Under consideration for publication in Formal Aspects of Computing

ProFeat: Modeling and Analyzing Feature-oriented Probabilistic Systems

Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier
Faculty of Computer Science, Technische Universität 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, i.e., perform 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, which comprises a modeling language to describe families of probabilistic systems and a framework that enables their family-based analysis using state-of-the-art probabilistic model checkers. ProFeat provides special support for the modeling and analysis of feature-oriented systems and (probabilistic) product lines with dynamic feature switches, multi-features and feature attributes. By means of several case studies we show how ProFeat eases family-based modeling and carry out quantitative analysis using the probabilistic model checker Prism to 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]. Within software product lines, variants of a software are developed to satisfy the needs of different customers, e.g., by providing a free but ad-financed

Correspondence and offprint requests to: Philipp Chrszon, e-mail: Philipp.Chrszon@tu-dresden.de

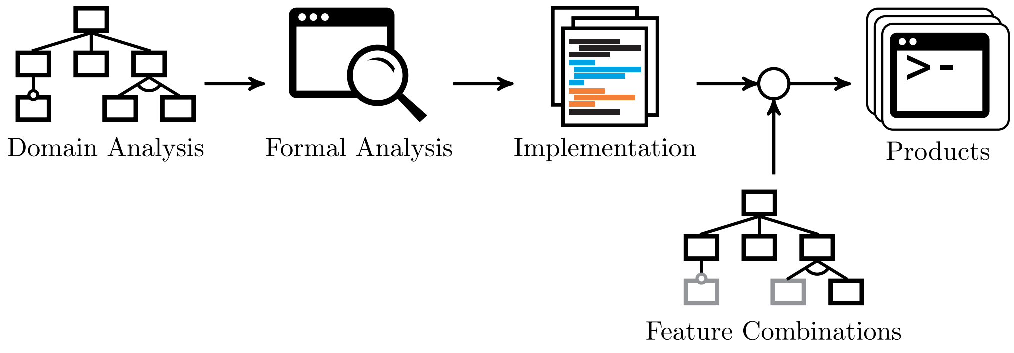


Fig. 1. Overview of the software-product-line development process

version or a professional version of the software including functionalities for enterprises. Usually, software product lines are developed following the approach depicted in Figure 1: 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 as illustrated in Figure 1. 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 re-design 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 focuses on the formal-analysis aspect of the described development process by introducing the tool ProFeat, which consists of a high-level modeling language for feature-oriented (probabilistic) systems that can be interpreted and then analyzed by a feature-aware (quantitative) analysis part of the tool. Our modeling language can be seen as a feature-aware extension of the input language of the probabilistic model checker Prism [KNP11]. To specify valid feature combinations and 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]. For specifying the (operational) behavior of the system, ProFeat follows the approach of [DBK15], which introduced a formal framework for feature-oriented probabilistic systems. The behaviors of each feature are specified in *feature modules* described by discrete- or continuous-time Markov chains or Markov decision processes (MDPs). The support for dynamic feature switches during runtime is maintained by another 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]. Probabilistic dynamic product lines as presented in [DBK15] allow, e.g., to model the frequencies of uncontrollable feature switches by stochastic distributions. The potential adaptations are then modeled by non-deterministic feature switches. In the ProFeat language, the feature modules and the feature controller are represented by an extension of modules from the guarded command language used in Prism. The close connection of the ProFeat language to prominent existing specification languages both on the feature modeling and the behavior description side, makes it easy to learn, understand, and use the ProFeat tool.

To analyze families of systems specified in our new modeling language, ProFeat follows a translational approach, illustrated in Figure 2. In a preprocessing step, ProFeat models and requirements are translated into the pure input language of Prism. As many state-of-the art probabilistic model checkers support the input language of Prism [KNP11], a formal analysis then can be carried out by choosing other analysis tools such as, e.g., MRMC [KZH+11], Marcie [HRS13], Param [HHWZ10], or PROPhESY [DJJ+15]. In an optional post-processing step, ProFeat then can interpret the analysis results provided by the external analysis tool and represent them in the feature-aware context. Its purpose is to enable the feature-aware interpretation of the result, e.g., through collapsing similar analysis results and representing them using

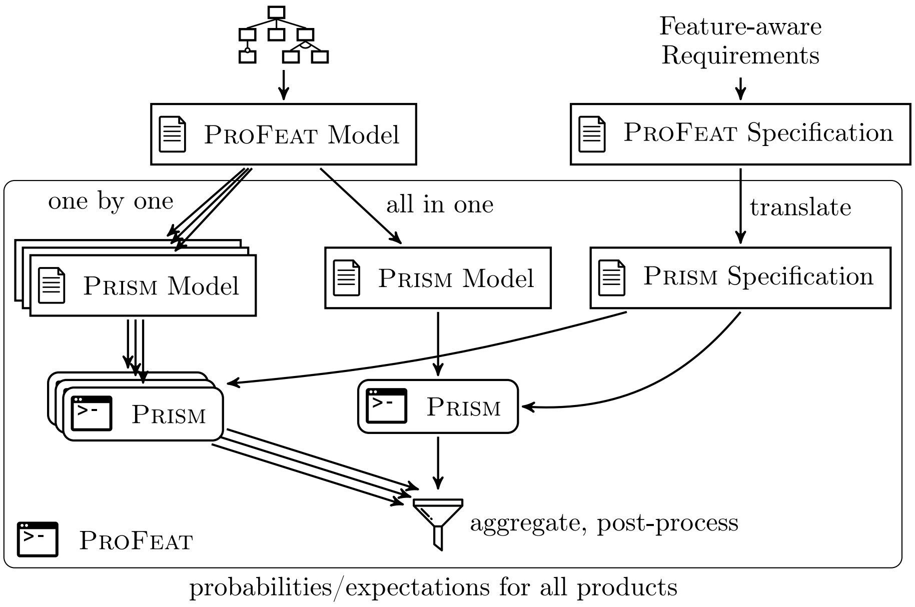


Fig. 2. Architecture of ProFeat

multi-terminal binary decision diagrams (MTBDD) [CFM+93]. Our implementation currently only provides the post-processing step for results returned by Prism. However, the architecture of ProFeat allows the integration of parsers for analysis results produced by other probabilistic model checkers.

