activate` and `deactivate` updates assign 0 or 1 to the corresponding variable, respectively.

Feature modules are translated to standard PRISM modules. In case of a feature module that implements a multi-feature, one module per feature instance is generated (with the `id` parameter set accordingly). Listing 12 shows the feature module of the **Worker** feature and its translation. Note that only the module corresponding to the first **Worker** feature is shown, as the other instances are nearly identical. In the translated module, all local variables are qualified with their corresponding feature name as the PRISM language does not support local scopes. Additionally, the guard of each command is extended by the `Worker_0_active` predicate, such that the module has no behavior if the feature is inactive. However, it must be ensured 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. This is achieved by adding an unconditional transition for each non-blocking action. Such a transition can only be taken if the feature is inactive (see line 9 in Listing 12b). This command is not generated if the user explicitly requests the blocking of an action by using the `block` keyword in the feature declaration.

<pre> 1 module Worker_impl 2 t : [0..max_work_size] init 0; 3 4 [] t > 0 -> 5 (t' = max(0, t - speed)); 6 [dequeue[id]] t = 0 -> 7 (t' = Buffer.cell[0]); 8 [cancel] true -> (t' = 0); 9 10 endmodule </pre>	<pre> module Worker_0_worker_impl Worker_0_t : [0..max_work_size]; [] Worker_0_active & Worker_0_t > 0 -> (Worker_0_t' = max(0, Worker_0_t - speed)); [dequeue_0] Worker_0_active & Worker_0_t = 0 -> (Worker_0_t' = Buffer_cell_0); [cancel] Worker_0_active -> (Worker_0_t' = 0); [cancel] !Worker_0_active -> true; endmodule </pre>
(a) Worker in PROFEAT model	(b) Worker 0 in PRISM model

Lst. 12. Feature module of a worker and its translation

```

1 controller
2   [] buffer_full & !active(Worker[1]) -> activate(Worker[1]);
3   [] buffer_full & !active(Worker[2]) -> activate(Worker[2]);
4   [] buffer_low -> deactivate(Worker[2]);
5   [] buffer_empty -> deactivate(Worker[1]) & deactivate(Worker[2]);
6 endcontroller

```

Lst. 13. Feature controller for the producer–consumer model

The feature controller is translated into a PRISM module as well. Updates of the feature combination must not lead to an invalid feature combination. Consider the update at line 4 of the feature controller in Listing 13. According to the feature model, at least one of the **Workers** must be active at all times. Thus, the update is only allowed if there is at least one other active **Worker**. If not, this command should block. The described semantics is achieved by extending the guard in the translated module, as shown in Listing 14 (line 2). This guard is synthesized as follows. First, all constraints regarding the features to update (only **Worker 2** in this example) are collected from the feature model. Then, all feature variables that would be changed by the update are substituted with their updated value. Here, the variable `Worker_2` is replaced by 0, because the update would deactivate this feature. The resulting expression only evaluates to true if the updated feature combination is valid. Thus, an update command cannot lead to 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. Consider again the feature controller shown in Listing 13. The command in line 4 implicitly synchronizes with the feature module in Listing 15a. To implement this synchronization, PROFEAT generates action labels for feature activation and deactivation, as shown in line 1 of Listing 14. The command in line 5 of Listing 13 deactivates two **Worker** instances at once, thus it also has to synchronize with both corresponding feature modules. However, in the PRISM input language each command can only be labeled with at most one action label. To circumvent this restriction, the set of action labels is merged into a single action label. This solution requires special care in the translation of the feature modules. First, we collect the action labels of all feature-controller commands that deactivate **Worker 2** (lines 4 and 5). Then, we create a copy of the feature-module command for each collected action label, as shown in Listing 15b. 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.

```

1 [Worker_2_deactivate] buffer_low &
2 (1 <= Worker_0 + Worker_1 + 0) & (Worker_0 + Worker_1 + 0 <= 3) ->
3 (Worker_2' = 0);

```

Lst. 14. Translated feature controller command

<pre> 1 module Worker_impl 2 t : [0..max_work_size] init 0; 3 4 [deactivate] t = 0 -> true; 5 6 7 8 endmodule </pre>	<pre> module Worker_2_Worker_impl Worker_2_t : [0..2]; [Worker_1_deactivate_Worker_2_deactivate] Worker_2_active & Worker_2_t = 0 -> true; [Worker_2_deactivate] Worker_2_active & Worker_2_t = 0 -> true; endmodule </pre>
(a) Worker in PROFEAT model	(b) Worker 2 in PRISM model

Lst. 15. Translation of synchronization with the feature controller

4.2. All-in-one and one-by-one translation

In case of a one-by-one translation, a model for each initial valuation of the system parameters and for each initial feature combination is created. The system parameters are constant for each instance and can therefore be replaced by constants in the translated models. However, the feature variables are not replaced by constants, as the feature combination may be changed by the feature controller.

For the all-in-one translation, PROFEAT 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 system parameters depend on the initial state and thus are not known at translation time. In the producer–consumer model for example, both the buffer size as well as the number of workers may be defined in terms of system parameters. Since all family instances must be contained in the all-in-one model, PROFEAT instantiates arrays with their maximal possible size, generates the maximal number of multi-feature instances and creates variables with the greatest possible bounds. These upper bounds can be computed from the range of the system parameters, which is known at translation time. In the example, the `buffer_size` system parameter may range from 1 to 5. Then, PROFEAT instantiates the `fifo` module (Listing 5) with size 5 in the all-in-one model. The need for instantiating all structures with their maximal size is the main reason that the all-in-one model is often substantially larger than (most of) the models generated by a one-by-one translation.

4.3. Post-processing of analysis results

As a consequence of the translational approach of PROFEAT, the results of a quantitative analysis are ultimately produced by the used analysis tool. In the default case, this tool is PRISM, which is the only analysis tool currently supported for the post-processing step by PROFEAT. Therefore, the analysis results actually refer to the translated PRISM model rather than the PROFEAT model. This means that variable names will not appear as written in the PROFEAT model. Furthermore, PRISM has no concept of features, thus feature variables are not easily distinguishable from other variables. However, the main issue is that the results as produced by PRISM are hard to read, which makes their interpretation challenging.

As a first step, PROFEAT rewrites variable names and feature names such that they appear as in the original PROFEAT model and rounds the results up to a given precision. If the model investigated does not contain a feature model, i.e., the family described arises from parametrization only, PROFEAT returns the resulting list of results. We already presented an example output in Listing 11. Within each line the active features are now indicated with their names as provided in the PROFEAT model which increases the readability of the results.

