Serdica J. Computing **9** (2015), No 1, 35–82

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

MODEL MINING AND EFFICIENT VERIFICATION OF SOFTWARE PRODUCT LINES

Siavash Soleimanifard, Dilian Gurov, Ina Schaefer,
Bjarte M. Østvold, Minko Markov

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.

ACM Computing Classification System (1998): D.2.4, D.2.7.

Key words: Product families, Compositional verification, Model mining, Variability models, Model checking, Maximal models.

1. Introduction.

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. *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 [36].

The variability of the different products in a software product line can be represented at different levels [11]. *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 *artifacts* that are used to realize the products. Problem-space variability based on features is at the requirements level, while *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.

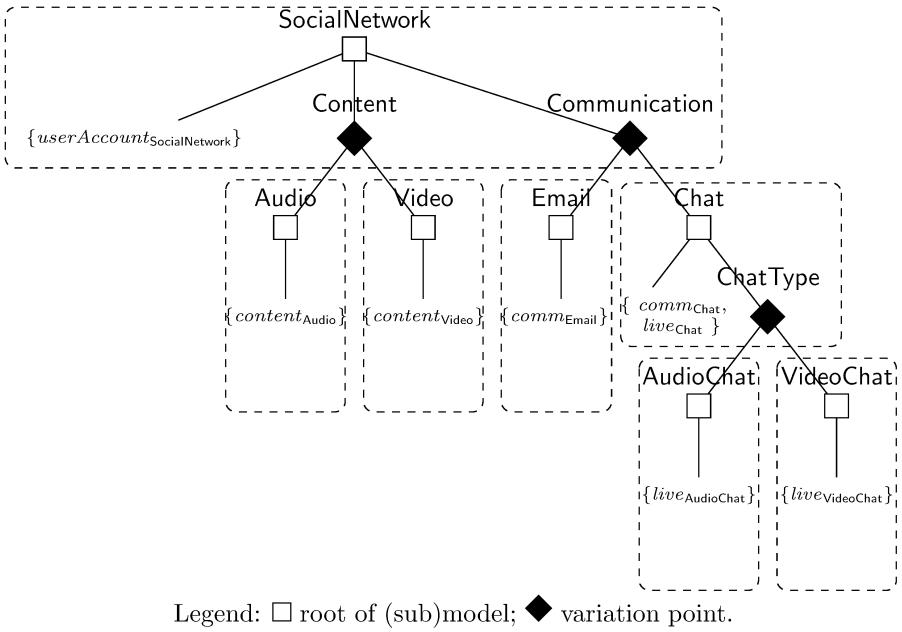

Fig. 1. The Social Network hierarchical variability model

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 [39]. 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 as, e.g., a component or a method. We fix a countably infinite set of artifact names Art.

Definition 1 (Product, family)**.** *An* artifact implementation *is an indexed artifact name; let a_i denote the i-th implementation of artifact name a. A* product P *is a finite set of artifact implementations, where for each artifact name there is at most one implementation. A* family $\mathcal{F}$ *is a finite non-empty set of products.*

Thus, products can be seen as partial maps from artifact names to natural numbers, having a finite domain; we use Nat^{Art} to denote the set of all products over Art. We refer to singleton set families as *core* families, or simply cores. The family consisting of the empty product is denoted $1_{\mathcal{F}}$.

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

$$\mathcal{F}_A = \{\ \{a_1, b_1, c_1, d_1, e_1\}, \{a_1, b_1, c_1, d_1, e_2\}, \{a_1, b_1, c_2, d_2, e_1\},$$
$$\{a_1, b_1, c_2, d_2, e_2\}, \{a_1, b_1, c_2, d_3, e_1\}, \{a_1, b_1, c_2, d_3, e_2\}\ \}$$
$$\mathcal{F}_B = \{\ \{a_1, b_1\}, \{a_1, b_2\}, \{a_2, b_1\}\ \}$$

