Feature-Family-Based Reliability Analysis of Software Product Lines

André Lanna[a], Thiago Castro[a,b], Vander Alves[a,d], Genaina Rodrigues[a], Pierre-Yves Schobbens[c], Sven Apel[d]

[a] *Computer Science Department, University of Brasília. Campus Universitário Darcy Ribeiro - Edifício CIC/EST, 70910-900, Asa Norte, Brasília -DF, Brazil*
[b] *Systems Development Center, Brazilian Army. QG do Exercito - Bloco G - 2o. andar, Setor Militar Urbano, Brasília - DF, Brazil*
[c] *PReCISE, NaDI, Faculty of Computer Science, University of Namur. Rue Grandgagnage 21, 5000 Namur, Belgium*
[d] *Department of Informatics and Mathematics, University of Passau Innstr. 33, 94032, Passau, Germany*

Abstract

Context: Verification techniques are being applied to ensure that software systems achieve desired quality levels and fulfill functional and non-functional requirements. However, applying these techniques to software product lines is challenging, given the exponential blowup of the number of products. Current product-line verification techniques leverage symbolic model checking and variability information to optimize the analysis, but still face limitations that make them costly or infeasible. In particular, state-of-the-art verification techniques for product-line reliability analysis are enumerative which hinders their applicability, given the latent exponential blowup of the configuration space.

Objective: The objectives of this paper are the following: (a) we present a method to efficiently compute the reliability of all configurations of a compositional or annotation-based software product line from its UML behavioral models, (b) we provide a tool that implements the proposed method, and (c) we report on an empirical study comparing the performance of different reliability analysis strategies for software product lines.

Method: We present a novel *feature-family-based* analysis strategy to compute the reliability of all products of a (compositional or annotation-based) software product line. The *feature-based* step of our strategy divides the behavioral models into smaller units that can be analyzed more efficiently.

The *family-based* step performs the reliability computation for all configurations at once by evaluating reliability expressions in terms of a suitable variational data structure.

Results: Our empirical results show that our feature-family-based strategy for *reliability* analysis outperforms, in terms of time and space, four state-of-the-art strategies (product-based, family-based, feature-product-based, and family-product-based) for the same property. It is the only one that could be scaled to a 2^{20}-fold increase in the size of the configuration space.

Conclusion: Our feature-family-based strategy leverages both feature- and family-based strategies by taming the size of the models to be analyzed and by avoiding the products enumeration inherent to some state-of-the-art analysis methods.

Keywords: Software Product Lines, Software Reliability Analysis, Parametric Verification

1. Introduction

Achieving a high quality, low costs, and a short time to market are the driving goals of software product line engineering. A software product line [11] is created to take advantage of the commonalities and variabilities of a specific application domain, by reusing artifacts when instantiating individual software products (a.k.a. *variants* or simply *products*). A domain variability is expressed in terms of features, which are distinguishable characteristics relevant to some stakeholder of the domain [13]. Nowadays software product line engineering is widely accepted in both industry [46, 31] and academia [1, 11, 26, 38].

Quality assurance of product lines has drawn growing attention [32, 42]. Particularly, model checking techniques for product lines explore the space of all products in a product line by searching for execution states where functional [7, 8, 9] or non-functional [17, 21, 28, 34, 39] properties are violated [5]. Nevertheless, employing model checking techniques to verify product lines is a complex task, posing a twofold challenge [8]: (1) the number of variants may grow exponentially with the number of features, which gives rise to an *exponential blowup* [10, 9, 4, 1]; and (2) model checking is inherently prone to the *state-explosion problem* [3, 5]. Therefore, model checking all products of a product line is often not feasible in practice [42].

In previous work, model checking techniques have been applied to analyze probabilistic properties of product lines, in particular, reliability [21, 39, 34]. These approaches attenuate the complexity of analyzing probabilistic properties by exploiting, to some extent, reuse in modeling and analysis. On the one hand, non-compositional techniques exploit commonalities across products resulting into a single model representing the variability and the behavior of the product line as a whole (covering the behaviors of all products), but it may not scale due to the large state space of models generated by this modeling approach [39, 34]. On the other hand, a compositional alternative is to create and analyze isolated models for each feature and then evaluate them jointly for each configuration [21]. This approach is space-efficient, but faces an exponential blowup by enumerating all valid configurations, which leads to time scalability issues. In essence, both approaches have limitations in reusing analysis effort in product lines. As a result, state-of-the-art verification techniques for product-line reliability analysis are enumerative (product-based), which hinders their applicability, given the latent exponential blowup of the configuration space. Consequently, unwanted redundant computational effort is wasted on modeling and analyzing product line's models [21].

As our key contribution, we present a strategy to efficiently compute the reliability of all products of both compositional and annotation-based product lines, without enumerating and analyzing each of these products. Our strategy employs a divide-and-conquer approach in which pre-computed reliabilities of individual features are combined to compute the reliability of the whole product line in a single pass. In a nutshell, in the first step, a feature-based analysis is applied to build a probabilistic model per feature and to analyze each such model using a parametric model checker, returning expressions that describe the reliability of features. Parameters in a feature's reliability expression represent the reliabilities of other features on which it depends at runtime. In the second step, our strategy performs a family-based step to evaluate each expression in terms of Algebraic Decision Diagrams [2] that are used to encode the knowledge about valid feature combinations and the mapping to their corresponding reliabilities. Since our strategy is a combination of feature-based and family-based analyses, it is effectively a feature-family analysis strategy [42], being the first of its kind for reliability analysis.

We implemented our approach in the tool ReAna (which stands for **Re**liability **Ana**lysis), whose source code is publicly available as free and open-source software[1]. The tool takes as input a set of UML behavioral models annotated with reliability information and a feature model of a product line, and it outputs the reliability values for the valid configurations (i.e., products) of this product line. To evaluate the time-space complexity, we performed 120 experiments to empirically compare our feature-family-based analysis strategy with the following state-of-the-art strategies [42]: product-based, family-based, feature-product-based, and family-product-based. We implemented these alternative strategies as variations of ReAna and used them to analyze twenty variants of each of six publicly available product-line models: a system for monitoring an individual's health [39], control systems for mine pumps [29] and lifts [37], an email system [43], inter-cloud configuration [19], and a game [43]. These product lines have been used widely as benchmarks; they have configuration spaces of different sizes, ranging from dozens to billions of billions of products.

Our experiment consisted of progressively increasing the number of features and the size of the behavioral models for each of the product lines, analyzing each of the evolved product lines with all analysis strategies. Our results indicate that the feature-family-based strategy has the best performance in terms of time and space, being the only one that could be scaled to a 2^{20}-fold increase in the size of the configuration space for *reliability* analysis when compared to four state-of-the-art strategies for the same property: product-based, family-based, feature-product-based, and family-product-based.

In summary, the contributions of this paper are the following:

- We introduce a novel feature-family-based strategy for reliability analysis that analyzes each feature in isolation and combines the resulting pieces of information to compute the reliability of a given product line (Section 3);
- We provide a novel tool, called ReAna, implementing such feature-family-based strategy, to carry out the analysis of reliability of a product line from its UML behavioral diagrams and its feature model (Section 4.1);
- We report on an empirical study comparing the performance of our

[1] `https://github.com/SPLMC/reana-spl/`

feature-family-based strategy to other state-of-the-art analysis strategies, implemented as an extension of our ReAna tool (Section 4.3).

Supplementary material, including the ReAna tool and its extensions (which include all evaluation strategies considered in this work), as well as models used in our empirical evaluation and respective experimental results are publicly available for replication purposes at `http://splmc.github.io/scalabilityAnalysis/`.

2. Background

In this section, we provide an overview of fundamental concepts related to our work and a running example to guide the presentation of our approach in later sections. We assume the reader is familiar with software product lines [11, 38] and discrete-time Markov chains (DTMC) [3].

2.1. Reliability Analysis and FDTMC

Probabilistic verification techniques have been used in the past to substitute the concept of absolute correctness by bounds on the probability that certain behavior may occur. Based on probabilistic models, it is possible to specify probabilistic system behavior due to, e.g., intrinsically unreliable hardware components and environmental characteristics. Reliability can be defined as a probabilistic existence property [22], in the sense that it is given by the probability of eventually reaching some set of *success* states in a probabilistic behavioral model of a system. (In our setting, we define success to mean that all tasks of interest have been accomplished as intended.)

Discrete-time Markov Chain (DTMC) is a well-known formalism to model such probabilistic behavior. In a DTMC, the reachability probability is defined as the sum of probabilities for each possible path that starts in an initial state and ends in a state belonging to the set of target states [3]. Thus, to compute reliability, we label success states with the atom *"success"* and compute the reachability probability of success states, expressed as $P_{=?}[\Diamond\text{“}\mathit{success}\text{”}]$ in the query language of the PARAM model checker [24].

To analyze the behavior of a product line, it is useful to embed its inherent variability in such a probabilistic model. A possible approach is to use *parametric DTMCs* (PDTMC) [15], which augment DTMCs with transition probabilities that can be expressed as variables. A PDTMC is a DTMC whose probability matrix takes values from a set X of strictly positive parameters. A PDTMC gives rise to a family of DTMCs by instantiating the formal parameters to values with an instantiation function $\kappa : \mathbb{Q}_+ \cup X \mapsto [0,1]$. For a parametric DTMC D_X and an instantiation function $\kappa, \kappa(D_X)$ denotes the DTMC whose probability matrix is given by instantiating D_X's formal parameters. For PDTMCs, the reliability analysis problem can be solved by a *parametric probabilistic reachability* algorithm [23], which outputs a rational expression (a fraction of two polynomials) on the same variables as the ones in the input parametric model. The idea behind this technique is that evaluating the variables in the rational expression yields the reliability value of the DTMC that would be obtained by an equivalent evaluation of the variables in the PDTMC. However, this behavioral representation does not take a variability model (e.g., a feature model) into account, and thus is not sufficient for representing *possible* behavior in a product line (i.e, behavior of actual products).

Featured Discrete-time Markov Chains (FDTMC) [39] are probabilistic models that properly handle product-line variability. They can be thought as DTMCs that, instead of transition probabilities, have transition *probability profiles*. These profiles are functions $[\![FM]\!] \rightarrow [0,1]$ that map a configuration to a probability value, where $[\![FM]\!]$ denotes the set of valid configurations of the feature model FM. Rodrigues et al. [39] proposed a method to encode an FDTMC as a PDTMC, enabling its analysis by off-the-shelf parametric model checkers. In the present work, we leverage the view of Rodrigues et al. [39] of FDTMCs as PDTMCs for the purpose of compositional reliability analysis.

2.2. Software Product Line Analysis

Several analysis techniques have been proposed by researchers for software product lines, each one taking a particular property into account. To help researchers and practitioners understand the similarities and differences among such techniques, Thüm et al. [42] propose a classification of the existing techniques, which we follow in this work. In our context, a *product-based* reliability analysis operates only on derived (non-variable) UML behavioral models, whereas the variability model may be used to generate the models. As it is a brute-force strategy, it is only feasible for product lines with few products. In contrast, the *family-based* strategy for reliability analysis operates over variant-rich UML behavioral models and incorporates the knowledge about valid feature combinations. In a *feature-based* analysis strategy, the reliability of UML behavioral models related to each individual feature is analyzed in isolation from the others, i.e., interactions among features and the knowledge about valid feature combinations are not incorporated into the analysis.

Other evaluation strategies may be formed by combining two or more strategies aforementioned [42]. For instance, a *feature-product* analysis consists of a feature-based analysis step followed by a product-based analysis, such that the result of the feature-based analysis is reused by the product-based analysis. In the context of reliability, the reliability of UML behavioral models related to each feature is first evaluated in isolation and then the analysis result is reused when enumerating and evaluating the reliability of each non-variant UML behavioral model of the product line.

Although other combined evaluation strategies are possible, the aforementioned strategies suffice as contrast to our proposed strategy. For more information regarding the remaining strategies, please refer to Thüm et al. [42].

2.3. Running Example

To illustrate the concepts presented throughout this paper, we introduce an example of a simple product line within the medical domain, for which reliability is considered the major requirement [25]: the Body Sensor Network (BSN) product line is a network of connected sensors that capture vital signs from an individual and send them to a central system to analyze the collected data and identify critical health situations [39]. This product line has software components that interpret data provided by the sensors and analyze an individual's health situation, as well as components for data persistence in a database or memory. The set of possible configurations for this product line is defined by its feature model (Figure 1), in which wireless sensors are grouped by the feature *Sensor*, software components for interpreting health information are grouped by the feature *SensorInformation*, and the alternatives for data persistence are grouped by the feature *Storage*.

To continuously monitor an individual's health situation, the BSN product line has a control loop comprised of four activities: capture data coming from sensors, process information about the health condition, identify health goal changes, and reconfigure the system if necessary. This control loop represents the coarse-grained behavior of the BSN product line and it is modeled by the activity diagram shown in Figure 2a, with each activity being represented in detail by a sequence diagram involving the components and their behavior. Therefore, every product instantiated from the BSN product line executes this control loop and, whenever the individual's health condition

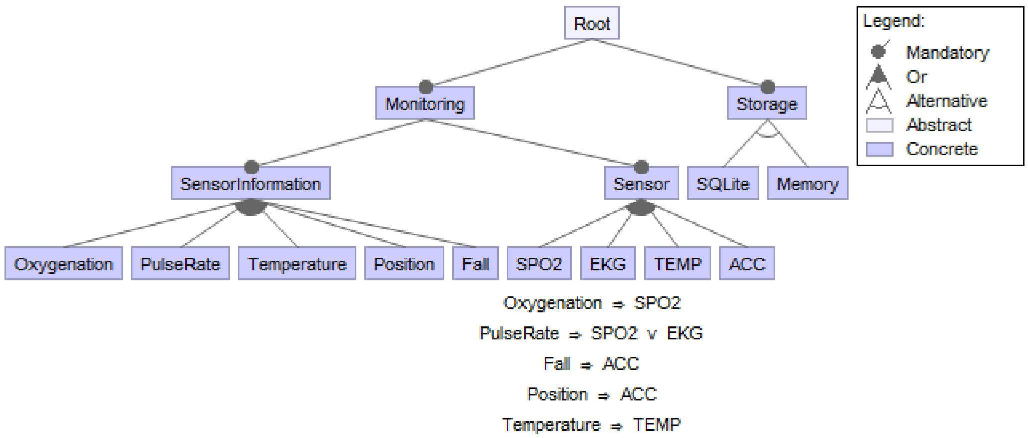
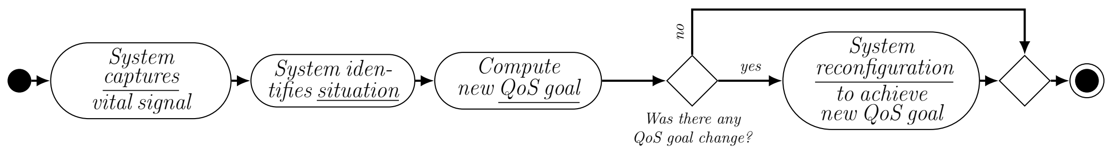


Figure 1: BSN-SPL Feature Model

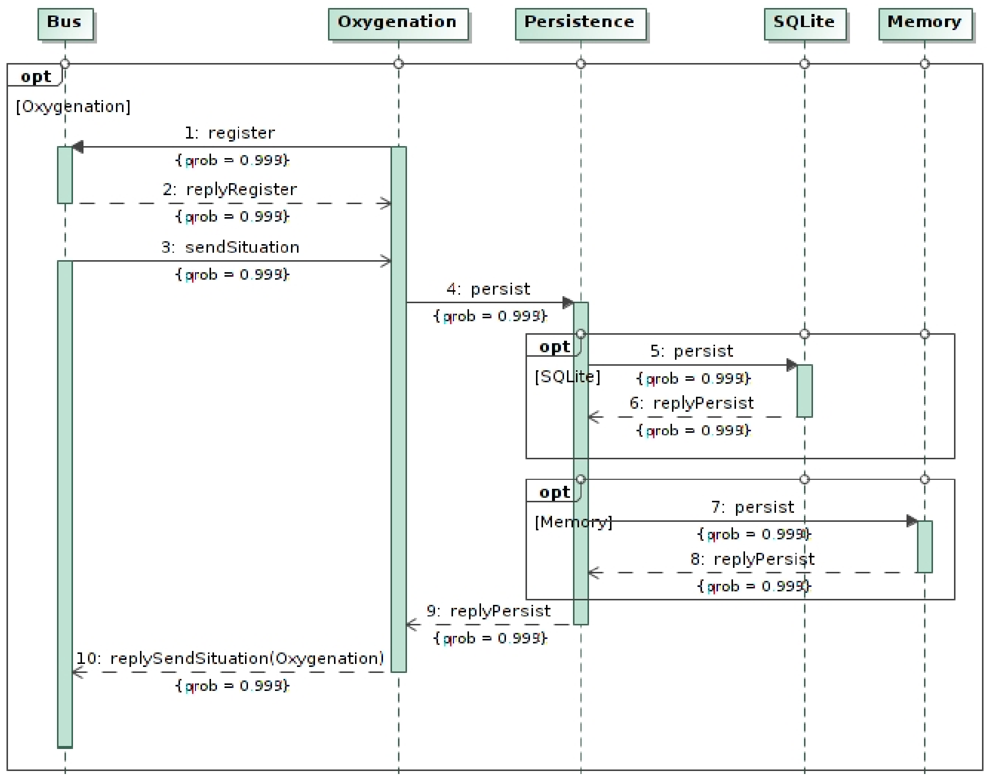
changes and this triggers a quality-of-service goal change, another product is instantiated from this product line with the desired behavior to reach the desired quality-of-service goal. The sequence diagrams play the role of representing the behavioral variability where necessary, by means of guard conditions involving the presence of features (a.k.a *presence conditions* [14]).