Symbolic representation of feature-oriented analysis results PROFEAT relies on standard model checking tools for the analysis.

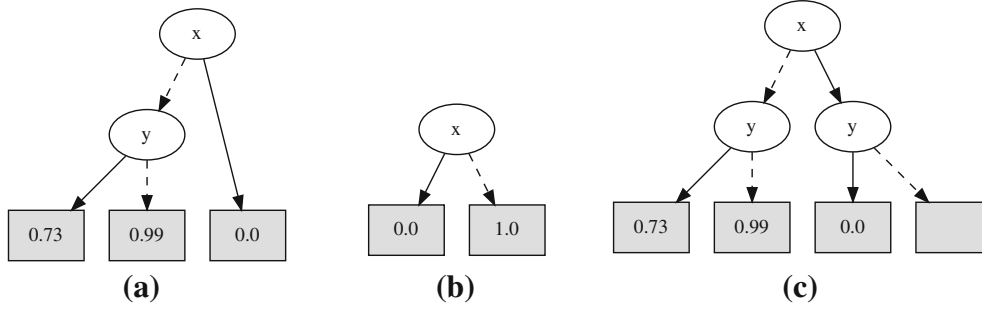


Fig. 3. Example MTBDD result representations within PROFEAT. **a** Without feature model, rounded up to 2 decimals, **b** without feature model, rounded to integers, **c** including feature model, rounded up to 2 decimals (*empty box* represents invalid feature combinations)

Therefore, the symbolic representations of the results are not directly provided by the model checking algorithm. Thus, the propositional formula or the binary decision diagram representing the satisfying feature combinations must be computed from the list of results provided by PRISM.

To generate a propositional formula representing the feature combinations satisfying some property, PROFEAT proceeds as follows. The set of satisfying feature combinations directly corresponds to a formula in canonical disjunctive normal form, where the literals are features. Of course, this CDNF is as big as the explicit representation. In order to minimize this formula, PROFEAT applies the Quine-McCluskey algorithm [McC56]⁴. As a consequence of this approach, the computed formula also encodes the feature model, or at least a part thereof. Since the feature model is already known before the analysis, one is usually more interested in the constraints that must hold in *addition* to the feature model, such that the given property is satisfied. Formally, given a propositional formula Φ encoding the feature model, we want to compute an additional constraint Ψ , such that all feature combinations of the (more restrictive) feature model $\Phi' = \Phi \wedge \Psi$ satisfy the given property. To compute the constraint Ψ , we apply the Quine-McCluskey algorithm to the set of satisfying feature combinations as described above, but also consider all invalid feature combinations as “don’t care” terms, which gives the algorithm more opportunities for minimization.

The BDD or MTBDD representation of the results is built by successively considering the result for each feature combination and adding the corresponding path to the MTBDD. PROFEAT supports two different modes to represent the MTBDD: including the feature model and not including the feature model. Because including the feature model often leads to a substantially larger and more cluttered MTBDD, the latter mode is the default. This representation can then be considered in addition to the feature model to guide the feature selection.

The MTBDDs in Fig. 3a+b show the default representation without the feature model. The variables in the MTBDD correspond to the feature variables, evaluating to true if the respective feature is activated and to false otherwise. Solid lines indicate that the respective feature is active, whereas dashed lines indicate that the feature is deactivated. The sink nodes of the MTBDD carry all the possible values w.r.t. the considered analysis query. A path from the unique root node of the MTBDD to a sink node stands for the set of valid feature combinations that share a common result. Whereas in Fig. 3a, the results are rounded to a precision of two digits, Fig. 3b shows the same result rounded to integers, i.e., rounding probabilities to either 0 or 1. For the representation including the feature model, a sink *inv* is included in the MTBDD to which all paths of invalid feature combinations lead to. Figure 3c shows the MTBDD representation output including the feature model, where the empty sink corresponds to *inv*. This MTBDD degenerates into a binary tree, as all feature combinations have different values and there is exactly one invalid feature combination (in which only feature *x* is active).

As the graphs provided can still be very large, PROFEAT provides a further reduction mechanism by applying the *sifting* algorithm [Rud93] for MTBDDs. The algorithm reorders feature variables to find better orderings that allow for more compact graph structures. This typically yields an order where the feature variables with the greatest impact on the result are considered first. Hence, one can easily extract the “importance” of individual

⁴ The time complexity of this algorithm is exponential. Since the algorithm solves an NP-hard problem, no algorithm of polynomial complexity can be expected. The families of systems suitable for quantitative analysis using model checking usually have a comparably small number of features. Thus, we have not encountered a model where the runtime of the Quine-McCluskey algorithm was the limiting factor.

features w.r.t. the considered query. We will provide an example how the size of the graph can be significantly reduced by reordering in the next section (see Fig. 6). In general, the height of the MTBDD scales linearly with the number of features. However, the number of MTBDD nodes largely depends on the structure of the analysis results and the variable ordering found by the sifting algorithm.

5. Experimental studies

To illustrate the benefits of the PROFEAT approach of analyzing families of stochastic systems, we carried out the following case studies:

A Producer–consumer example to provide a comparison between all-in-one, one-by-one sequentially, and one-by-one parallel analysis approaches.

Product-line case studies to show the additional support of PROFEAT for describing and analyzing probabilistic product lines. In particular, we investigate the body-sensor-net product line of [RAN⁺15] and a probabilistic variant of the elevator product line [PR01].

Benchmark suite examples which illustrate how PROFEAT can be used to analyze families arising from standard PRISM models by parameterizing model constants, e.g., the number of actors, queue size, etc.

A Feature-oriented network system model to show that PROFEAT is also beneficial for analyzing single systems where the concept of features is used to describe dynamics in operational modes.

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. When considering the one-by-one parallel approach, we restricted ourselves to an execution of 32 analyses in parallel.

5.1. 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 feature-oriented system 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.

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 4a 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 than 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 Fig. 4b–d.

In some cases, where the number of instances is exponential in the family parameter (cf. Fig. 4c—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. Fig. 4b—Best Buffer), the all-in-one approach was only superior up to a system size of 14. For the third model variant (cf. Fig. 4d—Distributions), the all-in-one and one-by-one approaches asymptotically displayed similar performance. Overall, there is no clear trend on which approach is favorable, the one-by-one or the all-in-one analysis.

