Model Mining and Efficient Verification of Software Product Lines

Siavash Soleimanifard\textsuperscript{a}, Dilian Gurov\textsuperscript{a}, Ina Schaefer\textsuperscript{b}, Bjarte M. Østvold\textsuperscript{c}, Minko Markov\textsuperscript{d}

\textsuperscript{a}KTH Royal Institute of Technology, Stockholm, Sweden
\{siavasha,dilian\}@csc.kth.se

\textsuperscript{b}Technical University of Braunschweig, Braunschweig, Germany
i.schaefer@tu-braunschweig.de

\textsuperscript{c}Norwegian Computing Center, Oslo, Norway
bjarte@nr.no

\textsuperscript{d}University of Sofia, Sofia, Bulgaria
minkom@fmi.uni-sofia.bg

\section*{Abstract}
Software product line modeling aims at capturing a set of software products in an economic yet meaningful way. We introduce a class of variability models that capture the sharing between the software artifacts forming the products of a software product line (SPL) in a hierarchical fashion, in terms of commonalities and orthogonalities. Such models are useful when analyzing and verifying all products of an SPL, since they provide a scheme for divide-and-conquer-style decomposition of the analysis or verification problem at hand. We define an abstract class of SPLs for which variability models can be constructed that are optimal w.r.t. the chosen representation of sharing. We show how the constructed models can be fed into a previously developed algorithmic technique for compositional verification of control-flow temporal safety properties, so that the properties to be verified are iteratively decomposed into simpler ones over orthogonal parts of the SPL, and are not re-verified over the shared parts. We provide tool support for our technique, and evaluate our tool on a small but realistic SPL of cash desks.

\textbf{Keywords:} Product families, Compositional verification, Model mining, Variability models, Model checking, Maximal models

\section{1. Introduction}

\textit{Software Product Lines.} System diversity is prevalent in modern software. In order to comply with the varying requirements of a potentially large number of customers, software systems often exist simultaneously in many different variants. \textit{Software product line} engineering aims at planning for and developing a family of system variants through managed reuse, in order to decrease time to market and improve software quality\cite{36}.

The variability of the different products in a software product line can be represented at different levels\cite{11}. \textit{Problem-space variability} describes product variation in terms of so-called features, that is user-visible product characteristics. The set of valid feature configurations defines the set of possible products. However, features are not necessarily related to the actual \textit{artifacts} that are used to realize the products. Problem-space variability based on features is at the requirements level, while \textit{solution-space variability} is at the design and implementation level. Solution-space variability describes product variation in terms of artifacts that are used to build the actual products of the product line.

In this paper, we aim to capture solution-space variability in terms of software artifacts that implement various functionalities. In the present context, an artifact is a software component at a suitable level of granularity, such as a Java method, a class, or a module.
Hierarchical Modeling. In order to describe the solution space variability in a software product line, we propose a hierarchical variability model, or HVM. Such a model represents, in a hierarchical manner, the artifacts that are common to all products, and the artifact variations that can occur between different products. On each hierarchical level, there is a common set of artifacts that represent parts shared by all products, while variation points represent parts that can vary from product to product. Every variation point is associated with a set of variants that represents choices for realizing the variation point in different ways. A variant is itself represented by a hierarchical variability model, potentially introducing a new level of hierarchy. A product described by a hierarchical variability model is obtained by selecting a variant at every variation point. The product line, or family, described by the model is the set of all its products.

Consider as an example a product line of a web-based social network application, shown graphically in Figure 1. This social network is to be used for audio or video sharing and communication between users. It provides basic user account support, content sharing facilities, and two communication environments, namely chat and email. The commonality of all social networks of the product line is that they all have user account support. This is modeled by the common artifact userAccount at the first level of hierarchy. The social networks, however, differ in the content they allow to share and the facilities they provide for communication: some allow only audio sharing, while others only allow video. In the model, this is represented by the variation point content (depicted as a diamond node) with the variants Audio and Video at the second level of hierarchy. Similarly, users of the social networks can either communicate via email or a chat system. Common for all social networks supporting chat is the text chat functionality, which only allows text exchange between users while at a third level of hierarchy, two alternative chat systems are realized, namely AudioChat and VideoChat. This hierarchical variability model gives rise to 6 products, corresponding to the 6 ways of resolving the variabilities.

Analysis and Verification of Software Product Lines. For any given program analysis, analyzing all products of a family individually may be infeasible for larger families. However, the number of products generated by a hierarchical variability model is at worst exponential in the size of the model; or equivalently, the model can be exponentially more succinct than the family. Exploiting the artifact commonalities at the different levels of hierarchy—as revealed by the model—is the key to achieving scalability of any analysis.
Factoring out common artifacts naturally reduces redundancy in the analysis: At variants with more than one variation point, the analysis problem is decomposed into simpler subproblems, as long as variation points at the same level of hierarchy expose orthogonality, while at variation points with more than one variant, the same problem is solved independently for each variant, as a case analysis, as long as variants at the same level of hierarchy expose alternative implementations. Thus, a hierarchical variability model can be viewed as a divide-and-conquer scheme for decomposing and splitting an analysis over a family of products.

In this paper, we develop the above idea by relativizing the correctness of the properties that are to hold for all products of a family on local specifications associated with the variation points. Thus, the number of verification tasks is reduced to the number of regions in the model (indicated by dotted lines in Figure 1), which is linear in its size rather than exponential. The associated overhead is that the designer has to provide specifications for the variation points. Here, we adapt for this scenario our previously developed compositional verification technique for temporal safety properties \[19\] and its automated tool support ProMoVer \[43\].

Model Mining. The above considerations lead to the natural problem of constructing a hierarchical variability model from an already realized software product line. The problem where a model is inferred from a set of programs is sometimes referred to as model mining.

In general, the HVMs giving rise to a particular software product line are not unique. We would like to measure how amenable a hierarchical variability model is to analysis by means of divide-and-conquer reasoning as suggested above. To this end we define a quality measure, called the separation degree of a model, as the ratio between the total number of artifacts from which products are constructed and the total number of artifact occurrences in the leaves of the model. High-quality models capture repetitions of products in a family without repetition in the model. The maximum theoretically possible separation degree of one is only reached in models where artifacts occur exactly once.

The problem then becomes to construct, from a given software product line, an HVM with maximum separation degree. We introduce a natural class of software product lines termed simple for which the optimal HVMs are unique and have separation degree one. We present a model mining transformation that constructs the unique optimal HVM from a given simple family.

Contributions. This paper combines and extends two of our earlier results: The hierarchical variability model for software product lines \[20\] and a technique for verification of families modeled in this way \[59\]. The combination essentially provides an efficient verification technique for simple families that have either been originally described in a modeling language that does not capture solution space variability, or families that have been produced in an ad hoc manner, for instance as a result of evolving and adapting a piece of software for different customers. For such families, the technique of the first paper allows the algorithmic extraction of a variability model, which is then used by the technique of the second paper to drive the verification of all products of the family. Thus, the main technical contributions of this paper are:

- A formal definition of simple hierarchical variability models (SHVM), together with a quality measure called separation degree and a set of well-formedness constraints yielding (by construction) models with maximal measure (Subsection 2.1).

- A formal semantics for hierarchical variability models in terms of family generation, and a proof that, for every well-formed variability model, the generated family is simple (Subsection 2.2).

- A characterization result stating that, for well-formed hierarchical variability models and simple families, family generation and hierarchical variability model construction are inverses of each other, thus implying correctness of model construction and uniqueness of well-formed models with respect to the families they generate (Subsection 2.2).

- A procedure to construct hierarchical variability models from simple families that produces well-formed models (Subsections 2.2 and 2.3).
• An adaptation of a previously developed compositional verification framework and its tool support, ProMoVer, for verifying control flow temporal safety properties of all products of simple families represented through (constructed) SHVMs (Section 3).

• Evaluation of the tools on a small but realistic case study (Section 4).

The proofs of all results presented in the paper can be found in the Appendix.

2. Hierarchical Variability Models

In this section, we present our variability models and their semantics, and relate them with families of products. We also illustrate our construction of variability models from families, by an example.

2.1. Families and Variability Models

Here, we first present product families as a semantic domain for our hierarchical variability models and then define formally these models.

We develop our formalization using a straightforward notation. However, the formalization can also be carried out in the terminology of relational algebra, or the one of regular languages. We choose a neutral notation here since our intended application domains are of a general nature.

Families. We consider products realized by a set of artifact implementations for a given set of artifact names. An artifact can be thought of, e.g., a component or a method. We fix a countably infinite set of artifact names $\text{Art} = \{a_1, a_2, a_3, \ldots\}$. A product $P$ is a finite set of artifact implementations, where for each artifact name there is at most one implementation. A family $F$ is a finite non-empty set of products.

Example 1. Here are some families that are used later to illustrate various notions.

\begin{align*}
F_A & = \{ \{a_1, b_1, c_1, d_1, e_1\}, \{a_1, b_1, c_1, d_1, e_2\}, \{a_1, b_1, c_1, d_2, e_1\}, \{a_1, b_1, c_2, d_1, e_1\}, \{a_1, b_1, c_2, d_2, e_2\}, \{a_1, b_2, c_1, d_1, e_1\}, \{a_1, b_2, c_2, d_1, e_2\}, \{a_2, b_1\} \} \\
F_B & = \{ \{a_1, b_1\}, \{a_1, b_2\} \}
\end{align*}

Next, we define two mappings for identifying the artifact names and artifact implementations that occur in a family.

Definition 2 (Family names and implementations). The mapping $\text{names}(F)$ from families to sets of artifact names and the mapping $\text{impls}(F)$ from families to sets of artifact implementations are defined as follows, where $a^1, \ldots, a^n \in \text{Art}$ and $i_1, \ldots, i_n \in \text{Nat}$:

\begin{align*}
\text{names}(F) & \equiv \bigcup_{P \in F} \text{names}(P) \\
\text{where names}(\{a_i^1, \ldots, a_i^n\}) & \equiv \{a^1, \ldots, a^n\} \\
\text{impls}(F) & \equiv \bigcup_{P \in F} \text{impls}(P) \\
\text{where impls}(\{a_i^1, \ldots, a_i^n\}) & \equiv \{a_i^1, \ldots, a_i^n\}
\end{align*}
In this definition we abuse notation by also defining mappings with the same names from products to the same co-domains.

We use two binary operations on families, the usual set union operation \( \cup \) and the product union operation \( \star \) over families with disjoint sets of artifact names defined by:

\[
F_1 \star F_2 \overset{\text{def}}{=} \{P_1 \cup P_2 \mid P_1 \in F_1 \land P_2 \in F_2\}
\]

and generalized through \( \prod_{i \in I} F_i \) to non-empty sets of families\(^1\). Intuitively, the product union of two families is the family having as products all possible combinations of products of the original families. Both operations are commutative and associative.

We now define a distinct class of families that we later relate to a specific class of hierarchical variability models. The class of families contains all single-product families consisting of a single artifact implementation, and is closed under product union of families over disjoint sets of artifact names, and under union of families over the same set of artifact names, but having disjoint implementations.

**Definition 3 (Simple family).** The class \( F \) of simple families is the least set of families closed under the formation rules:

\((F1)\) \( \{\{a\}_i\} \in F \) for any \( a \in \text{Art} \) and \( i \in \text{Nat} \).