For instance, Figure 2b presents an excerpt of the sequence diagram associated with the activity *system identifies situation* (Figure 2a). This activity consists of processing and persisting data regarding the individual's health condition, in particular sensor information, represented by feature *SensorInformation* and its child features in Figure 1. Figure 2b depicts the behavior associated with the computation and persistence of the individual's oxygenation. Such behavior is defined by the messages exchanged between five software components, whose roles are data processing ($Oxygenation$) and persistence ($Persistence$, $SQLite$ and $Memory$—$Persistence$ dispatches calls to the concrete persistence engines), and components for communication and coordination (Bus). Each message is named according to its task and has an associated probability value `prob` to represent the reliability of the channel between the components comprising the interaction. The reliability is given by the product of (a) the probability that the required message arrives at the receiver component and (b) the receiver component's reliability (i.e., the probability that it performs the required task without failure). For the BSN product line, we assume that all channels have reliability 0.999.

The guard condition at the top level of the sequence diagram presented in Figure 2b is the atomic proposition `Oxygenation`. This means that the



(a) Activity Diagram representing the control loop of BSN-SPL



(b) Sequence diagram (excerpt) associated with the activity *system identifies situation*, for processing and persisting Oxygenation information.

Figure 2: Behavioral diagrams for BSN-SPL

enclosed behavior is associated with the presence of the *Oxygenation* feature in a given configuration. This behavior, in turn, has two variants, according to the chosen mechanism for data persistence. The optional fragment whose guard condition is `SQLite` models the behavior of persisting data in a database whenever feature *SQLite* is part of a configuration. Likewise, the optional fragment associated to the presence of the feature *Memory* (i.e., the fragment with the `Memory` guard) models persistence on secondary memory.

Intuitively, the reliability of the BSN-SPL in terms of the UML behavioral diagrams shown in Figure 2 is defined by the probability of reaching the final elements of both activity (Figure 2a) and sequence (Figure 2b) diagrams without any error occurrence. This probability is given by the serial execution of the behavioral elements along the possible paths from the first until the final element in both diagrams. In Figure 2a, for instance, there are two possible executions leading to the end state: the first one considers that a reconfiguration is necessary to accomplish a new QoS goal, whereas the other bypasses the reconfiguration activity. The reliability for such diagram is the sum of the probabilities of both executions, considering that the reliability of each individual activity is represented by a variable named after its configuration parameter. Thus, assuming that the decision to reconfigure the BSN is taken 50% of the times, the reliability computed for the model represented in Figure 2a is given by

$$\begin{aligned} R(BSN) &= rCapture \cdot rSituation \cdot rQoSGoal \cdot 0.5 \\ &\quad + rCapture \cdot rSituation \cdot rQoSGoal \cdot rReconfiguration \cdot 0.5 \end{aligned}$$

Similarly, the reliability of the sequence diagram in Figure 2b is given by the probability that all messages are transmitted and processed without errors (the probability for any such message is noted in the corresponding arrow). The reliability of the `Oxygenation` fragment is then given by

$$\begin{aligned} R(Oxygenation) &= 0.999 \cdot 0.999 \cdot 0.999 \cdot 0.999 \\ &\qquad \cdot rSQLite \cdot rMemory \cdot 0.999 \cdot 0.999 \\ &= 0.999^6 \cdot rSQLite \cdot rMemory \end{aligned}$$

Similar to activities in the computation of *R(BSN)*, the reliability values of the fragments associated to the features *SQLite* and *Memory* are represented by variables. The reliability of each of these inner fragments is computed in the same fashion, leading to

$$R(SQLite) = R(Memory) = 0.999 \cdot 0.999 = 0.999^2$$

Although the reliabilities of the inner fragments are constant, we are not able to inline these values into the expression for *R(Oxygenation)*. Indeed, according to the feature model in Figure 1, features *SQLite* and *Memory* are alternative, meaning that *exactly one* of them is ever present in a given configuration. Thus, we leverage variables in the reliability expression to also encode product-line variability: whenever *SQLite* is present and *Memory* is absent, for instance, we evaluate *rSQLite* as *R(SQLite)* and *rMemory* as 1.

Note that the dynamic behavior of the BSN does not affect our approach to reliability analysis, since we only consider the execution of tasks up to reconfiguration (Figure 2a). Moreover, our approach is entirely based on design-time artifacts. For a deeper discussion on how the BSN is engineered for reconfiguration and how the reliability computation affects this dynamic behavior, please refer to the work by Pessoa et al. [36]

3. Feature-Family-based Reliability Analysis

In this section, we present our approach to evaluate the reliability property of product lines following a feature-family-based strategy [42]. It consists of three key steps, as shown in Figure 3.

First, the transformation step maps UML behavioral diagrams with variability into a graph structure called Runtime Dependency Graph (RDG), whose nodes represent the behavioral fragments and store corresponding FDTMCs (i.e. the probabilistic behavioral model), meanwhile the edges represent the runtime dependencies between such models. Next, the feature-based evaluation step analyzes each FDTMC with respect to a reliability property, with the support of a parametric model checker. Each FDTMC is analyzed in isolation, by abstracting the existing runtime dependencies as parameters. This results in rational expressions [23] (hereafter referred to simply as *expressions*), each giving the reliability of an FDTMC as a function of the reliabilities of the FDTMCs on which it depends. Lastly, the family-based evaluation step follows a topological sorting of the runtime dependency graph, computing the reliability value of each configuration by evaluating the expression in each node and reusing the evaluation results previously computed for the nodes on which it depends. This step also considers the variability model of the product line in question to prune invalid configurations. The following subsections describe these steps in detail, guided by the example of Section 2.3.

3.1. Transformation

To perform reliability analysis of a given product line, our approach first composes its inherent variability and probabilistic behavior into a Runtime Dependency Graph (RDG), which is then used for analysis in further steps. The probabilistic behavior can be derived from UML behavioral models,

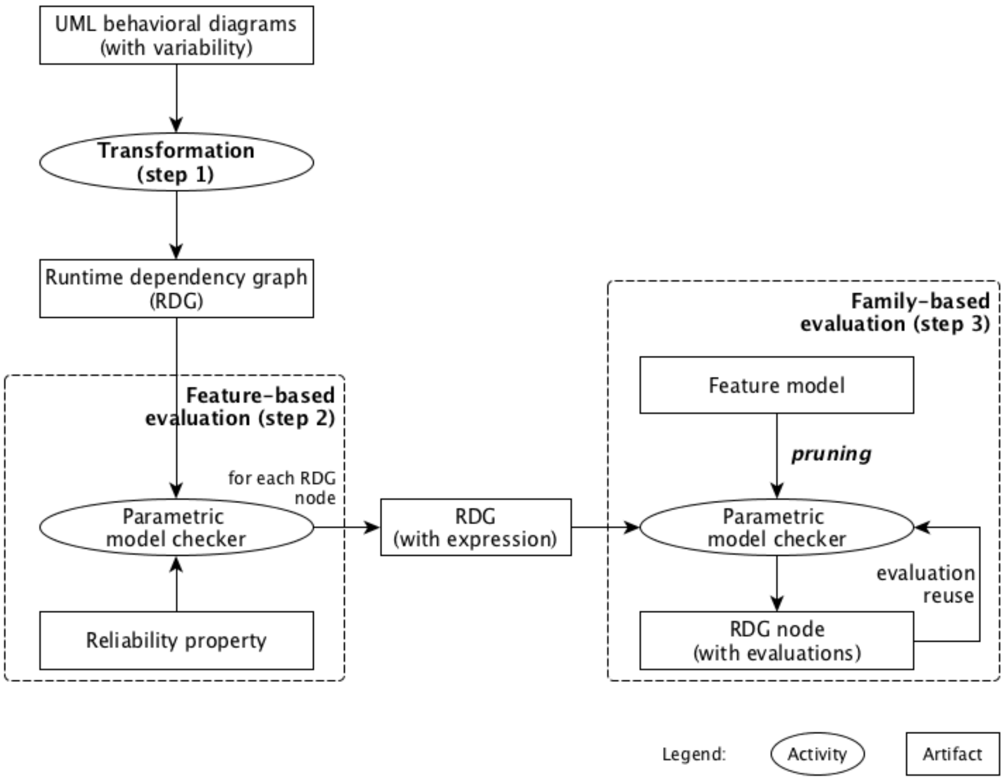


Figure 3: Feature-family-based approach for efficient reliability analysis of product lines.

representing the runtime interactions between software components, enriched with reliability information for such interactions. Next, we provide details on the behavioral models, the RDG, and the transformation of the former into the latter.

3.1.1. Behavioral Models

In our approach, the coarse-grained behavior of a product line is represented by a UML activity diagram, with each activity being refined into a sequence diagram [39]. The activity diagram is useful for representing whether the activities are performed in a sequential or parallel manner, whereas sequence diagrams represent how the probabilistic behavior of the interactions between software components varies according to the configuration space of the product line. To represent probabilistic behavior, each message in a sequence diagram is annotated with a probability value that represents the reliability of the channel—i.e., the probability that the interaction succeeds—by using the UML MARTE profile [35] (e.g., *prob* tags in Figure 2b).

As an example, Figure 2a shows a UML activity diagram describing, at a high level, the behavior of all products of the BSN product line. The behavior corresponding to the activity *system identifies situation* is modeled by an associated sequence diagram, partially depicted in Figure 2b.

Without loss of generality, behavior variability is defined by *behavioral fragments*, each of which can be an activity diagram (that has an associated sequence diagram), a sequence diagram, or an optional combined fragment within a sequence diagram such that this fragment has a guard condition denoting presence condition [14]. These conditions are propositional logical statements defined over features, that denote the set of configurations for which the guarded behavior is present. Optional combined behavioral fragments can be nested, which allows representing behavioral variability at several levels.

Note that the behavioral variability expressed by optional fragments may be implemented in two distinct ways: 1) in case the fragment's guard condition is expressed by an atomic proposition (i.e., a single feature), the feature may be implemented in its own module, which characterizes a compositional product line; 2) if the guard condition is a propositional formula comprising two or more features, such tangled behavior can be implemented in an annotation-based style by using, for example, the `#ifdef` and `#endif` macros of the C preprocessor. Therefore, our approach can be applied to analyze both compositional and annotation-based software product lines.

The sequence diagram shown in Figure 2b presents three behavioral fragments whose presence conditions are the atoms `Oxygenation`, `Memory`, and `SQLite`. The outermost behavioral fragment represents the optional behavior for processing the oxygenation information in the BSN product line, and it varies according to two nested behavioral fragments. These latter are optional combined fragments related to the features *SQLite* and *Memory* of the feature model in Figure 1 and, jointly with this model's constraints, ultimately represent alternative behavior for data persistence.

3.1.2. Runtime Dependency Graphs

A Runtime Dependency Graph (RDG) is a behavioral representation for variable systems, which combines the configurability view of a product line (expressed by presence conditions) with its probabilistic behavior (expressed by FDTMCs). Formally, it can be defined as follows.

Definition 1 (RDG)**.** *A Runtime Dependency Graph* $\mathcal{R}$ *is a directed acyclic graph* $\mathcal{R} = (\mathcal{N}, \mathcal{E}, x_0)$*, where* $\mathcal{N}$ *is a set of nodes,* $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ *is a set of directed edges that denote a dependency relation, and* $x_0 \in \mathcal{N}$ *is the* root *node with in-degree 0. An RDG node* $x \in \mathcal{N}$ *is a pair* $x = (m, p)$*, where* m *is an FDTMC representing a probabilistic behavior and* p *is a propositional logic formula that represents the presence condition associated with* m*.*

To build an RDG for a software product line, we extract the configurability and probabilistic information only from the UML behavioral diagrams, such that each RDG node is associated with an FDTMC derived from a behavioral fragment and its presence condition. Since we consider that the UML activity diagram represents the product line's *coarse-grained behavior* executed by all products and, each activity is further refined (detailed) into its respective sequence diagram. Thus, the behavioral variability is not considered at the representation at system level, which implies its related RDG nodes have *true* as presence condition (i.e., it is satisfied for all products). Edges represent dependencies between nodes, which are due to refinement or nesting relations between the respective behavioral fragments. RDG nodes that do not depend on any other node are called *basic*. The ones with dependencies are called *variant* nodes, which are represented with outgoing edges directed to the RDG nodes on which they depend.

The structure of UML sequence diagrams is tree-like, which suggests a tree could be a better model of their dependencies. Nonetheless, applications sometimes have behavioral fragments replicated throughout UML models. For instance, the data persistence behavior in Figure 2b is present in all fragments that denote sensor information processing. In our approach, redundant fragments are represented by a single RDG node, with as many incoming edges as its number of replications. When performing this reuse, the resulting graph will be acyclic, because the original UML model is a finite hierarchy.

Figure 7a illustrates an excerpt of the BSN product line's RDG that represents the behavioral fragment of Figure 2b. As the fragments related to the features *SQLite* and *Memory* are nested inside the fragment related to feature *Oxygenation*, the RDG for this fragment represents the dependencies between their respective nodes. The behavioral fragment related to *Oxygenation* is part of the sequence diagram representing the behavior of the activity *system identifies situation*. Therefore, this relation is also represented by the edge from the node *rSituation* to the node *rOxygenation*. For brevity, we

```
1 RDGNode transformAD(ActivityDiagram ad) {
2     RDGNode root = new RDGNode(ad.id);
3     root.model = adToFDTMC(ad);
4     root.presenceCondition = true;
5     for (Activity act : ad.activities) {
6         root.addDependency(transformSD(act.sequenceDiagram));
7     }
8     return root;
9 }
```

Listing 1: Activity diagram transformation

do not represent the internal structure of the nodes and the remaining RDG nodes (indicated by ellipses in Figure 7a).

3.1.3. From Behavioral Models to RDG

The transformation from behavioral models to an RDG can be described at two abstraction levels: the RDG topology and the generation of probabilistic models. Listings 1 and 2 both depict the transformation process from the topological point of view. Note that this step relies on uniquely generated identifiers for the behavioral models, which are then used as identifiers for the respective RDG nodes.

The process starts by calling the `transformAD` method (Listing 1), passing as argument the single activity diagram that embodies the coarse-grained behavior of the product line. This method creates the *root* node (Line 2), setting its presence condition to `true` (i.e., the overall behavior must always be present; Line 4). The root's probabilistic model is then generated by processing the input diagram with the `adToFDTMC` method (Line 3), to which we will come back later. We then create an RDG node for each sequence diagram that refines an activity (denoted by the property `act.sequenceDiagram`), subsequently creating edges that mark them as dependencies of the *root* node (Line 6). Note that the *root* node is the only RDG node created by the `transformAD` method, so the root's FDTMC models the behavior represented by the activity diagram.