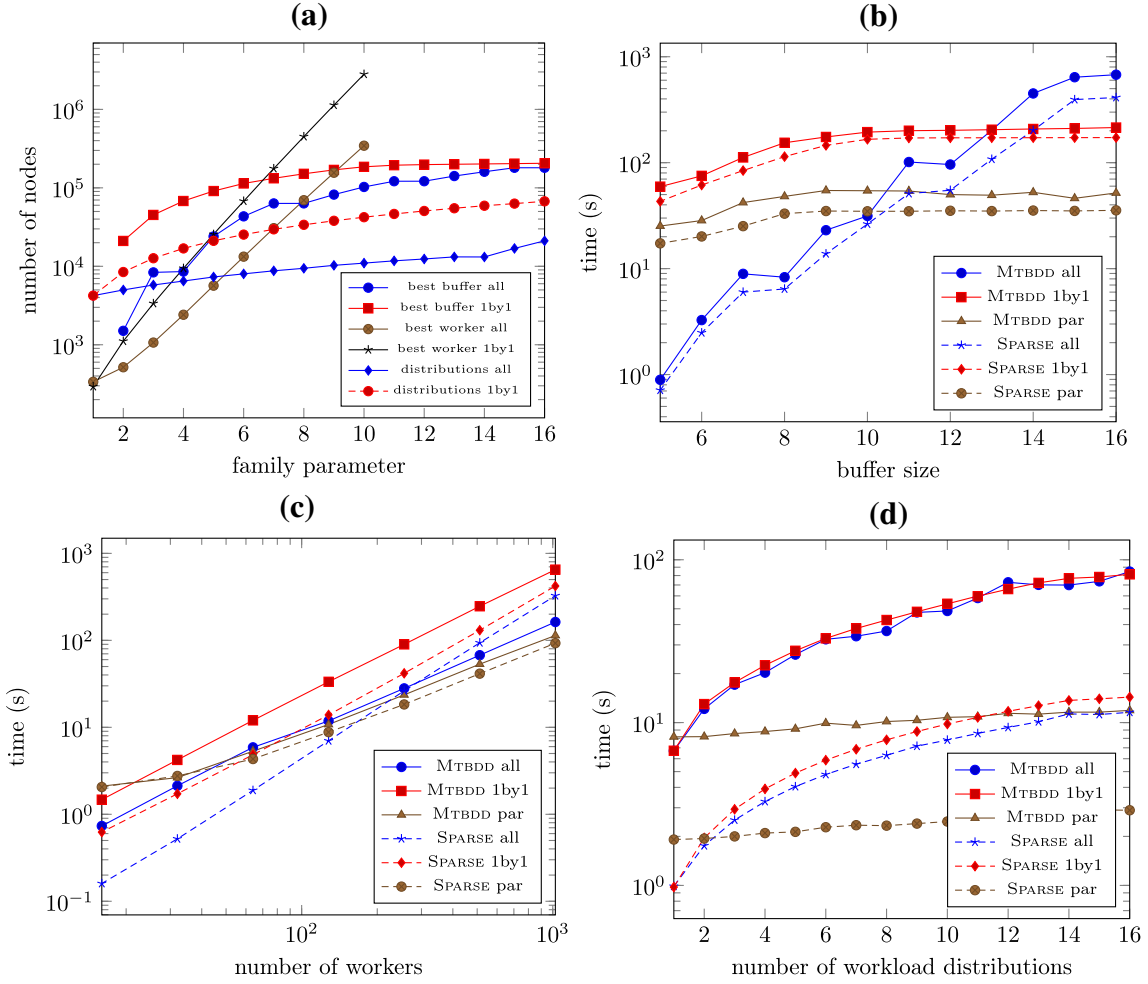


Fig. 4. Number of MTBDD nodes for the producer-consumer models **a**, analysis times of the variants best buffer **b**, best worker **c** and distributions **d**. Number of MTBDD nodes and analysis times are accumulated over all instances for the one-by-one approach. In **b-d** the x-axis shows the number of family instances

5.2. Product-line 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 (see, e.g., [CHS⁺10, TAK⁺14, DBK15]). In this section, we demonstrate how (probabilistic) versions of classical product lines can be modeled and analyzed with PROFEAT.

5.2.1. Body sensor network product line

A Body Sensor Network (BSN) system is a network of connected sensors sending measurements to a central entity that evaluates the data and identifies health critical situations. In [RAN⁺15], a (static) BSN product line with features for several sensors has been introduced, whose feature model is depicted in Fig. 5.

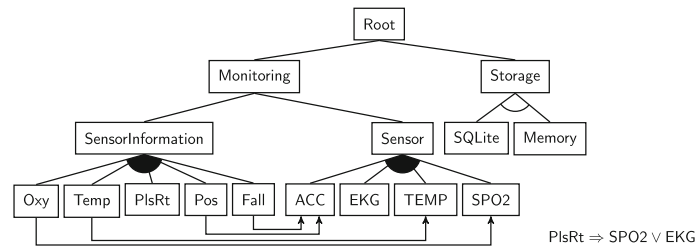


Fig. 5. A feature diagram corresponding to the BSN feature model

Results (non-zero only) for filter "init":

```

2924: (0,1,0,1,0,1,0,1,1,1,1,1,0,0,0,0,1,1,1)=0.9704387384917665
2977: (0,1,0,1,0,1,1,1,0,1,1,1,0,0,0,0,1,1,1)=0.9617396442452253
4041: (0,1,0,1,1,1,1,1,0,1,1,1,0,0,0,0,1,1,1)=0.953118529409139
4191: (0,1,1,0,0,0,0,1,0,0,0,1,0,0,0,0,1,1,1)=0.9792165174854354
5163: (0,1,1,0,0,1,1,1,0,0,1,1,0,0,0,0,1,1,1)=0.9617396442452253
... 293 lines omitted ...

```

Range of values over initial states: [0.9445746949695318,0.9792165174854354]

Lst. 16. Excerpt of analysis results provided by PRISM for the BSN product-line model

The approach presented in [RAN⁺15] follows the ideas by [GS13] 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 [RAN⁺15] 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 [RAN⁺15], as PROFEAT's representation of features as Boolean parameters is compatible with the approach by [GS13]. Thus, PROFEAT allows for an all-in-one approach on the same model as of [RAN⁺15] 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 [RAN⁺15]. The all-in-one approach turns out to be ≈ 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 [RAN⁺15].

