A Formal Framework of Software Product Line Analyses

THIAGO CASTRO, Systems Development Center – Brazilian Army, Brazil
LEOPOLDO TEIXEIRA, Federal University of Pernambuco, Brazil
VANDER ALVES, University of Brasília, Brazil
SVEN APEL, Saarland University, Saarland Informatics Campus, Germany
MAXIME CORDY, University of Luxembourg, Luxembourg
ROHIT GHEYI, Federal University of Campina Grande, Brazil

A number of product-line analysis approaches lift analyses such as type checking, model checking, and theorem proving from the level of single programs to the level of product lines. These approaches share concepts and mechanisms that suggest an unexplored potential for reuse of key analysis steps and properties, implementation, and verification efforts. Despite the availability of taxonomies synthesizing such approaches, there still remains the underlying problem of not being able to describe product-line analyses and their properties precisely and uniformly. We propose a formal framework that models product-line analyses in a compositional manner, providing an overall understanding of the space of family-based, feature-based, and product-based analysis strategies. It defines precisely how the different types of product-line analyses compose and inter-relate. To ensure soundness, we formalize the framework, providing mechanized specification and proofs of key concepts and properties of the individual analyses. The formalization provides unambiguous definitions of domain terminology and assumptions as well as solid evidence of key properties based on rigorous formal proofs. To qualitatively assess the generality of the framework, we discuss to what extent it describes five representative product-line analyses targeting the following properties: safety, performance, data-flow facts, security, and functional program properties.

CCS Concepts: • **Software and its engineering → Software product lines**; **Formal software verification**.

Additional Key Words and Phrases: Software Product Lines, Product-line Analysis

ACM Reference Format:
Thiago Castro, Leopoldo Teixeira, Vander Alves, Sven Apel, Maxime Cordy, and Rohit Gheyi. 2020. A Formal Framework of Software Product Line Analyses. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (December 2020), 37 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Thiago Castro, Systems Development Center – Brazilian Army, QG do Exército - Bloco G - 2º Andar, 70630-901, Setor Militar Urbano, Brasilia, Brazil, castro.thiago@eb.mil.br; Leopoldo Teixeira, Federal University of Pernambuco, Av. Jornalista Aníbal Fernandes, s/n - Cidade Universitária (Campus Recife), 50740-560, Recife, Brazil, lmt@cin.ufpe.br; Vander Alves, University of Brasília, Campus Universitário Darcy Ribeiro - Edifício CIC/EST, 70910-900, Brasilia, Brazil, valves@unb.br; Sven Apel, Saarland University, Saarland Informatics Campus, Campus E1 1, 66123, Saarbruecken, Germany, apel@cs.uni-saarland.de; Maxime Cordy, University of Luxembourg, 6, rue Richard Coudenhove-Kalergi L-1359, Luxembourg, Luxembourg, maxime.cordy@uni.lu; Rohit Gheyi, Federal University of Campina Grande, Av. Aprigio Veloso, 882, Bloco CN, Bairro Universitário 58.429-900, Campina Grande, Brazil, rohit@dsc.ufcg.edu.br.

1049-331X/2020/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software product line engineering is a means to systematically manage variability and commonality in software systems, enabling automated synthesis of product variants from a set of reusable assets [1, 22, 63]. Such discipline aims at improving productivity and time-to-market, as well as achieving mass customization of software [63]. However, for a product line with high variability, the number of possibly generated products may be intractably large. Because of this phenomenon, it is often infeasible to quality-check each of the products individually. Nonetheless, software analysis and verification techniques for single products are widely used in industry, and it is beneficial to exploit their maturity to increase reliability of software product lines while reducing costs and risks.

There is a number of product-line analysis approaches that lift analyses such as type checking, model checking, and theorem proving to the level of entire product lines [73]. *Product-line analyses* can be classified along three main strategies: product-based (the analysis is performed on generated products or models thereof), family-based (only domain artifacts and valid combinations thereof are checked), and feature-based (domain artifacts implementing a given feature are analyzed in isolation, regardless of their combinations) [73].

Although different in purpose, individual approaches to product-line analysis share features that suggest there is a yet unexplored potential for reuse of key analysis steps and properties, implementation, and verification efforts. For instance, Kowal et al. [47] employ a family-based analysis strategy for building a stochastic model that encodes all variability of the product line, and thus can be analyzed in a single run. Apel et al. [3] adopt the same strategy for generating source code of a program that encodes all variability in the product line as runtime variability, so that off-the-shelf software model checkers can be applied. Although the models and verified properties are different, there is a pattern of encoding all variability in a single product, which Apel et al. [3] call a product or variant simulator. This pattern, in turn, must have an underlying principle governing its applicability and validity [81]. Therefore, should one have the need of applying these and similar analyses implemented by different tools, a number of analysis steps—that are inherently time-consuming, error-prone, and require specialized knowledge to perform—would inevitably be repeated.

In the same vein, as research in this area progresses, different analysis strategies may be applied to the same property. As an example, Ghezzi and Sharifloo [36] first explored compositionality of probabilistic models to analyze product-line reliability, effectively proposing a feature-product analysis strategy. Lanna et al. [49] also aimed at compositionality while avoiding enumerating all products in a feature-family strategy. Nevertheless, for the sake of soundness, each strategy carries the burden of proving that it yields correct results. This is often overlooked, error-prone, and time-consuming to achieve, eventually partially achieved with incomplete methods (e.g., testing [9]), ultimately jeopardizing confidence in such analysis strategies.

The repetition and formalization gaps are a natural outcome of ongoing research activity, given that there are independent groups exploring similar issues. Nonetheless, practitioners and researchers would benefit from having an integrated body of knowledge. So, it is useful to periodically synthesize existing work in a given field. Indeed, the survey by Thüm et al. [73], in which the terminology of family-, feature-, and product-based analysis strategies was established, is an important step in this direction. In some sense, it is a coarse-grained domain analysis of the domain of product-line analysis. However, there still remains the underlying problem of not being able to describe product-line analyses and their properties precisely and uniformly. Indeed, more refinement is necessary to formalize concepts, to explore similarities in analysis steps and key properties, and to suitably manage variability across analysis approaches, providing practitioners and researchers with more efficient techniques and a theoretical framework for further investigation.

We propose a formal framework of key concepts and properties of the state of the art of product-line analysis. In particular, the framework defines abstract functions and types modeling essential abstractions in this problem domain, including central analysis steps, models, and intermediate analysis results. Product-line analyses are presented in a compositional manner, providing an overall understanding of the structure space of family-based, feature-based, and product-based analysis strategies, showing how the different types of product-line analyses compose and inter-relate. To ensure soundness, we formalize the framework using the PVS proof assistant [58], providing a precise specification of key concepts and properties as well as mechanized proofs of key soundness results (e.g., commutativity of intermediate analysis steps). Therefore, the novelty of this work lies in how we model and map the different strategies and how we prove certain properties. To qualitatively assess the generality of the framework, we discuss to what extent it is able to describe five representative product-line analyses targeting the following properties: safety, performance, data-flow facts, security, and functional program properties. Analytical assessment by instantiation of the PVS theory for concrete analysis techniques is a promising avenue of future work. Our framework lays the foundation for this.

Overall, our formal framework encodes knowledge on the product-line analysis domain in a concise, precise, and sound manner, thus providing unambiguous definitions of domain terminology and assumptions as well as solid evidence of key domain properties based on formal proofs. This way, the framework provides a principled understanding and further facilitates the communication of ideas and knowledge of this domain. Researchers and practitioners can safely leverage the framework to lift existing single-product analysis techniques to yet under-explored product-line analysis approaches and to explore further strategies.

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

- We review existing product-line analyses, revealing their key concepts and properties (Section 3);
- We present a formal framework of product-line analyses that provides an overall understanding of the space of family-based, feature-based, and product-based analysis strategies as well as relations among them (Section 4);
- We provide a mechanized formalization of the framework using the PVS proof assistant, comprising a specification of essential concepts and proofs of key soundness results (Section 4.2);
- We explore the framework's generality by discussing to what extent it can describe five representative product-line analyses (Section 5).

2 BACKGROUND

Before presenting our framework, we introduce basic concepts of software product lines (Section 2.1) and product-line analysis (Section 2.2). We also briefly review transition systems, the mathematical foundation of the analysis techniques that inspired this work (Section 2.3). Furthermore, we present algebraic decision diagrams (Section 2.4), which are leveraged as variational data structures to support family-based analyses within our framework.

2.1 Software Product Lines

A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [22]. The main goal in product-line engineering is managing variability, which is defined by van Gurp et al. [76] as the ability to change or customize a system. To accomplish this, it is useful to abstract variability in terms of *features*. The concept of a feature encompasses both intentions of stakeholders and implementation-level

concerns, and has been subject to a number of definitions [1]. Synthetically, it can be seen as a characteristic or end-user-visible behavior of a software system. The features of a product line and their relationships are documented in a *feature model* [27, 43], which can be graphically represented as a *feature diagram*.

A given software system that is part of a product line is referred to as a *product*. A product is specified by a *configuration*, which is a selection of features respecting the constraints established by the feature model. A product consists of a set of assets (e.g., source code files, test cases, documentation), which are derived from a common *asset base*. The mapping between a given configuration and the assets that constitute the corresponding product is called *configuration knowledge* [27]. Configuration knowledge may consist of selecting source files, for instance, but may also incorporate processing tasks over the selected assets, such as running the C Preprocessor.

The locations in the assets at which variation occurs are called *variation points*. There are three common approaches for representing variability at implementation level: annotative, compositional, and transformational [38, 44]. *Annotative* approaches annotate common assets with tags corresponding to features, such that product derivation can be done by removing the parts annotated with the features not selected (e.g., by using preprocessor directives [59]). *Compositional* approaches represent variability in a modular way by segregating assets or parts of assets that correspond to each feature in composable units; the ones corresponding to selected features in a given configuration are combined to derive a product. *Transformational* approaches, more generally, rely on transformations of base assets; these transformations usually manipulate assets at the syntactic level, but this is not a formal restriction of this category of techniques.

2.2 Analysis Taxonomy

Analysis of software product lines is a broad subject in the sense that it can refer to reasoning about any of the product line artifacts, including the feature model and configuration knowledge [1]. We focus on the possibly derivable products. This does not necessarily mean generating all products in a product line and analyzing each of them, as long as analyzed properties can be generalized to the product line as a whole. We refer to the latter case as *variability-aware analysis*.

Thüm et al. [73] conducted a literature survey defining three dimensions of analysis strategies for product lines:

Product-based. Product-based analysis analyzes the derived products or models thereof. This can be accomplished by generating all products (the *brute-force* approach) or by sampling a subset of them. The main advantage of this strategy is that the actual analysis can be performed exactly as in the single-system case using off-the-shelf tools. However, the analysis effort can be prohibitively large (exponential blowup) if the considered product line has a large number of products.

Feature-based. Feature-based analysis analyzes all domain artifacts implementing a given feature in isolation, not considering how they relate to other features. However, issues related to feature interactions are frequent [35, 51, 55], which renders the premise false that features can be modularly analyzed. Still, a feature-based approach is able to verify compositional properties (e.g., syntactic correctness) and has the advantage of supporting *open-world scenarios*—since a feature is analyzed in isolation, not all features must be known in advance.

Family-based. Family-based analysis operates only on domain artifacts (not generated ones) and incorporates the knowledge about valid feature combinations. It aims at sharing, thereby avoiding redundant computations across multiple products. Family-based analyses may operate by merging all variability into a single *product simulator* (also known as *150% model* [82]),

which is prone to single-product analysis techniques. Nonetheless, there are also approaches specifically tailored to product lines, leveraging custom-made tools and techniques [29, 50].

Specifically, our framework addresses the static analysis of properties of derivable products, and not of variability management artifacts, so automated analyses of feature models [8] are out of scope.

Beyond individual strategies, there is the possibility to employ more than one strategy simultaneously. This way, weaknesses resulting from one strategy can be overcome by another. This is particularly useful for feature-based approaches, which are generally not sufficient if there are feature interactions. For instance, Thüm et al. [74] propose formal verification of design-by-contract properties [53] restricted to individual feature modules. This is a feature-based strategy, but the actual contract of a given product cannot be known before the corresponding feature modules are composed. Hence, this approach defines *partial* proofs for the contracts of individual feature modules (feature-based step), then generate proof obligations for each derived product, and verify whether these obligations are satisfied by a composition of the partial proofs for the selected features (product-based step). Since the product-based phase leverages the proofs obtained in the feature-based phase, this composite strategy is called *feature-product-based*.

Product-line analyses combining different strategies are classified as follows [73, 78]:

Feature-product-based. This strategy consists of a feature-based analysis followed by a product-based analysis. It leverages the feature-based phase (e.g., computing properties that hold for individual features) to ease the analysis effort necessary for the product-based phase.

Feature-family-based. In this strategy, one performs a feature-based analysis to check properties that apply individually for each feature, then the results are combined maintaining variability to undergo a family-based analysis. The family-based phase considers the feature model constraints and the interactions between features all at once, enabling the analysis of properties that are not observable in the scope of a single feature.

Family-product-based. This strategy consists of a partial family-based analysis followed by a product-based analysis that leverages the intermediate results.

Feature-family-product-based. In this strategy, one performs a feature-based analysis followed by a family-product-based analysis that leverages the analysis effort of the feature-based phase.

Although this taxonomy of product-line analyses provides an overall understanding, more refinement is necessary to formalize the underlying analysis steps and interrelations, key properties (e.g., commutativity of intermediate analysis steps), and preconditions (e.g., assumption on compositionality of basic analyses). Only this way, one can effectively explore similarities and suitably manage variability among these approaches.

2.3 Transition Systems

Transition Systems are a formalism to represent the behavior of a system as states and transitions among them. A transition system consists of a set of states and transitions between these states annotated with actions. For example, $s \xrightarrow{\alpha} s'$ denotes a transition from state s to state s' due to some action α. Each state is labeled with a set of so-called *atomic properties*, which represent all the properties that hold when the system is in this state. Examples of atomic properties are *failure* and *sleep*, which signal the system is in a failure state or in sleep mode, respectively. Starting from these atomic properties, one can define properties across the transitions between the system states. A most simple example is "the next state of the system must not be a failure state". Another is "the system must never be in a failure state". A more complex property is "the system cannot enter sleep

mode until all failures are resolved". These properties are typically expressed in some temporal logic such as Computation Tree Logic (CTL) [18] and Linear Temporal Logic (LTL) [62]. They are considered as behavioral properties, i.e., they consider the sequence (and in the case of CTL, also the alternance) of visible states. The properties that are expressible this way include safety, reachability, and repetitive reachability.

2.4 Algebraic Decision Diagrams

An Algebraic Decision Diagram (ADD) [5] is a data structure that encodes k-ary Boolean functions of type $\mathbb{B}^k \rightarrow \mathbb{R}$. As an example, Figure 1 depicts an ADD representing a simple binary function.

$$f(x,y) = \begin{cases} 0.9 & \text{if } x \wedge y \\ 0.8 & \text{if } x \wedge \neg y \\ 0 & \text{otherwise} \end{cases}$$

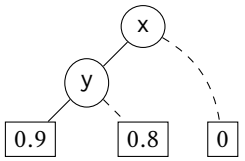


Fig. 1. ADD A_f representing the Boolean function f on the left

Each internal node of an ADD (one of the circular nodes) marks a decision over a single parameter. Function application is achieved by traversing the ADD along a path that denotes a decision over the values of actual parameters: if the parameter represented by the node at hand is 1 (*true*), we take the solid edge; otherwise, if the actual parameter is 0 (*false*), we take the dashed edge. The evaluation ends when we reach a terminal node (one of the square nodes at the bottom).