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

$$names\,(\mathcal{F}) \stackrel{\mathsf{def}}{=} \bigcup_{P \in \mathcal{F}} names\,(P)$$

$$\textit{where names}\left(\{a^1_{i_1},\ldots,a^n_{i_n}\}\right) \stackrel{\mathsf{def}}{=} \{a^1,\ldots,a^n\}$$

$$\textit{impls}(\mathcal{F}) \stackrel{\mathsf{def}}{=} \bigcup_{P\in\mathcal{F}} \textit{impls}(P)$$

$$\textit{where impls}(\{a^1_{i_1},\ldots,a^n_{i_n}\}) \stackrel{\mathsf{def}}{=} \{a^1_{i_1},\ldots,a^n_{i_n}\}$$

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 $\bowtie$ over families with disjoint sets of artifact names defined by:

$$\mathcal{F}_1 \bowtie \mathcal{F}_2 \stackrel{\mathsf{def}}{=} \{P_1 \cup P_2 \,|\, P_1 \in \mathcal{F}_1 \wedge P_2 \in \mathcal{F}_2\}$$

and generalized through $\prod_{i\in I} \mathcal{F}_i$ 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* $\boldsymbol{F}$ *of* simple *families is the least set of families closed under the formation rules:*

(F1) $\{\{a_i\}\} \in \boldsymbol{F}$ *for any* $a \in Art$ *and* $i \in Nat$*.*

(F2) $\mathcal{F}_1 \bowtie \mathcal{F}_2 \in \boldsymbol{F}$ *for any* $\mathcal{F}_1, \mathcal{F}_2 \in \boldsymbol{F}$ *such that* $\textit{names}\,(\mathcal{F}_1) \cap \textit{names}\,(\mathcal{F}_2) = \emptyset$*.*

(F3) $\mathcal{F}_1 \cup \mathcal{F}_2 \in \boldsymbol{F}$ *for any* $\mathcal{F}_1, \mathcal{F}_2 \in \boldsymbol{F}$ *such that* $\textit{names}\,(\mathcal{F}_1) = \textit{names}\,(\mathcal{F}_2)$ *and* $\textit{impls}(\mathcal{F}_1) \cap \textit{impls}(\mathcal{F}_2) = \emptyset$*.*

Example 2. The family $\{\{a_1,b_1\},\{a_1,b_2\}\}$ is simple, as it can be presented as $\{\{a_1\}\} \bowtie (\{\{b_1\}\} \cup \{\{b_2\}\})$ which follows the above formation rules.

[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 $\bowtie$.

Family $\mathcal{F}_A$ of Example 1 is also simple (as we shall see later in Example 6, while family $\mathcal{F}_B$ of Example 1 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 $a, b \in \mathit{names}\,(\mathcal{F})$ are termed *correlated* in a family $\mathcal{F}$, denoted $a\ C_{\mathcal{F}}\ b$, if there are implementations $a_i, b_j \in \mathit{impls}(\mathcal{F})$ such that no product in $\mathcal{F}$ contains both implementations simultaneously. Otherwise, names a and b are termed *uncorrelated* or *orthogonal*. The correlation relation $C_{\mathcal{F}}$ on $\mathit{names}\,(\mathcal{F})$ is symmetric, and hence, its reflexive and transitive closure $C^*_{\mathcal{F}}$ is an equivalence relation. As usual, we denote the partitioning induced by $C^*_{\mathcal{F}}$ on $\mathit{names}\,(\mathcal{F})$ by $\mathit{names}\,(\mathcal{F})\,/C^*_{\mathcal{F}}$ (quotient set).

Example 3. Consider family $\mathcal{F}_A$ of Example 1. 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 $\mathit{names}\,(\mathcal{F}_A)\,/C^*_{\mathcal{F}_A} = \{\{a\}, \{b\}, \{c, d\}, \{e\}\}$.

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

Similarly, we define the sharing relation $N_{\mathcal{F}}$ on $\mathcal{F}$ as $P_1\ N_{\mathcal{F}}\ P_2 \overset{\mathsf{def}}{\Leftrightarrow} P_1 \cap P_2 \neq \emptyset$, and use its reflexive and transitive closure $N^*_{\mathcal{F}}$ to partition the family $\mathcal{F}$.