For this case study, we highlight the capabilities of post-processing in PROFEAT, which provides output in the context of the feature model and thus eases the interpretation of the results. Note that a feature-aware output in the case studies by [RAN⁺15] was not possible within their approaches. Listing 16 shows an excerpt of the results provided by an all-in-one analysis with PRISM, not using the post-processing step provided by PROFEAT. Using the post-processing of PROFEAT, these results are interpreted by replacing feature variables by its feature names, which yields a more readable list of results as illustrated in Listing 17. Notably, the latter results are identical also when a one-by-one analysis method would have been chosen⁵ (which would, without the post-processing, have lead to 298 complete output logs by PRISM).

⁵ Even though the results obtained with the all-in-one approach might potentially differ from those obtained by the one-by-one approach (caused by the approximative analysis methods used in PRISM), we never encountered this case in any of our case studies.

Final result: [0.9445746949695318,0.9792165174854354]

Results for initial configurations:

(Mem, Fall, Oxy, PlsRt, Pos, Temp, SACC, SSP02, STemp)=0.9445746949695318

(Mem, Oxy, PlsRt, Pos, Temp, SACC, SSP02, STemp)=0.953118529409139

(Mem, PlsRt, Pos, Temp, SACC, SSP02, STemp)=0.9617396442452253

(Mem, Pos, Temp, SACC, STemp)=0.9704387384917665

(Mem, PlsRt, SECG, STemp)=0.9792165174854354

... 293 lines omitted ...

Lst. 17. Excerpt of analysis results after post-processing by PROFEAT

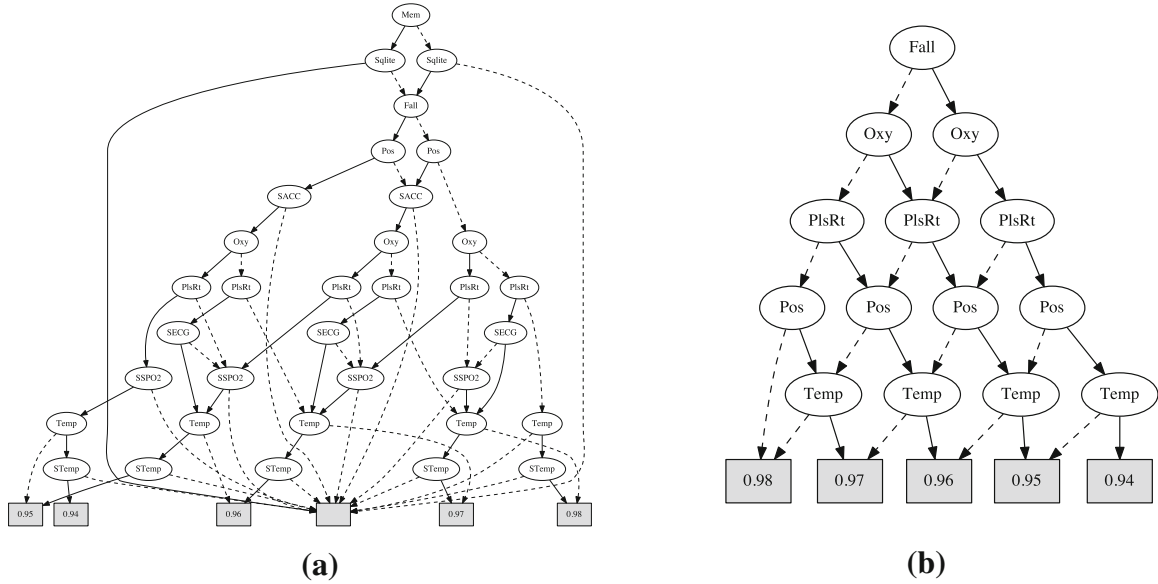


Fig. 6. MTBDD representations for the analysis results of a Body Sensor Network product line with rounding precision 2. Outgoing *solid edges* denote that the feature is included, *dashed edges* denote that it is excluded. The empty terminal node in **a** indicates an invalid feature combination. **a** Including feature model, precision: 2 digits. **b** Without feature model, reordered, precision: 2 digits

Using result grouping by rounding with a precision of two decimals and the MTBDD-representation capabilities of PROFEAT, the results of the analysis of the BSN case study yields the MTBDDs shown in Fig. 6. Especially the MTBDD representation not including the feature model impressively shows how clearly representable the results for each of the 298 valid feature combinations are within PROFEAT. Figure 6a shows the MTBDD generated for the BSN product-line analysis if the feature model is included. Omitting the constraints imposed by the feature model and applying the reductions described in Sect. 4.3 yields the MTBDD as shown in Fig. 6b. This allows the user to gain insights on the influence of certain features on the final result. For example, the user has to decide on the inclusion or exclusion of only 5 features (the BSN product line comprises 16 features, of which 11 are not mandatory). The selection of all other features either does not influence the result (as is the case for the SQLite and Memory features), or the inclusion/exclusion is dictated by the feature model. Consider, for example, following the path \neg Fall, \neg Oxy, \neg PlsRt, \neg Pos to the result 0.98. The feature model states that at least one of the features below the SensorInformation feature must be selected.

Table 1. Analysis times (in seconds) of feature and benchmark suite models (each row represents one query, analysis time for one-by-one is sum over all instances)

Model	inst.	MTBDD nodes		all	MTBDD		all	HYBRID		all	SPARSE	
		family	separate		lbyl	par		lbyl	par		lbyl	par
BSN	298	5651	111507	1	129	25	1	128	25	1	128	25
Elevator (2 floors)	64	42254	1329204	1	65	7	2	49	7	1	45	7
Elevator (3 floors)	64	151274	4924349	4	223	11	98	2531	96	7	286	18
Elevator (4 floors)	64	420448	13519274	15	910	32	2601	54262	1952	56	2008	83
Elevator (2-4 floors)	192	779569	19772827	29	1199	49	5089	56843	2052	74	2339	106
CSMA (2-4 processes)	4	633997	634076		timeout			not supported			1236	1251
		“	“		timeout		3660	3577	3384	1078	1013	954
Self-stabilization (3-21 processes)	19	4340	10662	2036	1643	932	251	37	22	129	33	20
		“	“	$\ll 1$	1	2		not supported		122	24	15
		“	“		timeout			not supported		2629	476	269
		“	“	13	10	7	12	10	6	12	10	7
		“	“	13	10	7	13	10	7	13	10	6
Philosophers (3-12)	10	82995	82689	9056	6212	3945	9722	5949	4009	out of memory		
PWCS (3 replicas, 1-9 writers)	9	134236	134190	49	26	15	232	165	130	314	271	220
		“	“	6564	2247	960		not supported		5473	1544	1230
PWCS (3 writers, 1-7 replicas)	7	955505	958033	752	2279	1628	968	348	306	738	2209	1265
		“	“		timeout			not supported		1221	3857	2735

Thus, when following the described path in the diagram, it is clear that the **Temp** feature must be selected and, in turn, also the **TEMP** sensor feature. Therefore, the nodes for **Temp** and **TEMP** are not present on this path of the diagram. Furthermore, the diagram clearly shows that the overall system reliability decreases with each additionally selected **SensorInformation** child feature. It is very hard to draw conclusions like this from a list of results as shown in Listing 17. Thus, the transformation into an MTBDD that succinctly represents the analysis results is a very useful tool to derive products with the desired properties and behavior.