Following existing results established in the field of product-line analysis (see, e.g., [MTS+14, DBK15]), our implementation supports two kinds of model translations and analysis: *all-in-one* and *one-by-one* analysis. Within the all-in-one approach, all family members are encoded into one single model and analyzed in a single run, whereas within a one-by-one analysis, each family member devotes a single model each analyzed separately. The advantage of an all-in-one analysis is to exploit commonalities between the family members by a symbolic representation of the single model and thus, speed up the analysis. For this, the tool chosen for the analysis step clearly has to support symbolic analysis algorithms. A one-by-one analysis however, can profit from parallel computations to speed up the analysis of the whole family. ProFeat seamlessly integrates both analysis approaches in the sense that the post-processing step yields the same output of the results, independent from the chosen kind of analysis.

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

To illustrate the capabilities of ProFeat, we considered a series of examples and compared the performance of all-in-one and one-by-one analyses using the three symbolic Prism engines Mtbdd, Hybrid and Sparse. While the Mtbdd engine is fully symbolic and carries out all computations using multi-terminal binary decision diagrams (MTBDD), the numerical computations of the Sparse engine are carried out using sparse matrices, and the Hybrid engine relies on an MTBDD-representation of the model and a sparse representation of probability or expectation vectors. Our experimental results indicate that there is no clear superiority of the all-in-one analysis approach, no matter which of the three Prism engines is used. However,

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

for well-known product-line models, where the base functionality contains most of the behaviors and features have comparably less behaviors, all-in-one approaches are feasible (especially within the MTBDD engine).

Related work. 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., [MTS+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 whole family and appropriate algorithms to reason about each member of the family 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.

We now briefly summarize related work concerning feature-aware analysis using model checking. 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. Gruler et al. [GLS08] introduced a feature-based extension of the process algebra CCS and presented model-checking algorithms to verify requirements expressed in the μ-calculus. We are not aware of any implementation of this approach. Lauenroth et al. [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. 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]. 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 support for family-based probabilistic model checking of dynamic product lines. 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 Markov decision processes to PRISM as done in [DBK15]. The benefits of probabilistic model checking for the analysis of adaptive software has been already drawn by Filieri et al. [FGT12]. 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 PROPHESY [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. The PROFEAT language can handle probability parameters as well and translate them to PRISM code. However, there is no direct connection between PROFEAT and the parametric probabilistic model checkers as they do not support multiple initial states. The recent work by Beek et al. [tBLLV15] presents a framework for the analysis of software product lines using statistical model checking. An approach towards a family-based performance analysis of dynamic probabilistic product lines arising from UML activity diagrams has been presented by [KST14].

Outline. Section 2 presents the main principles of the PROFEAT language. The standard workflow for analyzing PROFEAT models is described in Section 3. Details on the implementation of PROFEAT will be given in Section 4. Section 5 reports on experimental studies within ProFeat. A brief conclusion is provided in Section 6.

This paper is an extended version of the conference paper [CDKB16]. The source code of ProFeat can be obtained at `https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/ProFeat`.

2. Modeling Families of 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 probabilistic automata and particularly discrete- or continuous-time Markov chains (DTMCs, or CTMCs, respectively) and Markov decision processes (MDPs).

The first class of extensions that ProFeat provides (cf. Section 2.2), are metaprogramming language constructs not yet included in the Prism modeling language. For instance, ProFeat supports parameters, arrays, and loops. Besides required for the description of feature-oriented models, these language concepts are useful in general and bring Prism's input language closer to classical imperative programming languages. The second class (cf. Section 2.3) adds support for feature-based modeling. This is inspired by the application of feature-oriented formalisms for software product lines.

A ProFeat model usually consists of two 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 model 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. 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.

In what follows, we illustrate the modeling approach of ProFeat 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:

$$\texttt{[action-name] guard} \rightarrow \mathtt{p}_1\texttt{: update}_1 \texttt{ + } \mathtt{p}_2\texttt{: update}_2 \texttt{ + } \ldots \texttt{+ } \mathtt{p}_\mathrm{n}\texttt{: update}_\mathrm{n}$$

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 $\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n$. 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 3 shows an example of a stuttering

```
module counter
  c : [0..3] init 0;

  [] c < 3 -> 0.99: (c' = c + 1) + 0.01: true;
  [tick] c = 3 -> (c' = 0);
endmodule
```

Fig. 3. A module for a modulo-4-counter in the PRISM language

modulo-4-counter that increments the counter only with probability 0.99 at each step. The reset transition is labeled with the action name `tick`.

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 rewards[2] 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 model systems generically. 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
```

[2] 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")

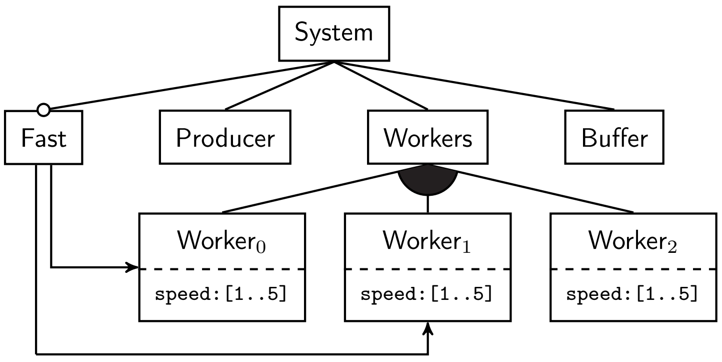


Fig. 4. Feature diagram of the producer-consumer product line

```
root feature
  all of Producer, Buffer, Workers, optional Fast;
  constraint active(Fast) => active(Worker[0]) & active(Worker[1]);
  constraint Worker[0].speed + Worker[1].speed < 7;
endfeature

feature Workers
  some of Worker[3];
endfeature

feature Worker
  speed : [1..5];
endfeature
```

Lst. 5. Excerpt of the feature model for the producer-consumer product line

Here, three distinct `dequeue` actions are generated.

2.3. Feature-oriented Modeling

A product line comprises a set of feature combinations, which are defined through a *feature model.* The standard formalism to define feature models in the software-engineering domain is provided by *feature diagrams* [KCH$^+$90]. Figure 4 depicts the feature diagram for our running example, the producer-consumer product line. Nodes in the diagram denote features and the root node stands for the so called root feature. Arcs connect an inner node with its children, namely the sub-features of the respective feature. They can be of different type imposing certain constraints on the the set of valid feature combinations. In the example, the root feature System has four sub-features. Three of them are mandatory in valid feature combinations, only the Fast feature is optional (indicated by the circle on the corresponding arc). The Workers feature has three sub-features for the workers. The arcs imply that at least one of the workers must be active in all valid feature combinations. Moreover, there can be arrows indicating additional dependencies possibly across the entire diagram. For instance, the arc between Fast and Worker$_1$ stands 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 Figure 4) and cardinalities indicating the minimum or maximum number of instances of the same feature that could be activated, etc.

Feature Models in ProFeat. Feature diagrams are usually conveniently described using the Textual Variability Language (TVL) [CBH11]. ProFeat provides similar language concepts for declaring the features, their attributes and dependencies. Listing 5 is an excerpt of the producer-consumer feature model.

Features are declared within feature blocks indicated by the keywords `feature` and `endfeature`. The