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, $\overline{A}$ denotes the complement of set A in a given universe of elements.

Proposition 1. *Let family $\mathcal{F}$ be simple. The following holds.*

(i) Let $a_i \in \mathit{impls}(\mathcal{F})$, and let $\mathcal{F}'$ be the projection of $\mathcal{F}$ on $\mathit{names}\,(\mathcal{F}) \setminus \{a\}$. The

name a_i occurs in all products of $\mathcal{F}$, i.e., $a_i \in \bigcap_{P \in \mathcal{F}} P$, *iff* $\mathcal{F} = \{\{a_i\}\} \bowtie \mathcal{F}'$. *Then either $\mathcal{F}' = 1_{\mathcal{F}}$ and thus rule ($\mathcal{F}1$) applies, or else $\mathcal{F}'$ is simple and rule ($\mathcal{F}2$) 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., $A_1 \times A_2 \subseteq \overline{C_{\mathcal{F}}}$, *iff $\mathcal{F} = \mathcal{F}_1 \bowtie \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule ($\mathcal{F}2$) 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., $\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_{\mathcal{F}}}$, *iff $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule ($\mathcal{F}3$) 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 ($\mathcal{F}2$) with some suitable $\mathcal{F}_1$ and $\mathcal{F}_2$ satisfying the rule's condition, then it cannot be formed by formation rule ($\mathcal{F}3$), and *vice versa*.

When restricted to simple families, the two operations on families do not distribute over each other. This entails that simple families have *unique* formation trees modulo commutativity and associativity of the two operations associated with the rules.

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) $\mathcal{S}$ *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 $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$, 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.

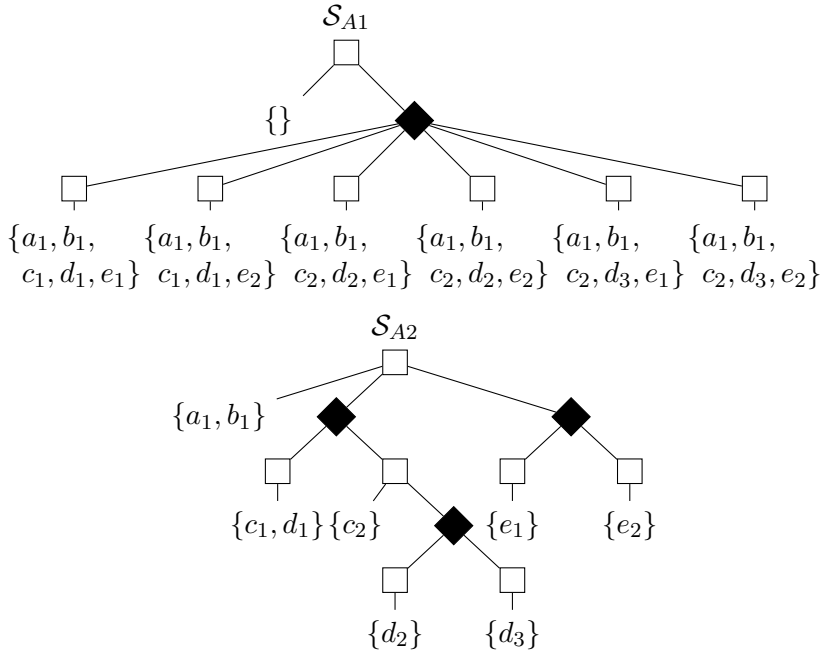

Fig. 2. SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ for the family $\mathcal{F}_A$ in Example 1

Example 4. Figure 2 and Figure 3 show four variability models named $\mathcal{S}_{A1}$, $\mathcal{S}_{A2}$, $\mathcal{S}_{B1}$, and $\mathcal{S}_{B2}$. 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* $(\mathcal{S})$ *from SHVMs to sets of artifact names and the mapping impls*$(\mathcal{S})$ *from SHVMs to sets of artifact implementations are defined as follows, where*