5.2.2. Elevator product line

A classical (non-probabilistic) product line comprises an elevator system, introduced by [PR01]. This product line has been then considered in several case studies issuing family-based verification (see, e.g., [ARW⁺13, CSHL13]). The elevator system is modeled by a cabin which can transport persons to floors of a building. 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 [PR01], 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 [DBK15]. 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 at 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.

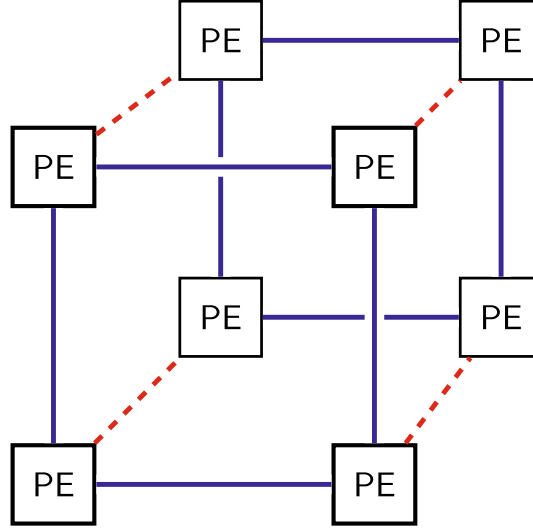


Fig. 7. A network system model with cube topology. The *blue lines* indicate wired connections, while the *dashed red lines* denote directed optical connections (color figure online)

5.3. Benchmark suite examples

We used PROFEAT also to model and analyze some examples taken from the PRISM benchmark suite [KNP12] and the probabilistic locking protocol PWCS [BEK⁺13] 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. This is mainly caused by the fact that there is almost no sharing of behaviors between the family instances. Consequently, the one-by-one approach outperforms the all-in-one approach in almost all cases, even for the self-stabilization protocol.

5.4. Feature-oriented network system model

PROFEAT is not only beneficial for specifying families of systems as illustrated in the above case studies, but also simplifies modeling systems with dynamic feature-oriented runtime characteristics. We consider a system which consists of several processing elements (PEs) interconnected by a communication network. The network is heterogeneous, i.e., it comprises network links with different characteristics. Figure 7 shows a system with both wired and (point-to-point) optical links. While the wired links are static, the optical links may be dynamically turned on and off. Furthermore, the links differ in their speed and energy consumption. We assume that activation of optical links costs energy as well as keeping them activated, even in case they are unused. In our model, the system processes concurrent tasks that are distributed among the PEs and communicate over the available network links. We focus on the network links and keep the PEs abstract. Thus, the state of the system is entirely determined by the load on the network links.

The model cycles through four phases. First, the number of PEs for an incoming task is determined. This number is binomially distributed to allow changing the externally imposed load on the system by a single parameter. In the second phase, the task is mapped onto the PEs, which increases the load on the network links between the selected PEs. Also, the mapping may cause the activation of optical links. In case no mapping is possible or no mapping is found, the task is dropped (raising a “fail” event). In the third phase, optical links can be activated or deactivated, the latter however only if they have no load. The fourth phase represents the processing of tasks.

```

controller
...
for i in [0..NUM_OPTIONAL_LINKS-1]
[] active(Link[i]) & load[i] = 0 & phase = RECONF & link = i -> deactivate(Link[i]);
[] phase = PHASE_RECONF & link = i -> true;
endfor
endcontroller

```

Lst. 18. Excerpt of the feature controller of the network system model. Unused links are deactivated nondeterministically.

Since the model focuses on the communication structure and its characteristics, we kept the PEs abstract and modeled processing of tasks by probabilistically decreasing the load on the network links.

We created three variants of the model that differ in the way the task mapping phase is implemented. The first two models are non-deterministic and are capable for a best- and worst-case analysis, whereas the latter one illustrates a randomized strategy for a possible implementation.

Non-deterministic For a task that requires a number n of network links, this model non-deterministically selects one of all possible spanning trees of size n over the network topology.

Non-deterministic with hopping This is a variation of the non-deterministic model, that may create mappings with additional *hops*, i.e., PEs that do not contribute to the processing of a task, but rather act as a communication relay between other PEs.

Heuristic A simple heuristic for mapping tasks to PEs follows the “greedy” principle. In the first step, the first PE with the most remaining capacity among its adjacent links is selected. In the second step, the task is randomly distributed among the PEs adjacent to the selected PE. The probability that an adjacent link is selected is indirectly proportional to its current load.

In the following, we illustrate notable modeling details. For the activation and deactivation of links, we utilize the feature-oriented modeling approach of PROFEAT. For each optical link i , the model contains a corresponding optional feature `Link[i]`, i.e., the set of optical links is compactly represented by a single multi-feature. This simplifies the definition of costs, since they only have to be defined once in the feature model and then are automatically applied to all optical links. Turning links on and off is handled by the feature controller. An excerpt of the controller of the non-deterministic variant is shown in Listing 18. Here, a `for` loop is used to generate a deactivation command for each optical link. The `load` array stores the current load of each network link. The excerpt in Listing 19 shows a part of the mapping implementation, where one of all possible spanning trees of size `task_size` is selected. An element i of the `selected` array is `true` if the PE with index i is a node in the spanning tree. The formulas `from(k)` and `to(k)` return the endpoints of a link with index k . The spanning tree is built up incrementally. In each step, another link, which must be connected to at least one other chosen link selected beforehand, is selected nondeterministically. The load on the selected link is then incremented and the `task_size` decremented. This process is repeated until the remaining `task_size` needing to be distributed among the network reaches zero. The meta-programming facilities of PROFEAT allowed us to define the model’s behavior completely independent from the network topology. Instead of being hard-coded into the model, the topology is described by an array of constants that specifies the set of network links (accessed using the `from` and `to` formulas described above).