To evaluate $f(1,0)$ in our example, we start on node x, take the solid edge to node y (since the actual parameter x is 1), then take the dashed edge to terminal 0.8. Thus, $f(1,0) = 0.8$. Henceforth, we will use a function application notation for ADDs, meaning that, if A is an ADD that encodes function f, then $A(b_1, \ldots, b_k)$ denotes $f(b_1, \ldots, b_k)$. For brevity, we also denote indexed parameters $b_1, \ldots, b_k$ as $\bar{b}$, and the application $A(\bar{b})$ by $[\![A]\!]_{\bar{b}}$.

ADDs are data structures that facilitate efficient application of arithmetics over Boolean functions. We employ Boolean functions to represent mappings from product-line configurations (Boolean tuples) to corresponding values for quality properties of interest. An important aspect that motivated the use of ADDs for this variability-aware arithmetics is that the enumeration of all configurations to perform Real arithmetics on the respective quality property values (e.g., performance measurements) is usually subject to exponential blowup. Arithmetic operations on ADDs are linear in the input size, which, in turn, can also be exponential in the number of Boolean parameters (i.e., ADD variables), in the worst case. However, given a suitable variable ordering, ADD sizes are often polynomial, or even linear [5]. Thus, for many practical cases, ADD operations are more efficient than enumeration.

An arithmetic operation over ADDs is equivalent to performing the same operation on corresponding terminals of the operands. Thus, we denote ADD arithmetics by corresponding real arithmetics operators. Formally, given a valuation for Boolean parameters $\bar{b} = b_1, \ldots, b_k \in \mathbb{B}^k$, it holds that:

(1) $\forall_{\odot \in \{+,-,\times,\div\}} \cdot (A_1 \odot A_2)(\bar{b}) = A_1(\bar{b}) \odot A_2(\bar{b})$
(2) $\forall_{i \in \mathbb{N}} \cdot A_1^i(\bar{b}) = A_1(\bar{b})^i$

More details on algorithms for ADD operations are outside the scope of this work and can be found elsewhere [5].

3 PRODUCT-LINE ANALYSIS STRATEGIES: TWO EXAMPLES

To motivate the definition of a framework of product-line analysis strategies, we first review two representative analyses for some models and properties in the following sections. Section 3.1 addresses reliability, and Section 3.2 qualitative temporal logic properties.

3.1 Reliability

The reliability of a software system in a given user environment is defined as the probability that the system will give the correct output with a typical set of input data from that user environment [16]. As a representative, we review the approach taken by Castro et al. [13], which models software behavior in a state-space-based fashion by means of a Discrete-Time Markov Chain (DTMC)—a stochastic process that can also be viewed as a transition system labeled with transition probabilities. In a DTMC, states represent (parts of) software modules and transitions represent either a possible transfer of control between modules (with an associated probability) or a module execution failure (with probability $1 - r$, where r is the module's reliability). For simplicity, we constrain this model to have a single initial state (representing the program entry point) and only two terminal (absorbing) states, representing program success (i.e., correct execution) and program failure.

We concentrate on *user-oriented* reliability of a system, which is the probability that, starting from the initial state, the system eventually reaches the success state [16]. This reliability property is computed as a *reachability probability* in the DTMC that serves as the reliability model—that is, 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 [6]. For instance, the calculation of the reliability of the DTMC in the top-left of Figure 2 multiplies the probabilities along the single path to the success state (s_{suc}), which is illustrated by the dotted arrow labeled α and whose result is 0.9801.

However, although DTMCs are convenient to model probabilistic behavior, they cannot cope with variability in the sense of product-line variability. Parametric Markov Chains (PMCs) extend DTMCs with the ability to represent *variable* transition probabilities. These variable transition probabilities can be leveraged to represent product-line variability [17, 36, 65]. For instance, the top-right model in Figure 2 is a PMC in which variability is represented in both transitions leaving state s_1 (highlighted in thick green). To compute the reliability of this model, one can label success states with the atom *"success"* and leverage *parametric model checking* to compute the reachability probability of such states, resulting in the rational expression [39] in the bottom-right corner of Figure 2. This expression has two operands, each of which is a sub-expression corresponding to a path in the PMC leading to the success state (s_{suc}). Indeed, the computed reliability is an expression and not a literal value, as in the case of reliability calculation of the DTMC in Figure 2, since variables in the expression encode variability in the PMC.

For reliability analysis, one can choose two product-line analysis strategies [65]: (1) bind variability in a PMC deriving a variability-free model (i.e., DTMC) for each valid configuration of the product line by evaluation of the variables (i.e., a *projection* π), and then analyze (α) each such DTMC using traditional (not variability-aware) model checking, effectively pursuing a product-based strategy; (2) apply parametric model checking ($\hat{\alpha}$) only once to the PMC, resulting in an expression, which is then evaluated (σ) for each valid configuration, pursuing a family-product analysis. The advantage of the latter approach is that it analyzes the PMC's non-variable transitions only once. In exchange, the product-based strategy can rely on the existence of well established model checking tools such as PRISM [48]; the family-based strategy requires the development of a variability-aware tool.

In our example, PMCs are used to build an *annotative* model of product-line behavior: system states for all variants are present, and variables are used as a means to bypass feature-specific

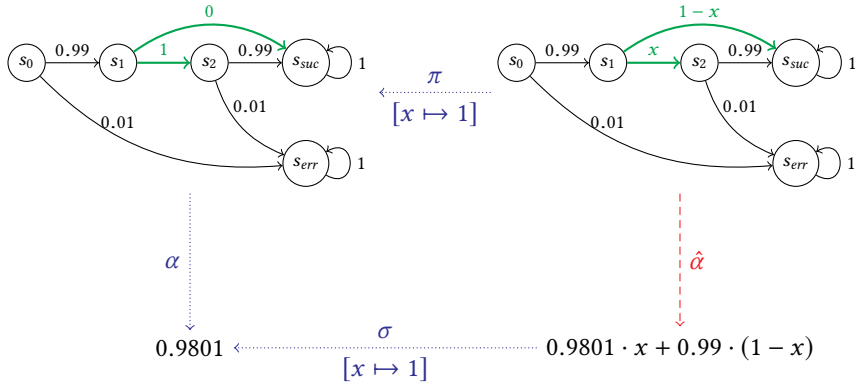


Fig. 2. Example of family-product-based analysis ($\hat{\alpha}$ followed by σ) in contrast to a product-based analysis (π followed by α) of an annotative PMC, for a configuration selecting transition x (i.e., both π and σ bind x to 1). Clockwise from top-right corner: PMC, expression resulting from reliability analysis of the PMC, reliability value of the DTMC corresponding to the configuration, and said DTMC.

states according to a feature selection, which is similar to preprocessor directives in source code. Correspondingly, the expression resulting from parametric model checking of an annotative PMC is call an *annotative expression*. Alternatively, one can also model behavior in a compositional way, also leveraging PMCs to denote variability. In *compositional* PMCs, variables are not meant to be directly bound to Real values, as in the annotative case; instead, they act as placeholders for variant behavior. Note that, to make sense of a set of compositional PMCs, one must resort to a notion of dependencies between them, so that each PMC can be composed in the intended placeholder in the PMC that depends on it.

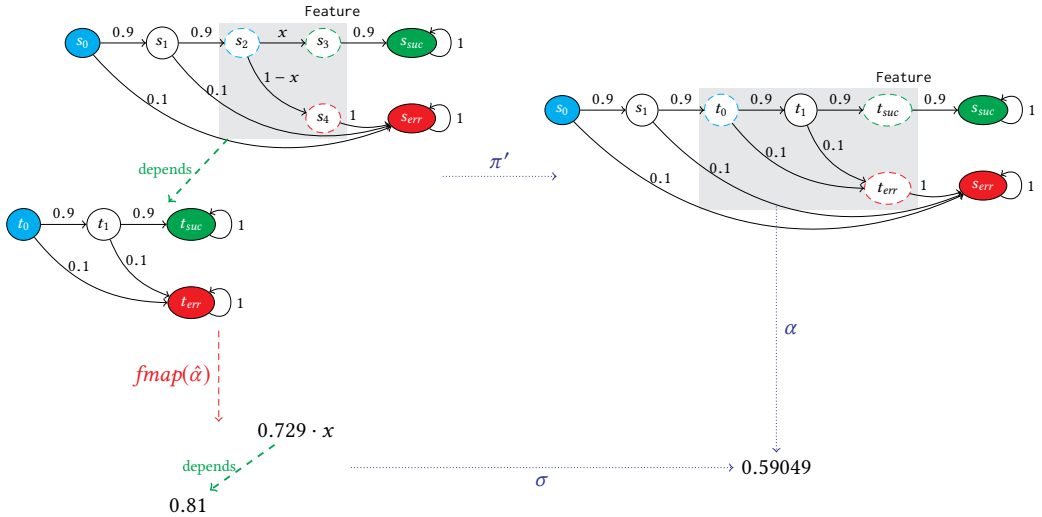
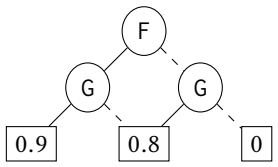


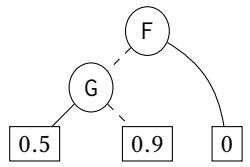
Fig. 3. Example of feature-product-based analysis ($fmap(\hat{\alpha})$ followed by σ) in contrast to a product-based analysis (π' followed by α) of compositional PMCs

Figure 3 illustrates the concept of compositional PMCs. On the top-left, there is a compositional model of a system, consisting of two PMCs: the base behavior with variability points and optional behavior. The base behavior has two parametric transitions (with values "x" and "$1-x$"), enclosed by a dashed rectangle for visualization. Such base PMC *depends* on further optional behavior, since not all behavior of possible products can be derived from the base case alone. The behavioral model of a product with the optional feature enabled can be derived by *composing* (π') the optional PMC into the corresponding placeholder of the base one (top-right corner of Figure 3). Then, the reliability of the composed model can be computed by regular model checking (α). Alternatively, one can perform parametric model checking on each PMC of the compositional model while preserving the dependency relation (using $fmap(\hat{\alpha})$), so that the reliability expression of a PMC is matched to the expressions representing the reliabilities on which that one originally depended (bottom-left corner of Figure 3). The resulting *compositional expressions* can be composed like in PMC composition, yielding a regular probability.

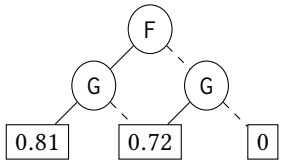
Compositional and annotative models are alternative means to express the behavior of a product line [1], both of which rely on parametric model checking to avoid performing regular model checking for all configurations. However, evaluating the resulting expressions would also need enumeration, which is often intractable in practice due to the large number of configurations. To cope with that problem, expressions can be *lifted* to a semantics based on ADDs [5], for which the encoded Boolean formulas (cf. Section 2.4) represent feature selections. Using this technique, the best-case time complexity of evaluating the reliability expressions for all valid configurations can be polynomial in the number of features, effectively taming the exponential blowup [49]. Note that lifting is semantic; lifted expressions (either compositional or annotative) are syntactically equal to the original one, but their variables are evaluated using ADDs that encode the possible values according to feature selections.



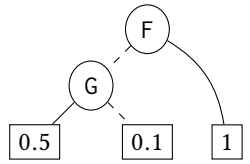
(a) ADD for the values of variable x



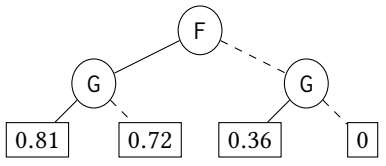
(b) ADD for the values of variable y



(c) ADD for the values of sub-expression $0.9 \cdot x$



(d) ADD for the values of sub-expression $1 - y$



(e) ADD for the expression $0.9 \cdot x \cdot (1 - y)$

Fig. 4. Example of lifted expression evaluation in terms of the presence of features F and G. Each figure denotes a sub-term of the expression $0.9 \cdot x \cdot (1 - y)$ when evaluated with ADD semantics.

As an example, Figure 4 depicts the evaluation of a lifted expression $0.9 \cdot x \cdot (1 - y)$ in a product line with only two features, F and G. In this example, we assume that the ADDs in Figures 4a and 4b denote the possible values for variables x and y, according to a given feature selection. That is, if feature G is selected, but F is not, then x and y evaluate to 0.8 and 0.5, respectively. Figures 4c and 4d present the results of multiplying x by the constant value 0.9 and subtracting y from 1; such operations only affect the leaf nodes and are performed in constant time. Figure 4e shows the result of multiplying the previous two ADDs, an operation that is performed in time proportional to the number of inner nodes.

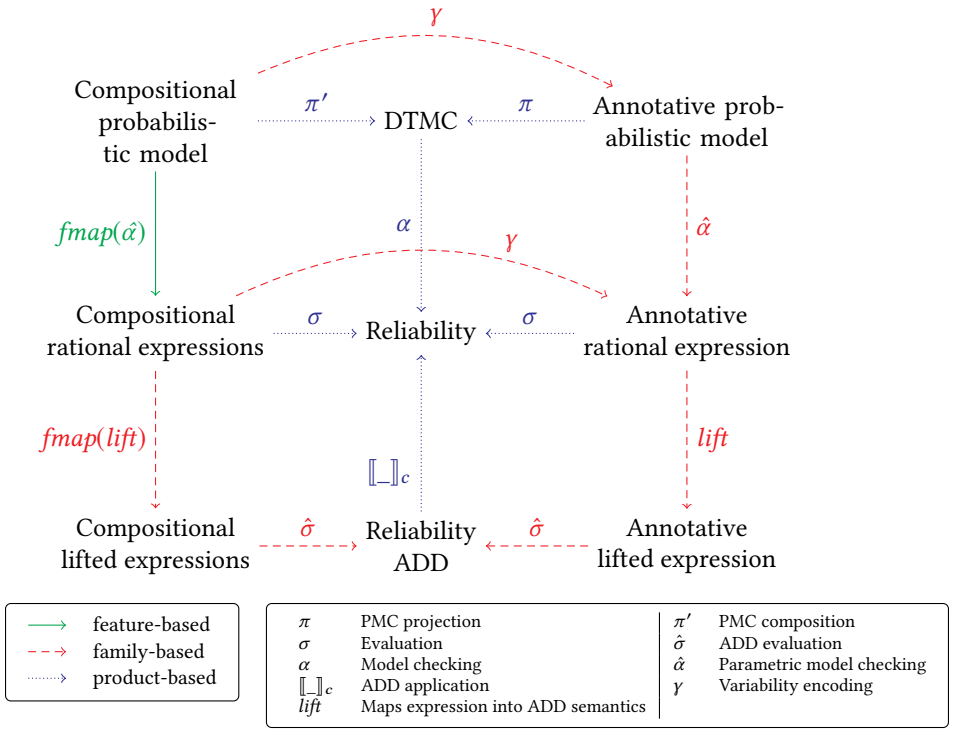


Fig. 5. Commutative diagram of product-line reliability analysis strategies, adapted from Castro et al. [13]

Summary. The analysis choices presented in this section are depicted in Figure 5 [13]. Starting from a compositional model (top-left corner) or an annotative model (top-right corner), different paths eventually lead to the *Reliability* or *Reliability ADD* nodes. Within each path, each arrow represents a function application or analysis step. Analysis steps can be feature-based (green solid arrows), product-based (blue dotted arrows), or family-based (red dashed arrows). Each node in the path represents an intermediate analysis result, the final ones being either Real-valued reliabilities or an ADD representing multiple values. Thus, each path ending in either node is a function composition defining an analysis strategy. Moreover, both compositional probabilistic models and compositional rational expressions can be transformed into corresponding annotative versions by means of *variability encoding* [80] (γ), which leverages a condition operator for PMCs to switch between possible states with a Boolean variable [13].