The creation of RDG nodes for sequence diagrams is similar: the method `transformSD` (Listing 2) takes a behavioral fragment as input and then creates a new RDG node whose FDTMC is derived by the `sdToFDTMC` method (Line 4). In this case, since behavioral fragments encode variability, their

```
1 RDGNode transformSD(BehavioralFragment sd) {
2     RDGNode thisNode = new RDGNode(sd.id);
3     thisNode.presenceCondition = sd.guard;
4     thisNode.model = sdToFDTMC(sd);
5     for (BehavioralFragment frag : sd.optFragments) {
6         thisNode.addDependency(transformSD(frag));
7     }
8     return RDGNode.reuse(thisNode);
9 }
```

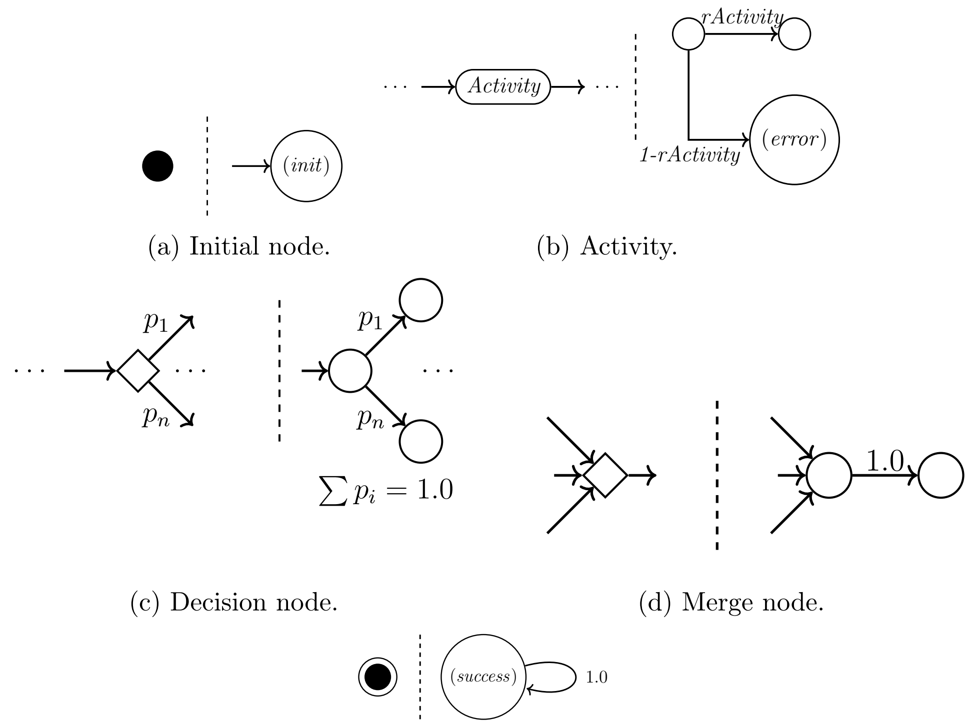
Listing 2: Sequence Diagram transformation

guard is assigned as the presence condition of the newly created node (Line 3). As with refined activities, we create RDG nodes for nested behavioral fragments and set them as dependencies of the node at hand (Line 6).

The reuse of behavior briefly mentioned in Section 3.1.2 is performed by calling the static method `RDGNode.reuse` (Listing 2, Line 8). This function maintains a registry of all RDG nodes created, and then searches among them for one that we consider *equivalent* to the one just created. This notion of equivalence is comprised of three conditions: (a) equality of presence conditions; (b) equality of FDTMCs; and (c) recursively computed equivalence of dependencies.

At the abstraction level of generating probabilistic models, the transformation of activity and sequence diagram elements into FDTMCs consists of applying transformation templates for each considered behavioral element represented on such diagrams. These templates are depicted by the UML behavioral element being transformed (left-hand side of the dashed line in Figures 4 and 5) and by its resulting probabilistic structure (right-hand side).

Figure 4 shows the templates for transforming an activity diagram into an FDTMC. The initial node of the activity diagram becomes the first state in the FDTMC and thus it is labeled as *init* (Figure 4a). Each activity abstracts behavior that is modeled with more detail in an associated sequence diagram. Accordingly, we abstract the reliability of an activity as a parameter that acts as a placeholder for the reliability of the corresponding sequence diagram. Therefore, each activity is represented in an FDTMC by the structure depicted in Figure 4b, where the upper edge denotes the reliability value of the associated sequence diagram (the parameter $rActivity$) and the lower



(a) Initial node. (b) Activity.

(c) Decision node. (d) Merge node.

(e) End node.

Figure 4: Templates for transforming activity diagram elements into FDTMCs.

edge denotes the probability of failure ($1 - rActivity$, the complement of the success probability).

A decision node in an activity diagram denotes a choice between alternative behaviors, each one represented by an outgoing transition directed to another activity diagram element (Figure 4c). Each transition has an associated guard condition that must be satisfied to allow the execution of its subsequent behavior. This decision is taken at runtime, but a domain expert is able to define the probability for each alternative. Therefore, the transformation of a decision node results into an FDTMC structure comprised of a state with as many outgoing transitions as the number of the direct subsequent elements of the decision node. Each outgoing transition has a probability value assigned by the domain expert, and these probabilities must sum up to 1 [2].

[2]States without variability are regular DTMC states, so the stochastic property holds:

A merge node denotes a place where different branches of an activity diagram join just before the execution of the next element proceeds. For each merging branch, there is an incoming edge directed to the merge node, and only one outgoing edge indicating the execution may proceed. The transformation of a merge node results into an FDTMC structure consisting of two states and one edge, as shown in Figure 4d. The first created state represents a synchronization point for a number of previous branches, and the edge to the second state (with probability 1.0) indicates that the execution can proceed. Lastly, the final node represents the coarse-grained execution have sucessfully reached its end. Since the reliability is given by the probability of a behavioral execution without errors occurrences, the transformation of a final node becomes a single FDTMC state labeled as *success*, with a reflexive edge whose probability is 1.0 (indicating it is an absorbing state), as shown in Figure 4e.

The sequence diagram elements considered by our approach are messages (synchronous or asynchronous) and combined fragments for representing the optional, alternative, and loop fragments. The optional combined fragment is used uniformly for representing the variation points of a product line, as its semantics allows representing behavioral fragments that may comprise a product (or not), according to its guard condition. Hence, whenever an optional fragment occurs within a behavioral fragment (sequence diagram or any other combined fragment), it represents a software product line variability (i.e. its condition denotes a presence condition statement) and it is transformed into an FDTMC structure comprised of three states and two edges, as illustrated in Figure 5. Accordingly, we abstract the reliability of the optional combined fragment's content by the parameter $rFragment$ which acts as a placeholder for the reliability of the whole combined fragment. The first edge is annotated with *rFragment* for representing the reliability values the fragment may assume, while the second edge is annotated with $1 -$ *rFragment* for representing the probability of failure occurrences.

Transformations of the remaining sequence diagram elements (synchronous, asynchronous and reply messages, and alternative and loop combined fragments) are performed according to Ghezzi and Sharifloo [21], except that our approach does not use alternative fragments to represent variation points re-

the probability of transitioning to a successor state must be 1, meaning that these transitions are the only possible events [3].

lated to alternative features. In our method, the behavioral variability is addressed uniformly by the optional fragment whose guard condition is expressed by a propositional logical formula denoting its presence condition statement. Such formula indeed expresses any kind of features relations, including OR and alternative features. In Section 3.3, we explain how the evaluation of a optional combined fragment with an arbitrarily associated presence condition statement is guided and constrained by the feature model's rules.

When the loop fragment is transformed into an FDTMC, it results into a structure that express the probabilistic conditions of an iteration. Both first and last states have two outgoing edges that denote the probability of executing (by the `loop` variable) and skipping (by the complement `1-loop`) the iteration behavior. The FDTMC representing the iteration behavior is represented between the first and last states.

The transformation of synchronous, asynchronous, and reply messages results into a structure comprised of three states and two edges. The first edge denotes the success probability of sending the message, while the complement edge denotes its failure probability [21]. The difference between the message types expresses the operational semantics of each message. The synchronous message denotes that the sender component holds its execution while it waits the call's answer that comes back by its associated reply message. In another way, in an asynchronous message the sender component continues its execution just after sending the message to the called component and it does not wait for a reply message.

Since the UML sequence diagram does not have a final element (as the end node represents in a UML activity diagram), the execution of a sequence diagram or an optional combined fragment is considered successful whenever the last element is reached and executed accordingly. As our approach considers that an FDTMC has a single and absorbing `error` state, when the last FDTMC's state is reached, it is ensured that no errors occurred during the behavioral execution, including the execution of the last sequence diagram element. Thus, when our approach transforms an sequence diagram or behavioral fragment and there is no remaining element, the last state in the FDTMC is labeled as *"success"*.

As an example, Figure 7a shows an excerpt of the RDG corresponding to the UML activity and sequence diagrams depicted in Figures 2a and 2b such there is an RDG node for each kind of behavioral fragment found on both figures. Note that whenever a behavioral fragment (activity or sequence diagrams and optional combined fragment) has to be transformed, its RDG

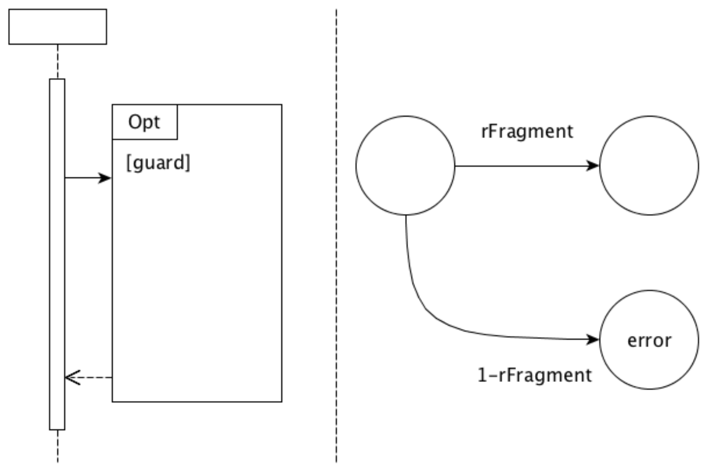
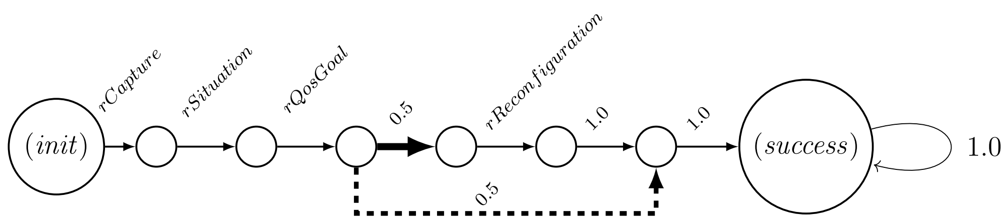


Figure 5: Transformation of optional combined fragment into FDTMC.

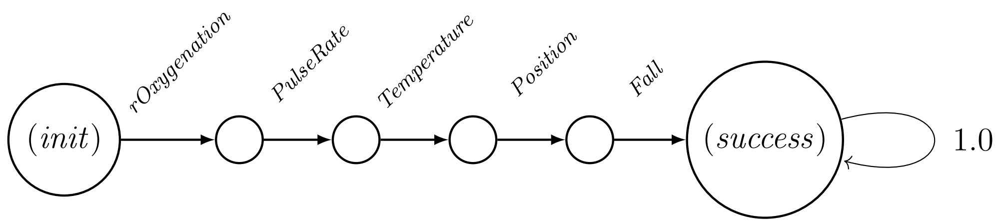
node and an edge are created to accommodate its FDTMC and represent the behavioral dependency, respectively. The node labeled *rRoot* is the root node of this RDG. The FDTMC assigned to this node (Figure 6a) is built by applying the transformation rules in Figure 4 to the activity diagram in Figure 2a. The decision node in this activity diagram gives rise to the bold and dashed transitions in Figure 6a, representing the *yes* and *no* branches.

The RDG node *rSituation* represents the sequence diagram depicted in Figure 2b, corresponding to the activity *System identifies situation* of BSN's control loop (Figure 2a). Since this activity is performed by all products, its presence condition is `true`. The node's FDTMC, depicted in Figure 6b, is obtained from the sequence diagram according to the transformation template in Figure 5 and the templates defined by Ghezzi and Sharifloo [21]. The outgoing edges of the node *rSituation* in Figure 7a correspond to its dependency on the availability of sensor information—one RDG node per optional behavioral fragment. (Most of the RDG nodes corresponding to such behavioral fragments are omitted for brevity).

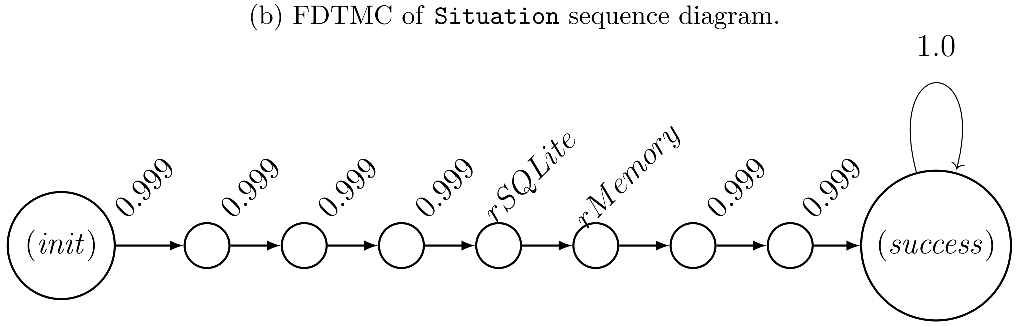
The node labeled *rOxygenation* in Figure 7a represents the behavior in the behavioral fragment whose presence condition is `Oxygenation` (Figure 2b). The corresponding FDTMC, presented in Figure 6c, is built by applying the transformation rules described in Section 3.1 in a stepwise fashion. Since the behavioral fragment consists of four messages, followed by two op-



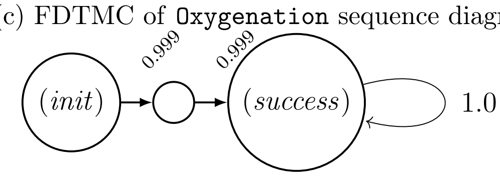
(a) FDTMC of the control loop of BSN-SPL.



(b) FDTMC of `Situation` sequence diagram.



(c) FDTMC of `Oxygenation` sequence diagram.

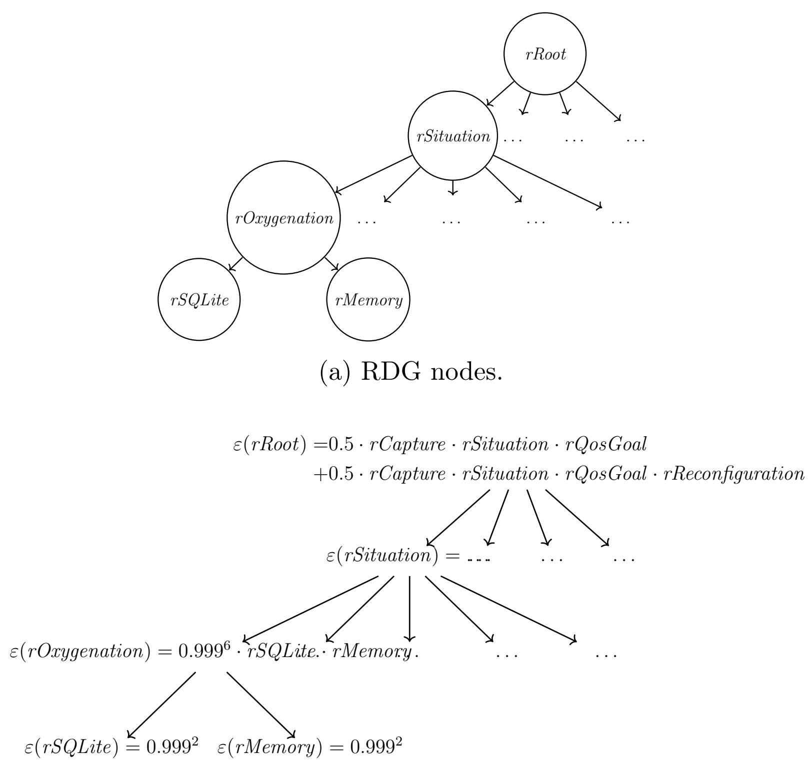


(d) FDTMC of `SQLite`/`Memory` sequence diagram.

