A Framework of Software Product Line Analyses

Vander Alves[a,*], Thiago Castro[a,b,*], Leopoldo Teixeira[c], Sven Apel[d], Pierre-Yves Schobbens[e], Maxime Cordy[e]

[a] *Computer Science Department, University of Brasília. Campus Universitário Darcy Ribeiro - Edifício CIC/EST, 70910-900, Asa Norte, Brasília - DF, Brazil*
[b] *Systems Development Center, Brazilian Army. QG do Exército - Bloco G - 2º Andar, 70630-901, Setor Militar Urbano, Brasília - DF, Brazil*
[c] *Informatics Center, Federal University of Pernambuco. Av. Jornalista Aníbal Fernandes, s/n - Cidade Universitária (Campus Recife), 50740-560, Recife - PE, Brazil*
[d] *Department of Informatics and Mathematics, University of Passau. Innstr. 33, 94032 Passau, Germany*
[e] *Faculty of Computer Science, University of Namur. rue Grandgagnage 21, 5000 Namur, Belgium*

Key words: Software Product Lines, Product-line Analysis, Verification

1. Introduction

Software Product Line Engineering (SPLE) is a means to systematically manage variability and commonality in software systems, enabling automated synthesis of similar programs from a set of reusable assets [15, 35, 1]. This methodology allows improving productivity and time-to-market, as well as achieving mass customization of software [35].

However, for a product line with high variability, combinatorial explosion may cause the number of possibly generated products to be very large. Because of this phenomenon, it is often infeasible to quality-check each of these products in isolation. Nonetheless, software verification techniques for the single-product case are widely used by the industry, and it is beneficial to exploit their maturity in order to increase reliability while reducing costs and risks.

There are a number of approaches to *product-line analysis* that adapt those standard techniques, such as type checking and model checking, according to a recent survey [39]. Although they are different to some extent, they share features which suggest there is a yet unexplored potential for reuse. Different analyses based on similar techniques represent redundancy in both implementation and verification efforts. We can note unexplored similarities between a number of analysis methods proposed so far:

Similar models. Kowal et al. [29] propose a method that takes UML activity diagrams annotated with performance as input to evaluate performance of the overall product line. In contrast, Ghezzi and Molzam Sharifloo [21] use UML sequence diagrams annotated with reliability and energy consumption

*Corresponding author
Email addresses: `valves@unb.br` (Vander Alves), `thiago.mael@aluno.unb.br` (Thiago Castro), `lmt@cin.ufpe.br` (Leopoldo Teixeira), `apel@uni-passau.de` (Sven Apel), `pierre-yves.schobbens@unamur.be` (Pierre-Yves Schobbens), `maxime.cordy@unamur.be` (Maxime Cordy)

values. As both take UML behavioral models as input and verify stochastic properties, there is an opportunity for investigating reuse, even though they employ different variability handling techniques.

Similar variability handling. Hähnle and Schaefer [25] devised a theoretical framework for verification of design-by-contract properties in Java programs with variability handled by Delta-Oriented Programming (DOP). Thüm et al. [40] check the same properties for Java source code, but using Feature-Oriented Programming (FOP) as the composition mechanism. Although DOP is classified by Haber et al. [23] as a transformational mechanism, according to Schaefer et al. [38] it can be seen as an extension of FOP. Nevertheless, the proposed analysis techniques do not explicitly share formalization.

Similar strategies. Kowal et al. [29] employ a family-based strategy for building a single stochastic model which encodes all variability in the product line, and thus can be analyzed in a single run. Apel et al. [5] adopt the same strategy for generating source code of a program which 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 seems to be a pattern of encoding all variability in a single product—which Apel et al. [5] call a simulator. This pattern, in turn, must have an underlying principle governing its application and validity.

The separate devising of analysis methods for software product lines yields unrelated tools, even if they, for instance, take similar models as input or perform computations over the same data structures. This can lead to redundant implementations and even to repetition in correctness proofs. Moreover, should one have the need of applying such similar analyses implemented by different tools, a number of analysis tasks—that are inherently time consuming, error-prone, and require specialized knowledge to perform—would inevitably be repeated.

This repetition is a natural outcome of ongoing research activity, given that there are independent groups exploring similar issues. Nonetheless, practitioners and researchers alike benefit from having an integrated body of knowledge. Because of this, it is useful to periodically synthesize existing work in a given field periodically. Indeed, the survey by Thüm et al. [39] is an important step in this direction, which could be seen as a coarse-grained domain analysis. However, more refinement is necessary to further explore similarities and suitably manage variabilities among these approaches, thus providing practitioners and researchers with more efficient techniques and theoretical framework for further investigation.

To address these issues, we propose a framework of product-line analyses. The framework presents such analyses in a compositional manner at a conceptual level, providing an overall understanding of the structure of the space of family-based, feature-based, and product-based analyses, showing how the different types of product-line analyses compose and inter-relate. The framework allows the organization and structuring of facts (e.g., commutativity of intermediate analysis steps) and knowledge in a concise and precise manner, thus further facilitating the communication of ideas and knowledge. This contributes to a more comprehensive and deeper understanding of underlying principles used in these techniques, which we envision could help other researchers to lift existing single-product analysis techniques to yet under-explored variability-aware approaches. To provide evidence of the generality of the framework, we present a number of instances in terms of existing product-line analyses. Lastly, to guide the use of the framework, we define a product line of product line analyses techniques (PL^2Ana), which exposes the configurability of the framework and systematizes the derivation of its instances. In summary, the contributions of this paper are the following:

- We review existing types of analyses, revealing their overall structure (Section 3);
- We present a framework of product-line analyses that provides an overall understanding of the structure of the space of family-based, feature-based, and product-based analyses (Section 4);
- We show instances of the framework, providing evidence of its generality (Section 4.3);
- We define a product line of product-line analysis strategies, built on top of the framework to systematically guide its use (Section 5).

2. Background

2.1. Software Product Line Analysis Strategies

2.2. Transition Systems

+Many formalisms can represent the behavior of a system. Yet in the end they all come down to a set of states and transition between these states. This is why *Transition Systems* (TS) are commonly chosen as a unified representation. A TS consists of a set of states and transitions between these states annotated with an action (e.g., $s \xrightarrow{\alpha} s'$ denotes a transition from s to s' due to some action α). Also, each state is labelled with a set of so-called *atomic properties* that represent all the properties that hold when the system is in this state. Examples of atomic properties are *failure* and *sleep*, which represent that 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 is "the system cannot enter sleep mode until all failures are resolved". These properties are typically expressed in some temporal logic like Computation Tree Logic (CTL) [11] and Linear Temporal Logic (LTL) [34]. They are considered as behavior al properties, i.e., they consider the sequence (and in the case of CTL, also the alternance) of visible states. The properties expressible include safety, reachability, and repetitive reachability.

2.3. Discrete-time Markov Chains

2.4. Algebraic Decision Diagrams

3. Product-Line Analysis Strategies

To motivate the definition of the framework of product-line analysis strategies, we first review existing analyses for some models and properties in the following subsections. Section 3.1 addresses reliability, and Section 3.2, behavior.

3.1. Reliability

Reliability can be defined as a probabilistic existence property [22] in the sense that it is given by the probability of eventually reaching some set of *success* states in a probabilistic behavioral model of a system. We define success to mean that all tasks of interest have been accomplished as intended.

Discrete-time Markov Chains (DTMC) is a well-known formalism to model such probabilistic behavior. DTMCs can be represented as transition systems where transitions are annotated with a probability value that describes their occurrence likelihood. These probabilities are defined such that they satisfy the usual probability axioms. In particular, for a given state the probability of its outgoing transitions must sum to 1. For example, the bottom-left model in Figure 1 is a DTMC that satisfies this property. In a DTMC, the reachability probability is defined as the sum of probabilities for each possible path that starts in an initial state and ends in a state belonging to the set of target states [6]. Accordingly, the calculation of the reliability of the DTMC in the bottom-left of Figure 1 multiplies the probabilities along the single path to the success state (s_{suc}), a computation which is represented in the figure by α.