Castro et al. [13] proved that Figure 5 is a commuting diagram, meaning that different reliability analysis strategies on compositional or annotative models are equivalent (i.e., they yield equal

results) if their corresponding paths share start and end points. For example, commutativity for the top-right quadrant means that, for all annotative probabilistic models *vModel*, and all valid configurations *conf* of the product line, $\sigma(\hat{\alpha}(vModel),conf) = \alpha(\pi(vModel,conf))$. Figure 2 is an instance of this commuting relation.

For illustration, the feature-product-based analysis strategy corresponds to the following choices in Figure 5: starting with a compositional probabilistic model (top-left corner), perform parametric model checking (move down), then evaluate them (move right), yielding a reliability value for a configuration. Such strategy was first proposed and implemented by Ghezzi and Sharifloo [36]. The feature-family-based analysis strategy corresponds to the following choices in Figure 5: starting with a compositional probabilistic model (top-left corner), perform parametric model checking (move down), and then lift the resulting expressions (move down one more step) and evaluate them (move right), yielding a reliability ADD for the product line as a whole. Lanna et al. [49] proposed this strategy.

Nevertheless, neither the strategy proposed by Ghezzi and Sharifloo [36] nor Lanna et al. [49] were conceived within the commutative diagram shown in Figure 5. As a result, the fact that they share the first transformation (down from the compositional probabilistic model) is implicit and thus their formalization and implementation were unnecessarily redundant. Furthermore, neither provide any soundness proofs. In contrast, Castro et al. [13] explore this commonality, providing reusable specifications and proofs of such analyses. In the resulting theory, the proof of the soundness of the feature-family-based strategy directly reuses the proof of the feature-product-based strategy. Nevertheless, Castro's work is limited to reliability analysis and DTMCs.

3.2 Qualitative Temporal Logic Properties

While reliability analysis, addressed in Section 3.1, is concerned with computing a quantitative property (i.e., the probability of eventually reaching success states of a system), in this section we focus on analysis of qualitative temporal logic properties, that is, properties that the system satisfies with certainty [7]. In what follows, we refer to such properties as behavioral properties.

Checking the behavior of a product line, as opposed to a single system, can be seen as the problem of verifying a set of transition systems, one for each product. A product-based verification strategy would check each of these transition systems individually. However, in a product line, products share common behavior. Similarly to reliability (cf. Section 3.1), one could thus reduce the verification time for the whole product line by checking behavior that is common to multiple products *only once*. Previous research aimed at defining concise formalisms to encode variable product line behavior and designing efficient verification algorithms that factorize the verification effort relying on modal transition systems [34, 42], multi-valued model checking [14] and featured transition systems [20]. A comprehensive survey and comparison of these approaches is available elsewhere [24, 70, 71, 73, 77]. In what follows, we analyze the work surrounding featured transition systems (FTS). We will demonstrate that the structure of the commutative diagram presented in Figure 5 also applies to this modeling formalism. This is a first indication that the diagram and the framework we devise generalize to other product-line analyses, as we further discuss in Section 5.1.

As an illustrative example, Figure 6 depicts an FTS modeling a product line of vending machines. In addition to an action, a transition in FTS is also labeled with an expression, called *feature expression*, which defines the set of products able to enable this transition. For instance, the transition from state 3 to state 6 is labeled with the feature expression t, meaning that it can be executed only by products that include the corresponding feature t. The transition from state 1 to state 2 is labeled with $\neg f$, and thus can be executed only by products that do not have the feature f. More generally, the expression can be any formula that represents a subset of products. The transition

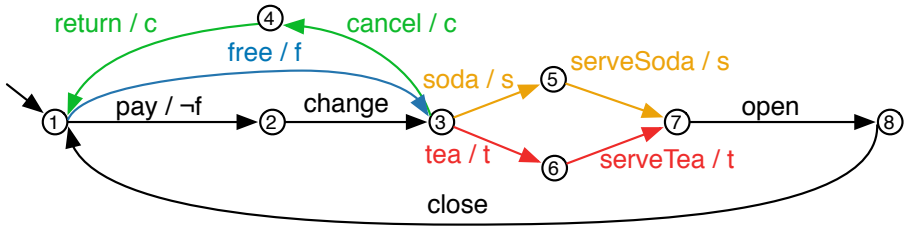


Fig. 6. The FTS modeling a vending machine product line

system representing the behavior of a particular product is obtained by removing all transitions not available to this product. This operation is called *projection* [20].

Most of FTS-based analyses rely on annotative models to design efficient CTL and LTL verification algorithms [19, 20, 25]. The result is a feature expression representing all the products that satisfy the property under verification. This feature expression can be represented in different ways. One representation sees a feature expression as two sets of features [21]: one set includes the features required to satisfy the property; the other contains the features that must be excluded. Alternatively, feature expressions can be represented as Boolean formulae in which included (resp. excluded) features appear as positive (resp. negative) literals [20] (as done in Figure 6). The result of a given verification analysis is another feature expression, produced by computing conjunctions and disjunctions of the individual expressions.

Figure 7 illustrates such variability-aware analysis of FTS, where the resulting feature expression is denoted as a Boolean formula. On the top right corner, an FTS models the behavior of the vending machine product line. Let us assume that this FTS is checked against the property "the machine can only `return` if `pay` occurred", which is violated by any product having both free drinks (f) and cancel (c) features. As an additional example, we consider a feature selection for the vending machine selling soda and tea without the free drinks and cancel features. The family-based analysis step $\hat{\alpha}$ applied to the FTS yields the feature expression $\neg(c \wedge f)$. Then, pursuing a family-product-based strategy, one can evaluate this formula under the feature valuation corresponding to the aforementioned product. Doing so would yield $True$, meaning that the product satisfies the property. Alternatively, one could follow a product-based strategy by projecting the FTS onto the specific product (π), yielding the transition system shown on the top left corner. Thus, a classical verification procedure α yields that the product indeed satisfies the property.

Whatever representation one chooses for a feature expression, it can be concisely encoded in a Binary Decision Diagram (BDD) [12, 19], similar to the ADD encoding of algebraic expressions in Figure 4. The analysis of Classen et al. [19] works fully symbolically: both the feature expressions and the state space are encoded as BDDs. The analysis proceeds classically, by induction on the structure of the CTL property, and by computing fixpoints backwards. In contrast, the analysis of Classen et al. [20] is semi-symbolic: the state space of the model and the automaton of the property are represented explicitly, but the feature expressions are represented as BDDs. Model checking is performed forwards, as in SPIN [41], but some states may be re-explored if they are reached with a new combination of features.

FTS is a formalism that falls into the category of *annotative* models. Higher-level formalisms are also availlable, most notably fPromela [20] (inspired by Promela [41]). Compositional models also exist. For example, Plath and Ryan [61] extended the SMV language to express how a feature modifies an SMV model, called fSMV. In fSMV, each feature is modeled in isolation, which paves

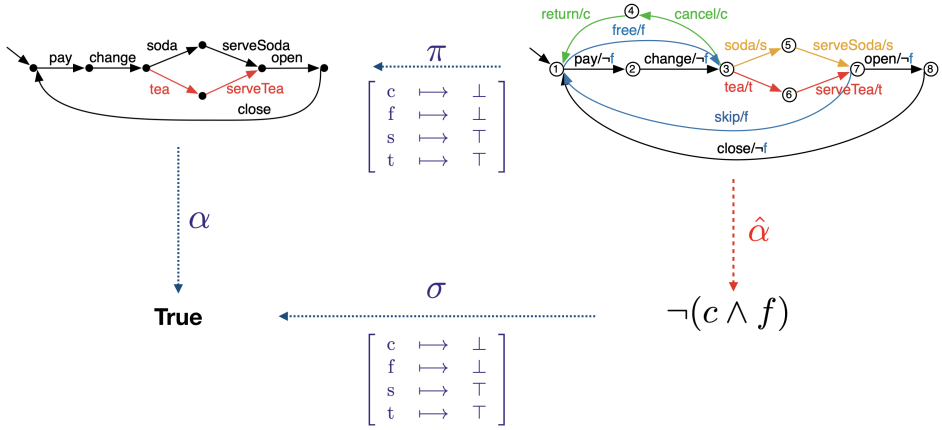


Fig. 7. Product-based and family-product-based strategies applied to FTS verification

the way for *compositional* reasoning. Classen et al. [19] showed that any fSMV model can be transformed into an FTS and vice versa, thereby proving that the compositional model is equivalent to the annotative one. This implies that any verification algorithm designed for one model also works for the other.

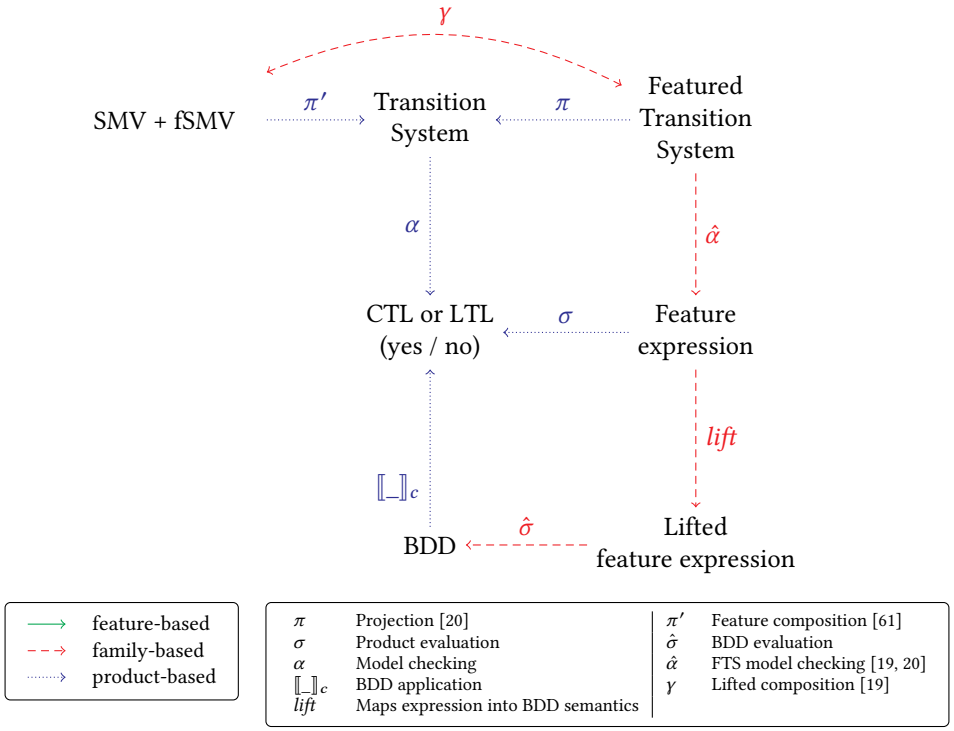


Fig. 8. Commutative diagram of product-line behavioral properties verification strategies

Summary. The above review of the state of the art of FTS verification suggests the potential partial reuse of the structure of the commutative diagram for reliability analysis (Figure 5), resulting in Figure 8. From an FTS (see top right of Figure 8), one can pursue a product-based strategy by computing the projection (π) of each product and applying a standard model checking algorithm (α). In contrast, family-based strategies start with the application of dedicated algorithms to FTS ($\hat{\alpha}$) to obtain the set of products satisfying the property. This set can either be enumerated (σ), giving rise to a family-product-based strategy, or lifted into a feature expression encoded in BDD semantics and then being evaluated without enumeration into a BDD ($\hat{\sigma}$).

As for the compositional fSMV model of Plath and Ryan [61], the transition system of a particular product can be obtained by composing the base model with the model of each of its features (π'). A family-based approach can be followed to produce directly an FTS from the *lifted composition* [19] of the base model and all the features (γ). Feature-based strategies have been proposed under specific assumptions (e.g., that features only add behavior) [23], but do not apply outside such restricted scope. The soundness of the analyses is proved by the commutativity of the upper-right quadrant of Figure 8 and by the equivalence result by Classen et al. [19], much like for reliability analysis (Figure 2).

4 A FORMAL FRAMEWORK FOR SOFTWARE PRODUCT LINE ANALYSIS

In this section, we generalize and formalize the discussion of Section 3. We first present an overview of the envisioned analysis framework (Section 4.1) and formalize key aspects (Section 4.2).

4.1 Analysis Framework Overview

The framework aims at precisely and uniformly describing product-line analyses and their key properties. To create the framework, we reviewed existing analyses for the different models and properties mentioned in Section 3, and we identified essential abstractions of such analyses and their structure. We started from a number of specific models and properties, such as the reliability analysis mentioned in Section 3.1. We then started a process of formulating more general concepts by means of abstraction. This also required assessing the impact on dependent concepts, which also had to be reformulated. Finally, we identified assumptions that such elements should fulfill to keep the diagram's structure.

For example, in Figure 5, by abstracting *DTMC* into *Product*, derivation by projection (π) also needs to be abstracted, so that π's domain becomes a generic *Product* model with annotations, its co-domain becomes *Product*, and its semantics is generalized to bind variability in the generic *Product* model with annotations. The goal is that the framework accommodates, at least, the analyses of Section 3, and becomes a generic theory, consisting of a structure of key concepts related to product-line analyses. Then, the theory should be further evaluated with instantiations for different product models and properties, as discussed in Section 5.1.

Figure 9 and Table 1 synthesize the outcome of this abstraction process. At a coarse grain, Figure 9 is a generic commutative diagram abstracting the structures presented in Figures 5 and 8. At a fine grain, Table 1 relates concrete elements of Figures 5 (reliability analysis) and 8 (behavioral properties) to corresponding generic elements in Figure 9, together with a brief description of the latter elements in the generic framework. These generic elements correspond to key abstractions of both the taxonomy proposed by Thüm et al. [73] and of the product-line representation by Kästner et al. [44] and von Rhein et al. [80], which have been used to describe and analyze a multitude of product lines [73].