\((F2)\) \( F_1 \star F_2 \in F \) for any \( F_1, F_2 \in F \) such that \( \text{names}(F_1) \cap \text{names}(F_2) = \emptyset \).

\((F3)\) \( F_1 \cup F_2 \in F \) for any \( F_1, F_2 \in F \) such that \( \text{names}(F_1) = \text{names}(F_2) \) and \( \text{impls}(F_1) \cap \text{impls}(F_2) = \emptyset \).

**Example 2.** The family \( \{\{a_1, b_1\}, \{a_1, b_2\}\} \) is simple, as it can be presented as \( \{\{a_1\}\} \star (\{\{b_1\}\} \cup \{\{b_2\}\}) \) which follows the above formation rules. Family \( F_A \) of Example [4] is also simple (as we shall see later in Example [4]), while family \( F_B \) of Example [4] is not: there is no way of building this family with the above formation rules.

Simplicity of families expresses that different functionalities in a product line are always orthogonal, and that alternative realizations of the same functionality have always disjoint implementations. These assumptions are rather heavy and may not always hold in practice. But only under such severe constraints can one hope for such a (strong) uniqueness result as the one obtained later (Section 2.1).

To characterize the applicability of the formation rules, we introduce the concept of correlation between artifact names as a restriction on the possible combinations of their implementations. If two artifact names are correlated, then not all possible combinations of artifact implementations occur in the family which means that the artifact implementations depend on each other.

Two distinct artifact names \( a, b \in \text{names}(F) \) are termed correlated in a family \( F \), denoted \( a C_F b \), if there are implementations \( a_i, b_j \in \text{impls}(F) \) such that no product in \( F \) contains both implementations simultaneously. Otherwise, names \( a \) and \( b \) are termed uncorrelated or orthogonal. The correlation relation \( C_F \) on \( \text{names}(F) \) is symmetric, and hence, its reflexive and transitive closure \( C^*_F \) is an equivalence relation. As usual, we denote the partitioning induced by \( C^*_F \) on \( \text{names}(F) \) by \( \text{names}(F)/C^*_F \) (quotient set).

**Example 3.** Consider family \( F_A \) of Example [4]. The only two correlated names are \( c \) and \( d \), evidenced by the lack of a product containing, for instance, \( c_1 \) and \( d_2 \). Thus, we have \( \text{names}(F_A)/C^*_F = \{\{a\}, \{b\}, \{c, d\}, \{e\}\} \).

Correlation (and orthogonality) extends naturally to products in a family: Products \( P \) and \( P' \) are correlated in \( F \) if some artifact name occurring in \( P \) is correlated to some artifact name occurring in \( P' \).

Similarly, we define the sharing relation \( N_F \) on \( F \) as \( P_1 N_F P_2 \overset{\text{def}}{=} P_1 \cap P_2 \neq \emptyset \), and use its reflexive and transitive closure \( N^*_F \) to partition the family \( F \).

The following result provides sufficient conditions for the applicability of the three formation rules for simple families from Definition 3. The proof of this proposition, as all other proofs can be found in the appendix. As usual, \( A \) denotes the complement of set \( A \) in a given universe of elements.

\(^1\)In relational algebra these are the usual union \( \cup \) and Cartesian product \( \times \) on relations with disjoint sets of attributes, a partial case of the more general join operation \( \# \).
Proposition 1. Let family $\mathcal{F}$ be simple. The following holds.

(i) Let $a_i \in \text{impls}(\mathcal{F})$, and let $\mathcal{F}'$ be the projection of $\mathcal{F}$ on names $(\mathcal{F}) \setminus \{a_i\}$. The name $a_i$ occurs in all products of $\mathcal{F}$, i.e., $a_i \in \bigcap_{P \in \mathcal{F}} P$, iff $\mathcal{F} = \{(a_i)\} \rtimes \mathcal{F}'$. Then either $\mathcal{F}' = 1\mathcal{F}$ and thus rule \((F1)\) applies, or else $\mathcal{F}'$ is simple and rule \((F2)\) applies.

(ii) Let $\{A_1, A_2\}$ be a non-trivial partitioning of names $(\mathcal{F})$, and let $\mathcal{F}_1$ and $\mathcal{F}_2$ be the projections of $\mathcal{F}$ on $A_1$ and $A_2$, respectively. Every name in $A_1$ is orthogonal to every name in $A_2$ in $\mathcal{F}$, i.e., $A_1 \times A_2 \subseteq \overline{\mathcal{F}}$, iff $\mathcal{F} = \mathcal{F}_1 \rtimes \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule \((F3)\) applies in this case.

(iii) Let $\{\mathcal{F}_1, \mathcal{F}_2\}$ be a non-trivial partitioning of $\mathcal{F}$. No product of $\mathcal{F}_1$ shares an artifact implementation with any product of $\mathcal{F}_2$, i.e., $\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \mathcal{N}_{\mathcal{F}}$, iff $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule \((F3)\) applies in this case.

The following important property of simple families follows from the above result: if a simple family $\mathcal{F}$ can be formed by formation rule \((F2)\) with some suitable $\mathcal{F}_1$ and $\mathcal{F}_2$ satisfying the rule’s condition, then it cannot be formed by formation rule \((F3)\), and vice versa.

Variability Models. In order to represent solution space variability of families in terms of shared artifact implementations, we consider simple hierarchical variability models.

Definition 4 (Simple hierarchical variability model). A simple hierarchical variability model (SHVM) $S$ is inductively defined as:

(i) a (possibly empty) common set of artifact implementations $M_C$, or

(ii) a pair $(M_C, \{VP_1, \ldots, VP_n\})$ where $M_C$ is defined as above and the set $\{VP_1, \ldots, VP_n\}$ of variation points is non-empty. A variation point $VP_i = \{S_{i,j} | 1 \leq j \leq k_i\}$, where $k_i \geq 2$, is a set of (at least two) SHVMs called variants.

We sometimes refer to an SHVM simply as a variability model. An SHVM consisting of only a common set of artifact implementations is called ground model. An SHVM generates a family $\mathcal{F}$ through all possible ways of resolving the variabilities of the SHVM. This process recursively selects exactly one variant for each variation point. We defer a formal definition of such a semantics for SHVMs to Section 2.2. Variability models can be naturally depicted as trees, where the leaves are common sets of artifact implementations, and the internal nodes are the roots of SHVMs or variation points.

Example 4. Figure 2 and Figure 3 show four variability models named $S_{A1}$, $S_{A2}$, $S_{B1}$, and $S_{B2}$. In these figures, (sub)trees showing variability models are rooted with boxes, and subtrees showing variation points are rooted with diamonds.

Analogously to Definition 2, we define two mappings for identifying the artifact names and artifact implementations that occur in SHVMs.

Definition 5 (SHVM names and implementations). The mapping names $(S)$ from SHVMs to sets of artifact names and the mapping impls$(S)$ from SHVMs to sets of artifact implementations are defined as
Figure 2: SHVMs $S_{A1}$ and $S_{A2}$ for the family $\mathcal{F}_A$ in Example 1.

Figure 3: SHVMs $S_{B1}$ and $S_{B2}$ for the family $\mathcal{F}_B$ in Example 1.
follows, where \(a_1, \ldots, a^n \in \text{Art} \) and \(i_1, \ldots, i_n \in \text{Nat}\):

\[
\text{names}(\{a_1, \ldots, a^n\}) \overset{\text{def}}{=} \{a_1, \ldots, a^n\}
\]

\[
\text{names}(\{M_C, \{VP_1, \ldots, VP_n\}\}) \overset{\text{def}}{=} \text{names}(M_C) \cup \bigcup_{1 \leq i \leq n} \text{names}(VP_i)
\]

where \(\text{names}(VP) \overset{\text{def}}{=} \bigcup_{S \in VP} \text{names}(S)\)

\[
\text{impls}(\{a_1, \ldots, a^n\}) \overset{\text{def}}{=} \{a_1, \ldots, a^n\}
\]

\[
\text{impls}(\{M_C, \{VP_1, \ldots, VP_n\}\}) \overset{\text{def}}{=} \text{impls}(M_C) \cup \bigcup_{1 \leq i \leq n} \text{impls}(VP_i)
\]

where \(\text{impls}(VP) \overset{\text{def}}{=} \bigcup_{S \in VP} \text{impls}(S)\)

Again we abuse notation by also defining mappings with the same names from variation points to the same co-domains.

Next we define a measure of the degree of separation in a variability model as the ratio between the cardinality of the set of artifact implementations and the sum of the cardinalities of the leaves of the SHVM tree. The separation degree is, thus, a number in the interval \((0, 1]\) that captures the degree to which orthogonal artifact implementations are delegated to separate variation points, and the extent to which disjointness of artifact implementations is delegated to separate variants. Since this is the original intention of variation points and variants in our model, separation degree is an obvious quality measure indicating how well the model is used for the purpose of hierarchically representing a software family.

The following definition provides a set of well-formedness constraints on SHVMs. Variability models satisfying these constraints always have separation degree one, as we show in Proposition 2.

**Definition 6 (Separation degree).** The separation degree \(sd(S)\) of a variability model \(S\) is defined as:

\[
sd(\{\}) \overset{\text{def}}{=} 1
\]

\[
sd(S) \overset{\text{def}}{=} \frac{|\text{impls}(S)|}{sd'(S)} \quad \text{if} \ S \neq \{\}
\]

where \(|S|\) denotes the cardinality of set \(S\), and \(sd'(S)\) is inductively defined as follows:

\[
\text{sd}'(M_C) \overset{\text{def}}{=} |M_C|
\]

\[
\text{sd}'(\{M_C, \{VP_1, \ldots, VP_n\}\}) \overset{\text{def}}{=} \text{sd}'(M_C) + \sum_{1 \leq i \leq n} \text{sd}'(VP_i)
\]

where \(\text{sd}'(VP_i) \overset{\text{def}}{=} \sum_{S \in VP} \text{sd}'(S)\)

Intuitively this definition captures the extent to which orthogonal artifact implementations are delegated to separate variation points, and the extent to which disjointness of artifact implementations is delegated to separate variants. Since this is the original intention of variation points and variants in our model, separation degree is an obvious quality measure indicating how well the model is used for the purpose of hierarchically representing a software family.

The following definition provides a set of well-formedness constraints on SHVMs. Variability models satisfying these constraints always have separation degree one, as we show in Proposition 2.

**Definition 7 (Well-formed variability model).** A ground variability model \(S = M_C\) is well-formed if constraint \([S1]\) below is satisfied. A variability model \(S = \{M_C, \{VP_1, \ldots, VP_n\}\}\) with variation points \(VP_i = \{S_{i,j} \mid 1 \leq j \leq k_i\}\) is well-formed if all variants \(S_{i,j}\) are well-formed, and furthermore, the following constraints are satisfied:

\((S1)\) \(M_C\) implements artifact names at most once.

\((S2)\) \(\text{names}(M_C) \cap \text{names}(VP_i) = \emptyset\) for all \(i\), and \(\text{names}(VP_{i_1}) \cap \text{names}(VP_{i_2}) = \emptyset\) whenever \(i_1 \neq i_2\).
(S3) names \((S_{i,j_1}) = names (S_{i,j_2})\) for all \(i, j_1, j_2\), and \(impls(S_{i,j_1}) \cap impls(S_{i,j_2}) = \emptyset\) whenever \(j_1 \neq j_2\).