Although DTMCs are convenient to model probabilistic behavior, they cannot cope with variability in the sense of product lines. Parametric Markov Chains (PMC) extend DTMCs with the ability to represent *variable* transition probabilities. These variable transition probabilities can be leveraged to represent product-line variability [37, 21, 10]. For instance, the top-left model in Figure 1 is a PMC in which variability is represented in both transitions leaving state s_1. To compute reliability, one can label success states with the atom *"success"* and compute the reachability probability of success states, resulting in a rational expression [24]. For instance, the reliability of the PMC in the top-left part of Figure 1 is the expression in the top-right part of the figure, which has two operands, each of which corresponds to a path in the PMC leading to the success state (s_{suc}). Indeed, the result is an expression and not a value, as in the case of reliability calculation of the DTMC in the same figure, since variables in the expression encode variability in the PMC.

Therefore, for reliability analysis, one can choose a number of product-line analysis strategies [8], as illustrated in Figure 1. One alternative is to first derive 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 model checking, thereby resulting in a product-based analysis strategy. Alternatively, we could first apply parametric model checking ($\hat{\alpha}$) only once to the PMC, resulting in an expression, then evaluate it (σ) for each valid configuration thus performing a family-product analysis. The latter approach has been explored before [37], and its advantage over the former is that it explores the commonality of the PMC, performing analysis of its common parts only once. But the product-based approach can rely on the existence of well established model checking tools such as PRISM, whereas the latter requires the development of a specific variability-aware tool.

In general, considering previous work [8], there are further analysis choices. Figure 2 illustrates these choices. Starting with a compositional (upper left corner) or an annotative model (upper right corner), one can follow any of the outgoing arrows while performing the respective analysis steps (abstracted as functions), until reliabilities are computed (either real-valued reliabilities or an

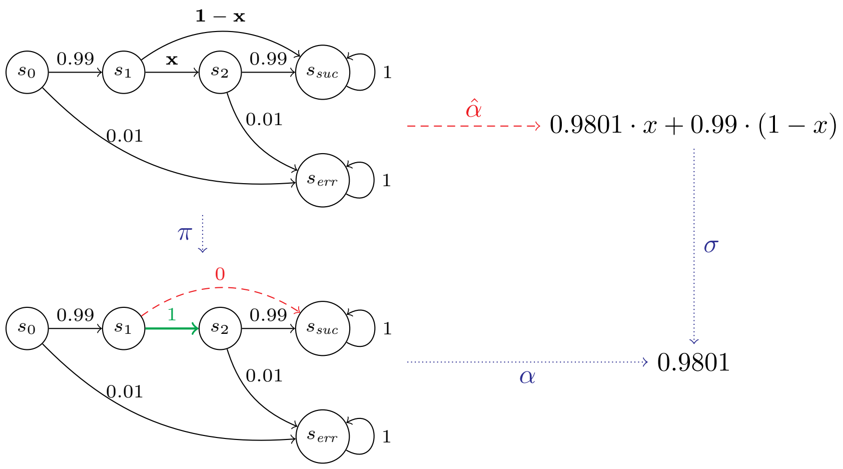


Figure 1: Example of family-product-based analysis ($\hat{\alpha}$ followed by σ) in contrast to a product-based analysis (π followed by α) of an annotative PMC, for a configuration satisfying x's presence condition

ADD representing all possible values). These analysis steps can be feature-based (green solid arrows), product-based (blue dotted arrows), or family-based (red dashed arrows). Thus, the arrows form an "analysis path" corresponding to a function composition, which defines the employed analysis strategy. Castro et al. [8] prove that Figure 2 is a commuting diagram, meaning that different analysis paths are equivalent (i.e., they yield equal results) if they share the start and end points.

After choosing a variability representation, the analysis of any of the resulting models presents another choice: either DTMC are derived for each configuration by *PMC projection* (π) or *PMC composition* (π') [8] and then analyzed (α), or variability-aware analysis is applied, using some form of parametric model checking ($\hat{\alpha}$). The first choice yields a product-based strategy, whereby each variant is independently analyzed. The second choice leverages parametric model checking to produce rational expressions denoting the reliability of PMCs in terms of expression variables. These variables carry the semantics they had in the model-checked PMC, so we correspondingly classify the resulting expressions as *annotative expressions* or *compositional expressions*. Both compositional probabilistic model and compositional rational expressions can be transformed into corresponding annotative versions by means of *variability encoding* (γ), which leverages an *if-then-else* operator for PMCs to switch between possible states with a Boolean variable [8].

Evaluating annotative and compositional expressions provides another choice: to evaluate the expressions for each valid configuration (σ), yielding family-product-based and feature-product-based strategies, respectively; or to interpret the expressions in terms of ADDs to obtain *annotative lifted expressions* or *compositional lifted expressions* (represented by function *lift*—a step we call *expression lifting*), effectively evaluating them for the whole family of models at once ($\hat{\sigma}$). The latter represents family-based and feature-family-based strategies, respectively.

As an example of walking through the choices of Figure 2, suppose we start with a compositional model (upper-left corner), perform parametric model checking (move down), and then lift the resulting expressions (move down one more step) and evaluate them (move right), reaching a reliability ADD for the family as a whole. The arrows in this path are, respectively, green solid, red dashed, and red dashed, meaning the analysis strategy is feature-family-based.

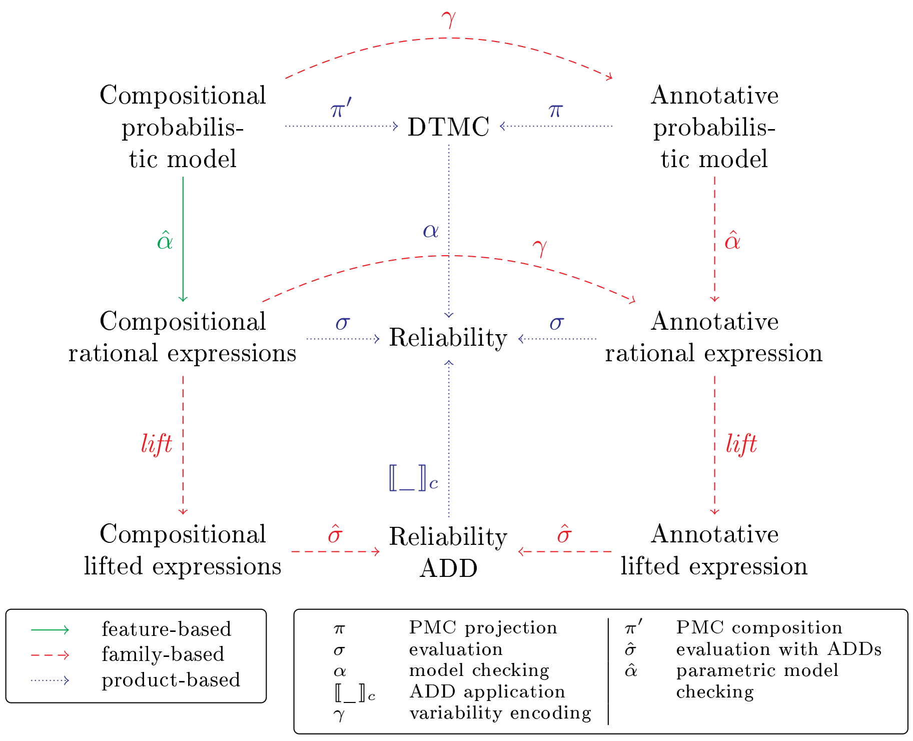

γ
Compositional probabilistic model
π'
DTMC
π
Annotative probabilistic model
$\hat{\alpha}$
α
γ
$\hat{\alpha}$
Compositional rational expressions
σ
Reliability
σ
Annotative rational expression
lift
$[\![\_]\!]_c$
lift
Compositional lifted expressions
$\hat{\sigma}$
Reliability ADD
$\hat{\sigma}$
Annotative lifted expression
feature-based
family-based
product-based
π PMC projection
σ evaluation
α model checking
$[\![\_]\!]_c$ ADD application
γ variability encoding
π' PMC composition
$\hat{\sigma}$ evaluation with ADDs
$\hat{\alpha}$ parametric model checking

Figure 2: Commutative diagram of product-line reliability analysis strategies [8]

3.2. Behavior

While reliability, addressed in Section 3.1, is concerned with ensuring that something eventually happens in a system with a given probability, what we call "behavior verification" focuses on properties that the system satisfies with certainty. That is, by behavior we mean all the behaviors that the system exhibits in any event.