The diagram shown in Figure 9 offers different ways to analyze a given *Property* of a product line of *Product* members. We can obtain a product (variability-free model) by *projection* (π), in the case of annotative product lines, or model *composition* (π'), in the case of compositional product

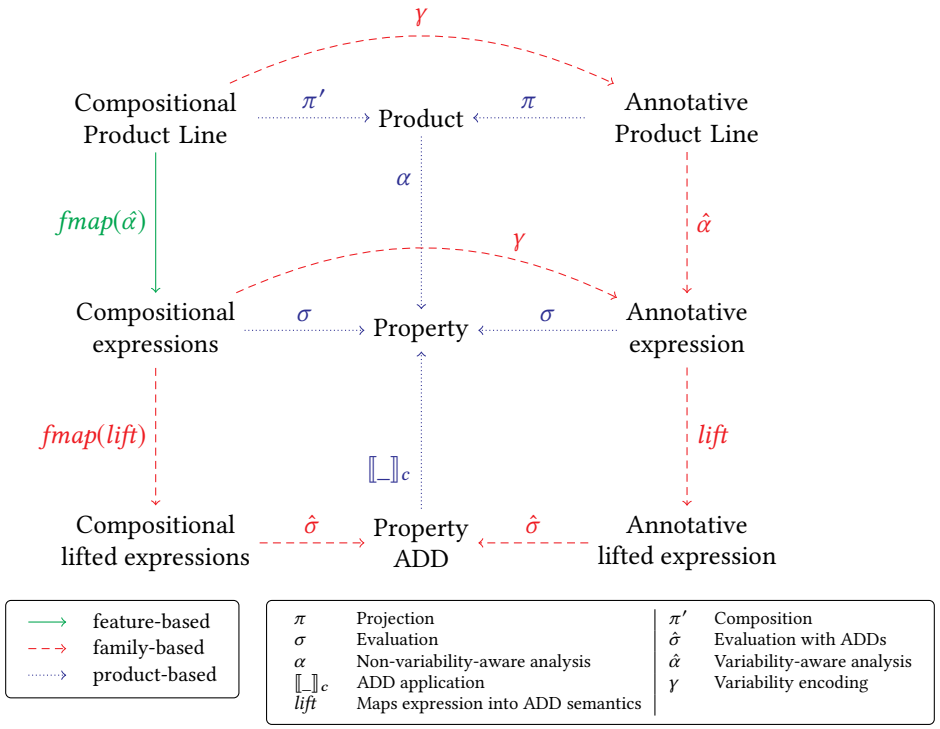


Fig. 9. Commutative diagram of our software product line analysis framework

lines, then proceed with a non-variability-aware analysis α. Alternatively, for annotative product lines, we can apply a *variability-aware analysis* $\hat{\alpha}$ to obtain an *annotative expression*, which is either evaluated for a given product (σ) or further abstracted into an *annotative lifted expressions* (*lift*). The latter is evaluated into a concise representation as an ADD ($\hat{\sigma}$). In the case of compositional product lines, *variability-aware analysis* is applied to the compositional model in a structure-preserving way (*fmap*($\hat{\alpha}$)), yielding a *compositional expression*, which is again either evaluated for a given product (σ) or further abstracted into a *compositional lifted expression* (*fmap*(*lift*)) for later evaluation into an ADD ($\hat{\sigma}$).

In either case, the purpose of lifting operations is to provide a representation of expressions enabling their non-enumerative evaluation (cf. Section 2.4), which is key to the family-based dimension of the analysis. Furthermore, an ADD-based representation is often used for a space-efficient encoding of values and also encompasses a BDD-based representation, since the value nodes in an ADD can represent values other than Booleans.

Compositional product lines and compositional expressions can be transformed into corresponding annotative versions by means of *variability encoding* [80] (γ). In addition to the examples presented in Section 3, this is also possible for Lightweight Java [68], as has been shown elsewhere [45], whereby complete refactorings enable transforming physical separation of features (compositional product line) to their virtual separation counterpart (annotative product line).

Similarly to Figures 5 and 8, each path from a compositional or annotative product line to a property value defines an analysis strategy, which amounts to function composition of the intermediate analysis steps (arrows in the diagram). The nodes in the diagram represent models

or abstractions thereof after applying a series of analysis steps. These nodes and their underlying structure are defined in Section 4.2 and further discussed in Section 5.2.

Our framework allows us to characterize a product-line analysis strategy in a compositional manner. For instance, the feature-family-based analysis (in Figure 9, from compositional product line, down, down, and right) and the feature-product-based analysis (from compositional product line, down, and right) share the feature dimension of the analysis (down from compositional product line). Furthermore, based on Figures 5 and 8, one could conjecture that the diagram in Figure 9 is also a commuting diagram: different analysis paths yield equal results if they share the start and end points. In fact, this is indeed the case, according to the framework formalization that we describe in the following section.

Table 1. Synthesis of abstraction process defining our software product line analysis framework (element view)

Framework Element	Description	Reliability Analysis Element	Qualitative Temporal Logic
Property	Computable property	Reliability	Temporal logic property
Product	Variability-free model	DTMC	Transition Systems
Analysis (α)	Variability-free analysis	Model checking	Model checking
Annotative Product Line	Product line with annotative representation	Annotative probabilistic model	Featured Transition Systems
Compositional Product Line	Product line with compositional representation	Compositional probabilistic model	SMV+fSMV
Annotative expression	Expression computing property value at products	Annotative rational expressions	Feature expression
Compositional expression	Compositional counterpart of annotative expressions	Compositional rational expressions	–
Annotative lifted expression	ADD-encoded annotative expressions	Annotative lifted expression	Lifted feature expression
Compositional lifted expression	ADD-encoded compositional expressions	Compositional lifted expressions	–
Property ADD	Maps product configurations to property values	Reliability ADD	BDD
Projection (π)	Product derivation by presence condition projection	PMC projection	projection
Composition (π')	Product derivation by model composition	PMC composition	feature composition
Variability-aware analysis ($\hat{\alpha}$)	SPL analysis exploring commonalities (sharing)	Parametric model checking	FTS model checking
Evaluation (σ)	Expression evaluation to a property value	rational expression evaluation	Product evaluation
Evaluation with ADDs ($\hat{\sigma}$)	Expression evaluation to an ADD	ADD evaluation	BDD evaluation
lift	Maps expression to ADD semantics	*lift*	*lift*
Variability encoding (γ)	Maps compositional into annotative representation	Variability encoding	Lifted composition
ADD application ($[\![\_]\!]_c$)	Property value mapped by a configuration	ADD application	BDD application

4.2 Formalization

To formally underpin our framework, we developed a machine-verified theory comprising formal specification and verification of key concepts and properties of product-line analysis. This theory is specified and checked using the PVS proof assistant [58], to ensure precise specification and to avoid unsound proof steps, since manual demonstrations are prone to human mistake. In particular, this formalization defines abstract functions and types modeling essential abstractions in this problem domain, including analysis steps, models, and intermediate analysis results. Hence, the details of concrete analysis strategies, such as reliability analysis using PMC or behavioral analysis over FTS, are abstracted. The mechanization allows us to derive machine-verified soundness proofs of key results, such as commutativity of analysis strategies. This way, we increase confidence that our framework can be reused to safely establish different product-line analysis strategies for specific models and properties.

4.2.1 The PVS Proof Assistant. PVS provides mechanized support for formal specification and verification, including a specification language and a theorem prover. Specifications consist of collections of theories. Each theory consists of signatures for the types introduced in the theory, and the axioms, definitions, and theorems associated with the signature. Specifications are strongly typed—every expression has an associated type. The specification language is based on classic, typed higher-order logic.

PVS also provides mechanisms such as theory parameterization and interpretation, enabling us to consider variability when specifying our theories. We may use parameters when defining a theory,

which provides support for *universal polymorphism*. PVS offers separate mechanisms for importing a theory with axioms, and for interpreting a theory by supplying a valid interpretation [57]. The theory interpretation mechanism enables us to show that a theory is correctly interpreted by another theory under a user-specified interpretation for the uninterpreted types and constants. We can use interpretations to show that an implementation is a correct refinement of a specification, that an axiomatically defined specification is consistent, or that an axiomatically defined specification captures its intended models. Axioms defined in the theory being interpreted generate proof obligations.

In what follows, we present our formalization using a simplified notation, abstracting from PVS syntax, to facilitate the communication of ideas and knowledge. We first present general framework definitions, and then proceed with a formalization of annotative product lines, which forms the basis for specifying compositional product lines. The full PVS mechanization, comprising all definitions and proofs, is available online.[1]

4.2.2 General Framework Definitions. We define *Product* as a variability-free model defining the type of the product line members. We do so by using uninterpreted types in PVS, which are a way of introducing types with almost no constraints, other than the fact that such types are disjoint from all other types. Products have a computable *Property* of interest, such as reliability. In existing analyses, this property usually has a numerical or Boolean type [73]—reliability, safety, performance, etc. To increase generality, we still define it using uninterpreted types. We introduce special elements $emptyproduct \in Product$ and $emptyproperty \in Property$, to be used as base cases for recursive functions in the compositional model evaluation. We use type *Conf* to represent a configuration. For generality, we also use uninterpreted types to abstract the specific syntax of product configuration. For instance, it could be a set of selected features from a product line or a Boolean formula. Finally, function α is a variability-free analysis that performs the computation of a *Property* for a given *Product*, such as reachability probability analysis for reliability (cf. Section 3.1). This function is also uninterpreted, but it must obey the constraint that analyzing an *emptyproduct* yields *emptyproperty*.

Definition 1 (Computing Properties from Products)**.** We compute a $Property$ from a $Product$, using a function $\alpha : Product \rightarrow Property$, such that $\alpha(emptyproduct) = emptyproperty$.

4.2.3 Annotative Product Lines. On the right-hand side of the diagram in Figure 9, *Annotative Product Line* comprises a feature model, configuration knowledge, and a variant-rich model called *annotative model*. The latter is an extension of *Product* including optional presence conditions attached to model elements. Therefore, *AnnotativeModel* is either a base variability-free *Product* model, or a variation point with choices depending on a presence condition. We use abstract data types to represent this, in which we provide a set of type constructors, such as *ModelBase* and *ModelChoice*, along with associated accessors. This allows us to extract arguments from the constructors, such as the product, or the presence condition. Definition 2 presents this data type in a simplified way.

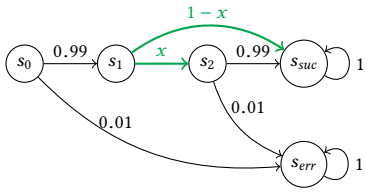
Definition 2 (Annotative Model)**.** An annotative model is either:

(1) $ModelBase(m : Product)$, denoting a variability-free product model; or
(2) $ModelChoice(pc : PresenceCondition, vm_1 : AnnotativeModel, vm_2 : AnnotativeModel)$, denoting variation according to a presence condition.

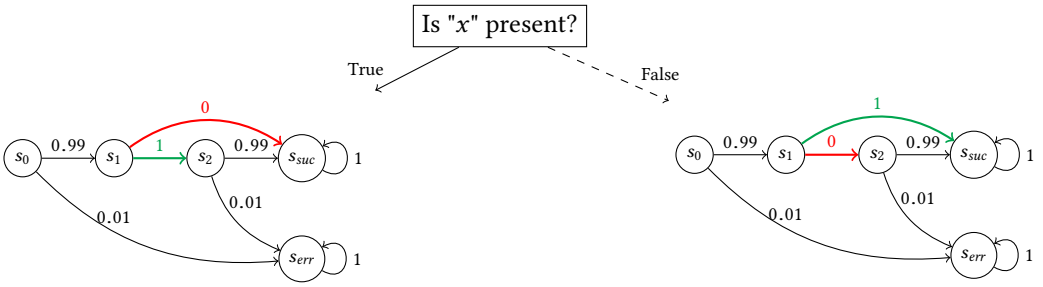
The intuition behind this specification is that *AnnotativeModel* is a data structure representing the *semantics* of an annotative product line. That is, an *AnnotativeModel* can be seen as a decision tree

[1] https://github.com/thiagomael/spl-analyses-mechanization/

in which the internal nodes are presence conditions (to be checked against a given configuration) and the leaf nodes are all possible products. For instance, Figure 10 illustrates how an annotative model of a product line (the example PMC from Figure 2) could be interpreted according to this concept.



(a) Concrete annotative PMC, denoting the behavioral model of a product line



(b) Variational semantics (*AnnotativeModel*) of the same PMC, represented as a decision tree

Fig. 10. Intuition of how an annotative model satisfies our specification in Definition 2 (assuming that x is bound to 1 if the corresponding feature is selected, and 0 otherwise)

Note that Definition 2 is not meant to be directly implemented; otherwise, modeling a product line would require that all products be modeled individually, which defeats the whole purpose of product line engineering. However, specification-wise, having an abstraction of all possible products helps on stating and proving theorems that quantify (universally or existentially) over the solution space.

Using this semantic notion of an *AnnotativeModel* in the form of a decision tree, product derivation becomes a matter of evaluating presence conditions until a leaf node (i.e., product) is reached. This product derivation process is denoted by function π, which receives an annotative model and a configuration, and yields a product.

Definition 3 (Product Derivation from Annotative Models)**.** Given an annotative model and a configuration c, product derivation is performed by function $\pi : \mathit{AnnotativeModel} \longrightarrow \mathit{Conf} \longrightarrow \mathit{Product}$, such that:

(1) $\pi\Big(\mathit{ModelBase}(m), c\Big) = m$; and

(2) $\pi\Big(\mathit{ModelChoice}(pc, vm_1, vm_2), c\Big) = \begin{cases} \pi(vm_1, c) & \text{if } c \models pc \quad \textit{(presence condition is satisfied)} \\ \pi(vm_2, c) & \text{otherwise} \end{cases}$

Similar to annotative models, annotative expressions are specified using a representation of choice semantics. Hence, we use the *BaseExpression* constructor for leaf nodes (denoting property values of products, such as the bottom-left corner of Figure 2) and *ChoiceExpression* to introduce decision nodes.

Definition 4 (Annotative Expression)**.** An annotative expression is either:

(1) *BaseExpression*(p: *Property*), denoting a property computed from a product; or
(2) *ChoiceExpression*(pc: *PresenceCondition*, e_1: *AnnotativeExpression*, e_2: *AnnotativeExpression*), denoting choices guarded by a presence condition.

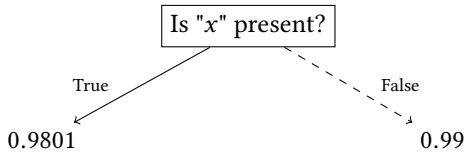


Fig. 11. Variational semantics (*AnnotativeExpression*) of the expression $0.9801 \cdot x + 0.99 \cdot (1 - x)$ in Figure 2, represented as a decision tree (assuming that x is bound to 1 if the corresponding feature is selected, and 0 otherwise)

Figure 11 illustrates the intuition of an *AnnotativeExpression* encoded in this way. It encodes the semantics of property values in terms of product-line configurations, so that annotative expression evaluation is defined recursively much like we did for product derivation.