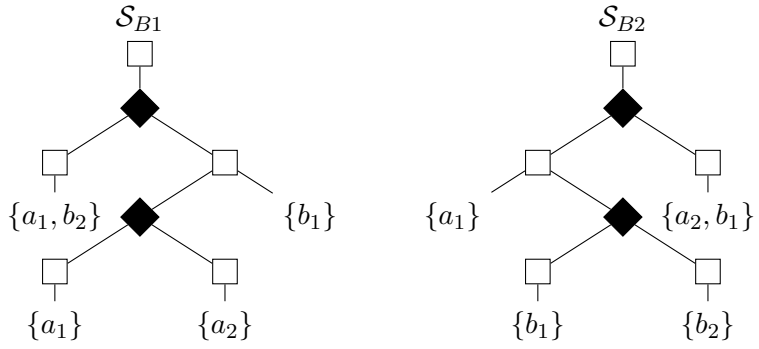

Fig. 3. SHVMs $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ for the family $\mathcal{F}_B$ in Example 1

$a^1, \ldots, a^n \in Art$ *and* $i_1, \ldots, i_n \in Nat$*:*

$$names\left(\{a^1_{i_1}, \ldots, a^n_{i_n}\}\right) \stackrel{\mathsf{def}}{=} \{a^1, \ldots, a^n\}$$

$$names\left((M_C, \{VP_1, \ldots, VP_n\})\right) \stackrel{\mathsf{def}}{=} names\left(M_C\right) \cup \bigcup_{1 \leq i \leq n} names\left(VP_i\right)$$

$$where\ \ names\left(VP\right) \stackrel{\mathsf{def}}{=} \bigcup_{\mathcal{S} \in VP} names\left(\mathcal{S}\right)$$

$$impls(\{a^1_{i_1}, \ldots, a^n_{i_n}\}) \stackrel{\mathsf{def}}{=} \{a^1_{i_1}, \ldots, a^n_{i_n}\}$$

$$impls((M_C, \{VP_1, \ldots, VP_n\})) \stackrel{\mathsf{def}}{=} impls(M_C) \cup \bigcup_{1 \leq i \leq n} impls(VP_i)$$

$$where\ \ impls(VP) \stackrel{\mathsf{def}}{=} \bigcup_{\mathcal{S} \in VP} impls(\mathcal{S})$$

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 the commonalities and orthogonalities of products are factored out as common sets and variation points in a variability model, respectively: the higher this degree, the less artifact implementations occur repeatedly in more than one leaf. The maximum value of 1 holds when every artifact implementation occurs in exactly one leaf; this is trivially the case for ground models.

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

$$sd(\{\}) \stackrel{\mathsf{def}}{=} 1$$

$$sd(\mathcal{S}) \stackrel{\mathsf{def}}{=} \frac{|impls(\mathcal{S})|}{sd'(\mathcal{S})} \qquad \text{if } \mathcal{S} \neq \{\}$$

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

$$sd'(M_C) \stackrel{\mathsf{def}}{=} |M_C|$$

$$sd'((M_C, \{VP_1, \ldots, VP_n\})) \stackrel{\mathsf{def}}{=} sd'(M_C) + \Sigma_{1 \leq i \leq n} sd'(VP_i)$$

$$\text{where } sd'(VP) \stackrel{\mathsf{def}}{=} \Sigma_{\mathcal{S} \in VP}\, sd'(\mathcal{S})$$

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

($\mathcal{S}1$*)* M_C *implements artifact names at most once.*

($\mathcal{S}2$*)* $names\,(M_C) \cap names\,(VP_i) = \emptyset$ *for all* i*, and*
$names\,(VP_{i_1}) \cap names\,(VP_{i_2}) = \emptyset$ *whenever* $i_1 \neq i_2$*.*