+

Checking the behavior of an SPL, as opposed to a single system, can be seen as the problem of verifying a set of TS, one for each product. A product-based verification strategy would thus check each of these TS individually. However, in a product line, products share common behavior. One could thus reduce the verification time for the whole SPL by checking *only once* behavior that is common to multiple products. Leaning on this assumption, previous research aimed at defining concise formalisms to encode variable SPL behavior and designing efficient verification algorithms that factorize the verification effort. Among these, one finds modal transition systems [28, 19], multi-valued model checking [9] and featured transition systems [13]. A more comprehensive survey of

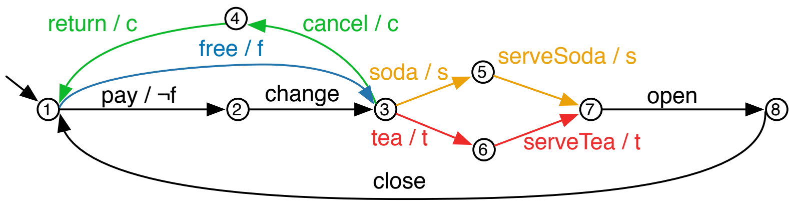


Figure 3: The FTS modelling the vending machine SPL.

these approaches can be read elsewhere [39]. In what follows, we analyze the work surrounding Featured Transition Systems (FTS) and argue that the structure of the commutative diagram (see Figure 2) also applies to this modeling formalism. This is a first indication that the diagram generalizes to other SPL analyses, as further discussed in Section 4.3.

As an illustrative example, Figure 3 depicts an FTS modelling an SPL of vending machines. In addition to an action, a transition in FTS is also labelled with an expression (named *feature expression*), which defines the set of products able to execute this transition. For instance, the transition from state 3 to state 6 is labelled with the feature expression t, meaning that it can be executed only by products including the corresponding SPL feature t. Transition from state 1 to state 2 is labelled 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 the SPL products. The TS representing the behavior of a particular product is obtained by removing all transitions not available to this product. This operation is named *projection* [13].

FTS is thus a low-level formalism that falls into the category of *annotative* models. Higher-level formalisms are also considered, namely fPromela in [13] (inspired by Promela [27]). Compositional models also exist. For example, Plath and Ryan [33] extended the SMV language to express how a feature modifies an SMV model. In this extension named fSMV, each feature is thus modelled in isolation, which paves the way for *compositional* reasoning. Classen et al. [12] 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 on the other.

Most of FTS-based analyses rely on the annotative model to design efficient family-based CTL and LTL verification algorithms [13, 12, 17]. The result of these algorithms 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 [14]: a set includes the features required to satisfy the property; the other contains the features that must be excluded. Another representation (used in Figure 3) represents the feature expressions labelling the transitions as Boolean formulae [13]. Then the result of a given verification analysis is another feature expression produced by computing conjunctions and disjunctions of the individual expressions. Whatever the chosen representation, it can be concisely encoded in a Binary Decision Diagram (BDD) [7, 12]. The analysis of Classen et al. [12] 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. The analysis of Classen et al. [13] 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. The model-checking is done forwards, as in SPIN [27], but some states may be re-explored if they are reached with a new combination of features.

Compositional reasoning on FTS has received less attention. Preliminary work [16, 18] studies the particular case where features only add or only remove behavior , and shows what properties are preserved upon the addition of these features. By applying this theory, one can thus infer the impact of each feature successively, and in the end determine which combinations of features lead to violations of the properties. Under the assumption that feature composition is commutative, BDD can again be used as an efficient encoding for these combinations. The general case where features have arbitrary impacts on the FTS remains to be addressed. Several papers deal with the compositional verification of features [20, 31]. Their application to FTS is a future work that is out of scope of this paper.

$$\begin{tikzcd}
\text{SMV + fSMV} \arrow[r, dotted, "\pi'"] \arrow[rr, dashed, bend left, "\gamma"] \arrow[d, "\hat{\alpha}"'] & \text{TS} \arrow[d, dotted, "\alpha"'] & \text{FTS} \arrow[l, dotted, "\pi"'] \arrow[d, dashed, "\hat{\alpha}"] \\
\substack{\text{Combinations}\\ \text{of features}} \arrow[r, dotted, "\sigma"] \arrow[d, dashed, "lift"'] & \substack{\text{CTL or LTL}\\ \text{(yes / no)}} & \substack{\text{Set of}\\ \text{products}} \arrow[l, dotted, "\sigma"'] \arrow[d, dashed, "lift"] \\
\substack{\text{Feature}\\ \text{expression}} \arrow[r, dashed, "\hat{\sigma}"] & \text{BDD} \arrow[u, dotted, "[\![\_]\!]_c"] & \substack{\text{Feature}\\ \text{expression}} \arrow[l, dashed, "\hat{\sigma}"']
\end{tikzcd}$$

π	projection [13]	π'	feature composition [33]
σ	product evaluation	$\hat{\sigma}$	BDD encoding
α	model checking	$\hat{\alpha}$	FTS model checking [13, 12]
$[\![\_]\!]_c$	BDD application	γ	lifted composition [12]

$\longrightarrow$	feature-based
$\dashrightarrow$	family-based
$\cdots\rightarrow$	product-based

Figure 4: Commutative diagram of product-line behavior verification strategies

The above review of the state of the art of FTS verification suggests the reuse of the structure of the commutative diagram in Figure 2, resulting in Figure 4.

From an FTS (see top right of Figure 4) one can follow a product-based strategy by computing the projection (π) of each product and apply standard model checking algorithm (α). The 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 as a BDD ($\hat{\sigma}$). On the other side, one finds the aforementioned fSMV model of Plath and Ryan [33]. The TS 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* [12] of the base model and all the features (γ). We believe that a feature-based strategy could arise by analyzing features individually (δ), yet this remains an open problem as of today. The soundness of the analyses is proved by the commutation of the upper-right diagram of Figure 4 and by the equivalence result by [12], similarly to Figure 1.

4. General and Configurable Analysis Framework

In this section, we generalize the discussion of Section 3. We first present an overview of a general and configurable analysis framework (Section 4.1), then we detail its construction process and elements (Section 4.2), and finally we illustrate some of its instances, providing initial evidence that the framework is indeed general (Section 4.3).

4.1. Analysis Framework Overview

By analyzing and abstracting the structures presented in the Figures 2 and 4, we arrive at the general diagram, shown in Figure 5.

In the simplest case, we analyze a variability-free model *Product* using some standard (non-variability aware) *analysis* technique.

In a nutshell, we apply the diagram shown in Figure 5 to analyze a given *Property* of a product line *Product* in a number of ways. In either *annotative* or *compositional product lines*, we can obtain a product (variability-free model) by *projection* (π) in annotative product lines or via model *composition* (π'), in the case of compositional product lines, then proceed with a non-variability-aware analysis. Alternatively, for annotative product lines, we can apply some *variability-aware analysis* to obtain *annotative expressions* that are either evaluated for a given product or further abstracted into *annotative lifted expressions* for a concise representation in an ADD, which generalizes a BDD. In the case of compositional product lines, some *variability-aware analysis* can be applied as well, but to obtain *compositional expressions* that are again either evaluated for a given product or further abstracted into *compositional lifted expressions* for a concise representation in an ADD.