Example 5. Consider the SHVMs \(S_{A_1}\) and \(S_{A_2}\) depicted in Figure 3. \(S_{A_1}\) is not well-formed whereas \(S_{A_2}\) is. The separation degrees are \(sd(S_{A_1}) = \frac{2}{9} = 0.3\) and \(sd(S_{A_2}) = \frac{9}{9} = 1\). Figure 3 depicts two other SHVMs, \(S_{B_1}\) and \(S_{B_2}\). Neither of these are well-formed and both have separation degree \(\frac{2}{9} = 0.8\).

The constraints in Definition 3 ensure that the separation degree of a well-formed SHVM is equal to 1 and is thus maximum.

Proposition 2. If variability model \(S\) is well-formed then \(sd(S) = 1\).

Note that the converse of Proposition 2 does not hold in general: The variability model \(\{a_1, a_2\}\) has separation degree 1, but well-formedness constraint \(\text{[ST]}\) is not satisfied.

Proposition 3. For a given SHVM, let AND and OR denote the maximum branching factors at SHVM and variation point nodes, respectively, and let ND be its nesting depth. The number of products generated by the SHVM is bound by \(\text{OR}^{\text{AND}(\text{AND}^{\text{ND}+1})-1}\) and is thus exponential in the size of the SHVM, which is bound by \(\frac{\text{AND} \cdot \text{OR}}{(\text{AND}^{\text{ND}} - 1)}\).

Inversely stated, SHVMs can be exponentially more succinct than the underlying family.

2.2. Relating Families and Variability Models

In this subsection we present translations between well-formed variability models and simple families and show that they are inverses of each other. In particular, this entails that the translation from simple families to variability models produces the unique well-formed model generating the respective family, thus giving a procedure for constructing a variability model from a given family.

From Variability Models to Families. The set of products generated by a ground model is the singleton set comprising the set of common artifact implementations and, thus, representing one product. The set of products generated by a variation point is the union of the product sets generated by its variants. Finally, the set of products generated by an SHVM with a non-empty set of variation points is the set of all products consisting of the common artifact implementations and of exactly one product from the set generated by each variation point.

Definition 8 (Family generation). The mapping family(\(S\)) from variability models to families is inductively defined as follows:

\[
\text{family}(M_C) \overset{\text{def}}{=} \{M_C\} \\
\text{family}((M_C, \{VP_1, \ldots, VP_n\})) \overset{\text{def}}{=} \{M_C\} \times \prod_{1 \leq i \leq n} \text{family}(VP_i) \\
\text{where family}(VP) \overset{\text{def}}{=} \bigcup_{S \in VP} \text{family}(S)
\]

We say that variability model \(S\) generates family(\(S\)).

Here we again abuse notation by also defining a mapping with the same name from variation points to the same co-domain. Family generation is well-defined in the sense that well-formed variability models generate simple families.

Proposition 4. If variability model \(S\) is well-formed, then family(\(S\)) is simple.

Example 6. SHVMs \(S_{A_1}\) and \(S_{A_2}\) in Figure 3 both generate family \(F_A\) in Example 2, implying that family \(F_A\) is simple since \(S_{A_2}\) is well-formed. SHVMs \(S_{B_1}\) and \(S_{B_2}\) in Figure 3 both generate family \(F_B\) in Example 4. Among these four SHVMs, \(S_{A_2}\), \(S_{B_1}\) and \(S_{B_2}\) have maximum separation degree in the sense that, for each of the families \(F_A\) and \(F_B\), no other SHVMs for the same family have higher separation degree.
From Families to Variability Models. We now present a reverse transformation from simple families to well-formed variability models. Recall that simple families have unique formation trees modulo commutativity and associativity of the two operations. Well-formed SHVMs can thus be seen as a uniform way of grouping the formation terms. Every family $F$ can be decomposed into the form:

$$F = \{P\} \times F_V, \quad F_V = \prod_{1 \leq i \leq n} F_i, \quad F_i = \bigcup_{1 \leq j \leq k_i} F_{i,j}$$

where $P$ is a product, or equivalently, as a single equation:

$$F = \{P\} \times \prod_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} F_{i,j} \quad (\ast)$$

The existence of the decomposition is ensured since every family $F$ can be trivially decomposed as $\{\emptyset\} \times \bigcup F$, i.e., with product $P$ being empty and $n = k_1 = 1$. Decomposition $(\ast)$ is only unique under additional constraints, under which the decomposition is called canonical.

Definition 9 (Canonical form of family). A family $F$, decomposed as equation $(\ast)$ above, is in canonical form if the following conditions hold:

(C1) The product $P$ is the set of artifact implementations that are common to all products in $F$.

(C2) The set of artifact names in $F_V$ has $n$ equivalence classes w.r.t. correlated artifact names $C_{F_V}$, and for the $i$-th equivalence class, the family $F_i$ is the projection of $F_V$ onto the artifact names of the class.

(C3) For all $i, 1 \leq i \leq n$, $F_{i,j}$ are the $k_i$ equivalence classes of $F_i$ w.r.t. implementation sharing $N_{F_i}$.

A consequence of the following proposition is that definitions and proofs may exploit the canonical form to proceed by induction on the size of simple families.

Proposition 5. If $F$ is a simple non-core family in canonical form then for all $i, 1 \leq i \leq n$, and $k_i \geq 2$ all $F_{i,j}$ are simple and of strictly smaller size than $F$.

The decomposition into canonical form is clearly unique for a simple family, and exposes one level of hierarchy. Thus, by iterative application of the decomposition, we obtain a mapping from families to hierarchical variability models.

Definition 10 (Variability model generation). The mapping $\text{shvm}(F)$ from simple families presented in canonical form to variability models is inductively defined as follows:

$$\text{shvm}(\{P\}) \overset{\text{def}}{=} P$$

$$\text{shvm}(\{P\} \times \prod_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} F_{i,j}) \overset{\text{def}}{=} (P, \{VP_1, \ldots, VP_n\})$$

where $VP_i \overset{\text{def}}{=} \{\text{shvm}(F_{i,j})\}_{1 \leq j \leq k_i}$

We say that family $F$ generates variability model $\text{shvm}(F)$.

Proposition 5 guarantees that the above mapping is well-defined, in the sense that $\text{shvm}(F)$ is indeed an SHVM. Furthermore, as the next result shows, the generated variability model is well-formed.

Proposition 6. If family $F$ is simple, then $\text{shvm}(F)$ is well-formed.

Example 7. Consider the family $F_A$ from Example 4.

- In the first step of the decomposition of $F_A$ into canonical form we obtain the common set $P = \{a_1, b_1\}$ and the family $F_V = \{(c_1, d_1, e_1), (c_1, d_1, e_2), (c_2, d_2, e_1), (c_2, d_2, e_2), (c_2, d_3, e_1), (c_2, d_3, e_2)\}$.

- In the next step, we analyze $F_V$ to find that only artifact names $c$ and $d$ are correlated. Projecting $F_V$ onto the two resulting equivalence classes $\{c, d\}$ and $\{e\}$ we obtain the two variation points $F_1 = \{(c_1, d_1), (c_2, d_2), (c_2, d_3)\}$ and $F_2 = \{(e_1), (e_2)\}$.
In the third step, we analyze $F_1$ and see that two products share the artifact implementation $c_2$, which gives us the variants $F_{1,1} = \{(c_1, d_1)\}$ and $F_{1,2} = \{(c_2, d_2), (c_2, d_3)\}$, and then analyze $F_2$ to obtain the variants $F_{2,1} = \{(e_1)\}$ and $F_{2,2} = \{(e_2)\}$.

Only $F_{1,2}$ is not a ground model. Applying the above steps decomposes it into a common set $\{c_2\}$ and a single variation point with two variants consisting of the common sets $\{d_2\}$ and $\{d_3\}$. It is easy to see that $\text{shvm}(FA)$ is the variability model $SA_2$ in Figure 2.

**Characterization Results.** Our first result establishes correctness of model extraction.

**Lemma 1.** For every simple family $F$ we have:

$$\text{family}(\text{shvm}(F)) = F$$

The second result establishes uniqueness of well-formed models w.r.t. the generated (simple) family.

**Lemma 2.** For every well-formed variability model $S$ we have:

$$\text{shvm}(\text{family}(S)) = S$$

An immediate consequence of the above two lemmas is our main characterization result, which essentially states that the two transformations relating variability models and families are inverses of each other.

**Theorem 1 (Characterization Theorem).** For every simple family $F$ and every well-formed variability model $S$ we have:

$$\text{family}(S) = F \iff \text{shvm}(F) = S$$

### 2.3. Model Extraction from Code

Here we explain, using an example, how to extract variability models from program code of simple product families. The example is written in Java, but our method is independent of the programming language.

**Example 8.** As a running example in the rest of this paper, we consider a product line of cash desks that is a simplified version of a case study from the HATS project [27]. A cash desk processes purchases by retrieving the prices for all items to be purchased and calculates the total price. After the customer has paid, a receipt is printed and the stock is updated. All cash desks have in common that every purchase is processed following the same process. However, the cash desks differ in how items are entered. Some cash desks allow entering products using a keyboard, others only provide a scanner, and a third group provides both options. Payment at some cash desks can only be made in cash. Other cash desks only accept credit cards, while a third group allows the choice between cash and credit card payment.

Figure 4 shows two of nine products from the product line where each product takes the form of a Java class called CashDesk. At the top is product code for a cash desk for entering with keyboard and paying with credit card only. At the bottom of the figure is product code for a cash desk that scans products and accepts cash payment only. These nine Java classes can be converted into a family of products in the sense of Definition 1 by considering public method names as artifact names and the corresponding method bodies as artifact implementations. This yields the following simple family:

$$F_{\text{CashDesk}} = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$$

where:

$$P_1 = \{\text{saleCashDesk}, \text{enterProd}^{\text{Keyboard}}, \text{payment}^{\text{Cash}}\}$$

$$P_2 = \{\text{saleCashDesk}, \text{enterProd}^{\text{Keyboard}}, \text{payment}^{\text{Card}}\}$$

$$P_3 = \{\text{saleCashDesk}, \text{enterProd}^{\text{Keyboard}}, \text{payment}^{\text{CashOrCard}}\}$$

$$P_4 = \{\text{saleCashDesk}, \text{enterProd}^{\text{Scanner}}, \text{payment}^{\text{Cash}}\}$$

$$P_5 = \{\text{saleCashDesk}, \text{enterProd}^{\text{Scanner}}, \text{payment}^{\text{Card}}\}$$

11
public class CashDesk {
    public void sale() {
        int prodNu = 10;
        for (int i = 0; i < 10; i++) {
            int prod = enterProd();
            writeReceipt(prod);
            prodNu = updateStock(prodNu);
            payment();
        }
    }
    public int enterProd() {
        return useScanner();
    }
    public void payment() {
        cashPay();
    }
    public static void main(String[] args) {
        (new CashDesk()).sale();
    }
}

The common purchase process of all cash desks is modeled by the artifact name sale and implementation (subscript) CashDesk. The artifact names enterProd and payment are common to all products, but their implementations vary: Cash, Card, or CashOrCard. Starting from family $F_{\text{CashDesk}}$, and following steps similar to those of Example 7, gives the following SHVM.

$$
\text{shvm}(\text{CashDesk}) = \{ \text{sale}_{\text{CashDesk}}, \text{enterProd}_{\text{Scanner}}, \text{payment}_{\text{CashOrCard}} \}
$$

$$
P_0 = \{ \text{sale}_{\text{CashDesk}}, \text{enterProd}_{\text{Scanner}}, \text{payment}_{\text{CashOrCard}} \}
$$

$$
P_1 = \{ \text{sale}_{\text{CashDesk}}, \text{enterProd}_{\text{KeyboardOrScanner}}, \text{payment}_{\text{Cash}} \}
$$

$$
P_2 = \{ \text{sale}_{\text{CashDesk}}, \text{enterProd}_{\text{KeyboardOrScanner}}, \text{payment}_{\text{Card}} \}
$$

$$
P_3 = \{ \text{sale}_{\text{CashDesk}}, \text{enterProd}_{\text{KeyboardOrScanner}}, \text{payment}_{\text{CashOrCard}} \}
$$

The two variation points $@\text{EnterProducts}$ and $@\text{Payment}$ represent the variabilities of the cash desks. Variation point $@\text{EnterProducts}$ has associated variants Keyboard, Scanner and KeyboardOrScanner, while variation point $@\text{Payment}$ has associated variants Cash, Card and CashOrCard. Figure 5 shows the model as a diagram.