($\mathcal{S}3$*)* $names\,(\mathcal{S}_{i,j_1}) = names\,(\mathcal{S}_{i,j_2})$ *for all* i, j_1, j_2*, and*
$impls(\mathcal{S}_{i,j_1}) \cap impls(\mathcal{S}_{i,j_2}) = \emptyset$ *whenever* $j_1 \neq j_2$*.*

Example 5. Consider the SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ depicted in Figure 2. $\mathcal{S}_{A1}$ is not well-formed whereas $\mathcal{S}_{A2}$ is. The separation degrees are $sd(\mathcal{S}_{A1}) = \frac{9}{6 \cdot 5} = 0.3$ and $sd(\mathcal{S}_{A2}) = \frac{9}{9} = 1$. Figure 3 depicts two other SHVMs, $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$. Neither of these are well-formed and both have separation degree $\frac{4}{5} = 0.8$.

The constraints in Definition 6 ensure that the separation degree of a well-formed SHVM is equal to 1 and is thus maximum.

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

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 ($\mathcal{S}1$) 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 $OR^{\frac{AND \cdot (AND^{ND}-1)}{AND-1}}$ and is thus exponential in the size of the SHVM, which is bound by $\frac{(OR \cdot AND)^{(ND+1)} - 1}{OR \cdot AND - 1}$.*

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*$(\mathcal{S})$ *from variability models to families is inductively defined as follows:*

$$family(M_C) \stackrel{\mathsf{def}}{=} \{M_C\}$$

$$family((M_C, \{VP_1, \ldots, VP_n\})) \stackrel{\mathsf{def}}{=} \{M_C\} \bowtie \prod_{1 \leq i \leq n} family(VP_i)$$

$$where\ family(VP) \stackrel{\mathsf{def}}{=} \bigcup_{\mathcal{S} \in VP} family(\mathcal{S})$$

We say that variability model $\mathcal{S}$ generates *family*$(\mathcal{S})$.

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* $\mathcal{S}$ *is well-formed, then family*$(\mathcal{S})$ *is simple.*

Example 6. SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ in Figure 2 both generate family $\mathcal{F}_A$ in Example 1, implying that family $\mathcal{F}_A$ is simple since $\mathcal{S}_{A2}$ is well-formed. SHVMs $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ in Figure 2 both generate family $\mathcal{F}_B$ in Example 1. Among these four SHVMs, $\mathcal{S}_{A2}$, $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ have maximum separation degree in the sense that, for each of the families $\mathcal{F}_A$ and $\mathcal{F}_B$, 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 $\mathcal{F}$ can be decomposed into the form:

$$\mathcal{F} = \{P\} \bowtie \mathcal{F}_V, \quad \mathcal{F}_V = \prod_{1 \leq i \leq n} \mathcal{F}_i, \quad \mathcal{F}_i = \bigcup_{1 \leq j \leq k_i} \mathcal{F}_{i,j}$$

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

$$(*) \qquad \mathcal{F} = \{P\} \bowtie \prod_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} \mathcal{F}_{i,j}$$

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

Definition 9 (Canonical form of family)**.** *A family $\mathcal{F}$, decomposed as equation $(*)$ 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 $\mathcal{F}$.

(C2) The set of artifact names in $\mathcal{F}_V$ has n equivalence classes w.r.t. correlated artifact names $C^*_{\mathcal{F}_V}$, and for the i-th equivalence class, the family $\mathcal{F}_i$ is the projection of $\mathcal{F}_V$ onto the artifact names of the class.

(C3) For all i, $1 \leq i \leq n$, $\mathcal{F}_{i,j}$ are the k_i equivalence classes of $\mathcal{F}_i$ w.r.t. implementation sharing $N^*_{\mathcal{F}_i}$.