We analyzed an instance of the network system model, where we allowed for tasks of size at most 2. The probability that at the beginning of a round a task of size 1 is scheduled is given by a parameter t_1 , i.e., the probability t_2 that a task of size 2 is scheduled is $t_2 = 1 - t_1$. First, we investigated the (minimal) probability that a mapping fails within 9 rounds and plotted the results depending on t_1 in Fig. 8a. Clearly, the resulting probability for the non-deterministic variant serves as theoretical optimum, i.e., is smaller than the probabilities for the hopping and heuristic variant. As expected, the probability of a mapping failure decreases when t_1 increases, as then it is more likely to have tasks of size 1. Tasks of size 1 can easily be mapped when there is at least one communication link without a task assigned, which does not depend on previous choices of mappings.

```

for i in [0..NUM_LINKS-1]
[] phase = MAP & task_size > 0 & load[i] < link_capacity(i) &
  selected[from(i)] & !selected[to(i)] ->
  (selected[to(i)]' = true) & (load[i]' = load[i] + 1) & (task_size' = task_size - 1);
// same command again, but with "to" and "from" switched
endfor

```

Lst. 19. Excerpt of the nondeterministic variant of the network system model showing how a spanning tree is built during the mapping phase

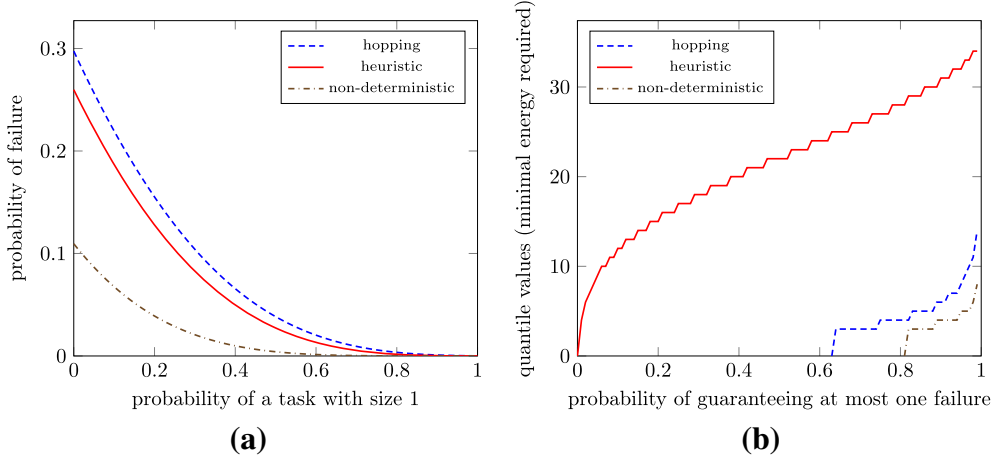


Fig. 8. Results for analyzing the net case study. **a** Minimal probability of having a mapping failure within 9 rounds, depending on the arrival probability of a task with size 1 in each round. **b** Minimal energy required such that there is a mapping strategy which guarantees with probability p at most one mapping failure within 9 rounds

However, mapping tasks of size 2 is more likely to fail, as two links without a task assigned have to be chosen that additionally have to be connected through processing elements. Thus, whether a mapping of a task of size 2 is successful also depends on previous mapping choices. The second property we investigated considers the trade-off between successful mappings and the energy required to guarantee them within a certain probability, which is formalized as an energy-utility quantile [BDD⁺14]. Here, we assumed that activating optical links consume 2 units of energy, while keeping them active requires 1 unit of energy. Wired links do not consume any energy. Figure 8b shows the quantile value as the minimal energy required depending on the probability within which at most one failed mapping has to be guaranteed. The higher p , the more likely it is to also use optical links for avoiding mapping failures, hence using more energy and thus, increasing the quantile value. The plot shows results for $t_1 = 0.5$, i.e., the probability distribution over the task sizes is uniform. Due to the combinatorial blowup from the number of possible mappings, the model suffers from the state-space-explosion problem, peaking at around 120 million states. We hence had to carry out all computations using the MTBDD-engine of PRISM, also profiting from automatic variable reordering mechanisms introduced in [KBC⁺16], which reduced the MTBDD-representation of the models by around 40%.

6. Conclusions and future work

We presented the tool PROFEAT, which eases the verification of quantitative requirements on families of probabilistic systems that either follow the feature-oriented compositional framework of [DBK15] and/or system families induced by varying parameters. For this, PROFEAT extends the input language of the prominent probabilistic model checker PRISM [KNP11] with feature-oriented concepts, templates and parametrization as well as

scripting constructs for meta-programming to describe families of probabilistic systems. Choosing a translational approach, PROFEAT operates in three phases: first, a PROFEAT family is translated into standard PRISM models. Then, the actual verification is performed by PRISM on the translated models. At the end, PROFEAT extracts the analysis results returned by PRISM and represents them in the feature-aware context. This approach has several advantages: no specialized feature-aware model-checking algorithm is required as standard PRISM is used. Furthermore, all advantages of PRISM take over to the quantitative analysis of families of systems described by PROFEAT, including possible future advances in PRISM development.

As the size of the described family usually grows exponentially in the number of features, both, the analysis itself and the results of the analysis can benefit from symbolic representations. On the analysis side, PROFEAT supports either an all-in-one or one-by-one analysis, i.e., checking the whole family on one single model or each family member separately. The all-in-one approach exploits commonalities of the family members through symbolic representations, whereas the one-by-one approach may profit from parallelization. PROFEAT provides only basic support for partitioning the family model into sub-families to be analyzed separately. However, extending this support to combine the benefits of both approaches is left for future work. In the present version of PROFEAT, a seamlessly integration of the automated variable-reordering mechanisms recently presented in [KBC⁺16] can be also be exploited to speed up and reduce memory consumption when using symbolic analysis engines.

To conveniently represent the analysis results symbolically, PROFEAT either uses propositional formulas or binary decision diagrams on the features of the family model. Currently, only static feature-oriented models, i.e., where the feature combination does not change during run time, have special support to represent the analysis results. We plan to integrate meaningful explanations of the results for dynamic feature-oriented systems in the future.

The practicality of the PROFEAT tool has been illustrated by a couple of case studies addressing different types of system families. These models have been used for comparing the all-in-one analysis approach with the one-by-one approach in the probabilistic setting. 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 (see, e.g., [TAK⁺14]), this cannot be generalized to arbitrary families, e.g., when only a few common behaviors exist within the family members. Moreover, we illustrated that the feature-based modeling approach is also useful for modeling single systems, i.e., where PROFEAT is not used to describe a family of systems but the dynamics of a single system running in several operational modes. 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 symbolic analysis engines).