Figure 6: Resulting FDTMCs. Error transitions are omitted for brevity.

tional combined fragments (with presence conditions `SQLite` and `Memory`) and other two messages (all messages having reliability 0.999), its resulting FDTMC comprises a sequence of four transitions with probability 0.999, two transitions with their probabilities represented by parameters (*rSQLite* and *rMemory*), and other two transitions with probability 0.999. The node *rOxygenation* depends on two basic RDG nodes, *rSQLite* and *rMemory*, corresponding to the nested behavioral fragments whose presence conditions are `SQLite` and `Memory`, respectively. Since both fragments have similar behavior (two sequential messages, each with reliability 0.999) their corresponding FDTMCs are equal (Figure 6d).

Finally, the approach relies on the divide-and-conquer strategy to decompose behavioral models. During the transformation of a behavioral fragment into a FDTMC, whenever another behavioral fragment is found, an RDG node is created with a parent-child dependency relation with the parent's RDG node. The way a software product line is decomposed results into a tree-like RDG if there is no behavioral fragment being reused. Otherwise, an RDG node representing a reused behavior fragment will have as many incoming edges as the times the fragment is reused. In this specific case, the structure of the resulting RDG will not be tree-like (that is why the RDG is a directed acyclic graph, in general).



(a) RDG nodes.

(b) Dependencies between expressions.

Figure 7: RDG excerpt for the BSN product line.

3.2. Feature-based Analysis

The role of the feature-based analysis step is to analyze the FDTMC for each RDG node in isolation, abstracting from the dependencies to other RDG nodes. That is, instead of evaluating a potentially intractable FDTMC for the product line as a whole, we perform multiple evaluations of smaller models, one per feature.

For each RDG node $x \in \mathcal{N}$, its FDTMC is subject to parametric probabilistic reachability analysis [24, 20]. This feature-based analysis yields x's reliability as an expression over the reliabilities of the n RDG nodes $x_1, \ldots, x_n$, on which it depends. This expression is denoted by a function $[0,1]^n \to [0,1]$, that is, the computation of a reliability value takes n reliability values as input. Therefore, there is a function $\varepsilon : \mathcal{N} \to ([0,1]^n \to [0,1])$ that yields the semantics of the reliability expression for a given RDG node. To remove possible ambiguities, the order of the formal parameters is determined by a total order relation over the corresponding RDG nodes x_i (e.g., a lexicographic order over node labels). When analyzing RDG nodes, the same reliability property of eventually reaching the *success* final state (expressed by the model checker query expression $P_{=?}[\Diamond \text{“}\mathit{success}\text{”}]$—see Section 2.1) is used for all FDTMCs.

Performing feature-based analysis over the RDG, as depicted in Figure 7a, yields the expressions shown in Figure 7b. These expressions illustrate that basic nodes have their reliabilities defined in terms of constants, whereas the reliabilities of variant nodes ultimately depend on the ones of basic RDG nodes. For the sake of simplicity, we overload the names of RDG nodes in Figure 7a as variables in the expressions in Figure 7b. This way, we map each variable to the RDG node whose reliability it represents.

For instance, in Figure 7b, the reliability expression of the node labeled *rOxygenation* is $0.999^6 \cdot \mathit{rSQLite} \cdot \mathit{rMemory}$, since the only path that reaches the `success` state in the corresponding FDTMC (Figure 6c) is a succession of four transitions with probability 0.999, two parametric transitions (*rSQLite* and *rMemory*), and two other 0.999-valued transitions. The reliability expressions of the nodes *rSQLite* and *rMemory* are constant, since these nodes are basic and, thus, their FDTMCs (Figure 6d) have only constant transitions. In this case, the single path to the `success` state in both FDTMCs has a reachability probability of 0.999^2. Hence, the reliability expressions for the feature-based analysis of the BSN product line are given by a function ε

such that

$$\varepsilon(rOxygenation) = 0.999^6 \cdot rSQLite \cdot rMemory$$
$$\varepsilon(rSQLite) = 0.999^2$$
$$\varepsilon(rMemory) = 0.999^2$$

3.3. Family-based Analysis

A possible next step would be to evaluate the obtained expressions once for each valid configuration, so that the reliability of every product would be computed. This enumerative approach would be, in fact, a *product-based* analysis, yielding an overall *feature-product-based* analysis, similar to the one described by Ghezzi and Sharifloo [21]. However, evaluating all products using this approach would be still prone to an exponential blowup, which would harm scalability.

To avoid this problem, we leverage a family-based analysis strategy to *lift* each expression to perform arithmetic operations over variational data, with the help of an appropriate *variational data structure* [45]. This way, we are able to represent all possible values under variation and efficiently evaluate results, sharing computations whenever possible. The data structure of choice is the Algebraic Decision Diagram (ADD)[3] [27], because it efficiently encodes a Boolean function $\mathbb{B}^n \to \mathbb{R}$. This is the same type as a mapping from configurations to reliability values would have, provided the Boolean values $b_1, \ldots, b_n \in \mathbb{B} = \{0, 1\}$ are taken to denote the presence (or absence) of the corresponding features $f_1, \ldots, f_n \in F$ (where F is the set of features in the feature model).

Given an expression $\varepsilon(x)$, obtained for an RDG node x in the feature-based step of the analysis (Section 3.2), the reliability ADD $\alpha(x)$ is obtained by first valuating the parameters $x_1, \ldots, x_k$ of the lifted expression with the ADDs for the reliabilities $\alpha(x_1), \ldots, \alpha(x_k)$ of the corresponding nodes upon which x depends. Then, arithmetic operations are performed using ADD semantics: for ADDs A_1 and A_2 over k Boolean variables and a binary operation $\odot \in \{+, -, \cdot, \div\}$, $(A_1 \odot A_2)(b_1, \ldots, b_k) = A_1(b_1, \ldots, b_k) \odot A_2(b_1, \ldots, b_k)$.

However, the computation of $\alpha(x)$ must take presence conditions into account. To accomplish this, we constrain the valuation of a variable x_i with

[3] ADDs, also called Multi-Terminal Binary Decision Diagrams (MTBDD), generalize Binary Decision Diagrams (BDD) to Real-valued Boolean functions.

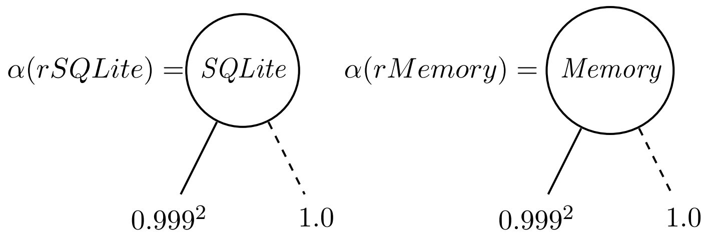
an ADD $p_x : [\![FM]\!] \rightarrow \mathbb{B}$ encoding its presence condition, with x ranging over x_1 to x_n, such n is the number of features. This ADD has the property that all configurations $c \in [\![FM]\!]$ that satisfy x_i's presence condition evaluate to 1, while all others evaluate to 0. The resulting constrained decision diagram φ_{x_i} is given by:

$$\varphi_{x_i}(c) = \begin{cases} \alpha(x_i)(c) & \text{if}\, p_{x_i}(c) = 1 \\ 1 & \text{otherwise} \end{cases}$$

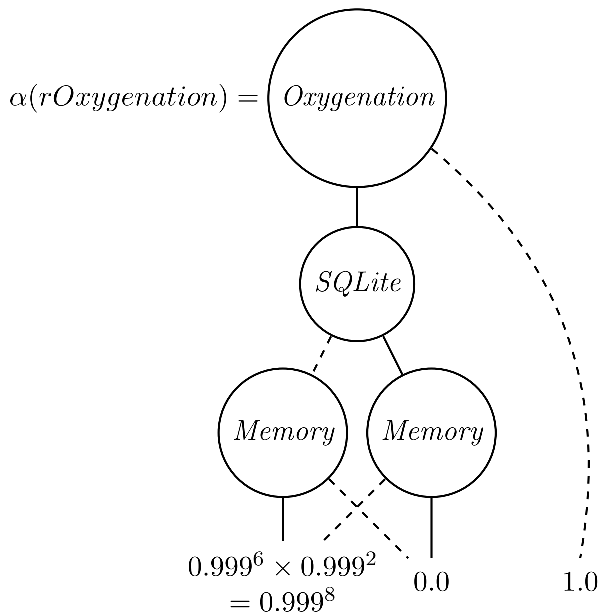
Notice the attribution of 1 to the reliability of a behavior that is absent in a given configuration. The intuition is that, for those configurations that do not satisfy the fragment's guard conditions (i.e., $p_{x_i}(c) = 0$), the behavior represented by the optional fragment will not be part of the resulting product's behavior. Since an absent behavioral fragment has no influence on the reliability of the overall system, in practice we can assume 1.0 as its reliability value (i.e., it cannot fail). The ADD φ_{x_i} is obtained by means of the *if-then-else* operator for decision diagrams, and the operational details of this construction are presented in Section 4.1.

This method of evaluating the expressions is inherently recursive, since the resulting value of computing the expression for a given RDG node depends on the results of computing the expressions for the nodes on which it depends. For example, Figure 7b shows that the expression $\varepsilon(rOxygenation)$ is defined in terms of the variables *rSQLite* and *rMemory*. Thus, before computing the lifted counterpart of expression $\varepsilon(rOxygenation)$, it is necessary to compute the lifted counterparts of expressions $\varepsilon(rSQLite)$ and $\varepsilon(rMemory)$. In a brief, the family-based step computes the reliabilities values each RDG node may assume by solving its ε expression using reliabilities values encoded by α for the nodes it depends on. Thus, it follows that the reliability of the product line as a whole is given by the ADD resulting from the computation of $\alpha(rRoot)$, where *rRoot* is the root RDG node.

Naturally, basic nodes are the base case of this recursion, since, by definition, they depend on no other node. Figure 8a depicts the ADDs representing the reliability encoding of the RDG nodes *rSQLite* and *rMemory*, respectively. Each ADD node represents a feature whose continuous outgoing edge denotes the feature's presence at the configuration, meanwhile the dashed outgoing edge means the feature is absent. Thus, $\alpha(rSQLite)$ encodes that the RDG node *rSQLite* assumes the reliability value of 0.999^2



(a) ADDs for $rSQLite$ and $rMemory$ nodes, respectively.



(b) ADD for $rOxygenation$ node.

Figure 8: ADDs for the running example.

when the feature *SQLite* is part of the configuration, and assumes the value 1.0 otherwise.

Figure 8b shows the reliability encoding computed for the *rOxygenation* RDG node. Since $\varepsilon(rOxygenation)$ is defined in terms of the variables representing the reliabilities of the nodes on which it depends, $\alpha(rOxygenation)$ is computed by assigning the ADDs previously computed to *rSQLite* and *rMemory* to the corresponding variables in $\varepsilon(rOxygenation)$, which is solved by employing ADD arithmetics. The resulting ADD is constrained to represent only the reliabilities of valid configurations when it is multiplied by the ADD representing the feature model's rules. In fact, all paths leading to non-zero terminal represent valid configurations. In the case that the feature *Oxygenation* is absent, its influence on the configuration's reliability is none, thus $\alpha(rOxygenation)$ assumes the value 1.0. Otherwise, for configurations containing *Oxygenation* and only one persistence feature (*SQLite* or *Memory*), the corresponding path in the ADD leads to the reliability value 0.999^8. Finally, the paths leading to the reliability value 0 represent ill-formed configurations. For example, since *SQLite* and *Memory* are alternative features, the paths representing that both features are present or absent will lead to 0. All these cases are also represented by the Table 1.

Table 1: Reliability of *Oxygenation* feature.

Configuration (c)	$\alpha(rOxygenation)(c)$
{Oxygenation, SQLite, ¬Memory}	995*(998/1000)*1/1000 = 0,99301
{Oxygenation, ¬SQLite, Memory}	995*1*(998/1000)/1000 = 0,99301
{Oxygenation, SQLite, Memory}	–

4. Evaluation

To assess the merits of a feature-family-based strategy, we first highlight key aspects of its implementation (Section 4.1) and analyze its complexity (Section 4.2). Then we report on an empirical evaluation (Section 4.3).

4.1. Implementation

We implemented our approach as a new tool named ReAna (**Re**liability **Ana**lysis), whose source code is open and publicly available[4]. ReAna takes as input a UML behavioral model, for example, built using the MagicDraw tool[5], and a feature model described in conjunctive normal form (CNF), for example, as exported by FeatureIDE [41]. It then outputs the ADD representing the reliability of all products of the product line to a file in DOT format, and it prints a list of configurations and respective reliabilities. The latter can be suppressed or filtered to a subset of possible configurations of interest.

[4]https://github.com/SPLMC/reana-spl
[5]http://www.nomagic.com/products/magicdraw.html

```
1 ADD evalReliability(RDGNode root) {
2     List<RDGNode> deps = root.topoSortTransitiveDeps();
3     LinkedHashMap<RDGNode, String> expressionsByNode =
          getReliabilityExpressions(deps);
4     Map<RDGNode, ADD> reliabilities =
          evalReliabilities(expressionsByNode);
5     return reliabilities.get(root);
6 }
```

Listing 3: ReAna's main evaluation routine

ReAna uses PARAM 2.3 [24] to compute parametric reachability probabilities and the CUDD 2.5.1 library[6] for ADD manipulation. However, any other tool or library providing the same functionality (e.g., the parametric model checker from Filieri and Ghezzi [20]) could be used too.

ReAna's main evaluation routine is depicted in Listing 3. After parsing and transforming the input models into an RDG structure (see Section 3.1), the method `evalReliability` is invoked on the RDG's root node. Its first task is to perform a topological sort of the RDG nodes, so that it obtains a list in which every node comes after all the nodes on which it (transitively) depends (Line 2). This implements the recursion described in Section 3.3 in an iterative fashion.

Then, it proceeds to the analysis of the reliability property in the FDTMC corresponding to each of the nodes (Line 3), with the support of a parametric model checker. Although this step does not depend on the ordering of nodes (because it handles dependencies as variables), it is useful that its output respects this order. This way, the resulting reliability expressions (ε in Section 3.2) can be evaluated in an order that allows every variable to be immediately resolved to a previously computed value, thus eliminating the need for recursion and null checking.

The third step is to evaluate each reliability expression, which yields an ADD representing the reliability function (α in Section 3.3) for each of the nodes. The evaluation of such reliability ADDs (method `evalReliabilities` in Line 4, Listing 3) invokes, for each node, method `evalNodeReliability`, which we present in Listing 4. It computes the φ functions of a node's depen-

[6]ftp://vlsi.colorado.edu/pub/cudd-2.5.1.tar.gz

```
1  ADD evalNodeReliability(RDGNode node,
2                          String reliabilityExpression,
3                          Map<RDGNode, ADD> relCache) {
4      Map<String, ADD> depsReliabilities = new HashMap();
5      for (RDGNode dep: node.getDependencies()) {
6          ADD depReliability = relCache.get(dep);
7          ADD presCond = dep.getPresenceCondition();
8          ADD phi = presCond.ifThenElse(depReliability,
9                                        constantAdd(1));
10         depsReliabilities.put(dep.getId(), phi);
11     }
12     ADD reliability = solve(reliabilityExpression,
13                             depsReliabilities);
14     return FM.times(reliability);
15 }
```

Listing 4: Evaluation of the reliability function for a single node

dencies (as in Section 3.3), encoding satisfaction of their presence conditions by means of conditionals in ADD `ITE` (*if-then-else*) operations (Line 8, Listing 4). The reliability function of each dependency is looked up in a reliability cache (`relCache`, in Line 6, Listing 4) and is then used as the *consequent* argument of the `ITE` operator, with the *alternative* argument being the constant ADD corresponding to 1.

After all these functions are computed, they are used to evaluate the lifted reliability expression (Line 12, Listing 4). Whenever a variable appears in this expression, function φ of the corresponding RDG node (on which the current one depends) is looked up in a variable–value mapping, indexed by the node id (`depsReliabilities`).

When this evaluation of α is done, it is necessary to consider only the valid configurations for the node at hand by discarding the reliability values of ill-formed products. We represent the feature model's rules by an ADD where all paths leading to terminal 1 represent a valid configuration, otherwise the path leads to terminal 0. Thus, for the node under evaluation we prune invalid configurations by multiplying its reliability ADD by the one representing the feature-model's rules (Line 14, Listing 4), so the resulting ADD yields the value 0, for ill-formed products and the actual reliability for the valid ones.