```
feature Worker
  speed : [1..5];
  block dequeue[id];
  modules Worker_impl;
endfeature

module Worker_impl
  t : [0..max_work_size] init 0;
  [working[id]] t > 0 -> (t' = max(0, t - speed));
  [dequeue[id]] t = 0 -> (t' = Buffer.cell[0]);
endmodule
```

Lst. 6. Declaration and implementation of the Worker feature

`root feature` is a designated feature that represents the base functionality on which the product line is built upon. 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.

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 our running example, the Worker feature is implemented by the `Worker_impl` module. Listing 6 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 Section 2.1). 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

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

Lst. 7. The feature controller of the producer-consumer model

```
feature Buffer
  modules fifo(buffer_size);
endfeature

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

Lst. 8. A FIFO buffer implementation parameterized over the capacity

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. 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 6). 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 7, 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 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. 9. A producer implementation with parameterized probability distribution over the work package size

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

Lst. 10. Parametrization of a model using the `family` block

Consider the `Buffer` feature and its accompanying feature module shown in Listing 8. 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 9) 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.

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 10). 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.

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

Lst. 11. Specification of the energy consumption of a Worker

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:

$$\Pr^{\min}\big(\mathsf{G}\,(\mathsf{level} < 0.75)\big) = 1$$

One can also ask for computing, e.g., the minimal probability that at some point Worker_2 is active, which could be expressed by

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

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 11 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.

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

Listing 12 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

As in the Prism input language, also in the ProFeat language, every model 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 a family of formal models indicated, i.e., a family of DTMCs, CTMCs or MDPs. 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,

```
formula level = (for i in [0..buffer_size-1]
                  (cell[i] > 0 ? 1 : 0) + ...
                endfor) / buffer_size;
Pmin>=1 [ G (level < 0.75) ];
Pmin=? [ F (active(Worker[2])) ];

const threshold = 20;
label "goal" = (counter = 0) &
               for w in [0..2]
                 (active(Worker[w])) => Worker[w].t = 0 & ...
               endfor;

R{"energy"}max<=threshold [ F "goal" ];
```

Lst. 12. Property specifications using the language extensions of ProFeat

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]. We refer to Section 4, where the details on the implementation how ProFeat models are translated to pure Prism models are provided.

3. The Family-based ProFeat Approach in a Nutshell

In this section, we describe the general workflow for analyzing families of systems specified in the ProFeat language using our implementation[3] to which we also refer to as "ProFeat". As illustrated in Figure 2, the ProFeat approach consists of a preprocessing and post-processing step. Before the actual translation step, the user may decide whether to create one Prism model for the entire system family (for an all-in-one analysis) or individual models for each family member (for a one-by-one analysis). 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 one Prism model or a collection of Prism models respectively. In the latter step, also the formula is being translated such that it fits the produced Prism model(s). After that, ProFeat can automatically invoke Prism and the results are being post-processed and presented such that they are readable in the feature context. Clearly, ProFeat can perform both steps, preprocessing and post-processing also separately. This is useful, e.g., when the translated model(s) should further be processed, be simulated within the Prism simulator, or analyzed using some alternative tool.

In the following, we illustrate the pre- and post-processing using a simple ProFeat model as a running example. Its code is provided in Listing 13, comprising 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 simple 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. Preprocessing: All-in-One and One-by-One Analysis

As mentioned above, ProFeat supports two different approaches for the translation of ProFeat models: The *one-by-one* (instance-based) approach and the *all-in-one* (family-based) approach. Within the one-by-one approach, ProFeat generates one Prism model for each instance of the family. In case that no `family` block is given in the ProFeat program, a model for each valid initial feature combination will be generated.

[3] For the Haskell source code of the tool, we refer to `https://wwwtcs.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. 13. Simple product line example in ProFeat

Within the all-in-one approach, the whole family is encoded into one Prism model with multiple initial states, one for each instance in the family. The all-in-one analysis is of particular interest when using symbolic representations the model(s), e.g., by algorithms based on binary discussion diagrams (BDDs). In this case, the shared behavior potentially dominates the size of the BDD, whereas the different feature combinations have only a moderate effect on the BDD size and possibly the analysis time. Taking this to an extreme, the rough idea is that analyzing the whole family of systems would then almost be as expensive as analyzing just a single family member. ProFeat does not provide mechanisms for explicitly exploiting commonalities among family members. Instead we rely on the symbolic representations. For instance, the Mtbdd engine and the Hybrid engine of Prism could be used to benefit from redundancies in the family model. However, since the all-in-one approach combines all family instances into one large model, the analysis usually requires more memory than sequentially performing one-by-one analysis for each member of the family. As we illustrate in our case studies (see Section 5), it depends on the model as well as the time and memory constraints which approach is more appropriate. Please note, that switching from one analysis approach to the other does not require any changes in the ProFeat model. In our running example, 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 of Listing 13. Thus, whereas within a one-by-one approach, 3 models are generated by ProFeat, an all-in-one approach yields one model with three initial states.

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. Within the `family` block, a set of feature instances can be selected.

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

Lst. 14. Output of analyzing the simple product line with ProFeat

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.

3.2. 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. Additionally, if the analysis was carried out using the one-by-one approach, ProFeat automatically collects the results such that the presentation of the final results is independent of the chosen analysis approach. An example result generated by ProFeat on our running example is given in Listing 14. Here, we asked for the minimal probability such that `state=1` is reached in the base feature (formally, asking for $\mathrm{Pr}^{\min}(\mathsf{F}\,(\mathsf{state} = 1))$). 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 provided by Prism only. However, as the parser for analysis tool outputs is implemented in a modular way, ProFeat can easily be adapted to provide the post-processing also for other analysis tools.