Besides the future prospects drawn above, we plan to integrate other model-checking tools that support PRISM's input language (such as STORM [DJKV16], MRMC [KZH⁺11]) into PROFEAT. Adaptions are first and foremost required for the feature-aware symbolic representations of the analysis results, as PROFEAT currently is capable of interpreting PRISM result logs.

Acknowledgements

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 the Research Training Group RoSI (GRK 1907), and Deutsche Telekom Stiftung.

References

- [AH99] Alur R, Henzinger TA (1999) Reactive modules. *Form Methods Syst Des*, 15(1):7–48
- [AH10] Apel S, Hutchins D (2010) A calculus for uniform feature composition. *ACM Trans Program Lang Syst* 32(5):19
- [AJTK09] Apel S, Janda F, Trujillo S, Kästner C (2009) Model superimposition in software product lines. In: *ICMT'09*, volume 5563 of LNCS, pp 4–19. Springer, Berlin
- [AK09] Apel S, Kästner C (2009) An overview of feature-oriented software development. *J Object Technol* 8(5):49–84
- [Ake78] Akers SB (June 1978) Binary decision diagrams. *IEEE Trans Comput* 27(6):509–516
- [ARW⁺13] Apel S, von Rhein A, Wendler P, Groesslinger A, Beyer D (2013) Strategies for product-line verification: case studies and experiments. In: *Proceedings of the 2013 international conference on software engineering, ICSE '13*. IEEE, pp 482–491

- [ASW⁺11] Apel S, Speidel H, Wendler P, von Rhein A, Beyer D (2011) Detection of feature interactions using feature-aware verification. In: International conference on automated software engineering (ASE). IEEE, pp 372–375
- [AtBGF11] Asirelli P, ter Beek MH, Gnesi S, Fantechi A (2011) Formal description of variability in product families. In: Proceedings of the 2011 15th international software product line conference, SPLC '11. IEEE Computer Society, Washington, DC, USA, pp 130–139
- [BdA95] Bianco A, de Alfaro L (1995) Model checking of probabilistic and non-deterministic systems. In: FSTTCS'95, volume 1026 of LNCS, pp 499–513
- [BDD⁺14] Baier C, Daum M, Dubslaff C, Klein J, Klüppelholz S (2014) Energy-utility quantiles. Springer, Berlin, pp 285–299
- [BEK⁺13] Baier C, Engel B, Klüppelholz S, Märcker S, Tews H, Völz M (2013) A probabilistic quantitative analysis of probabilistic-write/copy-select. In: Proceedings of the 5th NASA formal methods symposium (NFM), LNCS. Springer, pp 307–321
- [BFG⁺97] Bahar RI, Frohm EA, Gaona CM, Hachtel GD, Macii E, Pardo A, Somenzi F (1997) Algebraic decision diagrams and their applications. *Form Methods Syst Des* 10(2/3):171–206
- [BK98] Baier C, Kwiatkowska M (1998) Model checking for a probabilistic branching time logic with fairness. *Distrib Comput* 11(3):125–155
- [BK08] Baier C, Katoen J-P (2008) Principles of model checking. The MIT Press, Cambridge
- [Bry86] Bryant RE (1986) Graph-based algorithms for boolean function manipulation. *IEEE Trans Comput* 35:677–691
- [BSRC10] Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later: A literature review. *Inf Syst* 35(6):615–636
- [CBH11] Classen A, Boucher Q, Heymans P (2011) A text-based approach to feature modelling: syntax and semantics of TVL. *Sci Comput Program* 76(12):1130–1143
- [CCH⁺12] Classen A, Cordy M, Heymans P, Legay A, Schobbens P-Y (2012) Model checking software product lines with SNIP. *STTT* 14(5):589–612
- [CCH⁺13a] Cordy M, Classen A, Heymans P, Legay A, Schobbens P-Y (2013) Model checking adaptive software with featured transition systems. LNCS. Springer, Berlin, pp 1–29
- [CCH⁺13b] Cordy M, Classen A, Heymans P, Schobbens P-Y, Legay A (2013) ProVeLines: a product line of verifiers for software product lines. In: 17th International software product line conference (SPLC). ACM, pp 141–146
- [CCH⁺14] Classen A, Cordy M, Heymans P, Legay A, Schobbens P-Y (2014) Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci Comput Program* 80:416–439
- [CCS⁺13] Classen A, Cordy M, Schobbens P-Y, Heymans P, Legay A, Raskin J-F (2013) Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans Softw Eng* 39(8):1069–1089
- [CDKB16] Chrszon P, Dubslaff C, Klüppelholz S, Baier C (2016) Family-based modeling and analysis for probabilistic systems—featuring ProFeat. Springer, Berlin, pp 287–304
- [CFM⁺93] Clarke EM, Fujita M, McGeers PC, McMillan KL, Yang JC-Y, Zhao X-J (1993) Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. In: Proceedings of international workshop on logic and synthesis
- [CHE05] Czarnecki K, Helsen S, Eisenecker UW (2005) Formalizing cardinality-based feature models and their specialization. *Softw Process Improv Pract* 10(1):7–29
- [CHS⁺10] Classen A, Heymans P, Schobbens P-Y, Legay A, Raskin J-F (2010) Model checking lots of systems: efficient verification of temporal properties in software product lines. In: 32nd International conference on software engineering (ICSE). ACM, pp 335–344
- [CN01] Clements P, Northrop L (2001) Software product lines: practices and patterns. Addison-Wesley Professional, Reading
- [CSHL13] Cordy M, Schobbens P-Y, Heymans P, Legay A (2013) Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. IEEE Press, pp 472–481
- [DABW15] Dimovski AS, Al-Sibahi AS, Brabrand C, Wasowski A (2015) Family-based model checking without a family-based model checker. In: Model checking software—22nd international symposium, SPIN 2015, Stellenbosch, South Africa, August 24–26, 2015, Proceedings, pp 282–299
- [Daw04] Daws C (2004) Symbolic and parametric model checking of discrete-time Markov chains. In: Theoretical aspects of computing—ICTAC 2004, volume 3407 of LNCS, pp 280–294
- [DBK15] Dubslaff C, Baier C, Klüppelholz S (2015) Probabilistic model checking for feature-oriented systems. *Trans Aspect-Oriented Softw Dev XII*, 8989:180–220
- [Dij75] Dijkstra EW (1975) Guarded commands, nondeterminacy and formal derivation of programs. *Commun ACM* 18(8):453–457
- [DJJ⁺15] Dehnert C, Junges S, Jansen N, Corzilius F, Volk M, Buntjes H, Katoen J-P, Abraham E (2015) PROPhESY: a probabilistic parameter synthesis tool. In: 27th International conference on computer aided verification (CAV), volume 9206 of LNCS, pp 214–231
- [DJKV16] Dehnert C, Junges S, Katoen J-P, Volk M (2016) The probabilistic model checker Storm (extended abstract). [arXiv:1610.08713](https://arxiv.org/abs/1610.08713)
- [DKB14] Dubslaff C, Klüppelholz S, Baier C (2014) Probabilistic model checking for energy analysis in software product lines. In: 13th International conference on modularity, MODULARITY '14, Lugano, Switzerland, April 22–26, 2014, pp 169–180
- [DMFM10] Dinkelaker T, Mitschke R, Fetzner K, Mezini M (2010) A dynamic software product line approach using aspect models at runtime. In: Proceedings of the 1st workshop on composition and variability
- [DS11] Damiani F, Schaefer I (2011) Dynamic delta-oriented programming. In: Proceedings of the 15th International software product line conference, SPLC '11. ACM
- [FGT12] Filieri A, Ghezzi C, Tamburrelli G (2012) A formal approach to adaptive software: continuous assurance of non-functional requirements. *Form Asp Comput* 24(2):163–186
- [GH03] Gomaa H, Hussein M (2003) Dynamic software reconfiguration in software product families. In: PFE, pp 435–444
- [GS13] Ghezzi C, Sharifloo AM (2013) Model-based verification of quantitative non-functional properties for software product lines. *Inf Softw Technol* 55(3):508–524