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

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 $shvm(\mathcal{F})$ from simple families presented in canonical form to variability models is inductively defined as follows:*

$$shvm(\{P\}) \stackrel{\mathsf{def}}{=} P$$

$$shvm\big(\{P\} \bowtie \textstyle\prod_{1\leq i\leq n} \bigcup_{1\leq j\leq k_i} \mathcal{F}_{i,j}\big) \stackrel{\mathsf{def}}{=} (P, \{VP_1, \ldots, VP_n\})$$

$$\text{where } VP_i \stackrel{\mathsf{def}}{=} \{shvm(\mathcal{F}_{i,j}) \mid 1 \leq j \leq k_i\}$$

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

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

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

Example 7. Consider the family $\mathcal{F}_A$ from Example 1.

- In the first step of the decomposition of $\mathcal{F}_A$ into canonical form we obtain the common set $P = \{a_1, b_1\}$ and the family $\mathcal{F}_V = \{\{c_1, d_1, e_1\}, \{c_1, d_1, e_2\}, \{c_2, d_2, e_1\}, \{c_2, d_2, e_2\}, \{c_2, d_3, e_1\}, \{c_2, d_3, e_2\}\}$.
- In the next step, we analyze $\mathcal{F}_V$ to find that only artifact names c and d are correlated. Projecting $\mathcal{F}_V$ onto the two resulting equivalence classes $\{c, d\}$ and $\{e\}$ we obtain the two variation points $\mathcal{F}_1 = \{\{c_1, d_1\}, \{c_2, d_2\}, \{c_2, d_3\}\}$ and $\mathcal{F}_2 = \{\{e_1\}, \{e_2\}\}$.
- In the third step, we analyze $\mathcal{F}_1$ and see that two products share the artifact implementation c_2, which gives us the variants $\mathcal{F}_{1,1} = \{\{c_1, d_1\}\}$ and $\mathcal{F}_{1,2} = \{\{c_2, d_2\}, \{c_2, d_3\}\}$, and then analyze $\mathcal{F}_2$ to obtain the variants $\mathcal{F}_{2,1} = \{\{e_1\}\}$ and $\mathcal{F}_{2,2} = \{\{e_2\}\}$.

Only $\mathcal{F}_{1,2}$ 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 $\mathit{shvm}(\mathcal{F}_A)$ is the variability model $\mathcal{S}_{A2}$ in Figure 2.

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

Lemma 1. *For every simple family $\mathcal{F}$ we have:*

$$\mathit{family}(\mathit{shvm}(\mathcal{F})) = \mathcal{F}$$

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

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

$$\mathit{shvm}(\mathit{family}(\mathcal{S})) = \mathcal{S}$$

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 $\mathcal{F}$ and every well-formed variability model $\mathcal{S}$ we have:*

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

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.

```
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 useKeyboard();
    }

    public void payment() {
        cardPay(enterCard());
    }

    public static void main(String[] args) {
        (new CashDesk()).sale();
    }

    /* The implementation of the private
       methods, including methods
       writeReceipt, updateStock, cardPay,
       enterCard, and useKeyboard are not
       shown here.
     */
}
```

```
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 implementation of the private
       methods, including methods
       writeReceipt, updateStock, cardPay,
       enterCard, and useScanner are not
       shown here.
     */
}
```

Fig. 4. Products P_2 (left) and P_4 (right) from the Cash Desk product line

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 [37]. 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:

$$\mathcal{F}_{\mathsf{CashDesk}} = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$$

where:

$$P_1 = \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Keyboard}}, payment_{\mathsf{Cash}}\}$$
$$P_2 = \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Keyboard}}, payment_{\mathsf{Card}}\}$$
$$P_3 = \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Keyboard}}, payment_{\mathsf{CashOrCard}}\}$$
$$P_4 = \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Scanner}}, payment_{\mathsf{Cash}}\}$$
$$P_5 = \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Scanner}}, payment_{\mathsf{Card}}\}$$
$$P_6 = \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Scanner}}, payment_{\mathsf{CashOrCard}}\}$$
$$P_7 = \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{KeyboardOrScanner}}, payment_{\mathsf{Cash}}\}$$
$$P_8 = \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{KeyboardOrScanner}}, payment_{\mathsf{Card}}\}$$
$$P_9 = \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{KeyboardOrScanner}}, payment_{\mathsf{CashOrCard}}\}$$