The representation shown in Listing 14 lists the results for each initial configuration separately. This is acceptable if just a small number of configurations is of interest. However, it is often difficult to draw general conclusions about the whole family of systems from this representation, especially if the number of configurations is large. For example, one might wonder which combinations of features lead to an undesired result, or which influence a given feature has on the result. In the non-probabilistic case, this problem is solved by providing a *symbolic* representation of the verification result. Here, a set of features violating a certain property is given in terms of a propositional formula [CHS+10, CCH+13b]. In the probabilistic case, this is not possible since the analysis produces quantitative results. That is, the result for a given feature combination is not just true or false, but can be any rational number. ProFeat thus supports the symbolic representation of the results as a *Multi-Terminal Binary Decision Diagram* (MTBDD). Details on how MTBDDs are used 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 13 are provided in Figure 19.

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.

```
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
```

(a) Worker in ProFeat model

```
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
```

(b) Worker 0 in Prism model

Lst. 15. Feature module of a worker and its translation

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 Section 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.

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 (Figure 4), 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 15 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 15b). This command is not generated if the user explicitly requests the blocking of an action by using the `block` keyword in the feature declaration.

The feature controller is translated into a Prism module as well. Updates of the feature combination must not to lead to an invalid feature combination. Consider the update at line 4 of the feature controller in Listing 16. 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 17 (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

```
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. 16. Feature controller for the producer-consumer model

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

Lst. 17. Translated feature controller command

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 16. The command in line 4 implicitly synchronizes with the feature module in Listing 18a. To implement this synchronization, ProFeat generates action labels for feature activation and deactivation, as shown in line 1 of Listing 17. The command in line 5 of Listing 16 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 18b. 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.

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,

```
module Worker_impl
  t : [0..max_work_size] init 0;

  [deactivate] t = 0 -> true;