All reliabilities computed in this way are progressively added to the reliability cache *relCache*. At the end of this loop inside `evalReliabilities`, the cache contains the reliability function for every node and is then returned (Line 4, Listing 3). The reliability of interest is then the one of the root RDG node (the one argument to `evalReliability`, Listing 3), so it is queried in constant time because of the underlying data structure.

4.2. Analytical Complexity

The overall analysis time is the sum of the time taken by each of the sequential steps in Listing 3. First, the computation of an ordering that respects the transitive closure of the dependency relation in an RDG (Line 2) is an instance of the classical topological sorting problem for directed acyclic graphs, which is linear in the sum of nodes and edges [12].

Second, the computation of the reliability expression for an RDG node consists of a call to the PARAM parametric model checker, which requires n calls to cover all nodes (Line 3). The problem of parametric probabilistic reachability in a model of s states consists of $O(s^3)$ operations over polynomials, each of which depends on the number of monomials in each operand [23]. This number of monomials is, in the worst case, exponential in the number of existing variables. The number of variables for a given node is, in turn, dependent on its number of child nodes and on the modeled behavior (e.g., if there are loops or alternative paths). Thus, the time complexity of computing all the reliability expressions is linear in the number of RDG nodes, but depends on the topologies of the RDG and of the models represented by each of its nodes (we address such dependencies with more details later on).

Last, method `evalReliabilities` calls method `evalNodeReliability`, which corresponds to the reliability function α in Section 3.3, once for each node. `evalNodeReliability`'s complexity is dominated by that of ADD operations, which are polynomial in the size of the operands [27]. Indeed, for ADDs f, g, and h, the *if-then-else* operation $\texttt{ITE}(f, g, h)$ is $O(|f| \cdot |g| \cdot |h|)$. Likewise, $\texttt{APPLY}(f, g, \odot)$, where $\odot$ is a binary ADD operator (e.g., multiplication), is $O(|f| \cdot |g|)$. Here, $|f|$ denotes the size of the ADD f, that is, its number of nodes. Because of configuration pruning (Section 3.3), all ADD sizes in our approach are bound by $|FM_{ADD}|$ (i.e., the size of the ADD that encodes the rules in the feature model).

Since the evaluation of α for a given node comprises a number of operations on the reliability ADDs of the nodes on which it depends (Listing 4, Line 12), we must estimate an upper bound for polynomial arithmetics. If a node identified by x has c children (nodes on which it depends), $f'(x)$ is a polynomial in c variables and it has, at most, e_{max}^c monomials of c variables each, where e_{max} is the maximum exponent for any variable. Each monomial has in turn, at most, $2c$ operations: c exponentiations and c multiplications among variables and the coefficient. Also, no variable can have an exponent greater than the maximum number of transitions between the initial and the success states of the original FDTMC, and this number is itself bound by the number m of messages in the corresponding behavioral model fragment. Thus, the number of ADD operations needed to compute this reliability ADD is $O(c \cdot m^c)$. This leads to an evaluation time of $O(c \cdot m^c \cdot |FM_{ADD}|^2)$.

Since the reliability of each RDG node needs to be evaluated exactly once (due to caching), we have n computations of $f(x_i)$, one for each of the n RDG nodes x_i. Hence, the cumulative time spent on reliability functions computation is $O(n \cdot c_{max} \cdot m_{max}^{c_{max}} \cdot |FM_{ADD}|^2)$, where c_{max} is the maximum number of children per node, and m_{max} is the maximum number of messages per model fragment.

Although this complexity bound is quadratic in the number of features, the number of nodes in an ADD is, in the worst case, exponential in the number of variables. As the variables in FM_{ADD} represent features, this means $|FM_{ADD}|$ can be exponential in the number F of features. Hence, the worst-case complexity is $O(n \cdot c_{max} \cdot m_{max}^{c_{max}} \cdot 2^{2 \cdot F})$. This worst-case exponential blowup cannot be avoided theoretically, but, in practice, efficient heuristics can be applied for defining an ordering of variables that can cause the ADD's size to grow linearly or polynomially, depending on the functions being represented [3]. Thus, as the growth in the sizes of ADDs varies with the product line being analyzed [30] and is, at least, linear in the number of features, we can also say the best-case time complexity is $O(n \cdot c_{max} \cdot m_{max}^{c_{max}} \cdot F^2)$.

In summary, the time complexity of our feature-family-based analysis strategy lies between $O(n \cdot c_{max} \cdot m_{max}^{c_{max}} \cdot F^2)$ and $O(n \cdot c_{max} \cdot m_{max}^{c_{max}} \cdot 2^{2 \cdot F})$, where n is the number of RDG nodes, c_{max} is the maximum number of child nodes in an RDG node, m_{max} is the maximum number of messages in a behavioral fragment, and F is the number of features of the product line.

4.3. Empirical Evaluation

Our empirical evaluation aims at comparing our feature-family-based analysis strategy (cf. Section 3) with other state-of-the-art strategies for product-line reliability analysis, as identified by Thüm et al. [42]: product-based, family-based, feature-product-based, and family-product-based. It is expected that our feature-family-based approach performs better than the others, since it (a) decomposes behavioral models into smaller ones and (b) prevents an exponential blowup by computing the reliabilities of all products at once using ADDs. The comparison focuses on the practical complexity of the selected strategies and is guided by the following research question:

- **RQ1:** How do product-line reliability analysis strategies compare to one another in terms of time and space?

To address RQ1, we measured the time and space demanded by each strategy for the analysis of six available software product lines and augmented versions thereof. For the time measure, we considered the wall-clock time spent during analysis after model transformation, including the recording of reliability values for all configurations of a given product line. Transformation time was excluded from this measurement, because all of our implementations of the analysis strategies employ the same transformation routines (using the rules presented in Section 3.1.3). From the transformation step on, the analysis strategies start to differ as each one traverses the resulting FDTMC in its specific fashion. For the space measure, we considered the peak memory usage for each strategy during the evaluation of each product line. This empirical assessment is described in detail in the following subsections.

4.3.1. Subject Systems and Experiment Design

To empirically compare the complexity of the different analysis strategies, we started with the models of six available product lines. Table 2 shows the number of features, the size, and the characteristics of the solution space of each one of these product lines. The solution space is described in terms of the number of activities in the activity diagram and of the total number of behavioral fragments present in the sequence diagrams. The general criterion for choosing these systems was the availability of their variability model. We chose EMail, MinePump, BSN, and Lift due to the fact that they had been commonly used in previous work studying model checking of product lines [7, 8, 39]. We selected InterCloud and TankWar product lines due to the significant size of their configuration spaces.

Each of the six original systems was evolved 20 times, with each evolution step adding one optional feature and a corresponding behavioral fragment with random messages defining its probabilistic behavior. According

Table 2: Initial version of product lines used for empirical evaluation.

			Solution Space's Characteristics	
	# **Features**	# **Products**	# **Activities**	# **Behavioral fragments**
EMail [43]	10	40	4	11
MinePump [29]	11	128	7	23
BSN [39]	16	298	4	15
Lift [37]	10	512	1	10
InterCloud [19]	54	110592	5	51
TankWar [43]	144	4.21×10^{18}	7	81

to Section 3.1.1, the name of the newly introduced feature was assigned as the guard condition of each new behavioral fragment, and each message in a fragment received a probability value. Thus, each evolution step doubles the size of the configuration space of the subject product line, with an optional behavior for the added feature.

The independent variable of the experiment is the evaluation strategy employed to perform the reliability analysis. The dependent variables are the metrics for time and space complexity. Each subject system was evaluated by all treatments.

We analyzed the outcomes using statistical tests, to properly address outlying behavior and spurious results. This way, we are more likely to overrule factors that affect performance but are difficult to control (e.g., JVM warm-up time and OS process scheduling). Ideally (i.e., disregarding uncontrollable factors), we would expect all runs of a given analysis strategy over the same subject product line to yield the same result. Thus, instead of comparing isolated runs of different strategies, we compare the inferred distribution of results of all runs of a strategy to the corresponding distribution for another strategy. Since there were multiple analysis strategies to compare with, we did so pairwise with the feature-family strategy, for example, feature-based with feature-family-based or family-based with feature-family-based.

We applied standard statistical tests for equality of the pairs of samples. The null hypothesis was that both samples come from the same distribution, while the alternative hypothesis was that one comes from a distribution with larger mean value than the other. The specific statistical test was the Mann-Whitney U test whenever one of the samples, at least, was not normally distributed. Otherwise, we applied the t test for independent samples if the variances were equal, or Welch's t test in case of different variances. The significance level for all tests was 0.01.

4.3.2. Experiment Setup

Modeling We implemented each strategy as a variant of ReAna, thus relying on the same tools and libraries for parametric reachability checking, ADD manipulation, and expression parsing (see Section 4.1). These ReAna extensions are also publicly available at the supplementary Web site[7]. Graduate students created the input UML behavioral models using MagicDraw 18.3 with Marte UML profile. All models were validated by the authors.

Instrumentation For this experiment we implemented a tool called SPL-Generator to create valid feature and behavioral models of a product line, according to a set of parameters (more details in Appendix B). This tool was used to create evolution scenarios, in order to assess how each evaluation strategy behaves with the growth of the configuration space. To obtain data regarding analysis time, we used Java's standard library method `System.nanoTime()` to get the time (with nanoseconds precision) reported by the Java Virtual Machine immediately before and right after ReAna's main analysis routine (Listing 3). The difference between these two time measures is taken to be the elapsed analysis time. Space usage was measured using the maximum resident set size reported by the Linux `/usr/bin/time` tool. This value represents the peak RAM usage throughout ReAna's execution.

Evolution Scenarios We used our SPL-Generator tool to evolve each software product line we chose as a subject system of our empirical evaluation, according to the representation provided in Figure 9. This evolution was accomplished stepwise, and it started with the original feature model (created by FeatureIDE) and behavioral models (created by MagicDraw)—this set of models is hereafter refered to as *original seed* or $seed_0$. At each evolution step ev_i, the generator tool doubled the configuration space of the subject system by adding an optional feature in order to generate a new feature model FM_i(no cross-tree constraint was added, to avoid constraining configuration space growth). For the newly created feature, the generator tool also creates an optional behavioral fragment comprising 10 messages

[7] `http://splmc.github.io/scalabilityAnalysis/`

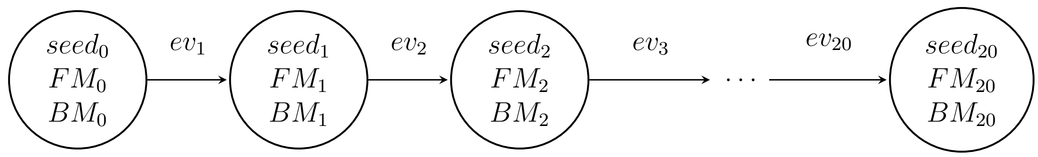


Figure 9: Evolution of subject systems accomplished by the SPL-Generator tool

randomly generated between 2 lifelines randomly chosen from a set of 10 lifelines. To establish a relation between the new feature and the corresponding new behavioral fragment, the fragment's guard condition is defined as being the atomic proposition containing the new feature's name, which characterizes the evolutions as being compositional. However, it is worth mentioning that our evaluation method also applies to the analysis of annotation-based software product lines since it was able to evaluate the original version of the EMail subject system ($seed_0$ that contains optional fragments expressed by a conjunction of two features thus, following an annotation-based implementation) and its evolutions. Each lifeline received a random reliability value from the range $[0.999, 0.99999]$. The guard condition of the behavioral fragment received an atomic proposition named after the feature, to relate the newly created items. The topological allocation method was used by the generator tool to create the new behavioral model BM_i, so the nesting of sequence diagrams follows the feature relations in the feature model. The end of an evolution step results into a new version of the product line ($seed_i$), which will be considered as a new *seed* for the next evolution step. Each subject system was evolved 20 times, as shown in Figure 9, and all artifacts are available at the paper's supplementary site.

Measurement Setup We executed the experiment using twelve Intel i5-4570TE, 2.70GHz, 4 hyper-threaded cores, 8 GB RAM and 1 GB swap space, running 64-bit CentOs Linux 7. The experiment environment (i.e., the set of tools, product line models, and automation running scripts) was defined as a Docker[8] container running 64-bit Ubuntu Linux 16.10, with access to 4 cores and 6 GB of main memory of the host machine. Each subject system was evaluated 8 times by each analysis strategy in each machine, thus summing up 96 evaluations for each pair of subject system and strategy. Because of the number of evaluations, we set a limit of 60 minutes for analysis

[8]https://www.docker.com/

execution time, after which the analysis at hand would be canceled. The results were then grouped to perform the time and memory consumption analysis. The evaluations that exceeded the time limit were discarded from the statistical analysis.

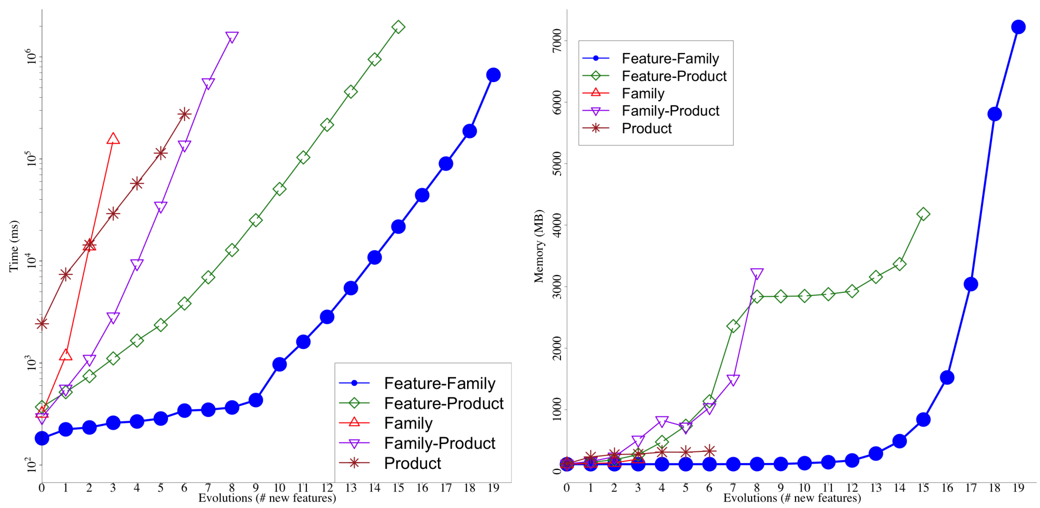
4.3.3. Results and Analysis

Figures 10, 11, 12, 13, 14 and 15 show plots with the mean time and memory demanded to analyze the Email, MinePump, BSN, Lift, InterCloud, and TankWar product lines (and corresponding evolutions), respectively. The horizontal axes represent the number of added features (with respect to the original product line) in the analyzed models. Thus, they range from 0 (the original model) to 20 (last evolution step). The vertical axes represent either the time in milliseconds (in logarithmic scale) or the space in megabytes.

The values of the plots are available in Tables A.4 and A.5 of Appendix A. Statistical tests over both time and space data rejected the null hypothesis for all pairs of strategies. Thus, within a significance level of 0.01, we can assume no two samples come from distributions with equal means.