At an abstract level, to perform an analysis of a given product line, one starts with a compositional (top-left corner in Figure 5) or an annotative product line (top-right corner), then follows any of the outgoing arrows while performing the respective analysis steps, until the desired properties are computed (either product-wise or as an ADD representing all possible values). Each path from a compositional or annotative product line until a property value has been computed defines an analysis strategy, which amounts to function composition of

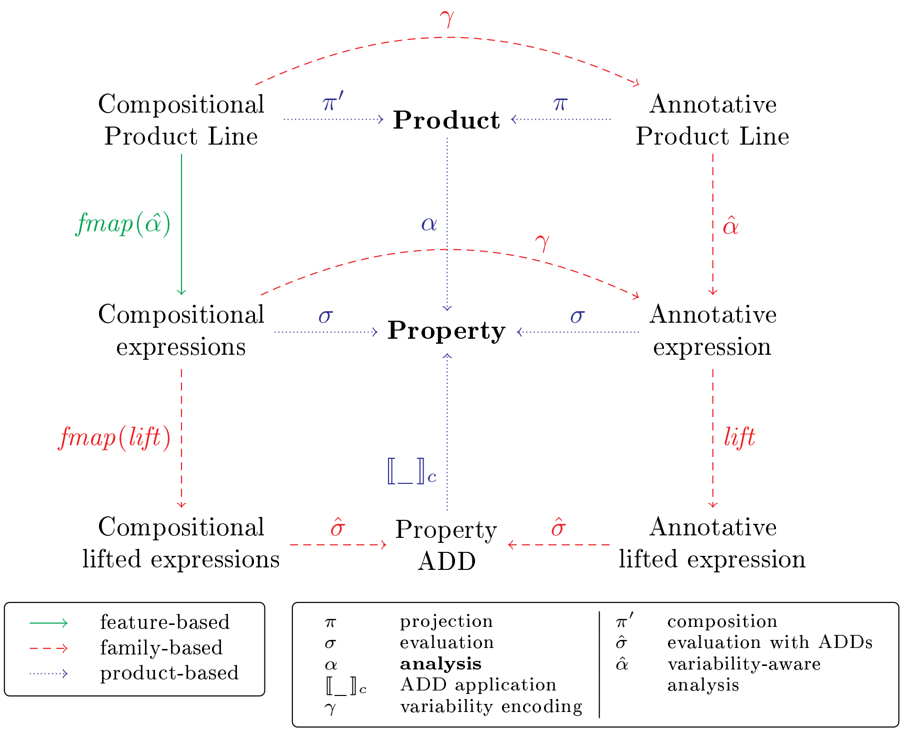


Figure 5: General and Configurable Analysis Framework

the intermediate analysis steps (arrows in the diagram). The nodes in the diagram represent models or abstractions thereof after application of such analysis steps.

As a result, the analyses are presented in a compositional manner as a conceptual framework, which provides an overall understanding of how the different types of product-line analyses compose and inter-relate. For instance, feature-family-based analysis (in Figure 5, from compositional product line, down, down, and right) and feature-product-based analysis (from compositional product line, down, and right) share the feature dimension of the analysis (down from compositional product line). Furthermore, like the Figures 2 and 4, we conjecture that the diagram in Figure 5 is a commuting diagram: different analysis paths yield equal results if they share the start and end points.

4.2. Abstraction Process and Framework Description

To create the framework, we reviewed existing analyses for the different models and properties mentioned in Section 3 and identified essential abstractions of such analyses and their structure. Specifically, by rewriting concrete models and properties into generic concepts, we assessed the impact on dependent elements (thereby also rewriting these latter recursively), while identifying assumptions that these elements should fulfill to keep the diagram's structure. For example, by abstracting *DTMC* into *Product*, derivation by projection (π) also needs to be rewritten so that π's domain becomes a generic *Product* model with annotations, and its codomain becomes *Product*. The ultimate goal is that the framework is consistent, having at least the instances of Section 3, and becomes a generic/parametric theory, consisting of a structure of generic concepts to be further evaluated with instantiations for different product models and properties like safety, performance, static analysis, and others discussed in Section 4.3.

Table 1 summarizes the analysis and abstraction process by showing in each row the corresponding elements of Figures 2 and 4 with the general corresponding element in Figure 5, together with a brief description of the latter element in the general framework. In the following, we describe each element in more detail, formalizing it in PVS.

Product is a variability-free model defining the type of the product line instances. Without loss of generality, such model is either a basic non-dividable construct or a nested composite structure, given by corresponding type constructors in the following PVS specification snippet

```
Product : DATATYPE
   BEGIN
      basic: basic?
      composite (m1,m2: Product): composite?
   END Product
```

Such model has a computable *Property* of interest such as reliability. It is assumed that the property has a numerical or Boolean type, which has a high prevalence in existing analyses [39], e.g., reliability, safety, performance, and static analysis. In PVS, the property is specified as the following non-empty subtype from the disjoint union type of types `bool` and `real`

```
 Property: TYPE+ FROM [bool+real]
```

In practice, we note that an ADD-based representation is often used for a space-efficient encoding of values and expressions and also that it encompasses a BDD-based representation, since the value nodes in a ADD can represent values other than Booleans.

The function α is a variability-free analysis that performs the computation of the property like reachability probability analysis for reliability (cf. Section 3.1). To analyze a non-variable Product model, we return some uninterpreted value in the basic case (p$_b$`ottom`) or apply a function to combine the intermediate results emerging from the recursive application of the analysis function in the nested structure of the model:

```
alpha(m: Product) : RECURSIVE Property =
   CASES m OF
      basic: p_bottom,
      composite(m1, m2): analyzeModelCompositeShell(alpha(m1), alpha(m2))
   ENDCASES
 MEASURE m by <<
```

where the `MEASURE` clause indicates a well-founded order relation to be checked on original and recursive argument of the function for termination purposes.

Product and *Property* are in bold in Table 1 and in Figure 5 because they are the parameters of general framework.

On the right-hand side of the diagram, *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 to annotate model elements. Specifically, an annotative model is either a base variability-free Product model, a variation point with choices depending on a presence condition, or a nested variability-rich model:

```
AnnotativeModel : DATATYPE
   BEGIN
      ModelBase(m:Product): ModelBase?
      ModelChoice(pc:PresenceCondition, vm1:AnnotativeModel, vm2:AnnotativeModel): ModelCh
      ModelComposite (vm1,vm2: AnnotativeModel): ModelComposite?
   END AnnotativeModel
```

The function π performs product derivation on annotative product lines by presence condition evaluation (*projection*) of its underlying annotative model, obtaining a variability-free model representing a derived product of the product line.

```
pi(vm:AnnotativeModel,c:Conf): RECURSIVE Product =
   CASES vm OF
      ModelBase(m): m,
      ModelChoice(pc, vm1, vm2):
                    IF c(pc) THEN pi(vm1,c)
                             ELSE pi(vm2,c)
                    ENDIF,
      ModelComposite(vm1, vm2): composite(pi(vm1,c),pi(vm2,c))
   ENDCASES
  MEASURE vm by <<
```

The function $\hat{\alpha}$ is a variability-aware analysis on annotative models, that is, it incorporates knowledge about variability and explores commonalities (sharing) in such models. Strategies for achieving this are *late splitting* (analyzing equal model elements only once after a common entry) and *early joining* (joining intermediate results as early as possible) [3]. In practice, it might difficult to achieve both, while still providing sound analyses, due to context dependencies on the model and property at hand (e.g., path sensitive analyses). In particular, we employ a generative approach whereby we derive $\hat{\alpha}$ from α by analyzing an annotative model via structural recursion, lifting variability from the structure of the model to the structure of annotative expressions representing values of the property. In case the model has no variability, a variability-free expression, i.e., the result of applying α to the model is returned. Otherwise, either an annotative choice expression or an annotative composite expression is returned.

```
 hatAlpha(vm:AnnotativeModel): RECURSIVE AnnotativeExpression =
   CASES vm OF
       ModelBase(m): BaseExpression(alpha(m)),
       ModelChoice(pc, vm1, vm2): ChoiceExpression (pc, hatAlpha(vm1), hatAlpha(vm2) ),
       ModelComposite(vm1, vm2): CompositeExpression (hatAlpha(vm1), hatAlpha(vm2) )
   ENDCASES
 MEASURE vm by <<
```

Annotative expressions, whose type constructors are `BaseExpression`, `ChoiceExpression`, and `CompositeExpression` (as seen above), denote property values of products (e.g., top-right side of Figure 1), which are either evaluated (σ) to a value or *lifted* into an ADD-based expression, the latter also being evaluable to a given ADD (e.g., Figure 13 in previous work [8]). For instance, annotative expression evaluation is defined by structural recursion on the expression as follows:

```
sigma(vp:AnnotativeExpression,c:Conf): RECURSIVE Property =
   CASES vp OF
      BaseExpression(p): p,
      ChoiceExpression(pc, vp1, vp2):
            IF c(pc) THEN sigma(vp1,c)
                     ELSE sigma(vp2,c)
            ENDIF,
      CompositeExpression(analysisShell, vp1, vp2): analysisShell(sigma(vp1,c), sigma(vp2,
   ENDCASES
  MEASURE vp by <<
```

Furthermore, the top-right quadrant means that the evaluation of the annotative expression obtained from $\hat{\alpha}$ for a given configuration yields the same result as if the variability had been bound for such configuration, and the resulting product analyzed via the non-variability aware analysis α. This theorem is specified in PVS as follows:

```
commutative_product_family_product:
THEOREM  sigma(hatAlpha(annotativeModel),conf) = alpha(pi(annotativeModel,conf))
```

Proof sketch. For an arbitrary configuration `conf`, we need to prove that `sigma(hatAlpha(annotativeModel),conf) = alpha(pi(annotativeModel,conf))` holds for any `annotativeModel`. We prove this by induction over `annotativeModel`. This generates three subgoals, which correspond to the three possibilities for the `AnnotativeModel` datatype. In the first subgoal, we need to prove that `sigma(hatAlpha(ModelBase(m)),conf) = alpha(pi(ModelBase(m),conf))`. By expanding `hatAlpha` on the LHS, and product derivation in the RHS, we then have to prove that `sigma(BaseExpression(alpha(m)),conf) = alpha(m)`. We then expand `sigma` on the LHS, resulting on `alpha(m) = alpha(m)`, which is trivially true.