Definition 5 (Annotative Expression Evaluation)**.** Given an annotative expression and a configuration c, evaluation is performed by function σ : *AnnotativeExpression* $\rightarrow$ *Conf* $\rightarrow$ *Property*, such that:

(1) $\sigma\Big(\mathit{BaseExpression}(p), c\Big) = p$; and

(2) $\sigma\Big(\mathit{ChoiceExpression}(pc, e_1, e_2), c\Big) = \begin{cases} \sigma(e_1, c) & \text{if } c \models pc \quad \textit{(presence condition is satisfied)} \\ \sigma(e_2, c) & \text{otherwise} \end{cases}$

Since the data types for both annotative models and annotative expressions denote the semantics of the corresponding artifacts, we specify function $\hat{\alpha}$ (variability-aware analysis on annotative models) by recursively mapping their structures. In the case that the model has no variability, a variability-free expression (i.e., the result of applying α to the model) is returned. Otherwise, an annotative choice expression is yielded.

Definition 6 (Variability-aware Analysis on Annotative Models)**.** Given an annotative model, we define the function $\hat{\alpha}$: *AnnotativeModel* $\rightarrow$ *AnnotativeExpression* to perform variability-aware analysis, such that:

(1) $\hat{\alpha}\Big(\mathit{ModelBase}(m)\Big) = \mathit{BaseExpression}\Big(\alpha(m)\Big)$; and

(2) $\hat{\alpha}\Big(\mathit{ModelChoice}(pc, vm_1, vm_2)\Big) = \mathit{ChoiceExpression}\Big(pc, \hat{\alpha}(vm_1), \hat{\alpha}(vm_2)\Big)$.

Furthermore, the structure of the top-right quadrant in Figure 9 means that, for a given configuration *conf*, the evaluation of the annotative expression obtained by $\hat{\alpha}$ from an annotative model *vModel* yields the same result as if the variability had first been bound for such configuration, and then the resulting product analyzed via the non-variability-aware analysis α. Formally specifying and proving this key property of family-based analyses is often overlooked [73].

Theorem 1 (Soundness of family-product-based analysis)**.**

$$\forall_{\mathit{vModel},\ \mathit{conf}} \cdot \sigma\Big(\hat{\alpha}(\mathit{vModel}), \mathit{conf}\Big) = \alpha\Big(\pi(\mathit{vModel}, \mathit{conf})\Big)$$

Proof sketch. For an arbitrary configuration $conf$, we need to prove that $\sigma(\hat{\alpha}(vModel), conf) = \alpha(\pi(vModel, conf))$ holds for any annotative model $vModel$. We prove this by induction over the structure of $vModel$. This generates two subgoals, which correspond to the possibilities for the type *AnnotativeModel*. In the first subgoal, we need to prove that $\sigma(\hat{\alpha}(ModelBase(m)), conf) = \alpha(\pi(ModelBase(m)), conf)$. By expanding $\hat{\alpha}$ on the left-hand side (LHS), and product derivation π on the right-hand side (RHS), we then have to prove that $\sigma(BaseExpression(\alpha(m)), conf) = \alpha(m)$. We then expand σ on the LHS, resulting on $\alpha(\text{m}) = \alpha(\text{m})$, which is trivially true.

In the second subgoal, for arbitrary presence condition pc and annotative models am_1 and am_2, we must prove that $\sigma(\hat{\alpha}(ModelChoice(pc, am_1, am_2)), conf) = \alpha(\pi(ModelChoice(pc, am_1, am_2), conf))$. As induction hypotheses, we have that $\sigma(\hat{\alpha}(am_1), conf) = \alpha(\pi(am_1, conf))$ and $\sigma(\hat{\alpha}(am_2), conf) = \alpha(\pi(am_2, conf))$. Expanding σ and $\hat{\alpha}$ on the LHS, and π on the RHS of the proof goal, we then have two possible situations, corresponding exactly to the two induction hypotheses. If $conf$ satisfies the presence condition pc, we must prove that $\sigma(\hat{\alpha}(am_1), conf) = \alpha(\pi(am_1, conf))$, which is already given by the induction hypotheses. Otherwise, we have to prove $\sigma(\hat{\alpha}(am_2), conf) = \alpha(\pi(am_2, conf))$, concluding the proof of this subgoal, since this is also given by the induction hypotheses. □

At this point, it is important to note that $\hat{\alpha}$ itself maps the semantics of an annotative model to the semantics of a corresponding annotative expression *by construction*. In other words, the framework assumes that each *BaseProduct* p (leaf node in an *AnnotativeModel*) is mapped to a corresponding *BaseExpression* $\alpha(p)$ (leaf node in an *AnnotativeExpression*). Therefore, to instantiate the framework, one is *required* to prove that such assumption holds for the concrete models and the concrete analysis at hand.

4.2.4 Compositional Product Lines. On the left-hand side of the diagram in Figure 9, *Compositional Product Line* comprises a feature model, configuration knowledge, and a variant-rich model called *compositional model*.

Definition 7 (Compositional Model)**.** A compositional model is a tuple $(idt, E, \prec, top)$, where:

- idt is a finite set of Natural numbers, meant to be *identifiers*;
- $E : \mathbb{N} \rightarrow AnnotativeModel$ is a function that maps identifiers in idt to *AnnotativeModel*s;
- $\prec$ is a well-founded dependency relation between identifiers from idt;
- $top \in idt$ denotes the identifier of the *root* model—i.e., a model on which no other depends.

A compositional model is a named finite set of annotative models with an associated dependency relation between them, which denotes possible compositions. For a *CompositionalModel* cm, $cm'top$ identifies the base for composition. The members of a *CompositionalModel* have type *AnnotativeModel* to denote that, in a concrete setting, such artifacts are expected to carry some sort of annotation that indicates where to perform composition (variation points). These annotations can be placeholders as in Figure 3, for instance. In some contexts, such as feature-oriented programming [64], it may seem that models should be specified as regular products; however, feature modules are essentially slices with a different semantics for `super` (which refers to the same method in another module) and `class` (which is actually a partial class or mixin). Previous work [45] explores integration strategies for annotative and compositional mechanisms.

We exploit the dependency relation among the constituent elements of a *CompositionalModel* to define function *dependents*. This function takes as input a *CompositionalModel* and an identifier i for a particular annotative model therein, and yields a list of dependents of such model, consisting of a pair (pc, id), formed by the id of the dependent model and its presence condition. The latter can be obtained from the structure of an *AnnotativeModel* through function *getPC*.

Definition 8 (Extracting dependent models)**.** Given a compositional model cm and a natural identifier $i \in cm'idt$, function *dependents* yields a list of pairs (pc, id) such that $pc = getPC(cm, id)$ and $id \prec i$.

To either bind or encode variability in compositional models, basic behavior is needed for the composition of elements in each node of the relation, incorporating dependent elements from the recursive composition along the structure. Therefore, we assume the existence of a function $partialModelComposition : AnnotativeModel \rightarrow AnnotativeModel \rightarrow AnnotativeModel$, which denotes the composition mechanism for *AnnotativeModel*.

Derivation by *composition* (π') operates on compositional product lines generating a product for a given configuration of its feature model. In particular, π' relies on auxiliary function π'_r to recursively perform a bottom-up composition of annotative models bound within the compositional model of the product line by leveraging function *partialModelComposition*. Since the dependency relation is well-founded, this recursion is guaranteed to terminate.

Definition 9 (Derivation by composition)**.** Given a compositional model cm and a configuration c, product derivation is performed by function $\pi' : CompositionalModel \rightarrow Conf \rightarrow Product$, such that $\pi'(cm, c) = \pi'_r(cm, cm`top, c)$. The auxiliary function π'_r expects a compositional model cm, an identifier $i \in cm'idt$, and a configuration c, such that:

$$\pi'_r(cm, i, c) = \pi\Big(foldl\big(partialModelComposition, cm'E(i), map(f, dependents(cm, i))\big), c\Big)$$

where f receives a pair $(pc, idt) \in dependents(cm, i)$ as an argument, and is defined as:

$$f(pc, idt) = \begin{cases} ModelBase\Big(\pi'_r(cm, idt, c)\Big) & \text{if } c \models pc \\ ModelBase(emptyproduct) & \text{otherwise} \end{cases}$$

In the same vein, compositional expressions are defined by type *CompositionalExpression* akin to *CompositionalModel*. The only difference is that field E now maps to *AnnotativeExpression* instead of *AnnotativeModel*. We obtain the list of dependent expressions through *dependents*, and *partialExpComposition* binds the composition mechanism for *AnnotativeExpression*. Evaluation of a *CompositionalExpression* is given by σ', which operates like π'. For brevity, we omit these definitions, which can be found in our repository.

The correspondence between *CompositionalExpression* and *CompositionalModel* is a foundation for applying $\hat{\alpha}$ to map component models to corresponding expressions, preserving the structure implied by the dependency relation. Placeholders in a model should have corresponding markers in the mapped expression, so as to preserve composition semantics. However, the mechanism by which *partialModelComposition* and *partialExpComposition* work depends on concrete models; hence, we leave both functions uninterpreted and specify a constraint that needs to be satisfied, given by the assumption below (Axiom 1). Essentially, such assumption means that $\hat{\alpha}$ is compositional.

Axiom 1 (Compositionality of $\hat{\alpha}$)**.**

$$\forall_{m_1,\ m_2} \cdot \hat{\alpha}\Big(partialModelComposition(m_1, m_2)\Big) = partialExpComposition\Big(\hat{\alpha}(m_1), \hat{\alpha}(m_2)\Big)$$

Similarly to the structure of the top-right quadrant of Figure 9, the top-left quadrant means that obtaining a compositional expression by mapping $\hat{\alpha}$ over a given compositional model *vModel* and then evaluating that expression against a configuration *conf* yields the same result as applying α to the product derived from *vModel* and *conf*.

Theorem 2 (Soundness of feature-product-based analysis)**.**

$$\forall_{vModel,\ conf} \cdot \sigma'\Big(fmap(\hat{\alpha}, vModel), conf\Big) = \alpha\Big(\pi'(vModel, conf)\Big)$$

Proof sketch. For an arbitrary configuration *conf* and compositional model *vModel*, we need to prove that $\sigma'(fmap(\hat{\alpha}, vModel), conf) = \alpha(\pi'(vModel, conf))$. By expanding σ' and π', we then have to prove that $\sigma'_r(fmap(\hat{\alpha}, vModel), fmap(\hat{\alpha}, vModel)'top, conf) = \alpha(\pi'_r(vModel, vModel'top, conf))$. By generalizing $vModel'top$ and applying well-founded induction, we must prove that, for a given identifier $x \in vModel'idt$, $\sigma'_r(fmap(\hat{\alpha}, vModel), x, conf) = \alpha(\pi'_r(vModel, x, conf))$. By expanding σ'_r and π'_r, we have to prove that

$$\sigma\Big(foldl\big(partialExpComposition, fmap(\hat{\alpha}, vModel)'E(x), map(\ldots)\big), conf\Big)$$
$$= \alpha\Big(\pi\big(foldl(partialModelComposition, vModel'E(x), map(\ldots)), conf\big)\Big)$$

We omit the arguments of *map* operations inside *foldl*, as their structure can be found in Definition 9. We can reuse Theorem 1, instantiated with $foldl(partialModelComposition, vModel'E(x), map(\ldots))$. By replacing that in our goal, we can use Axiom 1 instantiated with $cm'E(x)$ and the *map* operation used in the theorem instantiation (see above). We then have to prove the following:

$$\sigma(foldl(partialExpComposition, \hat{\alpha}(vModel'E(x)), map(\ldots)), conf)$$
$$= \sigma(foldl(partialExpComposition, \hat{\alpha}(vModel'E(x)), map(\hat{\alpha}, map(\ldots)), conf)$$

We see that the right-hand side consists of two nested map operations, the outermost being the application of $\hat{\alpha}$ to the list yielded by mapping over each pair p yielded by $p \in dependents(vModel, x)$. The left-hand side has a single map operation, which is applied over each pair p given by $p \in dependents(\text{fmap}(\hat{\alpha}, vModel), x))$. To conclude the proof, we need to establish the equivalence of those map operations, which follows from finite induction on the list of dependents. □

The proof relies on Axiom 1, which is an abstraction over the particular types of the model and the non-variability-aware analysis function α. Hence, when these types are instantiated, PVS automatically generates a corresponding theorem (a *proof obligation*) to establish that α is indeed compositional. This fact highlights one of the key advantages of using PVS: its type system *generates* theorems in the form of obligations, which could go unnoticed in a handcrafted specification. While addressing proof obligations requires some effort, it is inherent to the analysis at hand. Obligations aside, the proof of the theorems on the commutativity of the upper-left corner is completely reusable across different models, properties, and instantiations of non-variability-aware analysis.

5 DISCUSSION

The formalization described in Section 4.2, which was mechanized into PVS, provides formal evidence on the validity of the framework. In this section, we qualitatively evaluate the framework. We first discuss the framework's generality (Section 5.1). Then, we discuss findings, related strengths, and limitations (Section 5.2). Finally, we discuss threats to validity (Section 5.3).

5.1 Framework's Generality

In this section, we qualitatively assess the framework's generality by retrospectively discussing to what extent it can describe existing product-line analyses. In particular, we show how the framework's elements, previously listed in Table 1 and within the structure of the generic diagram in Figure 9, can be related to different models, properties, and analyses. These analyses were chosen because three of them (safety, data-flow facts, and functional program properties) represent different types as identified in a comprehensive survey [73] and the remaining two (performance and security) are more recent. An analytical assessment, by instantiating the mechanized theory (cf. Section 4.2) for concrete analyses, is yet to be performed. Such task is outside the scope of this work and is regarded as future work.

5.1.1 Safety Analysis. Apel et al. [3] have developed the tool chain SPLverifier for empirically comparing family-based, product-based, and sample-based model checking of domain-specific safety properties in product lines written in C and Java. To manage variability, the authors leverage a compositional representation, using feature modules and derivation via superimposition as supported by the tool FeatureHouse [2]. For product-based analysis, all products corresponding to valid configurations are derived via superimposition and then model-checked against a number of safety properties via explicit-state model checking. The sample-based analysis is an optimized version of product-based analysis, whereby different strategies are used: 1-wise, 2-wise, and 3-wise, each of which defines a corresponding sampling function for selecting subsets of products to be model-checked, after which the analysis is product-based.

In contrast, for family-based analysis, first the composition mechanism of FeatureHouse [2] is adjusted to perform variability encoding [80]: essentially, the variability induced by different combinations of features is encoded in the form of conditional program executions using if statements so that the product line is transformed into a product simulator, which simulates the behavior of all products, depending on the values of feature variables that represent the presence or absence of individual features. Then, off-the-shelf model checkers are used to perform verification of safety properties by means of explicit-state and symbolic model checking. The model checker exploits sharing between products using two principles: late splitting and early joining. The former refers to performing analysis without variability until encountering it, that is, the model checker explores execution paths of different products only once as long as such paths are equal; the latter refers to attempting to join intermediate results as early as possible, so analysis branches differing only on the value of feature variables should be analyzed only once. The result of model checking is a set of products that violate the given property, which are encoded in BDDs for efficient representation. Figure 12 depicts the safety analysis by Apel et al. [3] according to our framework.

5.1.2 Performance Analysis. Kowal et al. [47] present an approach for variability-aware analysis of software performance models. The underlying variability-free model is a Performance Annotated Activity Diagram (PAAD), which is an UML activity diagram with performance annotations. In a PAAD, nodes represent a service center with given multiplicity, initial client distribution, and service time distribution; edges are annotated with probabilities to model operational profiles. The property of interest is throughput, which is defined as the difference between the number of incoming and outcoming jobs out of all service stations of the model. The non-variability-aware analysis of a PAAD abstracts this model as a Markov population process and approximates its behavior by a compact system of ordinary differential equations (traffic equations), whose solution is the throughput of the model.