endmodule
```

(a) Worker in ProFeat model

```
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
```

(b) Worker 2 in Prism model

Lst. 18. Translation of synchronization with the feature controller 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 8) 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.

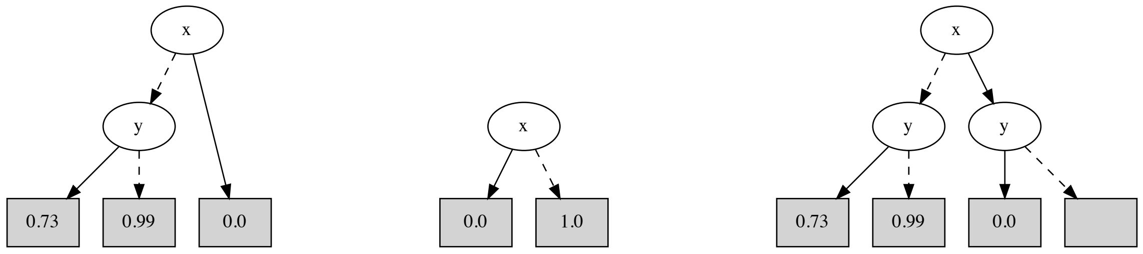
As a first step, ProFeat rewrites variable names and feature names such that they appear as in the original ProFeat model and rounds the results up to a given precision. If the model investigated does not contain a feature model, i.e., the family described arises from parametrization only, ProFeat returns the resulting list of results. We already presented an example output in Listing 14. 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. In the case of feature-oriented systems where a feature model is given in the ProFeat code, one has to expect many results returned by the analysis tool as the number of feature combinations might be exponential in the number of features. Then, the output can hardly provide deep insight on the impact of individual features on the product line w.r.t. to the considered quantitative property. Hence, ProFeat supports the translation of the table of result values for the feature combinations into a symbolic representation in terms of an MTBDD. This symbolic representation can be given to the user by exporting the MTBDD graph into the DOT-language (that can be rendered into graphs using Graphviz[4]). Figure 19 shows possible MTBDD outputs of ProFeat when applied to the ProFeat model given in Listing 13. 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.

In ProFeat, an MTBDD is built by successively considering the result for each feature combination and adding the corresponding path to the MTBDD. As usual in reduced BDDs structures, redundancies are automatically removed from the graph while adding new functions mapping feature combinations to analysis results. That is, at no point in time there are two MTBDD nodes that represent the same function.

ProFeat supports two different modes to represent the MTBDD: not including the feature model (which is standard) and including the feature model (which we call *full*). The MTBDDs in Figure 19a+b show the standard representation, including only valid features combinations. Whereas in Figure 19a, the results are grouped with precision of two digits, Figure 19b shows the same result with no precision, i.e., rounding probabilities to either 0 or 1. For the full representation, a sink `inv` is included in the MTBDD to which all paths of invalid feature combinations lead to. Figure 19c shows the full MTBDD representation output

[4] `http://www.graphviz.org`



(a) Standard, rounding 2 (b) Standard, rounding 0 (c) Full, rounding 2

Fig. 19. Example MTBDD result representations within ProFeat

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. The result is typically such that the feature variables that have the greatest impact on the result are considered first in the resulting order. Hence, one can easily extract the "importance" of individual 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 Figure 24).

5. Experimental Studies

As ProFeat follows a translational approach, all-in-one and one-in-one analyses can be carried out using the same model-checking tool Prism, allowing for a conceptual comparison of both approaches. Besides a sequential one-by-one analysis as usually performed within product-line verification (see, e.g., [ARW$^+$13]), we also provide results for analyzing the models generated by the one-by-one translation in parallel. Clearly, under the quite unrealistic assumption that lots of CPU cores (which allow for parallelization) and enough memory is provided, a parallel execution is likely to outperform an all-in-one approach. For our experiments we used a Linux machine with two 8-core Intel Xeon E5-2680 CPUs running at 2.7 GHz and equipped with 384 GBytes of RAM, hyper-threading enabled. Thus, we restricted ourselves to an execution of 32 analyses in parallel.

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

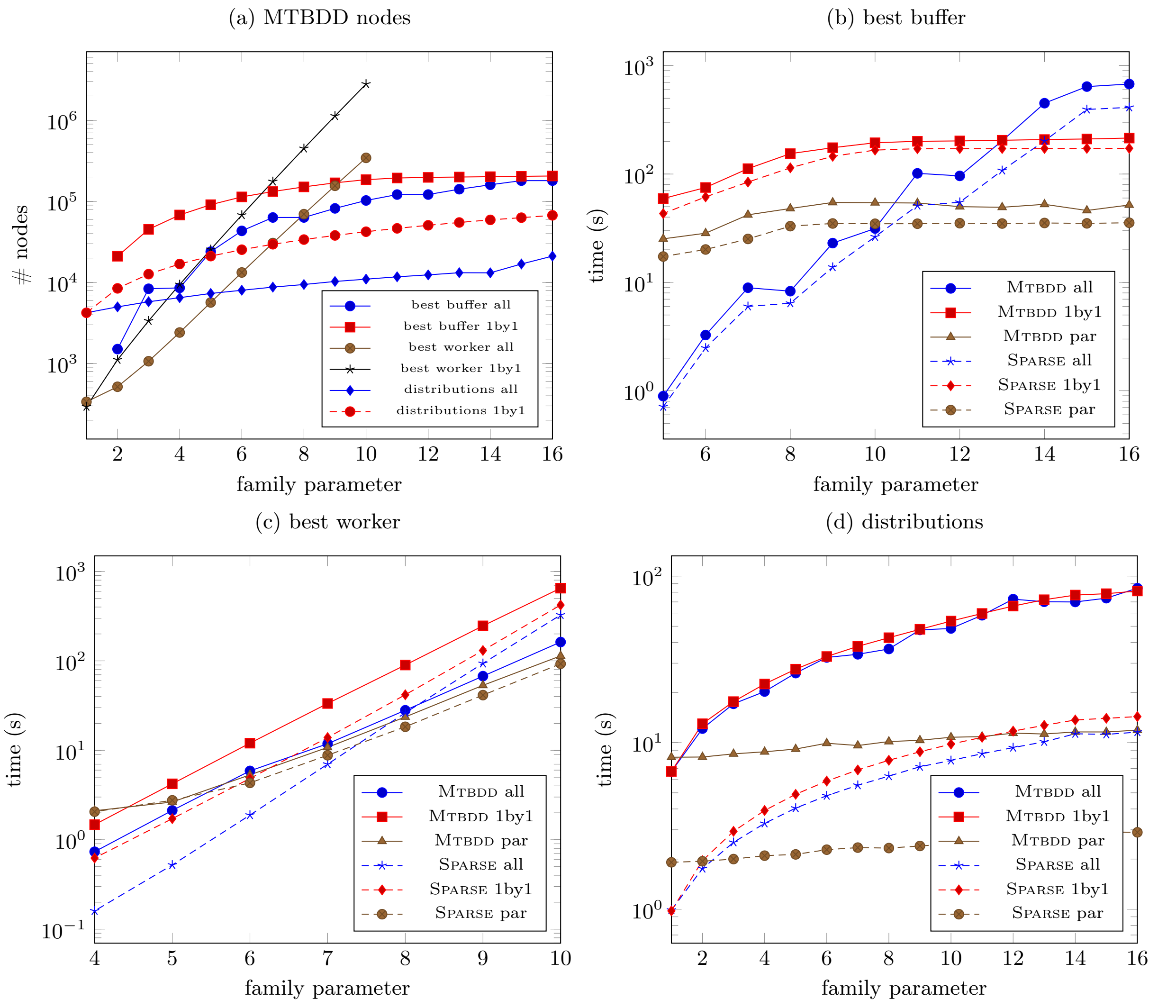


Fig. 20. Number of MTBDD nodes for the producer-consumer models (a), Analysis times of the variants best buffer (b), best worker (c) and distributions (d)

goal is to compute the distribution where the expected energy required to finish a certain number of jobs is minimal.

Figure 20a shows the number of MTBDD nodes for representing the three model variants depending on the family parameter. Within all variants, the number of nodes in the all-in-one model is significantly smaller that the sum of the MTBDD nodes for the separate models, indicating shared behaviors between the family members. We evaluated the quantitative queries stated above using both, the Mtbdd and the Sparse engine of Prism. In general, the Sparse engine turned out to perform slightly better than the Mtbdd engine, especially within expectation queries. The results are illustrated in Figure 20b–d.

In some cases, where the number of instances is exponential in the family parameter (cf. Figure 20c – Best Worker), the all-in-one analysis approach outperforms the one-by-one approach and can even keep up with the parallel computation. In other cases (cf. Figure 20b – Best Buffer), the all-in-one approach was only superior up to a system size of 14. For the third model variant (cf. Figure 20d – Distributions), the all-in-one and one-by-one approaches asymptotically displayed similar performance. Overall, there is a no clear trend on which approach is favorable, the one-by-one or the all-in-one analysis.

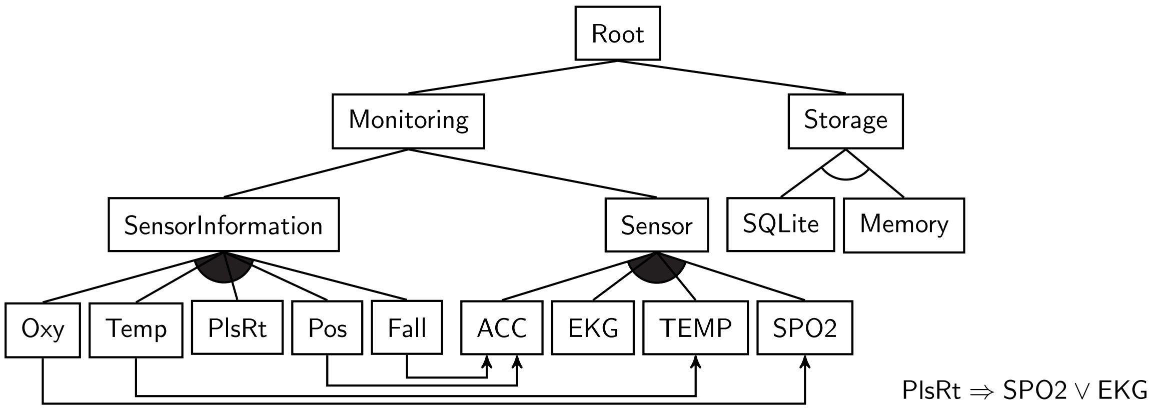


Fig. 21. A feature diagram corresponding to the BSN feature model

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, MTS+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, those feature model is depicted in Figure 21.

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 used. Listing 22 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 23. Notably, the latter results are identical also when a one-by-one analysis method would have been chosen