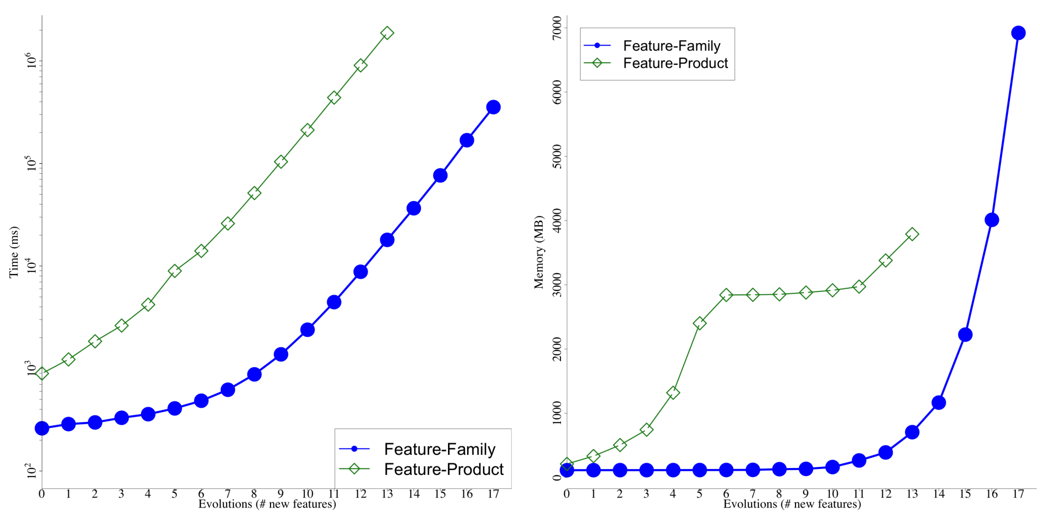
Overall, our experiments show with statistical significance that the feature-family-based strategy is faster than all other analysis strategies (as shown in Figures 10a, 11a, 12a, 13a, 14a, and 15a). Regarding execution time, in the worst case, our feature-family-based strategy performed 60% faster than the family-product-based strategy, when analyzing the original models of the Email product line (Figure 10a); in the best case, it outperformed the family-product-based analysis of the BSN product line with 4 optional features added (i.e., its 5^{th} evolution step—Figure 12a) by 4 orders of magnitude. Such cases are highlighted in yellow in Table A.4. Regarding memory consumption (Figures 10b, 11b, 12b, 13b, 14b, and 15b), the experiment also shows with statistical significance that, in the worst case, the feature-family-based strategy demanded 2% less memory than the family-based strategy when analyzing the original model of the Lift product line; in the best case, it saved around 4,757 megabytes when analyzing the 3^{rd} evolution step of the InterCloud product line. Such cases are highlighted in yellow in Table A.5.

Our feature-family-based strategy also scaled better in response to configuration space growth in comparison with other strategies. In the worst case, this strategy scaled up to a configuration space one order of magnitude larger than the limit of the nearest scalable strategy (the feature-product-based analysis of the Email, MinePump, BSN, and Lift systems). In the best case, the feature-family-based strategy supported a configuration space



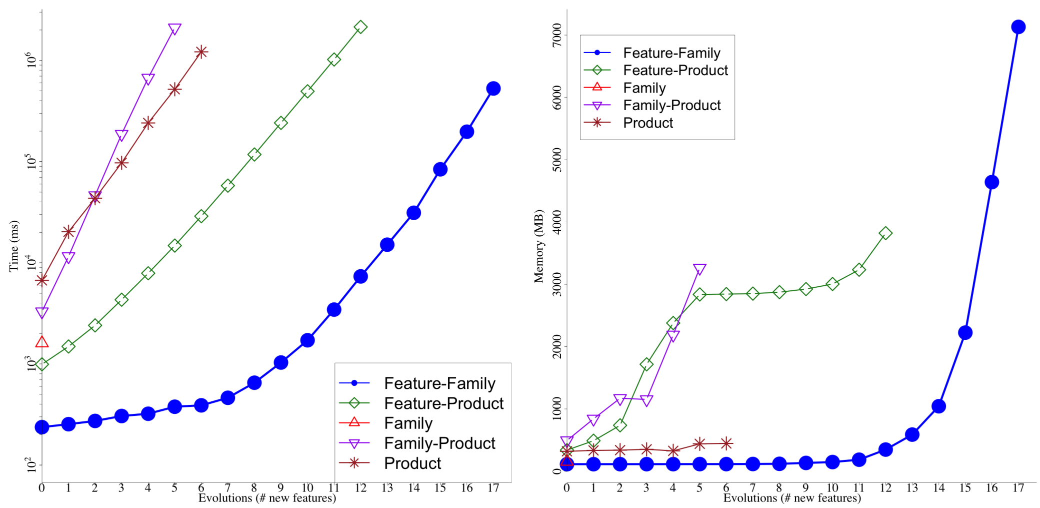
(a) Analysis time. (b) Demanded memory.

Figure 10: Time and memory required by different analysis strategies when evaluating evolutions of Email System.



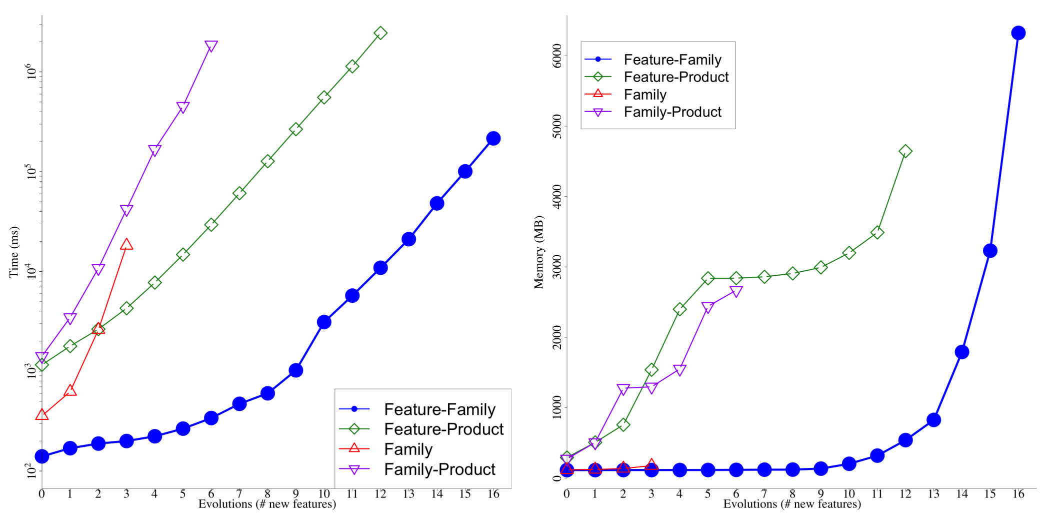
(a) Analysis time. (b) Demanded memory.

Figure 11: Time and memory required by different analysis strategies when evaluating evolutions of MinePump System.



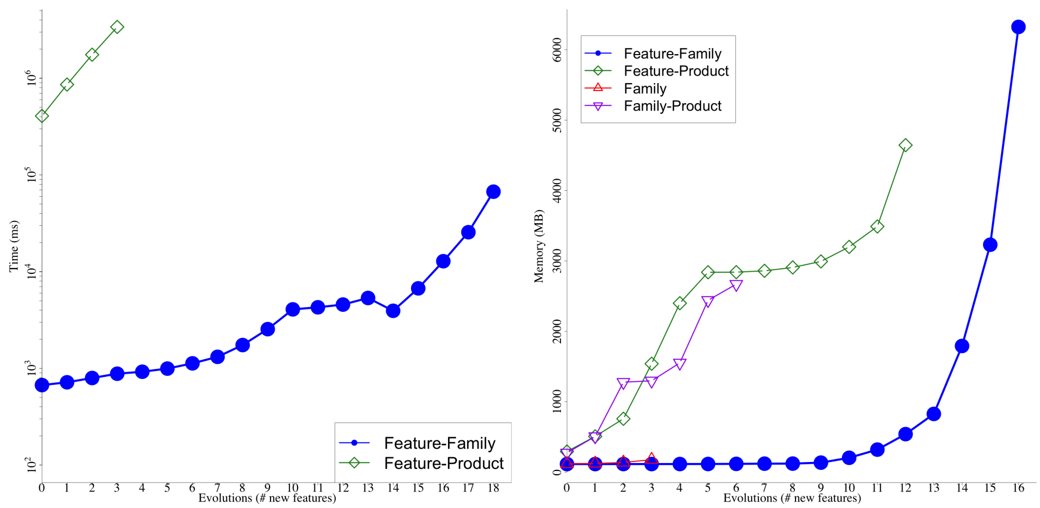
(a) Analysis time. (b) Demanded memory.

Figure 12: Time and memory required by different analysis strategies when evaluating evolutions of BSN-SPL.



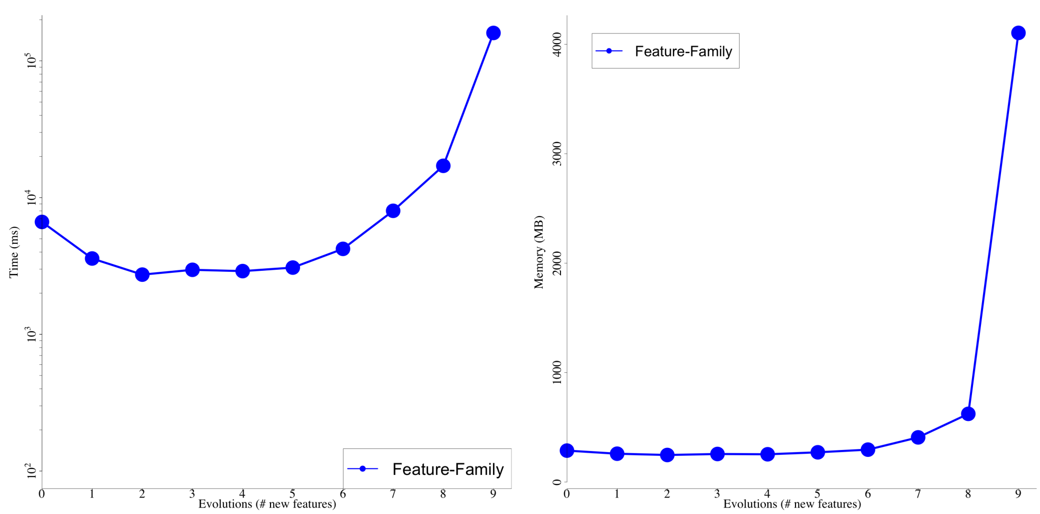
(a) Analysis time. (b) Demanded memory.

Figure 13: Time and memory required by different analysis strategies when evaluating evolutions of Lift System.



(a) Analysis time. (b) Demanded memory.

Figure 14: Time and memory required by different analysis strategies when evaluating evolutions of InterCloud System.



(a) Analysis time. (b) Demanded memory.

Figure 15: Time and memory required by different analysis strategies when evaluating evolutions of TankWar battle game.

5 orders of magnitude larger than supported by the feature-product-based strategy (when analyzing the InterCloud product line). Finally, we highlight that only our feature-family-based strategy was able to analyze the TankWar product line, from its original model up to its 9^{th} evolution step. That is, the feature-family-based strategy was able to analyze the reliability of up to 10^{21} products within 60 minutes.

Table 3: Probabilistic models statistics.

SPL	**Feature-***			**Family-***		**Product**	
	# states	# variables	# models	# states	# variables	# states	# models
EMail	12	0.93	14	182	9	115.8	40
MinePump	7.26	0.95	23	289	10	155.5	128
BSN	11.37	1.44	16	238	12	136.56	298
Lift	12.91	0.91	11	153	10	114	512
InterCloud	7.4	0.98	52	437	47	352.25	110592
TankWar	8.30	0.99	79	735	69	$\approx$500	4.21×10^{18}

4.3.4. Discussion

One reason for the feature-family-based strategy being faster than the alternatives is that it computes the reliability values of a product line by analyzing a small number of comparatively simple models. In contrast, family-based and family-product-based strategies yield more complex probabilistic models than the others, trading space for time. The complementary explanation for the performance boost is that the family-based analysis step leverages ADDs to compute reliability values, which leads to fewer operations than necessary if these values were to be calculated by enumeration of all valid product line configurations (cf. Section 4.2).

Table 3 shows the average number of states and variables present in the models created by each analysis strategy[9], with feature-family-based and feature-product-based strategies grouped under Feature-*, and family-based and family-product-based ones grouped under Family-*. Some values are omitted, because the number of models is always 1 for family-based approaches, and the number of variables is always 0 for product-based ones. In this table, all probabilistic models created by Feature-* analyses have, indeed, fewer states than the ones generated during Family-* and product-

[9]For TankWar, the average number of states in the product-based case is an estimate, because it is impractical to generate all models.

based approaches. Feature-based models also have fewer variables than the corresponding family-based ones.

The plots of the experiment results reveal some characteristics that depart from the expected behavior, which we discuss next. First, there is a single data point for the family-based analysis of the BSN product line (Figure 12), despite its analysis time being in the order of seconds (far from reaching the time limit). In fact, the family-based strategy was able to analyze BSN's models up to the 6^{th} evolution step. However, the resulting expression representing the family's reliability contained numbers that exceeded Java's floating-point representation capabilities. Thus, converting these numbers to the `double` data type yielded *not a number* (`NaN`). To the best of our knowledge, the overflow of floating-point representation was not reported yet by previous studies addressing reliability analysis of software product lines.

The second remarkable characteristic are the plateaus for feature-product-based analysis at the memory plots in Figures 10b, 11b, 12b, and 13b. Our hypothesis is that this behavior is related to the memory management of the Java Virtual Machine (JVM), but a detailed investigation was out of scope.

We also note that the plots for feature-family-based analysis are monotonically increasing, with two exceptions: a single decrease at the 14^{th} evolution step of the Intercloud product line (Figure 14) and a "valley" from TankWar's original model to its 4^{th} evolution step (Figure 15). These outliers result from different ordering of variables in ADDs. The inclusion of new variables for the mentioned cases led to a variable ordering that caused a decrease in the number of internal nodes of the resulting ADDs. Thus, the space needed by such data structures was reduced, and so was the time needed to perform ADD operations (which are linear in the number of internal nodes).

Moreover, our approach does not constrain the relation between the feature model's structure and the UML behavioral models implementing the SPL. For instance, the sequence diagram depicted in Figure 2b represents optional behavioral fragments that do not follow the structure of the feature model presented by Figure 1. The *Oxygenation* feature and the *Persistence* features (*SQLite* and *Memory*) are defined in different branches of the feature model, but the behavioral fragments related to them are nested. In general, the guard condition of an optional behavioral fragment is a propositional formula defined over features and can be defined arbitrarily, with no regard to the structure of the feature model.

Finally, the effect of having (many) cross-tree constraints in a feature model may affect our evaluation method in a twofold manner. First, by adding cross-tree constraints, the structure of the ADD representing the feature model's rules and the reliabilities values of each node is changed. However, it is not possible to foresee if the number of internal nodes will increase, decrease or stay the same, since this number also depends on the variable ordering. In our implementation, such ordering is defined by an internal heuristic defined by the CUDD library, on which our tool relies (namely, symmetric sifting). The second effect regards to the growth of the configuration space. In our experiments, the growth in the configuration space at each evolution step will be less than it is now, which will probably have a positive effect in the scalability of the strategies relying on a product-based step. However, since cross-tree constraints would have a random effect on the assessment, we decided not to add them, so as to have more control over the dependent variables.

4.3.5. Threats to Validity

A threat to internal validity is the creation of UML behavioral models of the product lines by graduate students. To mitigate this threat, the students received an initial training on modeling variable behavior of product lines. To validate the accuracy of the produced models, these were inspected by the authors.

A possible threat to construct validity would be an inadequate definition of metrics for the experiment. To address this, we tried to rule out implementation issues such as the influence of parallelism and reporting of results. Thus, we measured the total elapsed time between the parsing of behavioral models and the instant the reliabilities were ready to be reported, with all analysis steps taking place sequentially. In terms of memory usage, we tried to reduce the influence of garbage collection by measuring the peak memory usage during execution.

Finally, a threat to external validity arises from the selection of subject systems. To mitigate this threat, we selected systems commonly used by the community as benchmarks to evaluate work on model checking of product lines. To mitigate the risk of our approach not being generalizable, we applied it to further product lines (InterCloud and TankWar) whose configuration spaces resemble ones of real-world applications.

5. Related Work

In this section, we discuss related work to our approach, and we highlight the significant differences. For this purpose we use the classification of Thüm et al. [42]. Our approach differs from prior work [8, 21, 39] in that (a) it captures the runtime feature dependencies from the UML behavioral models, (b) which are enriched with variability information extracted from the feature model, and (c) we leverage ADDs to compute the reliability of all products of a product line with fewer operations than an enumeration would require.

5.1. Comparison to a Feature-Product-based Strategy

The evaluation method proposed by Ghezzi and Sharifloo [21] is the closest to our work and, to the best of our knowledge, it represents the state-of-the-art for reliability evaluation of software product lines. The whole behavior of a product line is modeled by a set of small sequence diagrams arranged in a tree, where each node has an associated expression resulting from the analysis performed by a parametric model checker. To compute the reliability of a product, the tree is traversed in a bottom-up fashion, when each node's expression is solved considering the configuration under analysis. The resulting value for the root node denotes the product's reliability. This method reduces time and effort required for evaluation by employing parametric instead of non-parametric reachability checking of probabilistic models, but it faces scalability issues as it is inherently enumerative (i.e., the decomposition tree is traversed for each product). The analysis strategy followed by the method is *Feature-Product-based*, as it decomposes the behavioral models into smaller units (feature-based step) and later composes the evaluation results of each unit to obtain the reliability of a product (product-based step).