Variability management is accomplished via delta-modeling such that a core PAAD is supplied along with a set of deltas adding, removing, or modifying vertices and edges (compositional representation). A variant can be obtained by applying such deltas and then the previous variability-free analysis can be applied. Alternatively, variability encoding of deltas transforms the compositional model into a 150% model (annotative representation), which subsumes every variant of the family, consisting of all nodes and transitions that are added or modified by some delta. This model undergoes variability-aware analysis, whereby a parametric[2] system of ordinary differential equations is solved, giving rise to an algebraic expression. Such expression is then evaluated for each possible configuration, yielding a family-product-based analysis, as shown in Figure 13. Like Classen et al. for model checking, Kowal et al. have not explored compositional (feature-based) reasoning.

[2]All elements that are added to, removed from, or modified in the 150% model are represented by parameters.

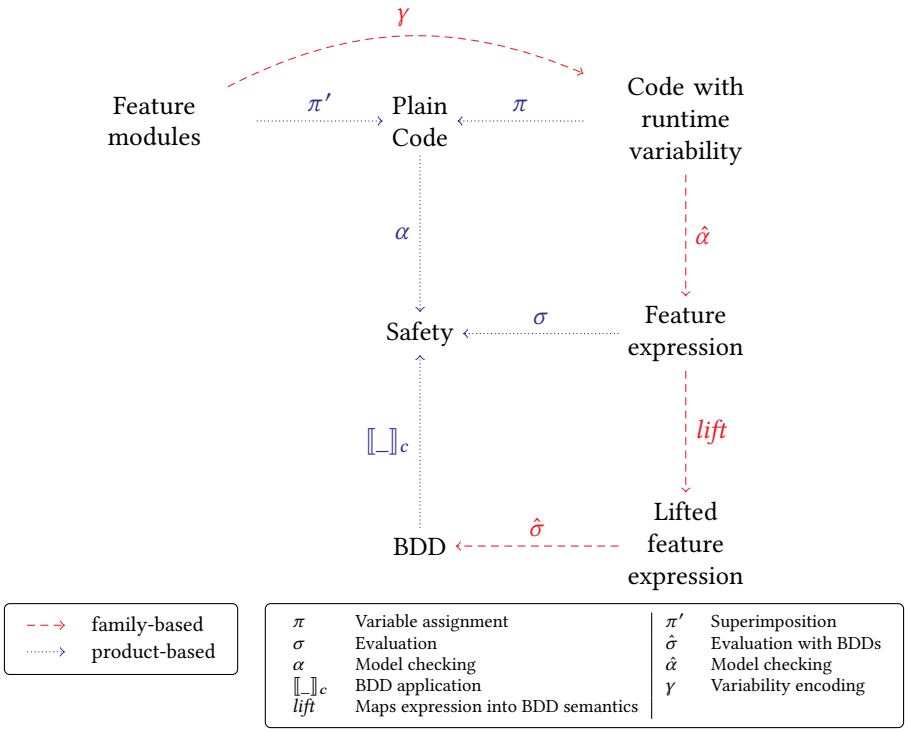


Fig. 12. Framework description of the safety analysis by Apel et al. [3] (Section 5.1.1)

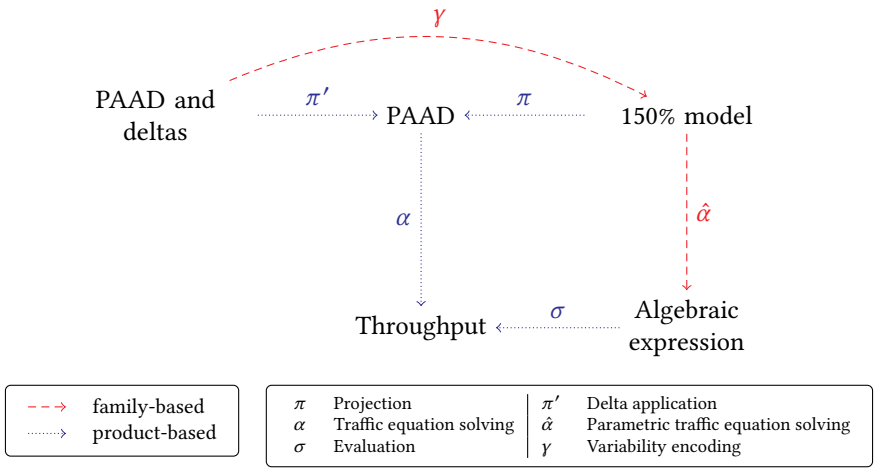


Fig. 13. Framework description of the performance analysis by Kowal et al. [47] (Section 5.1.2)

5.1.3 Security Analysis. Peldszus et al. [60] propose SecPL, a method for managing security requirements systematically in a product line, promoting model-based security analysis. SecPL extends UML's security profile UMLsec supporting the specification of security requirements and annotative variability in UML models with presence conditions. Users leverage UMLsec stereotypes for

encoding security specifications as OCL constraints. Such specifications can be done manually or automatically mined from annotated source code. The authors rely on *template interpretation* [28], effectively pursuing a family-based approach, this way lifting checks such as secure dependencies from the level of individual products to the entire product line. Accordingly, OCL constraints comprising the security property specification are analyzed on the annotative UML model, resulting in a feature expression, which is checked for satisfiability. If the formula is satisfiable, the feature set generates an unsafe product. Otherwise, one has a proof that each product satisfies the specified security properties. Figure 14 depicts the safety analysis by Peldszus et al. [60] according to our framework.

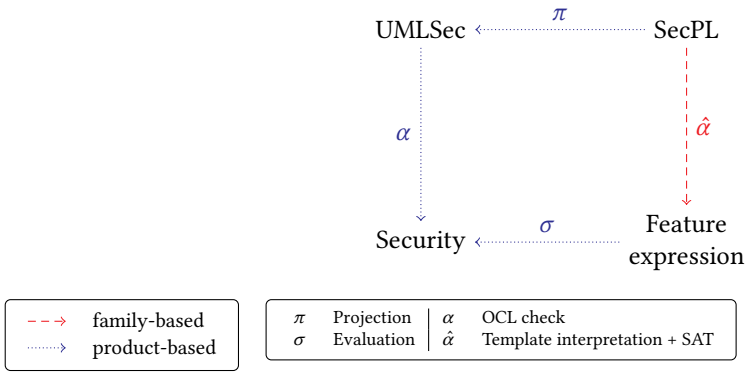


Fig. 14. Framework description of the security analysis by Peldszus et al. [60] (Section 5.1.3)

5.1.4 Data-flow Analysis. The generalization of the structure of our diagram is not limited to the verification of probabilistic and non-probabilistic behavioral properties. Consider, for example, the variability-aware static analysis method proposed by Bodden et al. [9]. Their method, called SPL^{LIFT}, seamlessly lifts inter-procedural data-flow analysis to product lines. To this end, they annotate an inter-procedural control-flow graph with a feature or its negation, to represent the cases where the feature is enabled or not. The variability-aware analysis then maps features expressions describing the configuration space of the annotated control-flow graph to corresponding data-flow facts. This representation is similar to the annotative FTS models used by Classen et al. [20]. The generalization of the structure of the diagram is therefore similar: from the annotated inter-procedural control-flow graph, one can perform a family-based analysis and find the set of products leading to a violation. This set can be compactly encoded as a constraint over features, which is equivalent to a feature expression. Figure 15 depicts the data-flow analysis by Bodden et al. [9] according to our framework. Like Classen et al. [20], Bodden et al. have not explored compositional (feature-based) reasoning.

5.1.5 Functional Program Properties Analysis. Hähnle and Schaefer [40] present a feature-family-based analysis strategy for product lines to analyze functional program properties expressed as contracts and invariants. Programs are expressed as a core and a number of delta modules that add, remove, or modify methods, fields, and contracts. To guarantee uniqueness of variant derivation for a given set of deltas, it is assumed that a partial order exists between the deltas. Program derivation proceeds by composing the core with a sequence of deltas.

The analysis proposed by Hähnle and Schaefer relies on an extended Liskov principle for delta modules, so the method contracts of subsequent deltas must be more specific; invariants cannot be removed; methods called in deltas use the contract of the first implementation of that method.

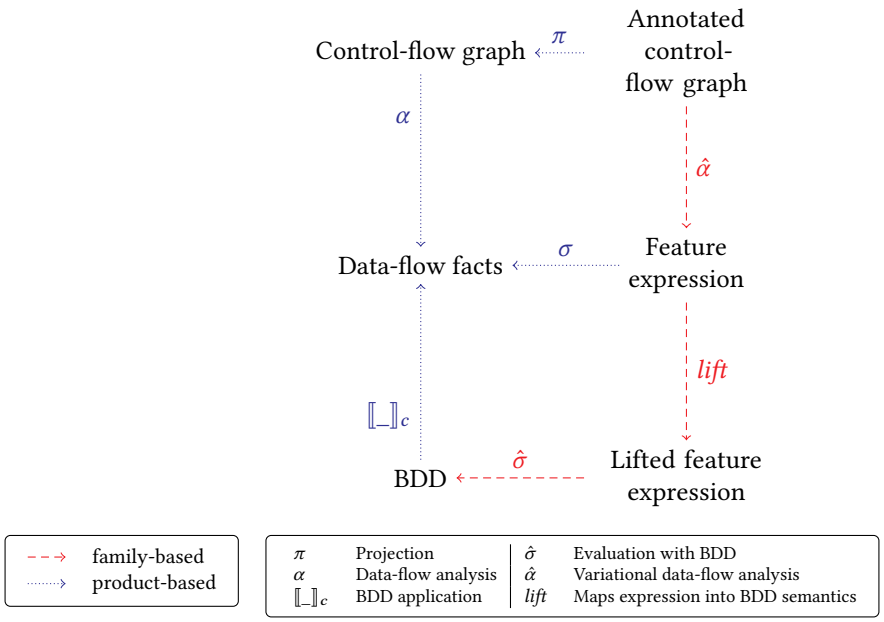


Fig. 15. Framework description of the data-flow analysis by Bodden et al. [9] (Section 5.1.4)

Leveraging these constraints, the core and each delta are analyzed in isolation with respect to method preconditions and postconditions (feature-based phase). After that, the global program invariants in the core and deltas are combined and checked (family-based phase), considering that method contracts are the ones computed in the previous step.

The resulting process can be characterized as a feature-family-based verification approach for delta-oriented product lines [73]. However, the analysis technique proposed by Hähnle and Schaefer [40] results in a yes/no answer to the question of whether all possible applications of deltas yield products that satisfy their corresponding specifications. This means that intermediate results can neither be represented as expressions nor lifted to ADD semantics. Hence, that approach cannot be described by our proposed framework.

5.2 Abstractions

In addition to generality, our framework rests on a number of central abstractions. Identifying these abstractions contributes to improving a principled understanding of product-line analyses:

- ***Inwards*—Variability binding**: From either side of Figure 9 to its center, there is variability restriction, which binds variability according to a given configuration. Variability restriction is performed at different types (models and expressions) and granularity levels (annotative and compositional models and expressions). For example, π' binds variability in compositional models, whereas π binds variability in annotative models.
- ***Left side*—Component functor**: In the leftmost models in Figure 9, there is a functor (a structure that can be mapped over) capturing the structure of the compositional model across compositional expressions and lifted compositional expressions (cf. dotted arrows in Figure 16). Function *fmap* maps the variability-aware analysis function $\hat{\alpha}$ over this structure of the compositional model (cf. edge labeled $fmap(\hat{\alpha})$ in Figure 9 and first large horizontal arrow in Figure 16), yielding a corresponding compositional expression. A further call to

fmap maps function *lift* over this expression, *fmap*(*lift*), resulting in a lifted compositional expression.

- ***Left side*—Folding functor with partial composition**: Additionally, from the left-hand side to the center, variability binding is performed within a folding operation over an ordering of the component functor structure to obtain a product (π') or a property value (σ and $\hat{\sigma}$). Folding implies the existence of both *partialModelComposition* and *partialExpComposition*, which bind the composition mechanism for *AnnotativeModel* and *AnnotativeExpression*, respectively.
- ***Lower quadrants*—Lifting to ADDs**: Both lower quadrants of the diagram illustrate a general principle for lifting analyses to product lines using ADDs: the intermediate analysis results represented by either compositional or annotative expressions are encoded using ADD operations for concise representation and avoiding enumeration during algebraic manipulation and evaluation.

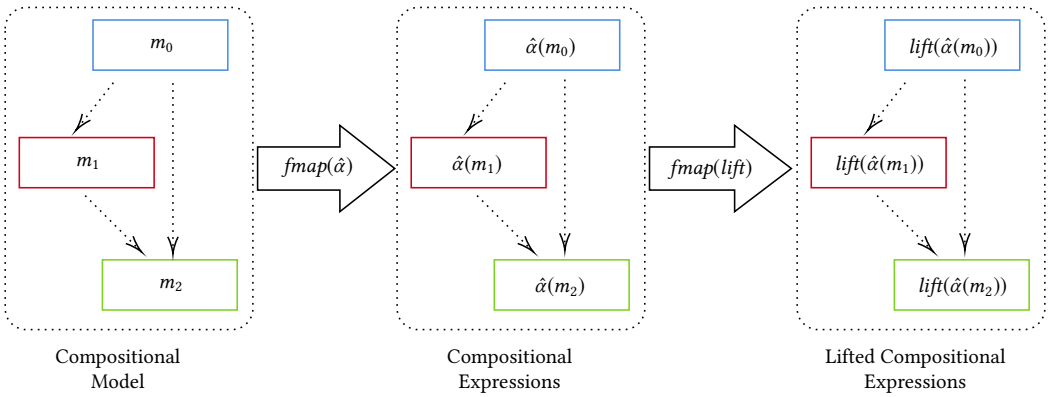


Fig. 16. Component Functor, allowing structure-preserving transformations

Regarding generality (cf. Section 5.1), we note that product-line analyses for both functional and non-functional properties can be described. In general, depending on the property and model, the total number of possible analysis strategies may change. Although for reliability Castro et al. [13] formalize seven analyses, this does not necessarily hold for all other properties, since some of these may not be amenable to compositional reasoning (e.g., performance). In principle, new analyses can also be conceived based on the original diagram. For instance, the performance analysis by Kowal et al. [47] results in algebraic expressions, which, in principle, could be subject to lifting and ADD-based evaluation. New analyses could also arise for other models and properties. As another example, the behavioral analyses described in Figure 8 could be extended along the compositional dimension of the framework.

At a finer grain, the extent to which our framework supports reuse depends on the property and on the model at hand. For instance, for reachability properties, the same model (PMC) for reliability can be reused. By analyzing the source code of the ReAna tool [49], which performs feature-family-based reliability analysis of product lines, we note that it is possible to achieve high level of reuse of both conceptual and implementation aspects. Similar considerations apply for other probabilistic properties. However, in other cases, the overall structure of the analyses could still be reused. For instance, in the product-based case, there is enumeration on products. In family-based strategies, the common part of the model is explored until variation points branch out the analysis. In feature-based analysis, each model fragment related to a feature is analyzed. This

overall control can be abstracted and shared across different properties and models thus helping the development of new analysis strategies.