```
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. 22. Excerpt of analysis results provided by Prism for the BSN product-line model

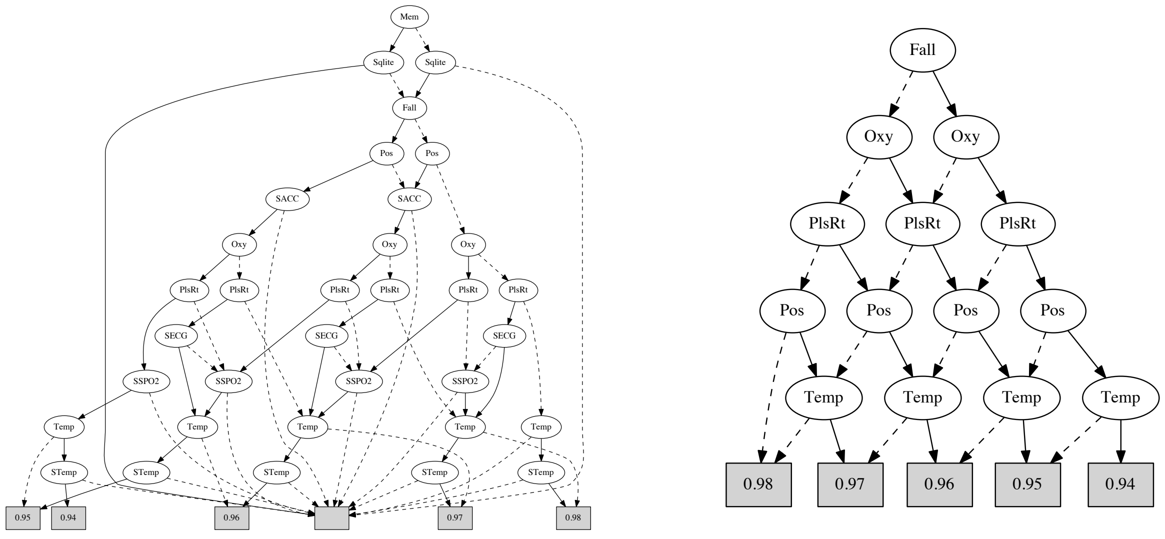
```
Final result: [0.9445746949695318,0.9792165174854354]
Results for initial configurations:
(Mem, Fall, Oxy, PlsRt, Pos, Temp, SACC, SSPO2, STemp)=0.9445746949695318
(Mem, Oxy, PlsRt, Pos, Temp, SACC, SSPO2, STemp)=0.953118529409139
(Mem, PlsRt, Pos, Temp, SACC, SSPO2, STemp)=0.9617396442452253
(Mem, Pos, Temp, SACC, STemp)=0.9704387384917665
(Mem, PlsRt, SECG, STemp)=0.9792165174854354
                  ... 293 lines omitted ...
```

Lst. 23. Excerpt of analysis results after post-processing by ProFeat

(which would, without the post-processing, have lead to 298 complete output logs by Prism). Using result grouping by rounding with precision 2 and the MTBDD-representation capabilities of ProFeat, the results of the analysis of the BSN case study yields the full and reordered MTBDDs shown in Figure 24. Especially the reordered MTBDD representation impressively shows how clearly representable the results for each of the 298 valid feature combinations are within ProFeat: Figure 24a shows the initial MTBDD generated for the BSN product-line analysis. Applying the reductions described in Section 4.3 yields a reduced MTBDD as shown in Figure 24b. This allows the user to gain insights on the influence of certain features on the final result. For example, the diagram clearly shows that the final result depends only on the inclusion or exclusion of 5 features (the BSN product line comprises 16 features). Furthermore, the inclusion of the Temp feature will always decrease the final result by approximately 0.01. It is very hard to draw conclusions like this from a list of results as shown in Listing 23. 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 considers an elevator system, introduced by [PR01] for detecting feature interactions. It has been then considered in several case studies issuing family-based product-line verification (see, e.g., [ARW$^+$13, CSHL13]). An elevator system is modeled by a cabin which can transport persons to floors of a building. The persons first have to push a button at the floor and then in the cabin for calling the elevator and defining a direction where to ride, respectively. In its basic version [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.



(a) Full, precision 2 (b) Reordered, precision 2

Fig. 24. Full and reordered 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.

Our product lines have 64 feature combinations each, parametrized over the number of floors (2-4) in the building. We finally consider the family of the three product lines, containing 192 single instances of the elevator system. We asked for the minimal probability that if the cabin is on the ground floor and the top floor is requested, the probability to serve the top floor within the next three steps is greater than 0.99. Our analysis results are depicted in Table 1, where especially for larger instances the Mtbdd all-in-one analysis outperforms other approaches and engines. Notice that the number of MTBDD nodes of the family model containing all three elevator product lines (cf. the row above the double rule) is greater than the sum of nodes of the family models for each product line. Possibly, other MTBDD variable orderings, e.g., provided by methods presented in [KBC$^+$16], could yield smaller model representations and faster all-in-one analyses.

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. The one-by-one approaches outperform the all-in-one approaches in almost all cases, even for the self-stabilization protocol.

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

Model	MTBDD nodes		Mtbdd			Hybrid			Sparse		
	family	separate	all	1by1	par	all	1by1	par	all	1by1	par
BSN	5651	111507	1	129	25	1	128	25	1	128	25
Elevator (2 floors)	42254	1329204	1	65	7	2	49	7	1	45	7
Elevator (3 floors)	151274	4924349	4	223	11	98	2531	96	7	286	18
Elevator (4 floors)	420448	13519274	15	910	32	2601	54262	1952	56	2008	83
Elevator (2-4 floors)	779569	19772827	29	1199	49	5089	56843	2052	74	2339	106
CSMA	633997	634076		timeout			not supported		1236	1251	1220
(2–4 processes)	“	“		timeout		3660	3577	3384	1078	1013	954
Self-stabilization	4340	10662	2036	1643	932	251	37	22	129	33	20
(3–21 processes)	“	“	$\ll 1$	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)	82995	82689	9056	6212	3945	9722	5949	4009		out of memory	
PWCS (3 replicas,	134236	134190	49	26	15	232	165	130	314	271	220
1–9 writers)	“	“	6564	2247	960		not supported		5473	1544	1230
PWCS (3 writers,	955505	958033	752	2279	1628	968	348	306	738	2209	1265
1–7 replicas)	“	“		timeout			not supported		1221	3857	2735

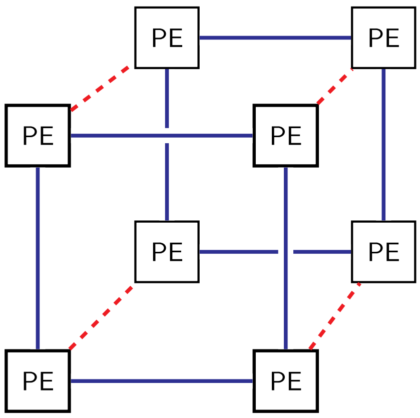


Fig. 25. A network system model with cube topology. The blue lines indicate wired connections, while the dashed red lines denote directed optical connections.