In the second subgoal, for an arbitrary presence condition `pc` and annotative models `am1` and `am2`, by the induction hypothesis, we have that `sigma(hatAlpha(am1), conf) = alpha(pi(am1, conf))` and `sigma(hatAlpha(am2), conf) = alpha(pi(am2, conf))`. We then have to prove that `sigma(hatAlpha(ModelChoice(pc, am1, am2)), conf) = alpha(pi(ModelChoice(pc, am1, am2), conf))`. By expanding `sigma` and `hatAlpha` on the LHS, and `pi` on the RHS, we then have two possible situations, corresponding exactly to the two induction hypothesis. If the presence condition `pc` is satisfied according to `conf`, we have to prove that `sigma(hatAlpha(am1), conf) = alpha(pi(am1, conf))`, which is already given by one of the induction hypothesis. If the presence condition is not satisfied, we then have to prove `sigma(hatAlpha(am2), conf) = alpha(pi(am2, conf))`, concluding the proof of this subgoal since this is also given by the induction hypothesis.

Finally, in the third subgoal, for arbitrary annotative models `am1` and `am2`, by the induction hypothesis we have that `sigma(hatAlpha(am1), conf) = alpha(pi(am1, conf))` and `sigma(hatAlpha(am2), conf) = alpha(pi(am2,` `conf))`. We then have to prove that `sigma(hatAlpha(ModelComposite(am1, am2)), conf) = alpha(pi(ModelComposite(am1, am2), conf))`. By expanding `sigma` and `hatAlpha` on the LHS, and `alpha` and `pi` on the RHS, we then have to prove that `analyzeModelCompositeShell(sigma(hatAlpha(am1),` [redacted] `conf), sigma(hatAlpha(am2), conf)) = analyzeModelCompositeShell(alpha(pi(am1,` [redacted] `conf)), alpha(pi(am2, conf)))`. This is trivially true given the two induction hypothesis.

On the left-hand side of the diagram, *Compositional Product Line* also comprises a feature model, a configuration knowledge, and variant-rich model called *compositional model*, which is a structure of related annotative models. For instance, a compositional model is depicted on top-right side of Figure 6 where each square could represent an annotative model like the one on the top-left side of Figure 1. Derivation by *composition* (π') operates on compositional product lines generating a product by composing the annotative models within the compositional model of the product line for a given configuration of its feature model. Compositional product lines can be analyzed by mapping ($fmap(\hat{\alpha})$ in Figure 5) the variability-aware analysis $\hat{\alpha}$ over the structure of their compositional model to yield a compositional expression with an isomorphic structure. For example, in the right-hand side of Figure 6, each φ within the compositional expression is similar to the top-right side of Figure 1 and is obtained by applying the variability-aware analysis $\hat{\alpha}$ to each annotative model (represented by a square in the figure) within the compositional model. These compositional expressions can be either evaluated to a value or further lifted into an compositional lifted expressions, which are ADD-encoded compositional rational expressions. Compositional product lines and compositional expressions can be mapped into corresponding annotative representations by means of *variability encoding*, (γ) which leverages an *if-then-else* operator to switch between possible behaviors.

Table 1: Analysis and Abstraction Process for Defining the General and Configurable Analysis Framework

Element in the General Framework	**Description**	**Reliability Analysis Element**	**Behavioral Analysis Element**
Property	Computable property	Reliability value	Temporal logic property
Product	Variability-free model	DTMC	TS
Analysis (α)	Variability-free analysis	Model checking	Model checking
Compositional Product Line	SPL with compositional asset base	Compositional probabilistic model	SMV+fSMV
Annotative Product Line	SPL with presence conditions annotating model elements	Annotative probabilistic model	FTS
Compositional expressions	Denotes property values at products	Compositional rational expressions	Combination of features
Annotative expressions	Denotes property values at products	Annotative rational expressions	Sets of products
Compositional lifted expressions	ADD-encoded compositional rational expressions	Compositional lifted expressions	Feature expression
Annotative lifted expression	ADD-encoded annotative rational expressions	Annotative lifted expression	Feature expression
Property ADD	Maps configurations to values	Reliability ADD	BDD
Projection (π)	Derivation by presence condition evaluation projection	PMC projection	projection
Composition (π')	Derivation by model composition	PMC composition	feature composition
Variability-aware analysis ($\hat{\alpha}$)	Explores commonalities (sharing)	Parametric model checking	FTS model checking
Evaluation (σ)	Maps expression to values	Evaluation	Product evaluation
Evaluation with ADDs ($\hat{\sigma}$)	Maps lifted expressions to ADD values	Evaluation with ADDs	BDD encoding
lift	Lifts expression to have ADD semantics	*lift*	*lift*
Variability encoding (γ)	Maps compositional into annotative representation	Variability encoding	Lifted composition
ADD application ($\llbracket \_ \rrbracket_c$)	Provides the value mapped by a configuration	ADD application	BDD application

4.3. Framework Instances

To provide evidence of the framework's generality and configurability, we show in Table 2 how the elements of the generic diagram in Figure 5 can be instantiated for different properties and existing analyses. Each column describes a group of analysis strategies for some model and property. The first column describes the diagram's elements. The remaining columns correspond to further research, which we briefly describe next. In Section 4.4, we further discuss the implications of this table. In every cell from columns two onwards, we refer to the instantiation of the diagram's elements in each work by mentioning the corresponding concrete concepts therein together with definition numbers when available.

Table 2: Instances of the framework

Element in the General Framework	**Safety Analysis Element**	**DOP Theorem Proving**	**Performance Analysis Element**	**Static Analysis Element**
Property	Safety	First-order logic formula	Throughput	IFDS properties
Product	Java and C code	Abstract Behavioral Specification	PAAD (Def. 1)	IFDS flow graph
Compositional Product Line	Feature modules	ABS core and deltas	PAAD and PAAD deltas (Def. 5)	–
Annotative Product Line	Simulator	–	150% model (Def. 7)	Lifted IFDS flow graph
Compositional expressions	–	–	–	–
Annotative expressions	Sets of products	–	–	Sets of products
Compositional lifted expressions	–	–	–	–
Annotative lifted expression	Feature expressions	–	–	Feature expressions
Property ADD	BDD	–	–	BDD
Projection	–	–	Projection (Def. 8)	Propagation
Composition	Superimposition	Delta application	PAAD delta application (Def. 6)	–
Analysis	Model checking	Theorem proving	Traffic equation solving (Def. 4)	IFDS analysis
Variability-aware analysis	Model checking	Theorem Proving	Parametric traffic equation solving	IFDS/IDE analysis
Evaluation	–	–	Expression evaluation	Boolean evaluation
Evaluation with ADDs	BDD encoding	–	–	Boolean evaluation
lift	*lift*	–	–	*lift*
Variability encoding	Variability encoding	–	–	–
ADD application	–	–	–	BDD application

4.3.1. Static Analysis

The generalization of the structure of our diagram is not limited to the verification of probabilistic and non-probabilistic behavioral properties. Consider, e.g., the SPL static analysis method proposed by Bodden et al. [**?**]. Their method, named SPL^{LIFT}, seamlessly lifts popular static analyses (e.g. taint analysis) to SPLs. To that end, they label data flow functions with a feature or its negation, to represent the cases where the feature is enabled or not, respectively. This leads to parametric flow functions which can be composed to represent the flow of all the SPL products in a concise representation. This representation is similar to the annotative FTS models introduced by Classen et al. [13] for behavioral model checking. The generalization of the structure of the diagram is therefore also similar: from the annotated flow functions, 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 the features, which is equivalent to a feature expression. As Classen et al. for model checking, Bodden et al. have not explored compositional (feature-based) reasoning.