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 form family $\mathcal{F}_{\mathsf{CashDesk}}$, and following steps similar

to those of Example 7, gives the following SHVM.

$$\begin{aligned}
\mathit{shvm}(\mathsf{CashDesk}) &= \left(\{\mathit{sale}_{\mathsf{CashDesk}}\}, \{@\mathsf{EnterProducts}, @\mathsf{Payment}\}\right) \\
\\
\mathit{where}\ @\mathsf{EnterProducts} &= \{\mathsf{Keyboard}, \mathsf{Scanner}, \mathsf{KeyboardOrScanner}\} \\
@\mathsf{Payment} &= \{\mathsf{Cash}, \mathsf{Card}, \mathsf{CashOrCard}\} \\
\\
\mathit{and}\quad \mathsf{Keyboard} &= \{\mathit{enterProd}_{\mathsf{Keyboard}}\} \\
\mathsf{Scanner} &= \{\mathit{enterProd}_{\mathsf{Scanner}}\} \\
\mathsf{KeyboardOrScanner} &= \{\mathit{enterProd}_{\mathsf{KeyboardOrScanner}}\} \\
\mathsf{Cash} &= \{\mathit{payment}_{\mathsf{Cash}}\} \\
\mathsf{Card} &= \{\mathit{payment}_{\mathsf{Card}}\} \\
\mathsf{CashOrCard} &= \{\mathit{payment}_{\mathsf{CashOrCard}}\}
\end{aligned}$$

The two variation points @EnterProducts and @Payment represent the variabilities of the cash desks. Variation point @EnterProducts has associated variants Keyboard, Scanner and KeyboardOrScanner, while variation point @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.

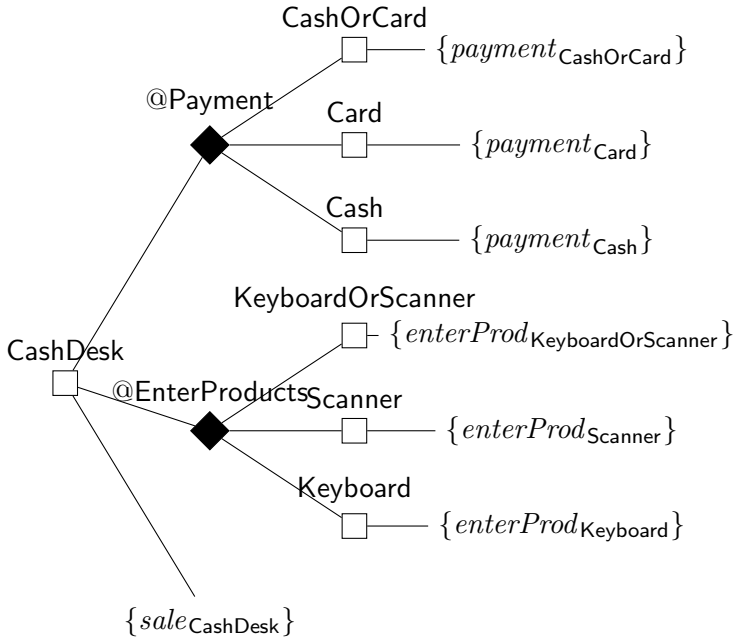

Fig. 5. The CashDesk hierarchical variability model (drawn sideways)

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 frame-

work 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 ϕ is the most general flow graph satisfying ϕ. Thus it can be used, for the purposes of verification, as a representation of *any* implementation of the component that satisfies ϕ. 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 $\mathcal{S}$ and global property ϕ, then all products of $\mathcal{S}$ satisfy ϕ.