- [HHWZ10] Hahn EM, Hermanns H, Wachter B, Zhang L (2010) PARAM: A model checker for parametric Markov models. In: 22nd International conference on computer aided verification (CAV), volume 6174 of LNCS, pp 660–664
- [HHZ11] Hahn EM, Hermanns H, Zhang L (2011) Probabilistic reachability for parametric Markov models. *Softw Tools Technol Transf* 13(1):3–19
- [Kat93] Katz S (1993) A superimposition control construct for distributed systems. *ACM Trans Program Lang Syst* 15(2):337–356
- [KBC⁺16] Klein J, Baier C, Chrszon P, Daum M, Dubslaff C, Klüppelholz S, Märcker S, Müller D (2016) Advances in symbolic probabilistic model checking with PRISM. In: Tools and algorithms for the construction and analysis of systems—22nd international conference, TACAS 2016, Proceedings, pp 349–366
- [KCH⁺90] Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University
- [KNP11] Kwiatkowska M, Norman G, Parker D (2011) PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan G, Qadeer S (eds) Proceedings of 23rd international conference on computer aided verification (CAV'11), volume 6806 of LNCS. Springer, pp 585–591
- [KNP12] Kwiatkowska MZ, Norman G, Parker D (2012) The PRISM benchmark suite. In: Proceedings of quantitative evaluation of systems (QEST'12), pp 203–204. IEEE <https://github.com/prismmodelchecker/prism-benchmarks/>.
- [KST14] Kowal M, Schaefer I, Tribastone M (2014) Family-based performance analysis of variant-rich software systems. In: Fundamental approaches to software engineering, volume 8411 of LNCS, pp 94–108
- [KZH⁺11] Katoen J-P, Zapreev IS, Hahn EM, Hermanns H, Jansen DN (2011) The ins and outs of the probabilistic model checker MRMC. *Perform Eval* 68(2):90–104
- [Lee59] Lee CY (1959) Representation of switching circuits by binary-decision programs. *Bell Syst Tech J* 38(4):985–999
- [LP17] Legay A, Perrouin G (2017) On quantitative requirements for product lines. In: Proceedings of the eleventh international workshop on variability modelling of software-intensive systems, VAMOS '17, New York, NY, USA. ACM, pp 2–4
- [LPT09] Lauenroth K, Pohl K, Toehning S (2009) Model checking of domain artifacts in product line engineering. In: 24th IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 269–280
- [McC56] McCluskey EJ (1956) Minimization of boolean functions*. *Bell Syst Tech J* 35(6):1417–1444
- [PR01] Plath M, Ryan M (2001) Feature integration using a feature construct. *Sci Comput Program* 41(1):53–84
- [PS95] Panda S, Somenzi F (1995) Who are the variables in your neighborhood. In: Proceedings of computer-aided design (ICCAD'95). IEEE, pp 74–77
- [RAN⁺15] Rodrigues GN, Alves V, Nunes V, Lanna A, Cordy M, Schobbens P-Y, Sharifloo AM, Legay A (2015) Modeling and verification for probabilistic properties in software product lines. In: High assurance systems engineering (HASE). IEEE, pp 173–180
- [Rud93] Rudell R (1993) Dynamic variable ordering for ordered binary decision diagrams. In: IEEE/ACM international conference on computer-aided design (ICCAD-93), pp 42–47
- [Sch10] Schaefer I (2010) Variability modelling for model-driven development of software product lines. In: VaMoS
- [Seg08] Segura S (2008) Automated analysis of feature models using atomic sets. In: SPLC (2), pp 201–207
- [TAK⁺14] Thüm T, Apel S, Kästner C, Schaefer I, Saake G (June 2014) A classification and survey of analysis strategies for software product lines. *ACM Comput Surv* 47(1):6:1–6:45
- [tBFGM16] ter Beek MH, Fantechi A, Gnesi S, Mazzanti F (2016) Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *J Log Algebraic Methods Program* 85(2):287 – 315
- [tBLLV15] ter Beek MH, Legay A, Lluch-Lafuente A, Vandin A (2015) Statistical analysis of probabilistic models of software product lines with quantitative constraints. In: 19th International conference on software product line (SPLC). ACM, pp 11–15
- [tBMS12] ter Beek MH, Mazzanti F, Sulova A (2012) VMC: a tool for product variability analysis. Springer, Berlin, pp 450–454
- [TKB⁺14] Thüm T, Kästner C, Benduhn F, Meinicke J, Saake G, Leich T (2014) FeatureIDE: an extensible framework for feature-oriented software development. *Sci Comput Program* 79:70–85
- [vR16] von Rhein Alexander (2016) Analysis strategies for configurable systems. PhD thesis, University of Passau
- [Weg00] Wegener I (2000) Branching programs and binary decision diagrams: theory and applications. Monographs on discrete mathematics and applications. SIAM Philadelphia

Received 19 November 2016

Accepted in revised form 3 July 2017 by Perdita Stevens, Andrzej Wasowski, and Ewen Denney

Published online 7 August 2017