4.3.2. Security Analysis

Peldszus el al. [ref] proposed SecPL, a method for managing security requirements systematically in an SPL, promoting a model-based analysis for security. The SecPL profile specifies security requirements and product-line variability in UML models. To do so, the authors extended UMLsec stereotypes with presence conditions, allowing users to create a profile for requirements that support variability. In this way, users use UMLsec stereotypes for security specifications, modeling products that have annotative structures and behavioral elements through presence conditions, expressions on a set of features. The SecPL profile consists entirely of 17 stereotypes and includes validation rules for presence conditions well-formedness. To analyze this method, the authors proposed a family-based approach, which lifts checks such as secure dependentes from the level of individual products to the entire SPL. The analysis assumes an encoding check as an OCL constraint. To evaluate the constraints on the model, we use a method called template interpretation, which supports the evaluation of OCL constraints on model in which model elements are annotated with present conditions. As a result, a propositional formula that can be evaluated by the SAT solver is generated. If the formula is satisfiable, the feature set generates an unsafe product. Otherwise, one have proof that each product contains the specified security properties. In addition to the initial intention to apply safety analyzes from the beginning, the method also allows to apply it to legacy SPLs, through reverse engineering. The key idea is that developers annotate security-critical parts of the source code of the SPL entry. From this, reverse engineering tools extract the UML model from the base code. Still, it is a parse with preprocessor annotations, whose conditional types become SecPL profile stereotypes, and other annotations become regular UMLsec stereotypes. From this, the model-based security analysis is done.

4.3.3. Safety

Apel et al. [4] provide the toolchain 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 [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 is leveraged to perform variability encoding: 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 means performing analysis without variability until encountering it, that is, the model checker explores only once execution paths of different products as long as such paths are equal; the latter means 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 are sets of products, which are encoded in BDDs for efficient representation.

4.3.4. Delta-oriented Theorem Proving

Hähnle and Schaefer [26] present a feature-family-based analysis strategy for product lines based on Delta Oriented Programming (DOP). 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 then proceeds by composing the core with a sequence of deltas. The proposed analysis relies on an extended Liskov principle for DOP, whereby the contracts of subsequent deltas must become more specific; invariants cannot be removed, but only added; methods called in deltas use the contract of the first implementation of that method. Therefore, the core and each delta are analyzed in isolation and then each delta is analyzed with respect to a subset of deltas regarding addition of more specific behavior. This leads to a compositional verification approach for product lines. The analysis is also family-based, since it is performed only on domain artifacts without the need to generate any products.

4.3.5. Performance

Kowal et al. [29] 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 such model, nodes represent a service center with given multiplicity, initial client distribution, and service time distribution; edges are annotated with probabilities to model operational profiles. Based on this model, 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 this 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 by expression evaluation give the throughput of the model.

Variability management is accomplished via delta-modeling such that a core PAAD is represented together 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 is a super-variant subsuming every concrete variant of the family, consisting of all nodes and transitions that are added or modified by some delta. This model undergoes variability-aware analysis such that a parametric (all elements that are added, removed, or modified are represented by parameters) system of ordinary differential equations is solved to analyze this model and then perform expression evaluation per configuration. Similar to Classen et al. for model checking, Kowal et al. have not explored compositional (feature-based) reasoning.

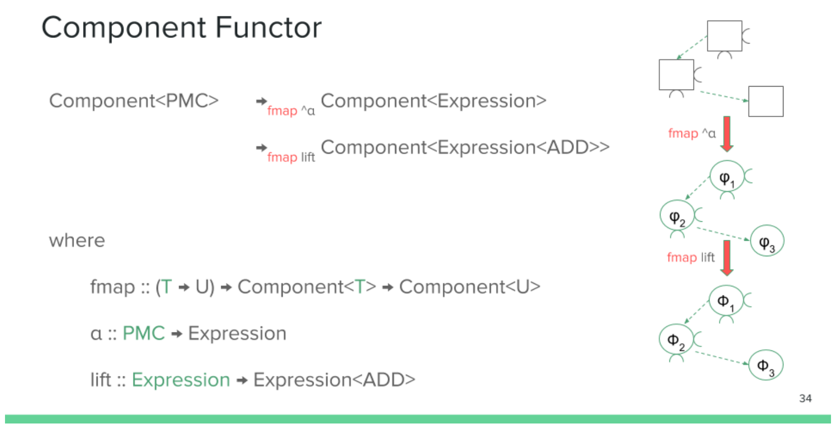
4.4. Discussion

We note in Table 2 that both functional and non-functional properties are instantiated. Depending on the property and model, the total number of possible analysis strategies may change. Although for reliability we 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). Additionally, in principle, new instances can also be conceived, in which case we rely on the original diagram, but have a different property and model.

The extent to which reuse is achieved depends on the property and on the model. For instance, for reachability properties, the same model (PMC) for reliability can be reused. Indeed, by analyzing the ReAna tool [30], we conjecture that it is possible to achieve high reuse level 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 the 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.

We identified higher-level abstractions in the framework involving a number of elements:

- **Component functor**: Both quadrants on the left-hand side of Figure 5 indicate that there is functor (a structure that can be mapped over), capturing the structure of the compositional model across the compositional expressions and lifted compositional expressions, as shown in Figure 6. The *Component* functor is applied to the annotative model to obtain the compositional model on the top-right, where the arrows indicate the relation between annotative models represented by the squares. The *fmap* function maps the variability-aware analysis function $\hat{\alpha}$ and this structure on the top-right to the application of the functor to the expressions (below the first vertical arrow in the figure), where circles represent expressions


Component Functor

Component<PMC> $\xrightarrow{}_{\text{fmap } \hat{}\alpha}$ Component<Expression>

$\xrightarrow{}_{\text{fmap lift}}$ Component<Expression<ADD>>

where

fmap :: (T → U) → Component<T> → Component<U>

α :: PMC → Expression

lift :: Expression → Expression<ADD>

34

Figure 6: Component Functor.

denoted by φs and arrows their relation. Furthermore, *fmap* maps this resulting structure and the *lift* function to the functor applied to ADDs, which is a structure relating lifted expressions, represented by arrows connecting the circles with Φs.

- **Variability restriction**: From the right-hand side of Figure 5 to its center, there is a variability restriction step (cf. von Rhein's Variability Restriction operator [36]), which binds the variablity according to a configuration. Such step is performed at different types: product line and expressions. Additionally, from the left-hand side to the center, a variability restriction step is combined with a folding behavior over the functor structure to obtain a product or a property value.

- **Lifting to ADDs**: Both lower quadrants of the diagram in Figure 5 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 obviating from enumeration during algebraic manipulation and evaluation. Further details can be checked elsewhere [8].

Although the commutative diagrams show logically equivalent analyses for a given model and property, they currently do not convey practical considerations in terms of efficiency. For instance, in Figure 5, for the feature-product-based analysis to be efficient it is necessary that the compositional expressions and their composition are less complex than evaluating the whole product. Otherwise, the product-based analysis is preferable. In general, such considerations 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.

Since the framework is generic and configurable, the process of defining its instances needs guidance. Indeed, the order of instantiation of its elements and

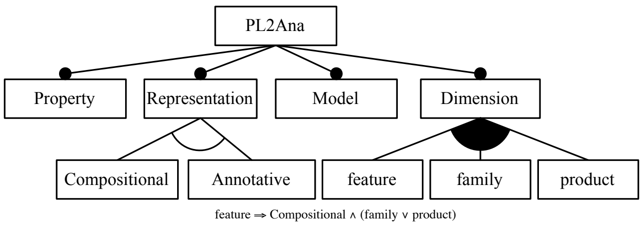


Figure 7: Feature Model of the Product Line of Product Line Analysis Strategies.

the valid choices to be made at each node need to be made systematically. To address this, we leverage Software Product Line Engineering to define a product line of product-line analysis strategies, as discussed next.