Accordingly, the formalization carried out in Section 4.2 supports reuse by focusing on key abstractions and properties in this domain. In terms of abstraction, reuse follows from types and functions defined directly or indirectly in terms of the framework's core concepts (i.e, the property, the model, and the corresponding non-variability-aware analysis). Regarding properties, our framework also supports reuse of commutative properties, which are essential for the soundness of product-line analysis strategies, but reuse is usually non-trivial involving scarce effort across independently developed analyses. As discussed in Section 4.2, proofs of commutative properties of our framework give rise to a high-level structure that is completely reused under different interpretations for different models and related properties. Proof reuse is also possible within the framework: The proof of commutativity of Figure 9's upper-left quadrant reuses the proof of its upper-right quadrant's commutativity. Essentially, this means that the soundness of compositional analyses, which are coarse-grained, rely on the soundness of standard analyses, which are finer-grained and non-necessarily compositional. Nevertheless, the commutativity proof of Figure 9's upper-left quadrant has the liability of an underlying proof obligation corresponding to Axiom 1 in Section 4.2, requiring basic compositional behavior of the model and its partial composition, which is model dependent. This way, by explicitly stating the abstract requirements that a model and related transformations must satisfy, the framework supports reuse.

Finally, although the commutative diagram of Figure 9 shows logically equivalent analyses for a given model and property, the diagram does not convey practical considerations in terms of efficiency. For instance, for the feature-product-based analysis to be efficient, it is necessary that recursive composition of expressions is less complex than building the whole product [36]. Otherwise, the product-based analysis is preferable. In general, such considerations may also depend on modeling pragmatics and could be used to define modeling styles and corresponding bad smells. Furthermore, for a given property and model, the alternative strategies have differing complexity costs, which could annotate the diagram, yielding another dimension. A still open question is whether these costs can be reused across properties or models [46, 49, 51, 79].

5.3 Threats to Validity

Our formal framework was inductively built starting from different product-line analysis strategies (cf. Section 3) and is based on the taxonomy proposed by Thüm et al. [73] and on the product-line representation by Kästner et al. [44], which have been used to describe and analyze a multitude of product lines [73]. Nonetheless, the analysis strategies that we considered in Section 3 operate over transition-system models (either DTMC [13, 49] or FTS [19, 20]), which threatens external validity. To mitigate this threat, we qualitatively assessed the framework under additional strategies, operating over transition systems (CTMC [47]), source code [3, 9], UML models [60], and formal artifacts [40]. We argue that such variety of analyzed models is a representative sample from the universe of product-line analysis techniques as surveyed by Thüm et al. [73]. Moreover, a recent survey on the state-of-practice of variability-aware static analyses in different application domains [55] found that reliability, correctness, and performance are critical properties of interest.

Furthermore, considering the generality assessment in Section 5.1, we see that the lower quadrants of our commuting diagram are somewhat misrepresented in the qualitative assessment. This raises the question of whether lifting to ADDs can actually be generalized to other analyses. Indeed, the proposition of ADD lifting as a technique to cope with product-line analysis is recent [49], so there are still no independent empirical studies that employ it. Still, we have analytical evidence that such technique can be employed to analyze any propoerty that can be expressed as an algebraic expression [13].

Besides human scrutiny, we provide evidence on the soundness of our commutativity framework by means of machine-based verification. To address the validity of the mapping between framework concepts and formal definitions, we created our mechanized theory by modeling the constructs that exist in the formal theory of commuting product-line reliability analysis strategies [13]. However, it is future work to assess the extent to which one can devise mechanized theories of concrete analysis strategies by instantiating our generic PVS theory.

6 RELATED WORK

Conceptual models and taxonomy: Thüm et al. [73] established the taxonomy for product-line analyses upon which we based our work, that is, the classification of analysis techniques in three basic strategies (product-based, feature-based, and family-based) and combinations thereof. Furthermore, Meinicke et al. [52] surveyed existing product-line analysis tools and categorized them along four criteria: product-line implementation technique (annotation-based *versus* composition-based approach), analysis technique (e.g., testing, type checking, model checking), strategies for product-line analysis (i.e., the analysis strategies taxonomy by Thüm et al. [73]), and strategy of the tool (product-based, variability-aware, and variability-encoding). In this work, we build upon these existing taxonomies to propose a framework relating analysis *steps* in all dimensions. Although the surveys by Thüm et al. [73] and Meinicke et al. [52] range over a larger number of primary studies, our work establishes finer-grained relationships between analysis steps, supported by formal reasoning. In principle, our framework could be applied to describe the studies surveyed by Thüm et al. [73] and Meinicke et al. [52]—for instance, the ones that are part of the qualitative analysis in Sections 5.1.1 and 5.1.4.

Von Rhein et al. [78] proposed the PLA model, a simple formal model for describing and comparing product-line analyses. That model consists of four operators that express possible manipulations of product line artifacts during analysis. Our commuting diagram (Figure 9) relates to the PLA model as follows: all downward arrows are instances of the *processing step* of von Rhein et al. [78], all straight arrows from either left or right to the center are instances of *variability restriction*, and all arcs from left to right are instances of *variability combinator*. Moreover, whereas von Rhein et al. [78] provide static building blocks for describing product line analyses, we present structurally related analysis steps and formalize conditions for their applicability.

Thüm et al. [75] discussed the analysis of product lines throughout their life cycle. The authors reviewed product-line analysis techniques that can be applied to regression analysis and, conversely, revision analyses that can be leveraged in a space–variant setting. Thüm et al. [75] conjecture that analyses of product-line variations in time (i.e., evolution) could be modeled as a fourth dimension in von Rhein's PLA model [78], combining both types of techniques. Within the scope of such modeling effort, we envision that our work can be extended to cover the time dimension as well, leveraging the finer-grained (yet generic) analysis steps.

Formal approaches to variability-aware analysis: The definition of product-line analysis techniques that are sound by construction has been investigated in different contexts [11, 15, 54]. Midtgaard et al. [54] presented a methodology to systematically derive family-based static analyses from single-product analyses based on *abstract interpretation*; Chen and Erwig [15] defined a framework for automatic lifting of static analyses that are expressible as type systems; and Brabrand et al. [11] proposed a technique to automatically lift *intraprocedural data-flow analyses* to handle variability in product lines. In contrast to their work, we provide a basis for structuring proofs of correctness without constraining them to a specific analysis technique formalism. In exchange, we expect users of our framework to perform more formalization activities to bridge their concrete setting to our

abstract one. Moreover, whereas Midtgaard et al. [54], Chen and Erwig [15], and Brabrand et al. [11] handle only the family-based dimension of analysis, we also address the feature-based dimension.

With their seminal work on FTS, Classen et al. [19, 20] laid the foundations for designing product-line model checking strategies. Classen et al. [20] were concerned with annotative strategies (i.e., the right-hand side of our framework diagram). The principles of their strategies were later reused and extended to solve automata-based verification problems (e.g. real-time model checking [26], 2-player games [37]). Classen et al. [19] proved the equivalence between compositional and annotative FTS (i.e., the existence of the encoding function γ for these models). Our framework makes apparent that purely compositional strategies (i.e., the left-hand side of our diagram) have not been investigated for FTS.

Earlier work on product-line model checking [33] represents the behavior of multiple products as modal automata (i.e., automata with optional and mandatory transitions). While such modeling allows one to check that given properties hold for the whole product line, they cannot trace back the features responsible for property violations. Framing this verification problem within our framework would indeed highlight the impossibility to construct annotative (lifted) expressions. This limitation was later circumvented by extending modal automata with variability constraints [4, 72]. This makes modal automata as expressive as FTS [70, 71] and paves the way for exploiting the benefits of both formalisms [77].

Dimovski et al. [31] proposed a formal approach of applying variability-aware analyses even in the case where they are not immediately feasible. Given a variability-aware analysis, this technique searches for a suitable abstraction that allows a pre-analysis to be performed. Such pre-analysis, in turn, is used to find the features that have the same effect on the property under evaluation (and thus can be grouped) and those that are irrelevant to the problem at hand (and can be ignored). The work by Dimovski et al. [31] addresses the optimization of already lifted analyses, whereas we focus on aspects of the variability-aware analysis itself. Hence, both approaches are complementary. Moreover, similar to this work, Dimovski et al. [31] propose that Binary Decision Diagrams be used to increase sharing of analysis results.

Castro et al. [13] formalized a number of strategies for user-oriented reliability analysis of product lines, covering all possible combinations in the taxonomy by Thüm et al. [73]. They provide mathematical specifications and soundness proofs (not machine checked) of these strategies, along with a commuting diagram relating intermediate steps. In contrast, our work does not formalize concrete analysis techniques, but provides a machine-verified theory regarding generic concepts involved in product-line analyses. Nonetheless, we used the commuting diagram by Castro et al. [13] as a starting point to elicit candidate concepts for reuse (cf. Section 3.1).

Schobbens et al. [67] presented a formalization of feature diagram semantics. They defined feature diagrams in a precise manner, thereby establishing a formal relationship between existing notations. Likewise, our framework aims at formally defining concepts that are otherwise expressed in natural language. However, whereas Schobbens et al. [67] deal with formalization and analysis of *variability management* artifacts, our work addresses the analysis of properties of *derivable products*.

Von Rhein et al.[79] handled practical aspects of static analysis by implementing variability-aware control-flow and data-flow analyses for large-scale and highly configurable systems, based on the TypeChef tooling infrastructure. Their evaluation demonstrates the applicability of variability-aware analysis to real-world systems, with a performance comparable to that of sampling techniques. To achieve such results, the authors focus on the family-based dimension and employ sharing techniques to leverage existing analyses in the variability context. Similar to our work, von Rhein et al. [79] present formal definitions of core concepts necessary for their technique. However, the

authors do not provide proofs of correctness. Given the relevance of analyzing industrial systems, we regard such an effort as an important step towards ensuring that the results can be trusted.

Last, we note that all aforementioned approaches of variability-aware analysis devise or evaluate concrete techniques for computing specific properties. In contrast, our work abstracts from such details in pursuit of a general framework that can be reused in different scenarios, with different techniques and strategies. Moreover, the definitions and theorems provided in previous work are often manually crafted, whereas the concepts presented in this work are specified and proved using a proof assistant, which further increases confidence in the results.

Mechanized specification of product lines: Researchers have leveraged theorem provers and proof assistants in the context of software product lines (e.g., Borba et al. [10], Delaware et al. [30], Neves et al. [56], Sampaio et al. [66], Teixeira et al. [69], Thüm et al. [74]). However, most of the existing work investigates the reuse of specification and proofs to assert soundness of different products in a given product line (product lines of theorems). Our work, in contrast, deals with properties of product-line analyses.

Borba et al. [10] devised a PVS theory about properties of product lines—in their case, for reasoning about safe product-line evolution. This work evolved into a product line of theories [69], where products are theories of safe evolution based on concrete product-line languages. Similar to our results, their work rests on PVS theories about properties of product lines. Sampaio et al. [66] extended the refinement theory to contemplate changes that do not preserve the behavior of the entire set of products in a product line, thus establishing the notion of partially safe evolution of product lines. All of their properties and theories are also specified and proved using PVS. Nonetheless, Neves et al. [56], Sampaio et al. [66] and Teixeira et al. [69] specified concepts in the domain of product-line engineering, whereby the targets of their theories are meta-models of product lines. Our work focuses on properties of product-line analysis strategies, instead.

Durán et al. [32] proposed the FaMa formaL frAMEwork (FLAME), which comprises a formalization of analysis operation over variability models, together with a reference implementation in Prolog. The semantics of different analysis operations were specified using Z, and defined over a common abstract layer to different variability model notations. Our goal is similar in the sense that we aim at abstracting analysis operations and representations. However, we go beyond *variability model* analysis, and we use a mechanized theorem prover to specify our theories.

Product line of theories: To leverage the machine-verified theory presented in this work, one needs to instantiate uninterpreted elements to obtain a concrete setting (property, product, and analysis technique). This way, our generalized theory of product-line analysis becomes itself a product line of mechanized theories.

Teixeira et al. [69] addressed a similar problem. They created a generic theory to reason about product-line evolution, based on a refinement notion that is independent of the concrete languages used to manage variability in a product line. Then, Teixeira et al. [69] employed product-line engineering techniques and leveraged the theory interpretation mechanism in PVS to systematically reuse soundness proofs across concrete scenarios. Similar to our theory, their work provides a generic "backbone" and requires instantiations to be manually developed. However, at this point, we do not provide a means to systematically manage the reuse of specifications and proofs from the generic theory. We plan to do so by employing similar techniques to the ones used by Teixeira et al. [69].

Another approach to manage a family of theories has been proposed by Delaware et al. [30]. In that work, the authors presented a means to manage features of programming languages along with corresponding theorems for reasoning about them. The specification and proofs for each feature are contained in a Coq module, and the modules containing selected features are manually

imported and used. This approach can be classified as a bottom-up composition, whereas our work establishes a generic theory to be instantiated in a top-down fashion.

7 CONCLUSION

To address the lack of a precise and uniform description of individual product-line analyses and their properties, we propose a framework for product-line analysis consisting of a machine-verified theory comprising formal specification and verification of key concepts and key properties of product-line analyses captured in suitable abstraction. In particular, the framework defines abstract functions and types modeling essential concepts in this problem domain, such as analysis steps, models, and intermediate analysis results. Product-line analyses are modeled in a compositional manner, providing an overall understanding of the structure of the space of family-based, feature-based, and product-based analyses, defining precisely how the different types of product-line analyses compose and inter-relate. Additionally, we provide mechanized proofs of commutativity of different analysis strategies. The novelty of this work lies in how we model and map the different strategies and how we prove certain properties.

To create the analysis framework, we reviewed a number of representative existing analyses for the different models and properties, and we identified essential abstractions of such analyses and their structure. Figure 9 depicts the patterns that were found, relating annotative and compositional models as well as the operations defined over them. This view facilitates the organization and structuring of facts (e.g., commutativity of intermediate analysis steps) in a concise and precise manner, facilitating the communication of ideas and contributing to a more comprehensive understanding of underlying principles used in current and future product-line analysis strategies. Indeed, we were able to fit different types of analyses into our framework, in terms of the employed techniques as well as the models and properties under analysis.

The commuting diagram in Figure 9 and the corresponding mechanized commutativity theory are meant to be leveraged as guidelines for the formalization of existing analysis strategies and for the design of new ones. Hence, our framework contributes to the ongoing search for a principled and possibly automated way to lift a given specification and analysis technique to product lines [73].

In future work, we aim at formalizing the lower quadrants of the commuting diagram (Figure 9) along the guidelines discussed in Section 4.2. We also plan to extend the qualitative assessment conducted in Section 5.1 to encompass more analyses as well as to formally instantiate existing and new analyses. An additional goal is to model product-line evolution within our analysis framework, so that one is able to reuse analysis effort throughout the lifetime of products. Finally, we intend to provide a reference implementation of the framework and tool support for its derivation process.