Despite the resemblances with this method, our approach presents some distinguishing characteristics. While Ghezzi and Sharifloo [21] must explore their decomposition tree each time a configuration is evaluated (thus employing a product-based analysis as an evaluation step), our approach employs a family-based evaluation for each RDG node, such that all reliability values it may assume are computed in a single step. Another difference refers to the usage of UML sequence diagram elements for representing behavioral variability. Ghezzi and Sharifloo [21] establish a direct relation from the feature model's semantics of optional and alternative features and the semantics of optional (`OPT`) and alternative (`ALT`) combined fragments, respectively. Although such relation is straightforward, it constrains the approach's expressiveness, as only single features can be associated to a combined fragment (i.e., the combined fragment's guard condition assumes only atomic propositions). In contrast, our approach represents behavioral variability uniformly by the optional combined fragment, with an arbitrary presence condition as a guard statement. This construct is simpler, because it does not leverage alternative fragments, but more expressive, as guards can be defined by propositional statements.

Another major difference concerns the underlying data structure for representing the dependencies between behavioral fragments. Ghezzi and Sharifloo [21] use a decomposition tree while our approach uses a directed acyclic graph that allows to represent a group of replicated behavioral fragments by a single node. This avoids the effort of performing redundant modeling and evaluation of the replicated model, which is not possible to accomplish in a tree structure.

A precise comparison of the tool implementing the method proposed by Ghezzi and Sharifloo and ReAna was not possible, since the former is not publicly available. Nonetheless, the feature-product-based variant of ReAna we created for our experiment closely resembles Ghezzi and Sharifloo's approach, the only exception being the parametric model checker of choice. Empirical results (Section 4.3) show with statistical significance that the feature-family-based approach performs faster and demands less memory than ReAna's feature-product-based variant. For the evaluation time, the feature-family strategy outperformed our feature-product-based strategy from 2 times (for the original seed of EMail system) up to 4 orders of magnitude (for the 3^{rd} evolution of Intercloud product line). Regarding space, the feature-family-based strategy required from 2.6% (original seed of Email system) up to 97% (3^{rd} evolution of InterCloud) less memory. Moreover, the feature-product-based strategy was not able to analyze the subject system with the largest configuration space (Tankwar), whereas our feature-family-based strategy succeeded up to Tankwar's 9^{th} evolution.

Ghezzi and Sharifloo's work [21] presents a theoretical analysis of time complexity, in which the authors devise a formula for computing the time needed to verify a number of properties for a product line with their approach. Their model transformation time is not comparable to ours, mainly because Ghezzi and Sharifloo do not handle activity diagrams in their work, and we do not handle reward models in ours. Also, both approaches use external tools with similar capabilities to perform parametric reachability analysis. In fact, Ghezzi and Sharifloo argue their tool [20] is actually faster than

PARAM, which is used by ReAna. Nonetheless, both model checkers could be used interchangeably, so we omit parametric reachability analysis time.

Because of that, we assume the output expressions from the parametric reachability phase to be correspondingly equal in both approaches. This way, the difference between the strategies is isolated in the way they solve each expression. While Ghezzi and Sharifloo perform a number k of floating-point operations for each configuration, our approach performs the same number k of ADD operations, but only once. Since the number of configurations is $O(2^F)$, the feature-product-based approach performs $O(k \cdot 2^F)$ computing steps. As no lowest number of steps is possible if one is to compute the reliability of all possible configurations, the number of computations in the best case is also $O(k \cdot 2^F)$. In contrast, an operation over ADDs in our approach comprises $O(2^{2 \cdot F})$ steps in the worst case, but is $O(F^2)$ in the best case (see Section 4.2). Thus, the feature-family-based approach performs between $O(k \cdot F^2)$ and $O(k \cdot 2^{2 \cdot F})$ computing steps.

Hence, we conclude that, in the worst case, the upper bound for our method's asymptotic complexity is worse than that of Ghezzi and Sharifloo's, but its best-case complexity is better, which is consistent with the empirical findings from the previous section.

5.2. Other Related Work

Rodrigues et al. [39] present and compare three family-based strategies to analyze probabilistic properties of product lines. Two of them leverage PARAM as model checker; the third one relies on FDTMCs representing the behavior of a whole product line by encoding its variability, resulting in an ADD expressing the reliability values of all configurations. Our feature-family-based strategy benefits even more from further breaking down probabilistic models. Indeed, the methods by Rodrigues et al. show a time-space tradeoff, but all of them presented scalability issues even for small product lines (around 12 features), whereas our approach is able to analyze a product line with 144 features and about 10^{18} products within reasonable time.

Further research has addressed efficient verification of other non-functional properties of product lines by exploiting family-based analysis strategies [40, 28, 17, 18, 44, 6, 8, 16]. Siegmund et al. [40] propose an approach for performance evaluation by simulating the behavior of all variants at runtime from the variability encoded in compile-time. Such simulator is created from the log of method calls traced by features. Kowal et al. [28] create a model representing the whole performance variability of a product line from UML activity diagrams annotated with performance-related annotations. Dubslaff et al. [17, 18] present an approach for modeling dynamic product lines and performing quantitative analysis of systems endowed of non-deterministic choices. Given the non-deterministic characteristic of the systems evaluated by this approach, the authors consider Markov Decision Processes as the suitable model for representing the model behavior. Similarly, Varshosaz and Khosravi [44] introduce a mathematical model named Markov Decision Process Family for representing the behavior of a product line as a whole, as well as a model checking algorithm to verify properties expressed in probabilistic computation tree logic. Classen et al. [8] establish the foundations of *Featured Transition Systems* (FTS) to create a model endowed with features expressions to represent the states variation of the whole software product line. The authors also present a family-based model checker [6] that is able to analyze *Linear Temporal Logic* (LTL) properties of the whole software product line by employing semi-symbolic algorithms to verify FTSs. All these pieces of work exploit symbolic computation on a model representing the whole variability of a product line as a better alternative to product-based strategies. Our study supports this conclusion, especially if a suitable variational data structure (e.g., ADD) is used for such analysis. However, our results indicate that feature-family-based analysis further improves performance.

Dimovski et al. [16] also present an efficient family-based technique to verify LTL properties of a software family. The authors leverage abstract interpretation to reduce the configuration space of an FTS, so that it can be verified by off-the-shelf model checkers (i.e., aimed and optimized to analyze single systems). Our method employs a divide-and-conquer strategy to reduce model size, without changing the configuration space. Moreover, our analysis method also employs off-the-shelf model checkers, but to analyze probabilistic properties of software product lines. Therefore, it is worth investigating the extent to which the technique proposed by Dimovski et al. [16] can be applied to the verification of PCTL properties. If that is the case, we conjecture that both strategies could be combined to further reduce verification effort.

6. Conclusion

We presented a feature-family-based strategy and corresponding tool for efficient reliability analysis of product lines. Our approach limits the effort needed to compute the reliability of a product line by initially employing a *feature-based* analysis to divide its behavioral models into smaller units, which can be verified more efficiently. For this purpose, we arrange probabilistic models in an RDG, which is a directed acyclic graph with variability information. This strategy facilitates reuse of reliability computations for redundant behaviors. The *family-based* step comes next when we perform the reliability computation for all configurations at once by evaluating reliability expressions in terms of ADDs. These decision diagrams encode presence conditions and the rules from the feature model, so that computation is inherently restricted to valid configurations.

The empirical evaluation was accomplished by conducting an experiment to compare our feature-family-based approach with the following evaluation strategies: feature-product-based, family-based, family-product-based, and product-based. Overall, the results show the product-based had the worst time and space performance among all strategies, as we expected. The family- and family-product-based strategies yield more complex probabilistic models than the other strategies, due to variability encoding in their models. The product, family-product and feature-product-based approaches were sensitive to the size of the configuration space of the software product line, given their inherent enumerative characteristic. Overall, our experiments show that the feature-family-based strategy is faster than all other analysis strategies and demanded less memory in most cases, being the only one that could be scaled to a 2^{20}-fold increase in the configuration space. Such results suggest that our feature-family-based strategy outperformed the alternative strategies due to the following: (a) the feature-based step explores a lower number of simpler models having fewer variables in comparison to family-based models; and (b) as the family-based step leverages ADD to compute reliability values, fewer operations are necessary to compute reliability values in comparison to the enumerative strategies.

As future work, we plan to extend the empirical evaluation to a larger number of subject systems. Furthermore, the present study investigated the sensitivity of analysis performance with respect to changes in the size of the configuration space of the subject product lines. Thus, we also plan to extend the study so as to evaluate the performance impact of changes in other characteristics, such as the number of decision nodes and the number of messages per behavioral fragment.

Acknowledgements

We would like to thank the following people for fruitful discussions and suggestions on how to improve this work: Alexandre Mota, Cecília Rubira, Azzedine Boukerche, Rodrigo Bonifácio, Marcelo Ladeira, Abílio Oliveira, Paula Gueiros, and Eneias Silva. Vander Alves would like to thank for the research grant CAPES ref. BEX 0557-16-1 / Alexander von Humboldt ref. 3.2-1190844-BRA-HFSTCAPES-E. Sven Apel's work has been supported by the German Research Foundation (AP 206/4 and AP 206/6).

Appendix A. Experiment Data

The following tables present the mean values for analysis time and memory consumption obtained in our experiment. Values typeset in boldface are the best values (i.e., the lowest) gathered from the experiments. Cells containing dashes represent unavailable data, meaning that the corresponding analysis violated the time limit of 60 minutes.

Table A.4: Time in milliseconds (fastest strategy in boldface).

		SPL's evolutions steps										
		0 10	1 11	2 12	3 13	4 14	5 15	6 16	7 17	8 18	9 19	20
Email	*Configuration space's order*	10^1	10^1	10^2	10^2	10^2	10^3	10^3	10^3	10^4	10^4	
	Feature-family	**183.04**	**223.78**	**233.69**	**259.65**	**267.32**	**285.79**	**341.65**	**348.46**	**366.73**	**433.30**	
	Feature-product	370.63	517.67	742.91	1108.95	1659.31	2358.51	3829.95	6919.98	12803.15	25110.63	
	Family	319.72	1167.27	13944.18	154067.34	–	–	–	–	–	–	
	Family-product	293.26	558.77	1095.75	2850.86	9451.57	34704.42	137866.42	562117.02	1607837.42	–	
	Product	2424.77	7387.35	14349.32	29137.45	57575.0	114084.61	275598.17	–	–	–	
	Configuration space's order	10^4	10^4	10^5	10^5	10^5	10^6	10^6	10^6	10^7	10^7	10^7
	Feature-family	**970.69**	**1613.76**	**2833.40**	**5425.14**	**10838.39**	**21719.17**	**44171.89**	**90015.26**	**187645.77**	**667138.0**	–
	Feature-product	50748.90	103510.61	215932.90	456329.22	945445.46	1966865.48	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
Minepump	*Configuration space's order*	10^2	10^2	10^2	10^3	10^3	10^3	10^3	10^4	10^4	10^4	
	Feature-family	**261.18**	**287.24**	**298.10**	**330.88**	**358.81**	**408.45**	**485.06**	**621.01**	**877.52**	**1375.22**	
	Feature-product	895.80	1226.97	1844.15	2624.96	4204.27	8952.61	14037.50	25989.10	51495.22	104090.89	
	Family	–	–	–	–	–	–	–	–	–	–	
	Family-product	–	–	–	–	–	–	–	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^5	10^5	10^5	10^6	10^6	10^6	10^6	10^7	10^7	10^7	10^8
	Feature-family	**2390.78**	**4445.44**	**8790.54**	**17995.17**	**36593.45**	**76513.51**	**168694.38**	**354887.72**	–	–	–
	Feature-product	211806.42	439411.29	905878.46	1876640.52	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
BSN	*Configuration space's order*	10^2	10^2	10^3	10^3	10^3	10^3	10^4	10^4	10^4	10^5	
	Feature-family	**237.14**	**253.65**	**273.01**	**305.48**	**321.69**	**377.40**	**389.41**	**462.66**	**651.84**	**1032.05**	
	Feature-product	991.30	1487.19	2404.18	4312.01	7875.91	14788.91	28881.71	57887.92	117630.81	241553.61	
	Family	1604.07	–	–	–	–	–	–	–	–	–	
	Family-product	3288.70	11543.38	46273.48	187134.89	672512.22	2109118.92	–	–	–	–	
	Product	6696.06	20259.05	43489.98	97280.21	241249.72	519495.92	1217404.53	–	–	–	
	Configuration space's order	10^5	10^5	10^6	10^6	10^6	10^6	10^7	10^7	10^7	10^8	10^8
	Feature-family	**1713.19**	**3443.81**	**7332.75**	**15090.83**	**31208.01**	**83984.25**	**197660.21**	**528948.31**	–	–	–
	Feature-product	495594.45	1022294.56	2145986.30	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–

continued in the next page

continued from last page

		SPL's evolutions steps										
		0	1	2	3	4	5	6	7	8	9	
		10	11	12	13	14	15	16	17	18	19	20
Lift	*Configuration space's order*	10^{2}	10^{3}	10^{3}	10^{3}	10^{3}	10^{4}	10^{4}	10^{4}	10^{5}	10^{5}	
	Feature-family	**140.32**	**169.51**	**188.56**	**199.95**	**223.20**	**266.20**	**339.85**	**472.01**	**601.46**	**1021.76**	
	Feature-product	1160.78	1786.75	1289.76	4281.11	7739.10	14769.15	29418.50	60785.39	127344.46	266609.58	
	Family	358.06	625.16	2606.86	18223.06	–	–	–	–	–	–	
	Family-product	1413.96	3462.52	10777.60	42156.47	167837.42	453830.76	1870142.24	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^{5}	10^{6}	10^{6}	10^{6}	10^{6}	10^{7}	10^{7}	10^{7}	10^{8}	10^{8}	10^{9}
	Feature-family	**3114.06**	**5728.56**	**10895.96**	**21081.40**	**48194.05**	**100755.69**	**215756.37**	–	–	–	–
	Feature-product	555525.51	1136506.33	2457317.34	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
InterCloud	*Configuration space's order*	10^{5}	10^{5}	10^{5}	10^{5}	10^{6}	10^{6}	10^{6}	10^{7}	10^{7}	10^{7}	
	Feature-family	**671.54**	**717.7**	**794.95**	**880.11**	**922.98**	**994.65**	**1126.91**	**1315.89**	**1742**	**2544.49**	
	Feature-product	407702.34	861181.31	1752682.98	3394277.78	–	–	–	–	–	–	
	Family	–	–	–	–	–	–	–	–	–	–	
	Family-product	–	–	–	–	–	–	–	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^{8}	10^{8}	10^{8}	10^{8}	10^{9}	10^{9}	10^{9}	10^{10}	10^{10}	10^{10}	10^{11}
	Feature-family	**4074.43**	**4280.4**	**4568.7**	**5344.4**	**3936.76**	**6719.68**	**12829.35**	**25588.69**	**67156.86**	–	–
	Feature-product	–	–	–	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
TankWar	*Configuration space's order*	10^{18}	10^{18}	10^{19}	10^{19}	10^{19}	10^{20}	10^{20}	10^{20}	10^{21}	10^{21}	
	Feature-family	**6643.88**	**3588.49**	**2734.86**	**2966.2**	**2902.18**	**3079.4**	**4221.14**	**8012.**	**17096.88**	**160259.19**	
	Feature-product	–	–	–	–	–	–	–	–	–	–	
	Family	–	–	–	–	–	–	–	–	–	–	
	Family-product	–	–	–	–	–	–	–	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^{21}	10^{21}	10^{22}	10^{22}	10^{22}	10^{23}	10^{23}	10^{23}	10^{24}	10^{24}	10^{24}
	Feature-family	–	–	–	–	–	–	–	–	–	–	–
	Feature-product	–	–	–	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–

Table A.5: Space in megabytes (smallest footprint in boldface).