5. Product Line of Product-Line Analyses

With the aim of providing a systematic way for using the general unified framework presented in Section 4, we define a product line of product-line analysis strategies (Section 5.1). By explicitly establishing product line elements, we can guide users in the configuration and instantiation of the general unified framework. Moreover, in Section 5.2, we also explain how this product line evolved.

5.1. Definition of Product Line of Product-Line Analyses

We use a three-space model to define the elements from the product line of product-line analysis strategies (PL^2Ana), as follows:
$PL^2Ana = (FM, CK, AB)$,
where

- FM is the variability model, given according to Figure 7;
- AB comprises the asset base of this product line, in this case functions and types of the framework in Figure 5;
- CK denotes the configuration knowledge, which maps feature expressions to assets; in this case, it conceptually maps feature expressions to path fragments in the framework.

The feature model shows that the PL^2Ana's configurability depends on the model and property to be analyzed, the representation (annotative versus compositional), and on analysis dimensions: feature, family, product (cf. [39]). We leverage attributes in the feature model to properly instantiate a given property, model, and specific compositional or annotative representations. With respect to the asset base, since the framework targets properties that can be represented in an ADD, as discussed in Section 4.2, the functions and types of the lower quadrants of Figure 5, except *Property*, have the specific interpretation mentioned in Section 4.2. However, since we are still interested in an abstract and conceptual description of analyses, the remaining types and functions in AB are uninterpreted, i.e., we provide no concrete specification nor implementation for them, since these also depend on the model and property to be analyzed.

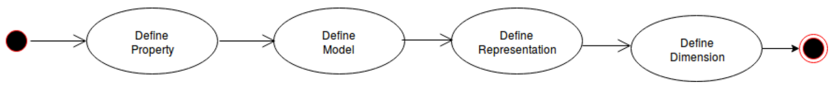


Figure 8: Product Derivation Workflow.

Additionally, at a coarse-grained level, they must comply with the structure in the diagram.

The configuration knowledge, as stated, is responsible for mapping features into assets. In our case, as mentioned, assets are functions and types, so the model illustrated in Table 3 associates feature expressions with function compositions and type instantiations. For instance, when we select both feature- and family- dimensions from the feature model, we compose the corresponding functions in the LHS of Figure 5. Additionally, given the cross-tree constraint of the feature model, the compositional representation is also selected, which implies that the type *Compositional Product Line* is instantiated, as specified by the corresponding row in the configuration knowledge. Furthermore, the feature expressions for *Property*, *Model*, and *Representation* evaluate to `true`, since these are mandatory features, so their corresponding instantiations are always performed. The root feature *PL2Ana* does not appear in the table because it is an abstract feature.

Table 3: Configuration Knowledge.

Feature Expression	**Transformation**
$\mathit{Property}(\texttt{true})$	$\texttt{instantiate}(\mathit{Property})$
$\mathit{Model}(\texttt{true})$	$\texttt{instantiate}(\mathit{Product})$
$\mathit{Representation}(\texttt{true})$	$\texttt{instantiate}(\mathit{AnnotativeModel})$
$\mathit{Annotative}$	$\texttt{instantiate}(\mathit{AnnotativeProductLine})$
$\mathit{Compositional}$	$\texttt{instantiate}(\mathit{CompositionalProductLine})$
$\mathit{family} \vee \mathit{feature}$	$\texttt{instantiate}(\hat{\alpha})$
family	$\hat{\alpha} \circ \mathit{lift} \circ \hat{\sigma}$
$\mathit{family} \wedge \mathit{product}$	$\hat{\alpha} \circ \sigma$
$\mathit{feature} \wedge \mathit{family}$	$\mathit{fmap}(\hat{\alpha}) \circ \mathit{fmap}(\mathit{lift}) \circ \hat{\sigma}$
$\mathit{feature} \wedge \mathit{product}$	$\mathit{fmap}(\hat{\alpha}) \circ \sigma$
$\mathit{product}$	$\texttt{instantiate}(\alpha)$
$\mathit{annotative} \wedge \mathit{product}$	$\pi \circ \alpha$
$\mathit{compositional} \wedge \mathit{product}$	$\pi' \circ \alpha$

The products of this product line are then product-line analysis strategies for a given model and property. The workflow in Figure 8 describes the product derivation process. Correspondingly, one first defines the model and the property to be analyzed (or vice-versa), then decides on the representation (annotative versus compositional), and finally on the analysis dimensions (feature, family, product). Derivation then proceeds by evaluating the *CK* according to the given configuration, composing the functions along its path.

An example of such workflow is given in Figure 9. We see that we might de-

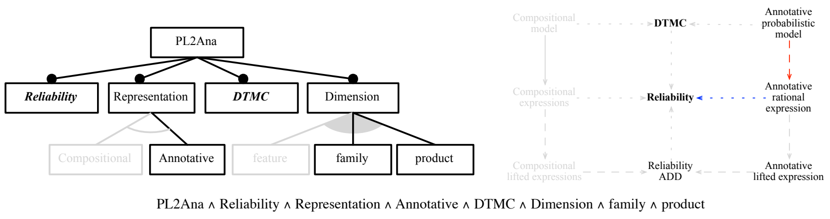

PL2Ana
Reliability
Representation
DTMC
Dimension
Compositional
Annotative
feature
family
product
Compositional model
DTMC
Annotative probabilistic model
Compositional expressions
Reliability
Annotative rational expression
Compositional lifted expressions
Reliability ADD
Annotative lifted expression
PL2Ana ∧ Reliability ∧ Representation ∧ Annotative ∧ DTMC ∧ Dimension ∧ family ∧ product

Figure 9: Workflow instantiated with Reliability Analysis.

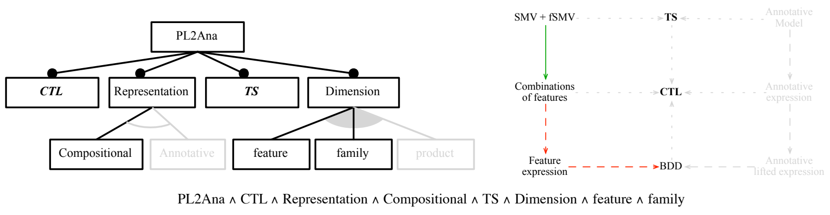


Figure 10: Workflow instantiated with Behavior Analysis.

rive a family-product reliability analysis for annotative models by choosing Reliability and DTMCs as property and model attributes, respectively. Moreover, we select the representation and analysis dimension accordingly, to produce an analysis strategy that consists of the RHS of Figure 5, particularly instantiated with the chosen property and model.

Another instance of the workflow is given in Figure 10. In this example, we derive a feature-family behavior analysis for compositional models (SMV and fSMV) by choosing CTL and TS as property and model attributes, respectively. Moreover, we select the representation and analysis dimension accordingly, to produce an analysis strategy that consists of the LHS of Figure 5, particularly instantiated with the chosen property and model.

5.2. Evolution of Product Line of Product-Line Analyses

Motivated by the need to empirically compare the proposed feature-family reliability analysis strategy to other strategies [30], the authors implemented the ReAna tool, which actually realizes the diagram in Figure 2. In this section, we first show how this development can be understood in terms of a principled evolution of previous works, as depicted in Figure 11. Essentially, we can apply the *merge template* [?] to incrementally evolve the product line, starting from the feature-family [30] and feature-product strategies [21], then incorporating the family-product [32] and lastly the family-based [37] strategies. The process' soundness follows from the soundness of the employed template.

We now describe in more detail how the diagram of Figure 2 evolves into the generic framework and the encompassing product line. As mentioned in the beginning of Section 4.2, we abstracted concrete into uninterpreted types with assumptions along the structure of the diagram. We illustrate the mechanics of the process showing initial types involved as follows.

Acknowledgements

Andreas Stahlbauer, Malte Lochau, Matthias Kowal, Ina Schaefer.