ACKNOWLEDGEMENTS

We would like to thank the following people for fruitful discussions and suggestions on how to improve this work: Tobias Sena, Danilo Caldas, Andreas Stahlbauer, Christoph Seidl, Malte Lochau, Matthias Kowal, Ina Schaefer, Thomas Thüm, Pierre-Yves Schobbens, and the anonymous reviewers. Vander Alves was partially supported by CNPq (grant 310757/2018-5), FAPDF (grant SEI 00193-00000926/2019-67), and the Alexander von Humboldt Foundation. Leopoldo Teixeira was partially supported by CNPq (grant 409335/2016-9) and FACEPE (APQ-0570-1.03/14), as well as INES 2.0,[3] FACEPE grants PRONEX APQ-0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0. Sven Apel was supported by the German Research Foundation (AP 206/6, AP 206/11). Maxime Cordy was supported by FNR Luxembourg (grant C19/IS/13566661/BEEHIVE/Cordy). Rohit

[3] http://www.ines.org.br

Gheyi was supported by CAPES (grants 117875 and 175956) and CNPq (grants 426005/2018-0 and 311442/2019-6).

REFERENCES

[1] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer. https://doi.org/10.1007/978-3-642-37521-7

[2] Sven Apel, Christian Kästner, and Christian Lengauer. 2013. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering* 39, 1 (2013), 63–79. https://doi.org/10.1109/TSE.2011.120

[3] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. 2013. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 482–491. https://doi.org/10.1109/ICSE.2013.6606594

[4] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. 2011. Formal Description of Variability in Product Families. In *Proceedings of the 15th International Conference on Software Product Lines (SPLC)*. Springer, 130–139.

[5] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1997. Algebraic Decision Diagrams and Their Applications. *Formal Methods in System Design* 10, 2/3 (1997), 171–206. https://doi.org/10.1023/A:1008699807402

[6] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series).* The MIT Press.

[7] Christel Baier and Marta Kwiatkowska. 1998. On the verification of qualitative properties of probabilistic processes under fairness constraints. *Inform. Process. Lett.* 66, 2 (1998), 71 – 79. https://doi.org/10.1016/S0020-0190(98)00038-6

[8] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615 – 636. https://doi.org/10.1016/j.is.2010.01.001

[9] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL^{LIFT}: 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)*. ACM, 355–364. https://doi.org/10.1145/2491956.2491976

[10] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. 2012. A theory of software product line refinement. *Theoretical Computer Science* 455 (Oct. 2012), 2–30. https://doi.org/10.1016/j.tcs.2012.01.031

[11] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. 2013. Intraprocedural Dataflow Analysis for Software Product Lines. In *Transactions on Aspect-Oriented Software Development*. Springer, 73–108. https://doi.org/10.1007/978-3-642-36964-3_3

[12] Randal E. Bryant. 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. *Comput. Surveys* 24, 3 (Sept. 1992), 293–318.

[13] Thiago Castro, André Lanna, Vander Alves, Leopoldo Teixeira, Sven Apel, and Pierre-Yves Schobbens. 2018. All roads lead to Rome: Commuting strategies for product-line reliability analysis. *Science of Computer Programming* 152 (2018), 116 – 160. https://doi.org/10.1016/j.scico.2017.10.013

[14] Marsha Chechik, Benet Devereux, and Arie Gurfinkel. 2001. Model-Checking Infinite State-Space Systems with Fine-Grained Abstractions Using SPIN. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN)*. Springer, 16–36.

[15] Sheng Chen and Martin Erwig. 2014. Type-based parametric analysis of program families. *ACM SIGPLAN Notices* 49, 9 (aug 2014), 39–51. https://doi.org/10.1145/2692915.2628155

[16] R. Cheung. 1980. A User-Oriented Software Reliability Model. *IEEE Transactions on Software Engineering* 6, 02 (March 1980), 118–125. https://doi.org/10.1109/TSE.1980.234477

[17] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. 2016. Family-Based Modeling and Analysis for Probabilistic Systems - Featuring ProFeat. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE) (Lecture Notes in Computer Science)*, Vol. 9633. Springer, 287–304. https://doi.org/10.1007/978-3-662-49665-7_17

[18] E. M. Clarke and E. A. Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs (LNCS)*, Vol. 131. Springer, 52–71.

[19] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2014. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming* 80 (2014), 416–439.

[20] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1069–1089. https://doi.org/10.1109/TSE.2012.86

[21] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*. ACM, 335–344.

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

[23] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2012. Managing Evolution in Software Product Lines : A Model-Checking Perspective. In *Proceedings of the 6th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS)*. ACM, 183–191.

[24] Maxime Cordy, Xavier Devroey, Axel Legay, Gilles Perrouin, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Jean-François Raskin. 2019. A Decade of Featured Transition Systems. In *From Software Engineering to Formal Methods and Tools, and Back - Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday (Lecture Notes in Computer Science)*, Vol. 11865. Springer, 285–312. https://doi.org/10.1007/978-3-030-30985-5_18

[25] Maxime Cordy, Patrick Heymans, Axel Legay, Pierre-Yves Schobbens, Bruno Dawagne, and Martin Leucker. 2014. Counterexample Guided Abstraction Refinement of Product-Line Behavioural Models. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 190–201. https://doi.org/10.1145/2635868.2635919

[26] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2013. Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 472–481.

[27] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., USA.

[28] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 211–220. https://doi.org/10.1145/1173706.1173738

[29] Benjamin Delaware, William R. Cook, and Don S. Batory. 2009. Fitting the pieces together: A machine-checked model of safe composition. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Hans van Vliet and Valérie Issarny (Eds.). ACM, 243–252. https://doi.org/10.1145/1595696.1595733

[30] Benjamin Delaware, William R. Cook, and Don S. Batory. 2011. Product lines of theorems. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. ACM, 595–608. https://doi.org/10.1145/2048066.2048113

[31] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. 2019. Finding suitable variability abstractions for lifted analysis. *Formal Aspects of Computing* 31, 2 (01 Apr 2019), 231–259. https://doi.org/10.1007/s00165-019-00479-y

[32] Amador Durán, David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2017. FLAME: A Formal Framework for the Automated Analysis of Software Product Lines Validated by Automated Specification Testing. *Software and Systems Modeling* 16, 4 (Oct. 2017), 1049–1082. https://doi.org/10.1007/s10270-015-0503-z

[33] Alessandro Fantechi and Stefania Gnesi. 2008. Formal Modeling for Product Families Engineering. In *Proceedings of the 12th International Conference on Software Product Lines (SPLC)*. IEEE, 193–202.

[34] Dario Fischbein, Sebastian Uchitel, and Victor Braberman. 2006. A Foundation for Behavioural Conformance in Software Product Line Architectures. In *Proceedings of the International Symposium on Software Testing and Analysis Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA '06)*. ACM, 39–48. https://doi.org/10.1145/1147249.1147254

[35] B. J. Garvin and M. B. Cohen. 2011. Feature Interaction Faults Revisited: An Exploratory Study. In *Proceedings of the IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 90–99. https://doi.org/10.1109/ISSRE.2011.25

[36] Carlo Ghezzi and Amir Molzam Sharifloo. 2013. Model-based verification of quantitative non-functional properties for software product lines. *Information and Software Technology* 55, 3 (March 2013), 508–524. https://doi.org/10.1016/j.infsof.2012.07.017

[37] Joel Greenyer, Christian Brenner, Maxime Cordy, Patrick Heymans, and Erika Gressi. 2013. Incrementally synthesizing controllers from scenario-based product line specifications. In *Proceedings of The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 433–443.

[38] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. 2011. Delta Modeling for Software Architectures. In *Proceedings of the 7th Tagungsband - Dagstuhl-Workshop: Modellbasierte Entwicklung eingebetteter Systeme (MBEES)*. 1 – 10.

[39] Ernst Moritz Hahn, Holger Hermanns, and Lijun Zhang. 2011. Probabilistic reachability for parametric Markov models. *International Journal on Software Tools for Technology Transfer (STTT)* 13, 1 (2011), 3–19. https://doi.org/10.1007/s10009-010-0146-x

[40] Reiner Hähnle and Ina Schaefer. 2012. A Liskov Principle for Delta-Oriented Programming. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA*

2012, Proceedings, Part I. Springer, 32–46. https://doi.org/10.1007/978-3-642-34026-0_4

[41] Gerard Holzmann. 2003. *Spin Model Checker, the: Primer and Reference Manual* (first ed.). Addison-Wesley Professional.

[42] Michael Huth, Radha Jagadeesan, and David Schmidt. 2001. Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In *Programming Languages and Systems*. Lecture Notes in Computer Science, Vol. 2028. Springer, 155–169.

[43] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University, Software Engineering Institute.

[44] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. ACM, 311–320.

[45] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2009. A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE '09)*. ACM, 157–166. https://doi.org/10.1145/1621607.1621632

[46] Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen, and Sven Apel. 2013. A comparison of product-based, feature-based, and family-based type checking. In *Proceedings of 12th International Conference on Generative Programming: Concepts and Experiences (GPCE'13)*. ACM, 115–124. https://doi.org/10.1145/2517208.2517213

[47] Matthias Kowal, Max Tschaikowski, Mirco Tribastone, and Ina Schaefer. 2015. Scaling Size and Parameter Spaces in Variability-Aware Software Performance Models. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 407–417. https://doi.org/10.1109/ASE.2015.16

[48] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science)*, Vol. 6806. Springer, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

[49] André Lanna, Thiago Castro, Vander Alves, Genaina Rodrigues, Pierre-Yves Schobbens, and Sven Apel. 2017. Feature-Family-Based Reliability Analysis of Software Product Lines. *Information and Software Technology* 94 (2017), 59–81. https://doi.org/10.1016/j.infsof.2017.10.001

[50] Michael Lienhardt, Ferruccio Damiani, Lorenzo Testa, and Gianluca Turin. 2018. On checking delta-oriented product lines of statecharts. *Science of Computer Programming* 166 (2018), 3–34. https://doi.org/10.1016/j.scico.2018.05.007

[51] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE) (ICSE '16)*. ACM, 643–654. https://doi.org/10.1145/2884781.2884793

[52] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. 2014. An overview on analysis tools for software product lines. In *Proceedings of the 18th International Software Product Line Conference (SPLC)*. ACM, 94–101. https://doi.org/10.1145/2647908.2655972

[53] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (Oct. 1992), 40–51. https://doi.org/10.1109/2.161279

[54] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. 2015. Systematic derivation of correct variability-aware program analyses. *Science of Computer Programming* 105 (July 2015), 145–170. https://doi.org/10.1016/j.scico.2015.04.005

[55] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 155–166. https://doi.org/10.1145/3238147.3238201

[56] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulezsa, and Paulo Borba. 2011. Investigating the safe evolution of software product lines. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering - GPCE '11*, Vol. 47. ACM, 33–42. https://doi.org/10.1145/2047862.2047869

[57] Sam Owre and Natarajan Shankar. 2001. *Theory Interpretations in PVS*. Technical Report. NASA Langley Research Center.

[58] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. 2001. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA.

[59] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wasowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of variability models and related software artifacts: A fresh look at evolution patterns in the Linux kernel. *Empirical Software Engineering* 21, 4 (05 2016), 1744–1793. https://doi.org/10.1007/s10664-015-9364-x

[60] Sven Peldszus, Daniel Strüber, and Jan Jürjens. 2018. Model-Based Security Analysis of Feature-Oriented Software Product Lines. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2018)*. ACM, 93–106. https://doi.org/10.1145/3278122.3278126

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

[62] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77)*. IEEE, 46–57. https://doi.org/10.1109/SFCS.1977.32

[63] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[64] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the 11th European Conference on Object-Oriented Programming (Lecture Notes in Computer Science)*, Vol. 1241. Springer, 419–443. https://doi.org/10.1007/BFb0053389

[65] Genaína Nunes Rodrigues, Vander Alves, Vinicius Nunes, André Lanna, Maxime Cordy, Pierre-Yves Schobbens, Amir Molzam Sharifloo, and Axel Legay. 2015. Modeling and Verification for Probabilistic Properties in Software Product Lines. In *Proceedings of the 16th IEEE International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 173–180. https://doi.org/10.1109/HASE.2015.34

[66] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2019. Partially safe evolution of software product lines. *Journal of Systems and Software* 155 (2019), 17 – 42. https://doi.org/10.1016/j.jss.2019.04.051

[67] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic Semantics of Feature Diagrams. *Computer Networks* 51, 2 (Feb. 2007), 456–479.

[68] Rok Strniša, Peter Sewell, and Matthew Parkinson. 2007. The Java Module System: Core Design and Semantic Definition. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA) (OOPSLA '07)*. ACM, 499–514. https://doi.org/10.1145/1297027.1297064

[69] Leopoldo Teixeira, Vander Alves, Paulo Borba, and Rohit Gheyi. 2015. A product line of theories for reasoning about safe evolution of product lines. In *Proceedings of the 19th International Conference on Software Product Line (SPLC)*. ACM, 161–170. https://doi.org/10.1145/2791060.2791105

[70] Maurice H. ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. 2015. From Featured Transition Systems to Modal Transition Systems with Variability Constraints. In *Proceedings of the 13th International Conference on Software Engineering and Formal Methods (SEFM)*. Springer, 344–359. https://doi.org/10.1007/978-3-319-22969-0

[71] Maurice H. ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. 2019. On the expressiveness of modal transition systems with variability constraints. *Science of Computer Programming* 169 (2019), 1–17. https://doi.org/10.1016/j.scico.2018.09.006

[72] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2016. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *Journal of Logical and Algebraic Methods in Programming* 85, 2 (2016), 287 – 315.

[73] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (June 2014), 1–45. https://doi.org/10.1145/2580950

[74] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. 2011. Proof Composition for Deductive Verification of Software Product Lines. In *2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 270–277. https://doi.org/10.1109/ICSTW.2011.48

[75] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC) (SPLC '19)*. ACM, 57–64. https://doi.org/10.1145/3307630.3342414

[76] J. van Gurp, J. Bosch, and M. Svahnberg. 2001. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*. IEEE, 45–54. https://doi.org/10.1109/WICSA.2001.948406

[77] Mahsa Varshosaz, Lars Luthmann, Paul Mohr, Malte Lochau, and Mohammad Reza Mousavi. 2019. Modal transition system encoding of featured transition systems. *Journal of Logical and Algebraic Methods in Programming* 106 (2019), 1–28. https://doi.org/10.1016/j.jlamp.2019.03.003

[78] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. 2013. The PLA model: on the combination of product-line analyses. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 1. https://doi.org/10.1145/2430502.2430522

[79] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 4, Article 18 (Nov. 2018), 33 pages. https://doi.org/10.1145/3280986

[80] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming* 85, 1 (jan 2016), 125–145. https://doi.org/10.1016/j.jlamp.2015.06.007

[81] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14*. ACM, 213–226. https://doi.org/10.1145/2661136.2661143

[82] Stephan Weißleder and Hartmut Lackner. 2013. Top-Down and Bottom-Up Approach for Model-Based Testing of Product Lines. In *Proceedings 8th Workshop on Model-Based Testing, (MBT)*, Alexander K. Petrenko and Holger Schlingloff (Eds.), Vol. 111. EPTCS, 82–94. https://doi.org/10.4204/EPTCS.111.7