		SPL's evolutions steps										
		0	1	2	3	4	5	6	7	8	9	
		10	11	12	13	14	15	16	17	18	19	20
Email	*Configuration space's order*	10^1	10^1	10^2	10^2	10^2	10^3	10^3	10^3	10^4	10^4	
	Feature-family	**113.70**	**113.84**	**113.93**	**114.30**	**114.45**	**114.33**	**114.52**	**114.91**	**115.86**	**117.64**	
	Feature-product	117.22	144.30	186.59	269.67	475.61	738.99	1136.73	2359.24	2839.02	2842.46	
	Family	116.97	125.48	136.57	196.99	–	–	–	–	–	–	
	Family-product	120.25	157.90	235.41	510.41	827.79	722.88	1037.62	1501.80	3231.31	–	
	Product	122.65	231.84	272.04	277.98	310.59	309.06	327.65	–	–	–	
	Configuration space's order	10^4	10^4	10^5	10^5	10^5	10^6	10^6	10^6	10^7	10^7	10^7
	Feature-family	**130.65**	**146.25**	**174.93**	**287.00**	**489.00**	**839.80**	**1523.88**	**3041.86**	**5807.80**	**7223.00**	–
	Feature-product	2849.01	2878.10	2927.46	3158.43	3367.68	4181.64	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
MinePump	*Configuration space's order*	10^2	10^2	10^2	10^3	10^3	10^3	10^3	10^4	10^4	10^4	
	Feature-family	**113.51**	**114.05**	**114.41**	**114.34**	**114.8**	**115.61**	**116.47**	**118.48**	**129.12**	**133.96**	
	Feature-product	210.97	333.42	504.98	743.93	1319.2	2400.89	2841.77	2844.10	2851.49	2879.39	
	Family	–	–	–	–	–	–	–	–	–	–	
	Family-product	–	–	–	–	–	–	–	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^5	10^5	10^5	10^6	10^6	10^6	10^6	10^7	10^7	10^7	10^8
	Feature-family	**162.48**	**265.31**	**390.03**	**705.39**	**1165.72**	**2224.17**	**4011.27**	**6921.67**	–	–	–
	Feature-product	2914.44	2971.55	3378.84	3789.31	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
BSN	*Configuration space's order*	10^2	10^2	10^3	10^3	10^3	10^3	10^4	10^4	10^4	10^5	
	Feature-family	**114.05**	**114.30**	**114.52**	**114.56**	**114.83**	**115.37**	**115.32**	**116.97**	**120.09**	**134.23**	
	Feature-product	339.91	490.76	737.50	1716.41	2379.06	2837.60	2843.36	2850.78	2874.49	2923.93	
	Family	156.54	–	–	–	–	–	–	–	–	–	
	Family-product	493.99	841.31	1171.71	1153.13	2189.89	3263.80	–	–	–	–	
	Product	320.43	335.18	339.40	352.72	327.95	440.60	446.75	–	–	–	
	Configuration space's order	10^5	10^5	10^6	10^6	10^6	10^6	10^7	10^7	10^7	10^8	10^8
	Feature-family	**148.34**	**186.03**	**348.58**	**588.99**	**1043.94**	**2225.13**	**4640.35**	**7130.79**	–	–	–
	Feature-product	3005.12	3234.19	3821.39	–	–	–	–	–	–	–	–
	Family	156.54	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–

continued in the next page

continued from last page

		SPL's evolutions steps										
		0 10	1 11	2 12	3 13	4 14	5 15	6 16	7 17	8 18	9 19	 20
Lift	*Configuration space's order*	10^2	10^3	10^3	10^3	10^3	10^4	10^4	10^4	10^5	10^5	
	Feature-family	**113.77**	**113.85**	**114.37**	**114.27**	**114.56**	**115.23**	**116.87**	**119.88**	**120.72**	**134.41**	
	Feature-product	292.23	507.43	757.52	1539.24	2399.04	2838.97	2840.51	2859.23	2907.41	2993.05	
	Family	116.54	122.63	136.83	177.02	–	–	–	–	–	–	
	Family-product	272.85	506.39	1277.44	1296.95	1551.49	2440.83	2669.75	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^5	10^6	10^6	10^6	10^6	10^7	10^7	10^7	10^8	10^8	10^7
	Feature-family	**203.42**	**319.45**	**539.66**	**826.57**	**1791.86**	**3230.47**	**6324.48**	–	–	–	–
	Feature-product	3199.10	3489.45	4644.73	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
InterCloud	*Configuration space's order*	10^5	10^5	10^5	10^5	10^6	10^6	10^6	10^7	10^7	10^7	
	Feature-family	**119.44**	**119.87**	**127.68**	**127.79**	**136.03**	**132.63**	**136.3**	**143.96**	**152.61**	**175.78**	
	Feature-product	3071.58	3158.59	3602.16	4884.81	–	–	–	–	–	–	
	Family	–	–	–	–	–	–	–	–	–	–	
	Family-product	–	–	–	–	–	–	–	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^8	10^8	10^8	10^8	10^9	10^9	10^9	10^{10}	10^{10}	10^{10}	10^{11}
	Feature-family	**224.06**	**223.7**	**251.48**	**275.81**	**237.67**	**378.87**	**635.76**	**1102.48**	**2628.21**	–	–
	Feature-product	–	–	–	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
TankWar	*Configuration space's order*	10^{18}	10^{18}	10^{19}	10^{19}	10^{19}	10^{20}	10^{20}	10^{20}	10^{21}	10^{21}	
	Feature-family	**286.99**	**258.64**	**246.91**	**256.09**	**253.42**	**271.44**	**295.76**	**407.85**	**622.82**	**4104.61**	
	Feature-product	–	–	–	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
	Configuration space's order	10^{21}	10^{21}	10^{22}	10^{22}	10^{22}	10^{23}	10^{23}	10^{23}	10^{24}	10^{24}	10^{24}
	Feature-family	–	–	–	–	–	–	–	–	–	–	–
	Feature-product	–	–	–	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–

Appendix B. SPLGenerator tool

To increase the number of subject systems and inspect how each evaluation strategy behaves with the growth of the configuration space, we implemented a product-line generator tool called SPL–Generator[10], which is able to create a software product line from scratch or modify an existing one by incrementally adding features and behavior to its models. For the feature model generation (i.e., to create a new feature model or change an existing one), the tool relies on the SPLAR tool [33]. The desired characteristics of the resulting feature model are obtained by defining accordingly the set of parameters provided by SPLAR. Examples of such parameters are the number of features to be created, the amount in percentage for each kind of feature (mandatory, optional, OR-inclusive and OR-exclusive), and the number of cross-tree constraints. As our SPL-Generator tool intends to create product lines that resemble real-world product lines, it produces only consistent feature-models (i.e., the SPLAR's parameter for creating consistent feature-models is always set to `true`).

To create behavioral models, the SPL-Generator tool considers the UML behavioral diagrams and follows the refinement of activity diagrams into sequence diagrams presented in Section 2.3. For creating activity and sequence diagrams, the generator tool is also guided by a set of parameters for each kind of behavioral diagram. For an activity diagram, it is possible to define how many activities it will comprise, the number of decision nodes, and how many sequence diagrams will refine each created activity. For a sequence diagram, it is possible to define its size in terms of numbers of behavioral fragments, the size of each behavioral fragment in terms of the number of messages, the number of lifelines, the number of different reliability values (such that each lifeline will randomly assume only one value) and the range for them. Thus, one possibly generated sequence diagram would have 5 behavioral fragments, each one containing 8 messages between 3 lifelines, whose reliability values are within the range $[0.99, 0.999]$.

Finally, the SPL-Generator tool also provides a parameter to define how the feature model and the behavioral models will be related. The allocation of a behavioral fragment (implementing a feature's behavior) can be fully *randomized* within the set of created sequence diagrams, or it can be *topological*, which means the relations between the behavioral fragments mimic

[10] `https://github.com/SPLMC/spl-generator/`

the relations between the corresponding features. In the latter, we assume a child feature refines its parent, so its behavioral fragment is nested into its parent's behavioral fragment.

References

[1] Apel, S., Batory, D., Kästner, C., Saake, G. (Eds.), 2013. Feature-oriented software product lines: concepts and implementation. Springer, Berlin.

[2] Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F., 1997. Algebric decision diagrams and their applications. Formal Methods in System Design 10 (2), 171–206.

[3] Baier, C., Katoen, J.-P., 2008. Principles of Model Checking (Representation and Mind Series). The MIT Press.

[4] Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M., 2013. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13. ACM, New York, NY, USA, pp. 355–364.

[5] Clarke, E. M., Grumberg, O., Peled, D. A., 1999. Model checking. MIT Press, Cambridge, Mass.

[6] Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.-Y., Oct 2012. Model checking software product lines with SNIP. International Journal on Software Tools for Technology Transfer 14 (5), 589–612.

[7] Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.-Y., Feb. 2014. Formal semantics, modular specification, and symbolic verification of product-line behaviour. Science of Computer Programming 80, Part B, 416–439.

[8] Classen, A., Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A., Raskin, J.-F., 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. IEEE Transactions on Software Engineering 39 (8), 1069–1089.

[9] Classen, A., Heymans, P., Schobbens, P., Legay, A., 2011. Symbolic model checking of software product lines. In: 2011 33rd International Conference on Software Engineering (ICSE). pp. 321–330.

[10] Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., Raskin, J.-F., 2010. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1. ICSE '10. ACM, New York, NY, USA, pp. 335–344.

[11] Clements, P., Northrop, L., 2002. Software product lines: practices and patterns. The SEI series in software engineering. Addison-Wesley, Boston.

[12] Cormen, T. H., Stein, C., Rivest, R. L., Leiserson, C. E., 2001. Introduction to Algorithms, 2nd Edition. McGraw-Hill Higher Education.

[13] Czarnecki, K., Eisenecker, U., 2000. Generative programming: methods, tools, and applications. Addison Wesley, Boston.

[14] Czarnecki, K., Pietroszek, K., 2006. Verifying feature-based model templates against well-formedness OCL constraints. ACM Press, p. 211.

[15] Daws, C., sep 2005. Symbolic and Parametric Model Checking of Discrete-time Markov Chains. In: Liu, Z., Araki, K. (Eds.), Proceedings of the First International Conference on Theoretical Aspects of Computing. Vol. 3407 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 280–294.

[16] Dimovski, A. S., Al-Sibahi, A. S., Brabrand, C., Wąsowski, A., 2015. Family-Based Model Checking Without a Family-Based Model Checker. Springer International Publishing, Cham, pp. 282–299.

[17] Dubslaff, C., Baier, C., Kluppelholz, S., 2015. Probabilistic Model Checking for Feature-Oriented Systems. In: Chiba, S., Tanter, E., Ernst, E., Hirschfeld, R. (Eds.), Transactions on Aspect-Oriented Software Development XII. No. 8989 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 180–220, dOI: 10.1007/978-3-662-46734-3_5.

[18] Dubslaff, C., Klüppelholz, S., Baier, C., 2014. Probabilistic Model Checking for Energy Analysis in Software Product Lines. In: Proceedings of the 13th International Conference on Modularity. MODULARITY '14. ACM, New York, NY, USA, pp. 169–180.

[19] Ferreira Leite, A., Alves, V., Nunes Rodrigues, G., Tadonki, C., Eisenbeis, C., Magalhaes Alves de Melo, A., Jun. 2015. Automating Resource Selection and Configuration in Inter-clouds through a Software Product Line Method. In: 2015 IEEE 8th International Conference on Cloud Computing (CLOUD). pp. 726–733.

[20] Filieri, A., Ghezzi, C., 2012. Further steps towards efficient runtime verification: Handling probabilistic cost models. In: Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches. FormSERA '12. IEEE Press, Piscataway, NJ, USA, pp. 2–8.

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

[22] Grunske, L., 2008. Specification patterns for probabilistic quality properties. In: ICSE '08. ACM, New York, NY, USA, pp. 31–40.

[23] Hahn, E., Hermanns, H., Zhang, L., 2010. Probabilistic reachability for parametric markov models. STTT, 1–17.

[24] Hahn, E. M., Hermanns, H., Wachter, B., Zhang, L., 2010. Param: A model checker for parametric markov models. In: CAV. pp. 660–664.

[25] Hao, Y., Foster, R., 2008. Wireless body sensor networks for health-monitoring applications. Physiological Measurement 29 (11), 27–56.

[26] Heradio, R., Perez-Morago, H., Fernandez-Amoros, D., Javier Cabrerizo, F., Herrera-Viedma, E., Apr. 2016. A bibliometric analysis of 20 years of research on software product lines. Information and Software Technology 72, 1–15.

[27] Iris, R., Erica, B., Frohm, A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., Somenzi, F., 1993. Algebraic Decision Diagrams and Their Applications. In: Proceedings of the 1993 IEEE/ACM International

Conference on Computer-aided Design (ICCAD '93). IEEE Computer Society Press, Santa Clara, California, USA, pp. 188–191.

[28] Kowal, M., Tschaikowski, M., Tribastone, M., Schaefer, I., Nov. 2015. Scaling Size and Parameter Spaces in Variability-Aware Software Performance Models (T). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 407–417.

[29] Kramer, J., Magee, J., Sloman, M., Lister, A., Jan. 1983. CONIC: an integrated approach to distributed computer control systems. Computers and Digital Techniques, IEE Proceedings E 130 (1), 1–.

[30] Liang, J. H., Ganesh, V., Czarnecki, K., Raman, V., 2015. SAT-based Analysis of Large Real-world Feature Models is Easy. In: Proceedings of the 19th International Conference on Software Product Line. SPLC '15. ACM, New York, NY, USA, pp. 91–100.

[31] Linden, F. v. d., Schmid, K., Rommes, E., 2007. Software product lines in action: the best industrial practice in product line engineering. Springer, Berlin ; New York.

[32] Machado, I. d. C., McGregor, J. D., Cavalcanti, Y. C., de Almeida, E. S., Oct. 2014. On strategies for testing software product lines: A systematic literature review. Information and Software Technology 56 (10), 1183–1199.

[33] Mendonca, M., Branco, M., Cowan, D., 2009. S.p.l.o.t.: Software product lines online tools. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. OOPSLA '09. ACM, New York, NY, USA, pp. 761–762.

[34] Nunes, V., Fernandes, P., Alves, V., Rodrigues, G., Sep. 2012. Variability Management of Reliability Models in Software Product Lines: An Expressiveness and Scalability Analysis. In: 2012 Sixth Brazilian Symposium on Software Components Architectures and Reuse (SBCARS). pp. 51–60.

[35] Object Management Group, 2011. The UML profile for MARTE: Modeling and analysis of real-time and embedded systems. Version 1.1.

[36] Pessoa, L., Fernandes, P., Castro, T., Alves, V., Rodrigues, G. N., Carvalho, H., 2017. Building reliable and maintainable dynamic software product lines: An investigation in the body sensor network domain. Information and Software Technology 86, 54 – 70.

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

[38] Pohl, K., Böckle, G., Linden, F. v. d., 2010. Software product line engineering: foundations, principles, and techniques. Springer, New York, NY.

[39] Rodrigues, G. N., Alves, V., Nunes, V., Lanna, A., Cordy, M., Schobbens, P.-Y., Sharifloo, A. M., Legay, A., Jan. 2015. Modeling and Verification for Probabilistic Properties in Software Product Lines. In: 2015 IEEE 16th International Symposium on High Assurance Systems Engineering (HASE). pp. 173–180.

[40] Siegmund, N., von Rhein, A., Apel, S., 2013. Family-based Performance Measurement. In: Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences. GPCE '13. ACM, New York, NY, USA, pp. 95–104.

[41] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., Jan. 2014. Featureide: An extensible framework for feature-oriented software development. Sci. Comput. Program. 79, 70–85.

[42] Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G., Jun. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Comput. Surv. 47 (1), 6:1–6:45.

[43] University of Magdeburg, O. v. G., 2011. SPL2go. Available at `http://spl2go.cs.ovgu.de/`, accessed: 2016-01-27.

[44] Varshosaz, M., Khosravi, R., Dec. 2014. Model Checking of Software Product Lines in Presence of Nondeterminism and Probabilities. In: Software Engineering Conference (APSEC), 2014 21st Asia-Pacific. Vol. 1. pp. 63–70.

[45] Walkingshaw, E., Kästner, C., Erwig, M., Apel, S., Bodden, E., 2014. Variational Data Structures: Exploring Tradeoffs in Computing with

Variability. In: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. Onward! 2014. ACM, New York, NY, USA, pp. 213–226.

[46] Weiss, D. M., 2008. The product line hall of fame. In: Proceedings of the 12th International Software Product Line Conference (SPLC). IEEE Computer Society, Washington, DC, USA, p. 395.