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 25 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. 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.

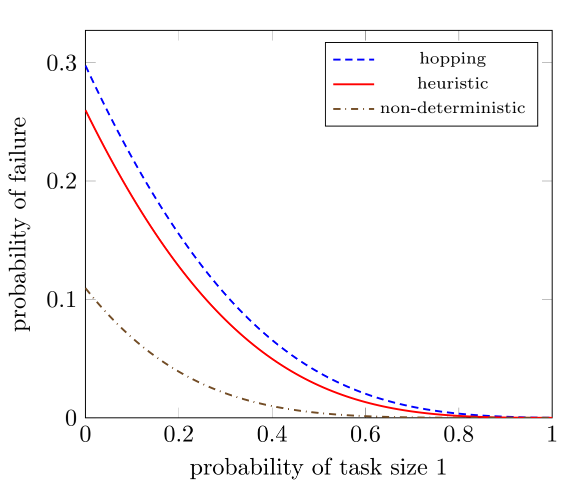
Non-deterministic. For a task that requires a number n of network links, this model non-deterministically selects one of all possible spanning trees of size n over the network topology.

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.

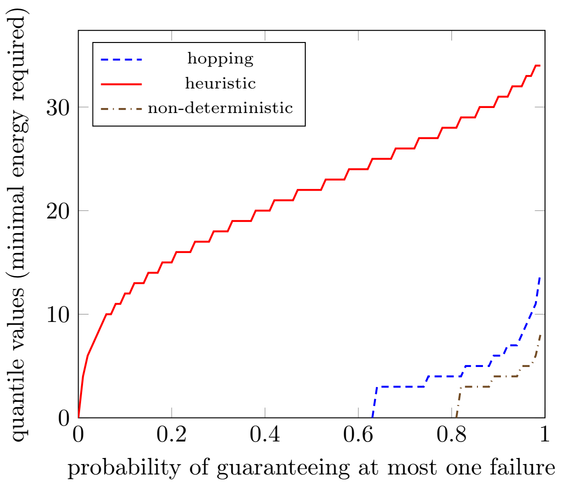
Heuristic. A simple heuristic for mapping tasks to PEs follows the "greedy" principle. In the first step, the first PE with the most remaining capacity among its adjacent links is selected. In the second step, the task is randomly distributed among the PEs adjacent to the selected PE. The probability 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, the model contains a corresponding optional feature. Thus, turning links on and off is handled by the feature controller. 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. The meta-programming facilities of ProFeat allowed us to define the model's behavior described above 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.

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 Figure 26a. 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. 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 of 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 26b 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 taking profit of automatic variable reordering mechanisms introduced in [KBC+16], which reduced the Mtbdd-representation of the models by around 40%.



(a) Minimal probability of having a mapping failure within 9 rounds, depending on the probability of task sizes 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

Fig. 26. Results for analyzing the net case study

6. Conclusions

We presented the language ProFeat for family-based modeling and analysis of probabilistic systems. System families comprise a feature-oriented model following the framework of [DBK15] and/or system instances induced by varying on parameters. ProFeat extends the input language of the prominent probabilistic model checker Prism with feature-based concepts, templates and parametrization as well as scripting constructs for meta-programming. To the best of our knowledge, ProFeat is the first modeling language for probabilistic feature-oriented systems, including support for dynamic product lines.

The ProFeat tool acts as a preprocessor that translates ProFeat models into standard Prism models. It provides basic support for partitioning the family model into sub-families enabling an all-in-one and one-by-one analysis, e.g., using the probabilistic model checker Prism [KNP11]. The ProFeat post-processor collects results for all family members and allows for presenting the results in a human-readable format.

The expressiveness of the input language as well as the practicality of the ProFeat tool have 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., [MTS$^+$14, CHS$^+$10]), this cannot be generalized to arbitrary families, e.g., when only a few common behaviors exist within the family members. Moreover, we illustrated that the feature-based modeling approach is also useful for modeling single systems, i.e., not describing a family of systems. The feature-based model served as input for a cost-utility analysis yielding insights to the tradeoff between energy consumption and performance. As ProFeat fully automatically translates into the Prism language, there are limited possibilities to influence the structure of the translated model, e.g., in order to adjust the variable ordering for a more compact symbolic model representation. However, ProFeat can rely and benefit from automatic variable-reordering mechanisms recently presented in [KBC$^+$16].

There are various directions for further work. For instance, partitioning the family model into arbitrary sub-families could be combined with symmetry-reduction methods to speed up the analysis. In the context of dynamic product lines it would be very interesting to synthesize feature controllers that are optimal with respect to some (multi-objective) quantitative criteria.

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. Icons by Freepik (flaticon.com).

References

[AH99] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[AH10] S. Apel and D. Hutchins. A calculus for uniform feature composition. *ACM Transactions on Programming Languages and Systems*, 32(5), 2010.

[AJTK09] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model superimposition in software product lines. In *ICMT'09*, volume 5563 of *LNCS*, pages 4–19. Springer, 2009.

[AK09] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[ARW$^+$13] S. Apel, A. von Rhein, P. Wendler, A. Groesslinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proc. of the 2013 Int. Conference on Software Engineering*, ICSE '13, pages 482–491. IEEE, 2013.

[ASW$^+$11] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Int. Conference on Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011.

[BdA95] A. Bianco and L. de Alfaro. Model checking of probabilistic and non-deterministic systems. In *FSTTCS'95*, volume 1026 of *LNCS*, pages 499–513, 1995.