As we describe in Section 4.1, the extraction of SHVM models from an existing simple family of products (explained by the above example) is implemented as a part of our tool support. These models can be used...
for hierarchical analyses of product families. In the next section, we show how they facilitate efficient verification of temporal safety properties.

3. Verification of Temporal Safety Properties of Software Product Lines

Suppose we have a large software family that has either been produced in an ad hoc manner (for instance as a result of evolving and adapting a software product for different customers) or that has been developed by some methodology that does not capture solution space variability. Suppose also that we want to apply some given standard static program analysis technique, such as formal verification, on the implementations (i.e., the code) of all products of the family. Naturally in such a case we should strive to minimize the overall effort by maximizing the reuse of partial verification results obtained for the shared artifacts. In the previous section we developed a technique to extract automatically SHVMs from the implementations of simple families. Since the extracted SHVMs capture the sharing of artifacts in the solution space, they contain, in a succinct representation (see Proposition 3), precisely the information that is needed to maximize the reuse of analysis results.

The exact way of utilizing SHVMs for software product line verification depends heavily on the concrete verification technique at hand. Especially suited for the task are compositional techniques, since they reduce the verification of whole products to the individual verification of their components (i.e., artifacts), and thus allow the reuse of the latter in case they are shared. In this paper, we illustrate this idea by adapting a previously developed compositional verification technique for temporal safety properties [19] and its automated tool support PROMOVER [43], to the setting of software product lines. Let us first explain intuitively our original compositional verification framework, and then describe its adaptation for product families.

Our original framework for compositional verification is a realization of assume-guarantee reasoning for the verification of incomplete programs, i.e., programs where the implementation of some of their components are not available. Hence, such programs consist of so-called concrete components available through their implementations and of unavailable abstract components. To verify incomplete programs, we require a user provided local specification for each abstract component that describes its legal behavior (assumption).
Our verification framework relativizes the correctness of global properties of such programs on the local specifications of their abstract components and the implementation of the concrete ones, thus dividing the verification task into the following two independent subtasks:

(a) a check that the composition of the local specifications of abstract components together with the implementation of concrete ones entails the global property, and

(b) a check that the implementation of each abstract component (once it becomes available) satisfies its local specification.

Technically, for subtask (b) a control flow graph is extracted from the code of each abstract component (once it becomes available), and is model checked against its local specification. A control flow graph, here called flow graph, is a collection of method graphs, each representing the control flow structure of the code of a method (see Definition 12 and Example 9). For subtask (a), however, so-called maximal flow graphs are constructed from the local specifications of abstract components. Intuitively, a maximal flow graph for a local specification \( \phi \) is the most general flow graph satisfying \( \phi \). Thus it can be used, for the purposes of verification, as a representation of any implementation of the component that satisfies \( \phi \). These maximal models are composed with the flow graphs extracted from the code of concrete ones, and then the behavior of the result represented as a pushdown automaton is model checked against the global property of the program.

To adapt our framework to the verification of temporal safety properties of SHVMs, we require user provided local properties at all variation points. These properties should abstractly express the legal behavior of all their underlying variants (see Example 11 for concrete properties). The idea is that for the verification of variants their underlying variation points and core methods (i.e., their children variation point and core nodes in the graph) can be viewed as abstract and concrete components, respectively. Then the verification of the variants is relativized on the properties of their underlying variation points, while the correctness of the variation points is established through verifying their underlying variants (i.e., their children variant nodes in the graph). This results in a hierarchical verification scheme that is realized by the following two steps:

1. Verify each variation point by checking, using step (2), that all its underlying variants satisfy its specification. This essentially means that underlying variants attached to a variation point inherit the property of their parent variation point.

2. Verify each variant by checking that the composition of maximal flow graphs constructed from the local specifications of its underlying variation points, together with the flow graphs extracted from its core methods, satisfy the property of the variant. By this, we basically verify that all sub-products constructed by composing the different artifact implementations below a variant satisfy its property.

Since the family corresponds to the root variant, the global property of the software family is the property of the top-level variant of its SHVM.

As we show in Section 3.2, this verification procedure is sound: If it succeeds for SHVM \( S \) and global property \( \phi \), then all products of \( S \) satisfy \( \phi \).

For example, to verify the CashDesk product line modeled by the SHVM in Figure 5, the variation points @EnterProducts and @Payment are locally specified, and the desired global property of all products would be the property of variant CashDesk. Then the verification procedure follows the steps below:

1. Verify that each individual variation point satisfies its property independently. This is achieved for instance for variation point @EnterProducts by independently checking that the variants Keyboard, Scanner, and KeyboardOrScanner satisfy the local specification of @EnterProducts.

2. Construct maximal flow graph for the variation points @EnterProducts and @Payment, compose these with the flow graphs extracted from the core method sale, and model check the result against the property of CashDesk.

In the remainder of this section, we first present our compositional verification framework formally, and then describe how it is adapted to the verification of software families represented by SHVMs.
3.1. A Framework for Compositional Verification

Here, we define our program models and specification language and present our compositional verification principle.

Program Model. In order to reason algorithmically about sequences of method invocations, we abstract the set of methods defining our program by ignoring all data. An initialized model serves as an abstract representation of a program’s structure and behavior.

Definition 11 (Model). A model is a (Kripke) structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ where $S$ is a set of states, $L$ a set of labels, $\rightarrow \subseteq S \times L \times S$ a labeled transition relation, $A$ a set of atomic propositions, and $\lambda : S \rightarrow \mathcal{P}(A)$ a valuation, assigning to each state $s$ the set of atomic propositions that hold in $s$. An initialized model is a pair $(\mathcal{M}, E)$ with $\mathcal{M}$ a model and $E \subseteq S$ a set of initial states.

A method graph is an instance of an initialized model which is obtained by ignoring all data from a method implementation. A flow graph is a collection of method graphs, one for each method of the program. It is a standard model for the analysis of control flow based properties [6].

Definition 12 (Method graph). Let Meth be a countably infinite set of methods names. A method graph for method $m \in$ Meth over a set of method names $M \subseteq$ Meth is an initialized model $(\mathcal{M}_m, E_m)$ where $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ is a finite model and $E_m \subseteq V_m$ is a non-empty set of entry points of $m$. $V_m$ is the set of control nodes of $m$, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, and $\lambda_m : V_m \rightarrow \mathcal{P}(A_m)$ so that $m \in \lambda_m(v)$ for all $v \in V_m$ (i.e., each node is tagged with its method name). The nodes $v \in V_m$ with $r \in \lambda_m(v)$ are return points.

Note that according to the above definition, methods can have multiple entry points. Flow graphs that are extracted from a program source have single entry points, but the maximal models that we generate for compositional verification can have multiple entry points.

Every flow graph $\mathcal{G}$ is equipped with an interface $I = (I^+, I^-)$, denoted $\mathcal{G} : I$, where $I^+ \subseteq$ Meth are the provided and externally required methods, respectively. Interfaces are needed when constructing maximal flow graphs. A flow graph is closed if its interface does not require any methods (i.e., $I^- = \emptyset$) and it is open otherwise. Flow graph composition is defined as the disjoint union $\uplus$ of their method graphs.

Example 9. Figure 6 shows a simple Java class and the (simplified) flow graph it induces. It consists of two method graphs, for method even and method odd, respectively. Entry nodes are depicted as usual by incoming edges without source. Its interface is $\{(\text{even, odd}). \emptyset\}$, thus the flow graph is closed.
The operational semantics of flow graphs, here called flow graph behavior, is also defined as an instance of an initialized model, induced through the flow graph structure. We use transition label \( \tau \) for internal transfer of control, \( m_1 \text{call}\ m_2 \) for the invocation of method \( m_2 \) by method \( m_1 \) when method \( m_2 \) is provided by the program and \( m_1 \text{ret}\ m_1 \) when method \( m_2 \) is external (e.g., API methods), and \( m_2 \text{ret}\ m_1 \) respectively for the corresponding return from the call.

**Definition 13 (Flow Graph Behavior).** Let \( \mathcal{G} = (\mathcal{M}, E) : (I^+, I^-) \) be a flow graph such that \( \mathcal{M} = (V, L, \rightarrow, A, \lambda) \). The behavior of \( \mathcal{G} \) is defined as an initialized model \( b(\mathcal{G}) = (\mathcal{M}_b, E_b) \), where \( \mathcal{M}_b = (\mathcal{S}_b, L_b, \rightarrow_b, A_b, \lambda_b) \), such that \( \mathcal{S}_b = (V \cup I^-) \times V^* \), i.e., states are pairs of control points \( v \) or required for the invocation of method \( m_2 \) | \( m_1 \in I^+, m_2 \notin I^+ \) \} \cup \{ m_2 \text{ret}\ m_1 | m_1 \in I^+, m_2 \notin I^+ \} \cup \{ \tau \}, A_b = A, \lambda_b((v, \sigma)) = \lambda(v) \) and \( \lambda_b((m, \sigma)) = m \), and \( \rightarrow_b \subseteq \mathcal{S}_b \times L_b \times \mathcal{S}_b \) is defined by the following rules:

1. \( \text{transfer} \) \( \left( v, \sigma \right) \xrightarrow{\tau} \left( v', \sigma \right) \) if \( m \in I^+, v \xrightarrow{\tau} m \cdot v', v \models \neg \rho \)
2. \( \text{call} \) \( \left( v, \sigma \right) m_1 \text{call}\ m_2 \xrightarrow{m_1 \cdot m_2}(v_1, \sigma) \) if \( m_1, m_2 \in I^+, v_1 \xrightarrow{m_2} m_1 \cdot v_1, v_1 \models \neg \rho, v_2 \models m_2 \cdot v_2 \in E \)
3. \( \text{ret} \) \( \left( v_2, v_1 \cdot \sigma \right) m_2 \text{ret}\ m_1 \xrightarrow{m_2 \cdot m_1}(v_1, \sigma) \) if \( m_1, m_2 \in I^+, v_2 \models m_2 \land \rho, v_1 \models m_1 \)
4. \( \text{call}! \) \( \left( v, \sigma \right) m_1 \text{call}\ m_2 \xrightarrow{m_1 \cdot m_2}(v_1, \sigma) \) if \( m_1 \in I^+, m_2 \in I^+, v_1 \xrightarrow{m_2} m_1 \cdot v_1, v_1 \models \neg \rho \)
5. \( \text{ret}? \) \( \left( m_2, v_1 \cdot \sigma \right) m_2 \text{ret}\ m_1 \xrightarrow{m_2 \cdot m_1}(v_1, \sigma) \) if \( m_1 \in I^+, m_2 \in I^+, v_1 \models m_1 \)

The set of initial states is defined by \( E_b = E \times \{ \varepsilon \} \), where \( \varepsilon \) denotes the empty sequence over \( V \cup I^- \).