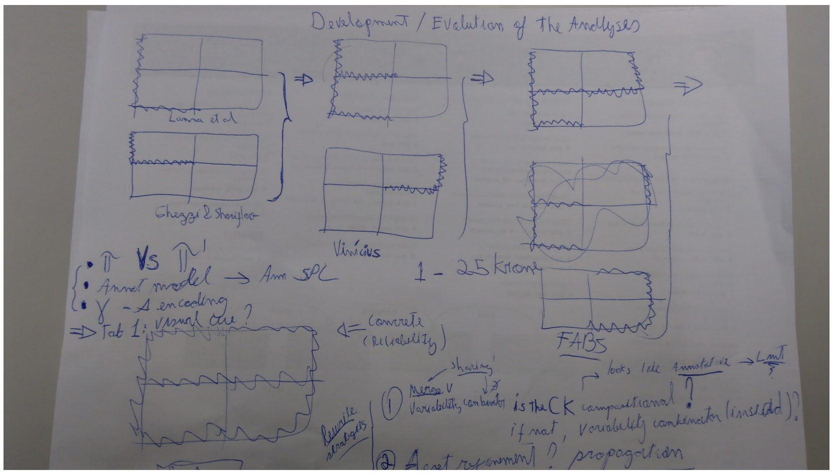


Figure 11: Evolution of Product Line of Product-Line Analysis Strategies.

References

References

[1] Apel, S., Batory, D.S., Kästner, C., Saake, G., 2013a. Feature-Oriented Software Product Lines – Concepts and Implementation. Springer. doi:10.1007/978-3-642-37521-7.

[2] Apel, S., Kästner, C., Lengauer, C., 2013b. Language-independent and automated software composition: The featurehouse experience. IEEE Trans. Software Eng. 39, 63–79. URL: https://doi.org/10.1109/TSE.2011.120, doi:10.1109/TSE.2011.120.

[3] Apel, S., von Rhein, A., Wendler, P., Größlinger, A., Beyer, D., 2013c. Strategies for product-line verification: case studies and experiments, in: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pp. 482–491. URL: https://doi.org/10.1109/ICSE.2013.6606594, doi:10.1109/ICSE.2013.6606594.

[4] Apel, S., von Rhein, A., Wendler, P., Größlinger, A., Beyer, D., 2013d. Strategies for product-line verification: case studies and experiments, in: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pp. 482–491. URL: https://doi.org/10.1109/ICSE.2013.6606594, doi:10.1109/ICSE.2013.6606594.

[5] Apel, S., Von Rhein, A., Wendler, P., Groslinger, A., Beyer, D., 2013e. Strategies for product-line verification: Case studies and experiments, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE Press, Piscataway, NJ, USA. pp. 482–491. doi:10.1109/ICSE.2013.6606594.

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

[] Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M., 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), pp. 355–364. doi:10.1145/2491956.2491976.

[7] Bryant, R.E., 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24, 293–318.

[8] Castro, T., Lanna, A., Alves, V., Teixeira, L., Apel, S., Schobbens, P.Y., 2017. All roads lead to rome: Commuting strategies for product-line reliability analysis. Science of Computer Programming , –URL: https://www.sciencedirect.com/science/article/pii/S0167642317302253, doi:https://doi.org/10.1016/j.scico.2017.10.013.

[9] Chechik, M., Devereux, B., Gurfinkel, A., 2001. Model-checking infinite state-space systems with fine-grained abstractions using spin, in: SPIN '01, pp. 16–36.

[10] Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C., 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), Springer. pp. 287–304. doi:10.1007/978-3-662-49665-7_17.

[11] Clarke, E.M., Emerson, E.A., 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Logic of Programs, Springer. pp. 52–71.

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

[13] Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., cois Raskin, J.F., 2013. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. Transactions on Software Engineering , 1069–1089.

[14] Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F., 2010. Model checking lots of systems: efficient verification of temporal properties in software product lines, in: ICSE'10, ACM. pp. 335–344.

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

[16] Cordy, M., Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., 2012a. Managing evolution in software product lines : A model-checking perspective, in: VaMoS'12, ACM. pp. 183–191.

[17] Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y., Dawagne, B., Leucker, M., 2014. Counterexample guided abstraction refinement of product-line behavioural models, in: FSE'14 (to appear), ACM.

[18] Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., 2012b. Towards an incremental automata-based approach for software product-line model checking, in: Proceedings of the 16th International Software Product Line Conference-Volume 2, ACM. pp. 74–81.

[19] Fischbein, D., Uchitel, S., Braberman, V., 2006. A foundation for behavioural conformance in software product line architectures, in: ROSATEA'06, ISSTA 2006 workshop, ACM Press. pp. 39–48.

[20] Fisler, K., Krishnamurthi, S., 2008. Decomposing verification around end-user features, in: Meyer, B., Woodcock, J. (Eds.), Verified Software: Theories, Tools, Experiments. Springer-Verlag, Berlin, Heidelberg, pp. 74–81.

[21] Ghezzi, C., Molzam Sharifloo, A., 2013. Model-based verification of quantitative non-functional properties for software product lines. Information and Software Technology 55, 508–524. doi:10.1016/j.infsof.2012.07.017.

[22] Grunske, L., 2008. Specification patterns for probabilistic quality properties, in: ICSE '08, ACM, New York, NY, USA. pp. 31–40. doi:10.1145/1368088.1368094.

[23] Haber, A., Rendel, H., Rumpe, B., Schaefer, I., 2011. Delta modeling for software architectures, in: MBEES, pp. 1–10.

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

[25] Hähnle, R., Schaefer, I., 2012. A liskov principle for delta-oriented programming, in: Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA), Springer, Berlin, Heidelberg. pp. 32–46. doi:10.1007/978-3-642-34026-0_4.

[26] Hähnle, R., Schaefer, I., 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, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I, pp. 32–46. URL: https://doi.org/10.1007/978-3-642-34026-0_4, doi:10.1007/978-3-642-34026-0_4.

[27] Holzmann, G., 2003. Spin Model Checker, the: Primer and Reference Manual. First ed., Addison-Wesley Professional.

[28] Huth, M., Jagadeesan, R., Schmidt, D., 2001. Modal transition systems: A foundation for three-valued program analysis, in: Sands, D. (Ed.), Programming Languages and Systems. Springer Berlin Heidelberg. volume 2028 of *Lecture Notes in Computer Science*, pp. 155–169.

[29] Kowal, M., Tschaikowski, M., Tribastone, M., Schaefer, I., 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), pp. 407–417. doi:10.1109/ASE.2015.16.

[30] Lanna, A., Castro, T., Alves, V., Rodrigues, G., Schobbens, P.Y., Apel, S., 2017. Feature-family-based reliability analysis of software product lines. Information and Software Technology URL: http://www.sciencedirect.com/science/article/pii/S0950584917302197, doi:https://doi.org/10.1016/j.infsof.2017.10.001.

[31] Liu, J., Basu, S., Lutz, R.R., 2011. Compositional model checking of software product lines using variation point obligations. Automated Software Engg. 18, 39–76. doi:http://dx.doi.org/10.1007/s10515-010-0075-7.

[32] Nunes, V., Fernandes, P., Alves, V., Rodrigues, G., 2012. Variability management of reliability models in software product lines: An expressiveness and scalability analysis, in: Proceedings of the Sixth Brazilian Symposium on Software Components Architectures and Reuse (SBCARS), pp. 51–60. doi:10.1109/SBCARS.2012.23.

[33] Plath, M., Ryan, M., 2001. Feature integration using a feature construct. SCP 41, 53–84.

[34] Pnueli, A., 1977. The temporal logic of programs, in: FOCS'77, pp. 46–57.

[35] Pohl, K., Böckle, G., van der Linden, F.J., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Secaucus, NJ, USA.

[36] von Rhein, A., Apel, S., Kästner, C., Thüm, T., Schaefer, I., 2013. The PLA model, in: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS), ACM Press, New York, New York, USA. p. 1. doi:10.1145/2430502.2430522.

[37] Rodrigues, G.N., Alves, V., Nunes, V., Lanna, A., Cordy, M., Schobbens, P., Sharifloo, A.M., Legay, A., 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 Computer Society. pp. 173–180. doi:10.1109/HASE.2015.34.

[38] Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N., 2010. Delta-oriented programming of software product lines, in: Proceedings of the 14th International Conference on Software Product Lines (SPLC), Springer, Berlin, Heidelberg. pp. 77–91.

[39] Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G., 2014. A classification and survey of analysis strategies for software product lines. ACM Computing Surveys 47, 1–45. doi:10.1145/2580950.

[40] Thüm, T., Schaefer, I., Apel, S., Hentschel, M., 2013. Family-based deductive verification of software product lines. ACM SIGPLAN Notices 48, 11–11–20–20. doi:10.1145/2480361.2371404.