[BDD$^+$14] C. Baier, M. Daum, C. Dubslaff, J. Klein, and S. Klüppelholz. *Energy-Utility Quantiles*, pages 285–299. Springer International Publishing, 2014.

[BEK$^+$13] C. Baier, B. Engel, S. Klüppelholz, S. Märcker, H. Tews, and M. Völp. A probabilistic quantitative analysis of probabilistic-write/copy-select. In *Proc. of the 5th NASA Formal Methods Symposium (NFM)*, LNCS, pages 307–321. Springer, 2013.

[BK98] C. Baier and M. Kwiatkoswka. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.

[BSRC10] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[CBH11] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76(12):1130–1143, 2011.

[CCH$^+$12] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with SNIP. *STTT*, 14(5):589–612, 2012.

[CCH$^+$13a] M. Cordy, A. Classen, P. Heymans, A. Legay, and P.-Y. Schobbens. *Model Checking Adaptive Software with Featured Transition Systems*, pages 1–29. LNCS. Springer, 2013.

[CCH$^+$13b] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. ProVeLines: a product line of verifiers for software product lines. In *17th Int. Software Product Line Conference (SPLC)*, pages 141–146. ACM, 2013.

[CCH$^+$14] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming*, 80:416–439, 2014.

[CCS$^+$13] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, 2013.

[CDKB16] P. Chrszon, C. Dubslaff, S. Klüppelholz, and C. Baier. *Family-Based Modeling and Analysis for Probabilistic Systems – Featuring ProFeat*, pages 287–304. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[CFM$^+$93] E. M. Clarke, M. Fujita, P. C. McGeers, K. L. McMillan, J. C.-Y. Yang, and X.-J. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. International Workshop on Logic & Synthesis*, 1993.

[CHE05] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[CHS$^+$10] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *32nd Int. Conference on Software Engineering (ICSE)*, pages 335–344. ACM, 2010.

[CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2001.

[CSHL13] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *Proc. of the 2013 Int. Conference on Software Engineering*, ICSE '13, pages 472–481. IEEE Press, 2013.

[Daw04] C. Daws. Symbolic and parametric model checking of discrete-time Markov chains. In *Theoretical Aspects of Computing - ICTAC 2004*, volume 3407 of *LNCS*, pages 280–294, 2004.

[DBK15] C. Dubslaff, C. Baier, and S. Klüppelholz. Probabilistic model checking for feature-oriented systems. *Transactions on Aspect-Oriented Software Development XII*, 8989:180–220, 2015.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[DJJ$^+$15] C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, H. Bruintjes, J.-P. Katoen, and E. Abraham. PROPhESY:

a probabilistic parameter synthesis tool. In *27th Int. Conference on Computer Aided Verification (CAV)*, volume 9206 of *LNCS*, pages 214–231, 2015.

[DMFM10] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A dynamic software product line approach using aspect models at runtime. In *Proc. of the 1st Workshop on Composition and Variability*, 2010.

[DS11] F. Damiani and I. Schaefer. Dynamic delta-oriented programming. In *Proc. of the 15th Int. Software Product Line Conference*, SPLC '11. ACM, 2011.

[FGT12] A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Aspects of Computing*, 24(2):163–186, 2012.

[GH03] H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. In *PFE*, pages 435–444, 2003.

[GLS08] A. Gruler, M. Leucker, and K. D. Scheidemann. Modeling and model checking software product lines. In *10th Int. Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, LNCS, pages 113–131, 2008.

[GS13] C. Ghezzi and A. M. Sharifloo. Model-based verification of quantitative non-functional properties for software product lines. *Information & Software Technology*, 55(3):508–524, 2013.

[HHWZ10] E. M. Hahn, H. H., B. Wachter, and L. Zhang. PARAM: A model checker for parametric Markov models. In *22nd Int. Conference on Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 660–664, 2010.

[HHZ11] E. M. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric Markov models. *Software Tools and Technology Transfer*, 13(1):3–19, 2011.

[HRS13] M. Heiner, C. Rohr, and M. Schwarick. *MARCIE – Model Checking and Reachability Analysis Done Efficiently*, pages 389–399. Springer Berlin Heidelberg, 2013.

[Kat93] S. Katz. A superimposition control construct for distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(2):337–356, 1993.

[KBC+16] J. Klein, C. Baier, P. Chrszon, M. Daum, C. Dubslaff, S. Klüppelholz, S. Märcker, and D. Müller. Advances in symbolic probabilistic model checking with PRISM. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Proceedings*, pages 349–366, 2016.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University, 1990.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[KNP12] M. Z. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *Proc. Quantitative Evaluation of Systems (QEST'12)*, pages 203–204. IEEE, 2012. `https://github.com/prismmodelchecker/prism-benchmarks/`.

[KST14] M. Kowal, I. Schaefer, and M. Tribastone. Family-based performance analysis of variant-rich software systems. In *Fundamental Approaches to Software Engineering*, volume 8411 of *LNCS*, pages 94–108, 2014.

[KZH+11] J.-P. Katoen, I.S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Performance Evaluation*, 68(2):90–104, 2011.

[LPT09] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *24th IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 269–280. IEEE, 2009.

[MTS+14] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, and G. Saake. An overview on analysis tools for software product lines. In *18th Int. Software Product Lines Conference (SPLC)*, pages 94–101. ACM, 2014.

[PR01] M. Plath and M. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84, 2001.

[RAN+15] G. N. Rodrigues, V. Alves, V. Nunes, A. Lanna, M. Cordy, P.-Y. Schobbens, A. M. Sharifloo, and A. Legay. Modeling and verification for probabilistic properties in software product lines. In *High Assurance Systems Engineering (HASE)*, pages 173–180. IEEE, 2015.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93).*, pages 42–47, 1993.

[Seg08] S. Segura. Automated analysis of feature models using atomic sets. In *SPLC (2)*, pages 201–207, 2008.

[tBLLV15] M. H. ter Beek, A. Legay, A. Lluch-Lafuente, and A. Vandin. Statistical analysis of probabilistic models of software product lines with quantitative constraints. In *19th Int. Conference on Software Product Line (SPLC)*, pages 11–15. ACM, 2015.

[TKB+14] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[vR16] Alexander von Rhein. *Analysis Strategies for Configurable Systems.* PhD thesis, University of Passau, 2016.