Notice that return transitions always hand back control to the caller of the method. Calls to external methods are modeled with an intermediate state, from which only an immediate return is possible. In this way possible callbacks from external methods are not captured in the behavior. This simplification is justified, since we abstract away from data in the model and the behavior is thus context-free, but has to be kept in mind when writing specifications; in particular one cannot specify that callbacks are not allowed.

**Example 10.** Consider the flow graph of Example 7. One example run through its (branching, infinite-state) behavior, from an initial to a final configuration, is:

\[
\left( v_0, \varepsilon \right) \xrightarrow{\tau} \left( v_1, \varepsilon \right) \xrightarrow{\text{even call odd}} \left( v_2, \varepsilon \right) \xrightarrow{\text{odd ret even}} \left( v_3, \varepsilon \right)
\]

Now, consider just the method graph of method \( \text{even} \) as an open flow graph, having interface \( \{ \text{even}, \{ \text{odd} \} \} \). The local contribution of method \( \text{even} \) to the above global behavior is the following run:

\[
\left( v_0, \varepsilon \right) \xrightarrow{\tau} \left( v_1, \varepsilon \right) \xrightarrow{\text{even call odd}} \left( v_2, \varepsilon \right) \xrightarrow{\text{odd ret even}} \left( v_3, \varepsilon \right)
\]

An alternative way to express flow graph behavior is by means of pushdown systems (PDS). We exploit this by using pushdown system model checking to verify behavioral properties [31].

**Specification Language.** To specify global and local properties we have use the safety fragment of linear temporal logic (LTL) that uses the weak version of until.

**Definition 14 (Safety LTL).** The formulas of sLTL are inductively defined by:

\[
\phi ::= p \mid \neg p \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid X \phi \mid G \phi \mid \phi_1 W \phi_2
\]

\[\text{The theoretical underpinnings of our compositional verification framework are actually based on a slightly more expressive specification language, namely simulation logic, the fragment of the modal \( \mu \)-calculus [23] with boxes and greatest fixed-points only. For details see again our previous work [19].}\]
where \( p \in A_b \) denotes the set of atomic propositions.

*Satisfaction* on states \((M_b, s) \models \phi\) is defined in the standard fashion [14] as validity of \( \phi \) over all runs starting from state \( s \in S_b \) in model \( M_b \). For instance, formula \( \Sigma \phi \) holds of state \( s \) in model \( M_b \) if \( \phi \) holds in the second state of every run starting from \( s \), while \( \phi \mathbin{\mathcal{W}} \psi \) holds in \( s \) if for every run starting in \( s \), either \( \phi \) holds in all states of the run, or \( \psi \) holds in some state of the run and \( \phi \) holds in all previous states. Satisfaction of a formula \( \phi \) in flow graph \( G \) with behavior \( b(G) = (M_b, E_b) \) is defined as satisfaction of \( \phi \) on all initial states \( s \in E_b \).

Satisfaction is generalized to product lines in the obvious way: A product line described by a variability model \( S \) satisfies a formula \( \phi \) if the behavior \( b(G_p) \) of the flow graph \( G_p \) of every product \( p \in \text{products}(S) \) satisfies \( \phi \).

**Compositional Verification.** As mentioned, our method for compositional verification is based on the construction of maximal flow graphs for properties of sets of methods. For a given property \( \psi \) and interface \( I \) consisting of provided and required methods, consider the class of all flow graphs with interface \( I \) satisfying \( \psi \). A maximal flow graph for \( \psi \) and \( I \) is a flow graph \( \text{Max}(\psi, I) \) that satisfies exactly those properties that hold for all members of the class. Thus, the maximal flow graph can be used as a representative of the class for the purpose of checking properties. Using maximal models for compositional verification was first proposed by Grumberg and Long [17] for finite-state systems, and was generalized for flow graphs by Gurov and others in [15] [18].

Suppose a system with \( n \) components that are partitioned into two sets: The set of abstract components \( G_1, \ldots, G_k \) specified with their local properties and interfaces \((\psi_1, I_1), \ldots, (\psi_k, I_k)\), and the set of concrete components \( G_{k+1}, \ldots, G_n \). The main principle of compositional verification based on maximal flow graphs, can relativize the global correctness of such systems on the local specifications \((\psi_1, I_1), \ldots, (\psi_k, I_k)\), by the proof rule presented below.

\[
G_1 \models \psi_1 \cdots G_k \models \psi_k \quad \bigcup_{j=k+1, \ldots, n} G_j \equiv \bigcup_{i=1, \ldots, k} \text{Max}(\psi_i, I_i) \models \phi 
\]

\[
\bigcup_{i=1, \ldots, n} G_i \models \phi
\]

The principle states that the composition of \( n \) components (here a set of methods), in which \( k \) of them are specified by their local specifications, satisfies global property \( \phi \) if (i) each specified (abstract) component \( G_i \) satisfies its respective local property \( \psi_i \) and (ii) the composition of the \( k \) maximal flow graphs \( \text{Max}(\psi_i, I_i) \) with the flow graphs extracted from the code of the other components (concrete components) \( G_{k+1}, \ldots, G_n \) satisfies \( \phi \).

As we proved previously [19], the rule is sound and complete when interfaces describe all provided and required methods\(^3\).

3.2. SHVM-driven Algorithmic Verification

For efficient verification of product families represented by SHVMs, we introduce the notion of *regions* in SHVMs, each of which is formed by an SHVM node (variant) and its underlying variation points and artifacts implementations, e.g., regions of the SHVM in Figure 1 are indicated by dotted lines. In this section, we propose a compositional reasoning approach that is linear in the number of regions in the SHVM description of the product line rather than linear in the number of generated products (which is exponential in the number of regions). This approach is an instantiation of the compositional verification principle presented above to SHVMs.

To show that all products generated from an SHVM satisfy global property \( \Phi \), the top-level region of the SHVM is specified with \( \Phi \), and also every variation point \( \text{VP} \) of the SHVM is specified by a behavioral

\(^3\)Our proof [19] is for global properties \( \phi \) written in behavioral simulation logic and local properties \( \psi_i \) in structural simulation logic; here in the context of sLTL we use translations into the respective logic.
property $\psi_{VP}$ and its interface $I_{VP} = (I_{VP}^+, I_{VP}^-)$ declaring the names of the provided and required methods. The underlying variants attached to a variation point inherit the corresponding variation point specification. Then, our verification procedure for SHVMs is as follows.

**Verification Procedure.** For every region $M = (M_C, \{VP_1, \ldots, VP_n\})$ of the SHVM with the property $\phi$, perform the following two tasks:

(i) For every artifact name $a \in Art(M_C)$, extract the flow graph $G_a$ from $Imp(a)$.

(ii) For all variation points $VP_i$ with specification $(\psi_{VP_i}, I_{VP_i})$, construct the maximal flow graph $Max(\psi_{VP_i}, I_{VP_i})$.

Then, compose the constructed graphs with the flow graphs of task (i), and model check the resulting flow graph against the region property $\phi$, i.e.,

$$\biguplus_{a \in Art(M_C)} G_a \uplus \biguplus_{1 \leq i \leq n} Max(\psi_{VP_i}, I_{VP_i}) \models \phi \quad (2)$$

For properties given in sLTL, the behavior of $G_{Max}$ is represented as a PDS and standard PDS model checking is used.

The presented verification procedure is sound, as established by the following theorem.

**Theorem 2.** Let $S$ be an SHVM with global property $\phi$. If the verification procedure succeeds for $S$, then $p \models \phi$ for all its products $p \in \text{products}(S)$.

The total number of verification tasks needed to establish the global product line property is, thus, equal to the number of regions, since we have to complete one verification task per region. In contrast, the number of products is exponential in the number of regions.

**Example 11.** To illustrate our compositional verification approach, we use the cash desk product line described in Example 8. The global behavioral property we want to verify is informally stated as follows:

The entering of products has to be finished before the payment process has started.

Taking into account the distribution of functionality to artifact intended by the variability model from the example, the specification can be approximated as:

If control starts in method $sale$, it cannot reach method $payment$ before it has already been in method $\text{enterProd}$ and then back in $sale$.

In terms of the (global) behavior of the flow graphs of the products induced by the product line, this property can be formalized in sLTL as follows:

$$\varphi_{CD} = sale \rightarrow (\neg payment \mathcal{W} (\text{enterProd} \land r \land X sale))$$

where the subformula $\text{enterProd} \land r \land X sale$ captures a return from $\text{enterProd}$ to $sale$.

First, we have to specify all variation points of the cash desk SHVM. The specification of the $@\text{EnterProd}$ and $@\text{Payment}$ variation points are as follows:

- The interface of variation point $@\text{EnterProducts}$ is $I_{EP} = (\{\text{enterProd}\}, \{\text{payment}\})$. The property required for the variation point is that the $\text{enterProd}$ method never makes calls to $\text{payment}$ method. Formally, this property can be expressed by the formula$^4$:

$$\varphi_{EP} = G \neg payment$$

$^4$This and the following property would trivialize if we specified the set of required methods to be empty. For now, however, our tool does not check interfaces.
The interface of variation point @Payment is \( I_P = (\{\text{payment}\}, \{\text{enterProd}\}) \). Similarly to the variation point above, the property required for this variation point is that the payment method never makes calls to the enterProd method:

\[ \varphi_P = G \neg \text{enterProd} \]

The variants Keyboard, Scanner, and KeyboardOrScanner inherit their specifications from the @EnterProducts variation point, and the variants Cash, Card and CashOrCard from the @Payment variation point.

Finally, we have to establish that all regions satisfy their respective property. For the top-level region, we construct the maximal flow graphs for the specifications of the variation points @EnterProducts and @Payment and compose these with the flow graph of method sale, and model check \( \varphi_{CD} \) against the composition result. Then variants Keyboard, Scanner, KeyboardOrScanner, Cash, Card and CashOrCard are verified also by model checking the flow graph extracted from their implementation against their inherited verification point property.

4. Tool Support and Evaluation

Our tool support for the verification of product families consists of two tools: A tool that constructs SHVMs from families, and another one that automatically verifies temporal properties of SHVMs. Using these tools, we verify a simple family in two steps; first we construct the SHVM representation of the family and then we verify temporal safety properties of the constructed SHVM.

4.1. Construction of Simple Hierarchical Variability Models

We have implemented an algorithm that takes as input a simple family and produces its SHVM decomposition. The algorithm is not written explicitly in this paper but can be unambiguously inferred from Definition 9 and Definition 10. Our implementation is written in OCaml. Its input is a text file containing the products of the family. The constructed family is a list of sets, each set representing one product. The sets’ elements are records, each record having two fields: name and number. Each record represents an implementation, the name being the name of the artifact and the number, the corresponding index. Having constructed the family \( F \) we proceed as dictated by Definition 10. First we factor out the common implementations, if any, and then we proceed with the remainder \( F_V \). We identify the equivalence classes of the \( C_{F_V} \) relation using Union-Find structures. For each equivalence class \( F_i \) we identify the equivalence classes of the \( N_{F_i} \) relation. If there are no common implementations and each of the two equivalence relations has a single equivalence class, by Proposition 5 the family \( F \) is not simple and the program exits with an appropriate message. Otherwise, recursive calls are made on each of the equivalence classes of the \( N_{F_i} \) relation. A very crude upper bound on the running time is \( O(n^4) \), \( n \) being the size of the family.

4.2. Automated Modular Verification of SHVMs

ProMoVer [43] is a fully automated tool for the procedure-modular verification of control flow temporal safety properties of Java programs\(^5\). It supports compositional verification by relativizing the correctness of a global program property on properties of individual methods and their interfaces. All interfaces, variation points and global properties are provided to the tool as assertions in the form of program annotations. ProMoVer accepts a JML-like syntax for annotations (cf. [27]) as special comments called pragmas. For scalability, ProMoVer provides a proof storage and reuse mechanism which stores flow graphs, maximal models and model checking results and reuses these the next time the same program is verified. To reuse the stored information, ProMoVer checks for each method of the program: if the source code of the method has not changed, the stored flow graph of the method is used, if a local specification has not changed the stored maximal model for the specification is used. Further, it provides users with a library of global properties which contains platform as well as application specific properties. For details about ProMoVer, the reader is referred to [43].

\(^5\)ProMoVer is available via the web interface [www.csc.kth.se/~siavashs/ProMoVer](http://www.csc.kth.se/~siavashs/ProMoVer)
/**
 * @variation_point: EnterProd
 * @variation_point_interface: provided enterProd
 * @variation_point_ltl_prop: G ! payment
 * @variants: Keyboard, Scanner, KeyboardOrScanner
 */

/** @variant: Keyboard
 * @variant_interface: provided enterProd()
 * @variation_points:
 */

public int enterProd() {
    ...
}

Figure 7: Annotations for variation point @EnterProd and its variant Keyboard

<table>
<thead>
<tr>
<th>Product Line</th>
<th>Depth</th>
<th># Modules</th>
<th># Products</th>
<th>t_ind[s]</th>
<th>t_comp[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>79</td>
<td>9</td>
</tr>
<tr>
<td>CD/CH</td>
<td>1</td>
<td>9</td>
<td>18</td>
<td>177</td>
<td>10</td>
</tr>
<tr>
<td>CD/CT</td>
<td>2</td>
<td>15</td>
<td>27</td>
<td>278</td>
<td>11</td>
</tr>
<tr>
<td>CD/CH/CT</td>
<td>2</td>
<td>17</td>
<td>54</td>
<td>652</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 1: Evaluation Results

We have adapted ProMoVer for verifying properties of SHVMs according to the compositionality principle described in Section 3.2. For this adaptation, we have extended the annotation language to support the definition of variants and variation points and the associated specifications by designated pragmas. The tool takes as input a source code file in which the SHVM to be analyzed is represented by annotations. The product property and the variation point properties are also provided by annotations. Figure 7 shows in the left column the annotation for the @EnterProd variation point, while the annotation for its Keyboard variant is shown in the right column. ProMoVer fully automatically extracts the SHVM modules and the corresponding flow graphs from the annotated source code and performs the associated model checking tasks.

For evaluating our compositional verification approach, we considered the verification of the safety property explained in Example 11 for different versions of the trading system product line [37]. The product lines of cash desks were described as SHVMs with different hierarchical depths and different total numbers of modules. As a basis, we used the product line described in Example 8 and extended it by an optional coupon handling functionality within the sale method, and a variation point for accepting different card types as a hierarchical refinement of variant Card. For each product line, we compared the time required to verify all induced products individually with the time for compositional verification. The experiments were performed on a SUN SPARC machine.

The results are summarized in Table 1 where CD denotes the product line of Example [37], CD/CH the version with coupon handling, CD/CT the version with different card types and CD/CH/CT the version with coupon handling and different card types. As can be observed from the table, the processing time $t_{ind}$ for verifying every product individually grows dramatically when new modules and levels of hierarchy are added to the SHVM. This is easily explained by the analytical bounds presented in Section 3.2. In contrast, the growth of the processing time $t_{comp}$ for compositional SHVM verification is insignificant, since the preprocessing and flow graph extraction is only performed once by ProMoVer for the complete SHVM. The experiment suggests that for large software products comprising many products, the compositional verification technique based on the SHVM representation of the product line increases efficiency of verification dramatically.

Scalability of our method comes at the price of having to provide specifications for variation points. This additional effort is justified for large systems that render infeasible the verification of the product line by

---

6The focus of the evaluation is on comparing the times required for verification, and not on the total times themselves.
verifying all its products individually. Also, the specifications only need to be written once and are later reused when the code has been changed, or for proving other global properties.

SHVMs do not allow to express that a variant requires or excludes another variant. Without these constraints, the set of products that can be derived from an SHVM is larger than with requires/excludes constraints. If a desired property can be shown for the larger set of products defined by an SHVM, the property immediately holds for the original product set defined by the hierarchical variability model. However, this leaves the possibility that not all products defined by an SHVM satisfy a property such that verification procedure fails, while the property is satisfied by the products defined by an hierarchical variability model containing variant constraints. In this case, an additional check of the excluded products would be required.

5. Related Work

Variability Modeling. Hierarchical variability models represent solution space variability. The existing approaches to represent solution space product line variability can be divided into three directions [40]. First, annotative approaches consider one model representing all products of a product line. Variant annotations, e.g., using UML stereotypes [51, 15], presence conditions [10], or separate variability representations, such as orthogonal variability models [36], define which parts of the model have to be removed to generate the model of a concrete product. Second, compositional approaches [4, 49, 34, 3] associate product fragments with product features which are composed for particular feature configurations, such as hierarchical variability models. Third, transformational approaches [22, 8] represent variability by rules determining how a base model has to be changed for a particular product model. All these approaches consider a representation of artifact variability without any hierarchy.

Our hierarchical variability model generalizes the ideas of the Koala component model [48] for the implementation of variant-rich component-based systems. In Koala, the variability of a component is described by the variability of its subcomponents which can be selected by switches and explicit diversity interfaces. Diversity switches and interfaces in Koala can be understood as concrete language constructs at the implementation level targeted to express variation points and associated variants. Plastic partial components [35] are an architectural modeling approach where component variability is defined by extending partially defined components with variation points and associated variants. However, variants cannot contain variable components so this modeling approach is not truly hierarchical. Hierarchical variability modeling for software architectures [21] applies the modeling concepts for solution space variability presented in this paper to component-based software engineering and provides a concrete modeling language for variable software architectures that is truly hierarchical.

However, none of these approaches formally defines the semantics of hierarchical variability models, nor reasons about their well-formedness or uniqueness. Simple hierarchical variability models strike a balance between the expressiveness of the modeling formalism—no bindings and being grammar-like—and the desirable property of uniqueness of models: With a more expressive modeling formalism, uniqueness may not be achievable. To the best of our knowledge, this work is the first to provide a formal semantics for hierarchical variability models in the solution space, and to characterize a class of variability models through the class of generated product families.

Variability Model Mining. This paper presents the first approach for constructing a hierarchical variability model for solution space variability from a given product family. So far, there have only been approaches to construct feature models for representing problem space variability for a given set of products. Czarnecki et al. [12] re-construct a feature model from a set of sample feature combinations using data mining techniques [1]. Other approaches aim at constructing feature models from sample mappings between products and their features using formal concept analysis [14], for instance, to derive logical dependencies between code variants from pre-processor annotations [42], or to construct a feature model for function-block based systems after determining model variants by similarity [38]. Loesch and Ploedereder [29] use formal concept analysis to optimize feature models in case of product line evolution, e.g., to remove unused features or to combine features that always occur together. Niu and Easterbrook [33] apply formal concept analysis to functional and non-functional product line requirements in order to construct a feature model as a more
abstract representation of the requirements. Also, information retrieval techniques are applied to obtain a feature model from heterogeneous product line requirements [2]. Using hierarchical clustering, a tree structure of textually similar requirements is constructed. Requirement clusters in the leaves are more similar to each other than requirements clusters closer to the root giving rise to the structure of a feature model.

In our work, we abstract from the need to determine the different variants of the same conceptual entity by assuming fixed artifact names and corresponding artifact implementations. However, if we relax this assumption, techniques, such as similarity analysis [38] or formal concept analysis [14] could be applied to infer the relationship between different variants of the same conceptual entity, and thus make our approach applicable.

Regular expressions and relational algebras. Regular expressions (regexps) were introduced by Kleene [24]. Several variants of the original definition are known [45]. A certain analogy between simple families and regexps without Kleene star can be noticed, where individual implementations, the \( \cup \) operation, and the \( \sqcup \) operation on families correspond to alphabet symbols, the + operation, and concatenation \( \cdot \), respectively. There are two major differences, however: in our domain there is a two-level hierarchy names-implementations with no analogue in Formal Languages, and, since products are sets, there is no repetition of implementations in them, while strings can have arbitrary repetitions of symbols. Our goal to construct an optimal SHVM for a given family corresponds to constructing a smallest regexp for a given (finite) language. It is known that regexp minimization is intractable: even without Kleene star or complement it is still co-NP-complete [45] problem INEQ(\( \{0, 1\}, \{\cup, \cdot\} \)) while in general it is PSPACE-complete [31] [23]. That discouraging result, however, is with respect to languages that have no restriction of non-repeating symbols. It is worth investigating whether the problem still remains intractable after the said restriction.

Our problem domain bears substantial similarity to relational algebra as well. Our concepts of name, product, family, and product union translate to active domain, tuple, database relation, union, and join of relations, a minor difference being that database theory allows join of relations that share attributes. For a detailed introduction to relational databases and relational algebra, see [30]. Using database terminology, our goal is, given a database relation to deduce aspects of its design. That is, to perform some sort of model mining. Database decomposition has been intensely studied for the purposes of forward design. To the best of our knowledge there are no results on mining the relational database model from a given database.

Verification of Product Families. Most approaches to algorithmic verification of behavioral properties of software product lines rely on an annotative model of the product line comprising all possible product variants in the same model [51] [47]. Existing model checking techniques are adapted to deal with optional behavior defined by variant annotations. For instance, in [13], modal transition systems are extended by variability operators from deontic logic. In [16], the process calculus CCS is extended with a variant operator to represent a family of processes. In [26], transitions of I/O-automata are related to variants. In [9], product families are modeled by transition systems where transitions are labeled with features, so that state reachability modulo a set of features can be computed. Also, in [5], safety specifications of features are identified and combined for the analysis of the products.

These approaches do not scale for large product lines since the used annotative product line models easily get very large. To counter this, Blundell et al. [17], Liu et al. [28], and Beek et al. [19] propose techniques for compositional verification of product features. In these approaches, the behavior of a feature is represented by a state machine to which other features may attach in designated states (interface states or variation points). For a temporal property of a feature, constraints for these states are generated which have to be satisfied by composed features. In another work, Millo et al. [32] check the conformance of variability information at the requirement and design level in a feature-based compositional fashion, but they do not address the reuse of verification results. In all these works, the compositionality results are based on the applied notion of features and feature composition, while SHVMs provide a more flexible means to define product variability.

The presented approach is one of the first compositional verification techniques for software product lines. It allows to guarantee efficiently that all products of a product line satisfy certain desired control-flow based safety properties. With respect to model checking behavioral properties of product lines, only
Blundell et al. [7] and Liu et al. [28] propose compositional verification techniques based on assume-guarantee style reasoning for product features. Other model checking approaches for product lines [13, 16, 26, 9] use a monolithic model of the complete product line such that they face severe state-space explosion problems since all possible products are analyzed in the same analysis step.

6. Conclusion

In this article, we present hierarchical solution space variability models for software product lines and we generalize a previously developed compositional technique and tool set for the automatic verification of control-flow based temporal safety properties to software families that can be described by such models.

We give a formal semantics of hierarchical variability models in terms of sets—or families—of products, where each product is a set of artifact implementations. We introduce the separation degree as a quality measure of hierarchical variability models. We identify well-formed variability models as a class of models for which the measure is maximal (and equal to one) and which are unique for the family they generate; the class of families generated by such models is the class of simple families. Furthermore, we present an algorithm that accepts as input a simple family and outputs the unique well-formed model that generates it. We prove uniqueness by showing that family generation and model construction are inverses of each other for this class of models. While maximum separation degree and uniqueness of models with maximal measure are theoretically appealing, in practice, product families might not be simple. Still, the separation degree is a useful measure for hierarchical variability models, and, as Examples 5 and 6 suggest, searching for the set of models with a maximal measure (not necessarily equal to one) for a given family is equally meaningful.

Using the introduced variability model, we adapt a previously developed method and tool set for compositional verification of procedural programs, which allows to avoid the combinatorial explosion of verifying all products individually. The number of verification tasks resulting from our method is linear in the size of the variability model rather than in the number of products. This is achieved by introducing variation point specifications on which product properties are relativized, and by constructing maximal flow graphs that replace the specifications when model checking specifications on the next higher level of hierarchy. The class of properties that can be handled fully automatically is the class of control flow-based temporal safety properties, specifying illegal sequences of method calls. The input to our verification tool is the description of a product line in form of an annotated Java program defining the variability model and the necessary specifications. Our first experiments with the tool show a dramatic gain in performance even for models with a low hierarchical depth.

Future work. Future work will focus on the practical evaluation of the proposed method for variability model mining, considering in particular sets of (legacy code) products that have not been designed as a family from the outset. Further effort is planned on generalizing the model with optional and multiple variant selections and with requires/excludes constraints between variants, and on adapting accordingly the model reconstruction transformation. Another generalization will deal with the more abstract domain of products over implementations only, where the names are not given in advance, but must be inferred. Additionally, the restriction that all variants associated to a variation point have to provide the same artifact names will be lifted.

References

Appendix A. Proofs

Proposition 1. Let family $F$ be simple. The following holds.

(i) Let $a_i \in \text{impls}(F)$, and let $F'$ be the projection of $F$ on $\text{names}(F) \setminus \{a\}$. $a_i$ occurs in all products of $F$, i.e., $a_i \in \bigcap_{\phi \in F} P$, iff $F = \{a_i\} \times F'$. Then either $F' = 1_F$ and thus rule (F1) applies, or else $F'$ is simple and rule (F2) applies.

(ii) Let $\{A_1, A_2\}$ be a non-trivial partitioning of $\text{names}(F)$, and let $F_1$ and $F_2$ be the projections of $F$ on $A_1$ and $A_2$, respectively. Every name in $A_1$ is orthogonal to every name in $A_2$ in $F$, i.e., $A_1 \times A_2 \subseteq F'$. $F = F_1 \times F_2$ if $F = F_1 \times F_2$ and $F_1$ and $F_2$ are simple. Formation rule (F3) applies in this case.

(iii) Let $\{F_1, F_2\}$ be a non-trivial partitioning of $F$. No product of $F_1$ shares an artifact implementation with any product of $F_2$, i.e., $F_1 \times F_2 \subseteq \overline{N_F}$, iff $F = F_1 \cup F_2$ and $F_1$ and $F_2$ are simple. Formation rule (F3) applies in this case.

Proof. The if parts of each case are immediate from Def. 1. The only-if parts are established by structural induction on the formation of $F$.

(i) Let $a_i \in \text{impls}(F)$, $F'$ be the projection of $F$ on $\text{names}(F) \setminus \{a\}$, and let $a_i \in \bigcap_{\phi \in F} P$. We consider the three possible ways of forming the simple family $F$.

(a) Let $F = \{b_j\}$. Then $a_i = b_j$, and so $F = \{a_i\} \times 1_F$.

(b) Let $F = F_1 \times F_2$ for simple $F_1$ and $F_2$ such that $\text{names}(F_1) \cap \text{names}(F_2) = \emptyset$. Assume w.l.o.g. that $a \in \text{names}(F_1)$. Then $a_i \in \bigcap_{\phi \in F_1} P$ and, by the induction hypothesis, $F_1 = \{a_i\} \times F_1'$ where either $F_1' = 1_F$ or else $F_1'$ is simple. In both cases, by the associativity of $\times$, $F = \{a_i\} \times F' = F_1' \times F_2$, and hence $F'$ is simple.
(c) The case $F = F_1 \cup F_2$ for simple $F_1$ and $F_2$ such that $\text{names}(F_1) = \text{names}(F_2)$ and $\text{impls}(F_1) \cap \text{impls}(F_2) = \emptyset$ is not possible when $a_i \in \bigcap_{P \in F} P$.

(ii) Let $\{A_1, A_2\}$ be a non-trivial partitioning of $\text{names}(F)$, let $F_1$ and $F_2$ be the projections of $F$ on $A_1$ and $A_2$, respectively, and let $A_1 \times A_2 \subseteq \overline{C_F}$. Again we consider three cases.

(a) The case $F = \{\{b_1\}\}$ is not possible when $\{A_1, A_2\}$ is non-trivial.

(b) The case $F = F_1 \times F_2'$ for simple $F_1'$ and $F_2'$ such that $\text{names}(F_1') \cap \text{names}(F_2') = \emptyset$. Let $A_1' = \text{names}(F_1')$ and $A_2' = \text{names}(F_2')$. If $A_1' = A_1$ then $A_2' = A_2$ and the result follows immediately. Otherwise, let $A_1' \cap A_1, A_1' \cap A_2, A_2' \cap A_1$ and $A_2' \cap A_2$. Then $\{A_1', A_1'\}$ and $\{A_1', A_1'\}$ are non-trivial partitionings of $A_1'$ and $A_2'$, respectively. Furthermore, $A_1' \times A_1' \subseteq \overline{C_{F_1}}$ and $A_1' \times A_2' \subseteq \overline{C_{F_2}}$. Then, by the induction hypothesis, $F_1 = F_1' \times F_1'$ and $F_2 = F_2' \times F_2'$, where, for all $i, j \in \{1, 2\}$, $F_{i,j}$ is the projection of $F_i$ on $A_j$ and is simple. Then $F_1 = F_1' \times F_2'$ and $F_2 = F_2' \times F_2'$ are simple, and, by the associativity of $\times$, $F = F_1 \times F_2$.

(c) The case $F = F_1' \cup F_2'$ for simple $F_1'$ and $F_2'$ such that $\text{names}(F_1') = \text{names}(F_2')$ and $\text{impls}(F_1') \cap \text{impls}(F_2') = \emptyset$ is not possible when $\{A_1, A_2\}$ is non-trivial and $A_1 \times A_2 \subseteq \overline{C_F}$.

(iii) Let $\{F_1, F_2\}$ be a non-trivial partitioning of $F$ and let $F_1 \times F_2 \subseteq \overline{N_F}$. Again we consider three cases.

(a) The case $F = \{\{b_1\}\}$ is not possible when $\{F_1, F_2\}$ is non-trivial.

(b) The case $F = F_1' \times F_2'$ for simple $F_1'$ and $F_2'$ such that $\text{names}(F_1') \cap \text{names}(F_2') = \emptyset$ is also not possible when $\{F_1, F_2\}$ is non-trivial and $F_1 \times F_2 \subseteq \overline{N_F}$.

(c) The case $F = F_1' \cup F_2'$ for simple $F_1'$ and $F_2'$ such that $\text{names}(F_1') = \text{names}(F_2')$ and $\text{impls}(F_1') \cap \text{impls}(F_2') = \emptyset$. If $F_1' = F_1$ then $F_2' = F_2$ and the result follows immediately. Otherwise, let $F_{1,1}' \equiv F_1' \cap F_1, F_{1,2}' \equiv F_1' \cap F_2, F_{2,1}' \equiv F_2' \cap F_1$ and $F_{2,2}' \equiv F_2' \cap F_2$. Then $\{F_{1,1}', F_{1,2}'\}$ and $\{F_{2,1}', F_{2,2}'\}$ are non-trivial partitionings of $F_1'$ and $F_2'$, respectively. Furthermore, $F_{1,1}' \times F_{1,2}' \subseteq \overline{N_{F_1}}$ and $F_{2,1}' \times F_{2,2}' \subseteq \overline{N_{F_2}}$. Then, by the induction hypothesis, $F_{1,1}', F_{1,2}', F_{2,1}'$ and $F_{2,2}'$ are all simple, and hence $F_1 = F_{1,1}' \cup F_{1,2}'$ and $F_2 = F_{2,1}' \cup F_{2,2}'$ are simple, too. Furthermore, since $F_1 \times F_2 \subseteq \overline{N_F}$ implies $\text{impls}(F_1) \cap \text{impls}(F_2) = \emptyset$, rule $[F3]$ applies.

This concludes the proof. \hfill \Box

**Proposition 2.** If variability model $S$ is well-formed then $sd(S) = 1$.

**Proof.** We show $sd'(S) = |\text{impls}(S)|$ by structural induction. First, let $S$ be a ground model with common set $M_C$. We have:

$$sd'(M_C) = |M_C| \quad \text{(Def. 5)}$$

Next, let $S$ be a variability model $(M_C, \{VP_1, \ldots, VP_n\})$ with variation points $VP_i = \{S_{i,j} | 1 \leq j \leq k_i\}$. As the induction hypothesis, assume the result holds for all $S_{i,j}$. We have:

$$sd'((M_C, \{VP_1, \ldots, VP_n\})) = |M_C| + \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq k_i} sd'(S_{i,j}) \quad \text{(Def. 6)}$$

$$= |M_C| + \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq k_i} |\text{impls}(S_{i,j})| \quad \text{(Ind. hyp.)}$$

$$= |M_C| \cup \bigcup_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} |\text{impls}(S_{i,j})| \quad \text{(Def. 7)}$$

$$= |\text{impls}((M_C, \{VP_1, \ldots, VP_n\})| \quad \text{(Def. 5)}$$

26
This concludes the proof. \(\square\)

**Proposition 3.** For a given SHVM, let \(\text{AND}\) and \(\text{OR}\) denote the maximum branching factors at SHVM and variation point nodes, respectively, and let \(ND\) be its nesting depth. The number of products induced by the SHVM is bound by \(\frac{\text{AND}^{ND}\cdot(\text{AND}^{ND}-1)}{\text{AND}-1}\) and is thus exponential in the size of the SHVM, which is bound by \(\frac{\text{OR}\cdot\text{AND}^{ND}\cdot(\text{AND}^{ND}-1)}{\text{OR}\cdot\text{AND}-1}\).

**Proof.** The bounds on the number of products and size of an SHVM is obtained by solving the following recurrence equations in a routine fashion.

\[
T(0) = T_0 \\
T(n) = \text{OR} \cdot T(n-1)^{\text{AND}}
\]

\[
T(0) = T_0 \\
T(n) = \text{OR} \cdot \text{AND} \cdot T(n-1) + 1
\]

**Proposition 4.** If variability model \(S\) is well-formed, then \(\text{family}(S)\) is simple.

**Proof.** By structural induction. First, let \(S\) be a well-formed ground model with common artifact implementations \(M_C\). In that case \(\text{family}(M_C)\) has a single product \(M_C\) that implements artifact names at most once. Then \(M_C\) can be represented as a product union over its artifact implementations taken as single-product families, and is hence simple.

Next, let \(S\) be a well-formed variability model \((M_C,\{VP_1,\ldots, VP_n\})\) with variation points \(VP_i = \{S_{i,j} | 1 \leq j \leq k_i\}\). As the induction hypothesis, assume the result holds for all \(S_{i,j}\). Since \(S\) is well-formed, so are all \(S_{i,j}\) by Definition 7 and hence, by the induction hypothesis, all families \(\text{family}(S_{i,j})\) are simple. For every variation point \(VP_i\) we have

\[
\text{family}(VP_i) = \bigcup_{1 \leq j \leq k_i} \text{family}(S_{i,j})
\]

by Definition 8. Further, by well-formedness constraint \((S3)\) of Definition 7, we have that \(\text{names}(S_{i,j_1}) = \text{names}(S_{i,j_2})\) for all \(i,j_1,j_2\), and \(\text{impls}(S_{i,j_1}) \cap \text{impls}(S_{i,j_2}) = \emptyset\) whenever \(j_1 \neq j_2\). Hence, by formation rule \((F3)\) of Definition 3, all \(\text{family}(VP_i)\) are simple. Furthermore, we have

\[
\text{family}(S) = \{M_C\} \times \prod_{1 \leq i \leq n} \text{family}(VP_i)
\]

by Definition 8. Further, by well-formedness constraint \((S2)\) of Definition 7, we have that \(\text{names}(M_C) \cap \text{names}(VP_i) = \emptyset\) for all \(i\), and \(\text{names}(VP_{i_1}) \cap \text{names}(VP_{i_2}) = \emptyset\) whenever \(i_1 \neq i_2\). Now, \(M_C\) is simple due to well-formedness constraint \((S1)\) of Definition 7 (see base case), and since all \(\text{family}(VP_i)\) are simple, by formation rule \((F2)\) of Definition 3, \(\text{family}(S)\) is also simple. \(\square\)

**Proposition 5.** If \(F\) is a simple non-core family in canonical form then for all \(i, 1 \leq i \leq n\), and \(k_i \geq 2\) all \(F_{i,j}\) are simple and of strictly smaller size than \(F\).

**Proof.** For every \(i\), by Proposition 1, \(F_i\) is simple. Furthermore, by condition \((C2)\) all names of \(F_i\) are correlated, and hence, by Proposition 1, \(F_i\) is not formed by rule \((F2)\). Since \(F\) is non-core, \(F_i\) is also non-core and is therefore formed by \((F3)\). Hence, again by Proposition 1, there are at least two equivalence classes of \(\text{impls}(F_i)\) w.r.t. implementation sharing \(N_{\text{imp}}^2\), and thus \(k_i \geq 2\).

That all \(F_{i,j}\) are simple is guaranteed by the three properties of simple families stated in Proposition 1 that match the three conditions in Definition 9.
That all $F_{i,j}$ are strictly smaller is enforced through the formation rules for simple families from Definition 3. Rule (F1) requires the existence of a shared artifact implementation, rule (F2) requires at least two equivalence classes on names, and rule (F3) requires at least two equivalence classes on implementations, and thus the decomposition into canonical form is never trivial. □

**Proposition 6.** If family $F$ is simple, then $shvm(F)$ is well-formed.

**Proof.** By induction on the size of $F$. First, let $F$ be a core $\{P\}$. Then, by Definition 10, $shvm(F) = P$, which is a well-formed variability model.

Next, let $F$ be a non-core family decomposed into canonical form. As the induction hypothesis, assume the result holds for all families smaller than $F$. We have:

\[
shvm(\{P\} \times \prod_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} F_{i,j})
\]

where $VP_i = \{shvm(F_{i,j}) | 1 \leq j \leq k_i\}$

By Proposition 5, all $F_{i,j}$ are simple and strictly smaller than $F$ and hence, by the induction hypothesis, all $shvm(F_{i,j})$ are well-formed variability models. Now, since $F$ is in canonical form, conditions (C1) to (C3) hold, ensuring the well-formedness constraints (S1) to (S3) respectively, and hence also $shvm(F)$ is a well-formed variability model. □

**Lemma 1.** For every simple family $F$ we have:

\[ \text{family}(shvm(F)) = F \]

**Proof.** By induction on the size of $F$. First, let $F$ be a core $\{P\}$. We have:

\[
\text{family}(shvm(\{P\}))
\]

\[ = \text{family}(P) \quad \{\text{Def. 10}\}
\]

\[ = \{P\} \quad \{\text{Def. 8}\}
\]

Next, let $F$ be a non-core family decomposed into canonical form presented as above. As the induction hypothesis, assume the result holds for all families smaller that $F$, and thus, by Proposition 5, for all $F_{i,j}$. We have:

\[
\text{family}(shvm(\{P\} \times \prod_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} F_{i,j}))
\]

where $VP_i = \{shvm(F_{i,j}) | 1 \leq j \leq k_i\}$

\[ = \text{family}((P, \{VP_1, \ldots, VP_n\})) \quad \{\text{Def. 10}\}
\]

\[ = \{P\} \times \prod_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i, \text{family}(shvm(F_{i,j}))} \quad \{\text{Def. 8}\}
\]

\[ = \{P\} \times \prod_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} F_{i,j} \quad \{\text{Ind. hyp.}\}
\]

This concludes the proof of the lemma. □

**Lemma 2.** For every well-formed variability model $S$ we have:

\[ shvm(\text{family}(S)) = S \]
Proof. By structural induction. First, let \( S \) be a ground model with common set \( M_C \). We have:

\[
shvm(family(M_C))
\]

\[
= shvm(\{M_C\}) \quad \text{(Def. 8)}
\]

\[
= M_C \quad \text{(Def. 10)}
\]

Next, let \( S \) be a variability model \((M_C, \{VP_1, \ldots, VP_n\})\) with variation points \( VP_i = \{S_{i,j} \mid 1 \leq j \leq k_i\}\). As the induction hypothesis, assume the result holds for all \( S_{i,j} \). We have:

\[
shvm(family((M_C, \{VP_1, \ldots, VP_n\})))
\]

\[
= shvm(\{M_C\} \times \prod_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} family(S_{i,j})) \quad \text{(Def. 8)}
\]

\[
= (M_C, \{VP_1', \ldots, VP_n'\}) \quad \text{(Def. 10)}
\]

where \( VP'_i = \{shvm(family(S_{i,j})) \mid 1 \leq j \leq k_i\} \)

\[
= (M_C, \{VP_1', \ldots, VP_n'\}) \quad \text{(Ind. hyp.)}
\]

\[
= (M_C, \{VP_1, \ldots, VP_n\}) \quad \text{(Def. } S\text{)}
\]

To justify the second step above we need to show that

\[
\{M_C\} \times \prod_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} family(S_{i,j})
\]

is in canonical form. This is established as follows, using that \( S \) is simple. First, the restriction that variation points have at least two variants and the constraint \([S2]\) guarantee that just the artifact implementations in \( M_C \) and no other artifact implementations are shared by all products of \( S \), and thus condition \([C1]\) is satisfied.

Next, constraint \([S2]\) guarantees that artifact names implemented by different variation points are orthogonal. On the other hand, the restriction that variation points have at least two variants and the constraint \([S3]\) guarantee that artifact names implemented by the same variation point must be correlated, and thus condition \([C2]\) is satisfied.

And finally, constraint \([S3]\) guarantees that variants do not share any artifact implementation. On the other hand, the restriction guarantees that any two products of the same variant share an artifact implementation, and thus condition \([C3]\) is satisfied. This concludes the proof of the lemma. \( \square \)

Theorem 2. Let \( S \) be an SHVM with global property \( \phi \). If the verification procedure succeeds for \( S \), then \( p \models \phi \) for all its products \( p \in \text{products}(S) \).

Proof. The proof is by induction on the structure of \( S \). For the base case, let \( S \) be a ground model with common set \( M_C \). Assume the verification procedure succeeds for \( S \). It has then established: \( \bigcup_{a \in Art(M_C)} G_a \models \phi \). From this, and by soundness of rule \([1]\), it follows that \( M_C \models \phi \). Since \( \text{products}(S) = \{M_C\} \) in this case, we have \( p \models \phi \) for all \( p \in \text{products}(S) \).

For the induction step, let \( S \) be a non-ground model \((M_C, \{VP_1, \ldots, VP_n\})\) with variation points \( VP_i = \{S_{i,j} \mid 1 \leq j \leq k_i\} \), where \( k_i \) is the number of variants of \( VP_i \). Further, let \( (\psi_{VP_i}, I_{VP_i}) \) be the specification of \( VP_i \). Assume the result for all \( S_{i,j} \) (induction hypothesis). Next, assume that the verification procedure succeeds for \( S \). The following has then been established for the top-level module:

\[
(i) \quad \bigcup_{a \in Art(M_C)} G_a \cup \bigcup_{1 \leq i \leq n} Max(\psi_{VP_i}, I_{VP_i}) \models \phi
\]

By the assumption, the verification procedure has also succeeded for all \( S_{i,j} \). Thus, by the induction hypothesis, and since the SHVM nodes of variants attached to a variation point inherit the corresponding variation point specification, we have:

\[
\forall i: 1 \leq i \leq n. \forall j: 1 \leq j \leq k_i. \forall p \in \text{products}(S_{i,j}). p \models \psi_{VP_i}
\]
By Definition 8 we have \( \text{products}(VP_i) = \bigcup_{1 \leq j \leq k_i} \text{products}(S_{i,j}) \), and hence:

\[(ii) \quad \forall i : 1 \leq i \leq n. \forall p \in \text{products}(VP_i). p \models \psi_{VP_i}.\]

Also by Definition 8, we know that every product \( p \) of \( S \) is the union of the core \( M_C \) and exactly one subproduct from every variation point. Due to \((ii)\), all subproducts meet their respective specifications. Also, by \( (i)\) and from soundness of rule \( (i)\) follows that \( p \models \phi \). This concludes the proof. \( \square \)