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 $I^+, I^- \subseteq Meth$ 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*

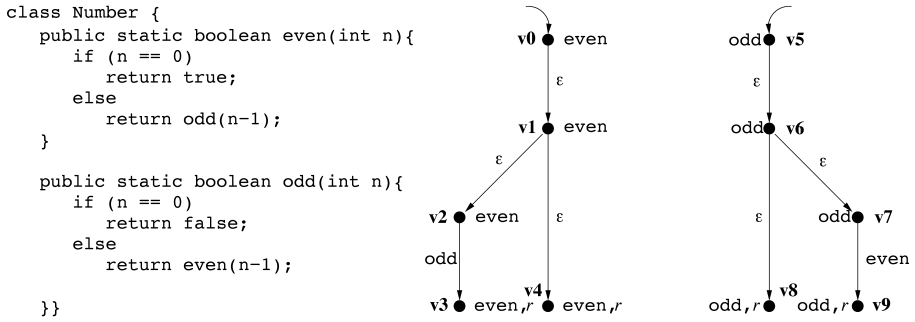

Fig. 6. A simple Java class and its flow graph

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* $(\{\texttt{even}, \texttt{odd}\}, \emptyset)$*, 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 τ for internal transfer of control, m_1 `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 call! m_2 when method m_2 is external (e.g., API methods), and m_2 `ret` m_1 respectively m_2 ret? m_1 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* $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$*, such that* $S_b = (V \cup I^-) \times V^*$*, i.e., states are pairs of control points* v *or required method names* m*, and stacks* σ*,* $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{m_1\ \mathsf{call!}\ m_2 \mid m_1 \in I^+, m_2 \notin I^+\} \cup \{m_2\ \mathsf{ret?}\ m_1 \mid m_1 \in I^+, m_2 \notin I^+\} \cup \{\tau\}$*,* $A_b = A$*,* $\lambda_b((v, \sigma)) = \lambda(v)$ *and* $\lambda_b((m, \sigma)) = m$*, and* $\rightarrow_b \subseteq S_b \times L_b \times S_b$ *is defined by the following rules:*

[transfer] $(v, \sigma) \xrightarrow{\tau} (v', \sigma)$ if $m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r$

[call] $(v_1, \sigma) \xrightarrow{m_1\ \texttt{call}\ m_2} (v_2, v_1' \cdot \sigma)$ if $m_1, m_2 \in I^+, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r,$ $v_2 \models m_2, v_2 \in E$

[ret] $(v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \texttt{ret}\ m_1} (v_1, \sigma)$ if $m_1, m_2 \in I^+, v_2 \models m_2 \wedge r, v_1 \models m_1$

[call!] $(v_1, \sigma) \xrightarrow{m_1\ \mathsf{call!}\ m_2} (m_2, v_1' \cdot \sigma)$ if $m_1 \in I^+, m_2 \in I^-, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r$

[ret?] $(m_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret?}\ m_1} (v_1, \sigma)$ 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* ε *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 9. One example run through its (branching, infinite-state) behavior, from an initial to a final

configuration, is:

$$(v_0,\varepsilon)\xrightarrow{\tau}(v_1,\varepsilon)\xrightarrow{\tau}(v_2,\varepsilon)\xrightarrow{\texttt{even call odd}}(v_5,v_3)\xrightarrow{\tau}(v_6,v_3)\xrightarrow{\tau}(v_8,v_3)\xrightarrow{\texttt{odd ret even}}(v_3,\varepsilon)$$

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

$$(v_0,\varepsilon)\xrightarrow{\tau}(v_1,\varepsilon)\xrightarrow{\tau}(v_2,\varepsilon)\xrightarrow{\texttt{even call! odd}}(\texttt{odd},v_3)\xrightarrow{\texttt{odd ret? even}}(v_3,\varepsilon)$$

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 [41].

Specification Language. To specify global and local properties we hare use the safety fragment of *linear temporal logic* (LTL) that uses the weak version of until[2].

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

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \mathtt{X}\,\phi \mid \mathtt{G}\,\phi \mid \phi_1\,\mathtt{W}\,\phi_2$$

where $p \in A_b$ denotes the set of atomic propositions.

Satisfaction on states $(\mathcal{M}_b, s) \models \phi$ is defined in the standard fashion [44] as validity of ϕ over all runs starting from state $s \in S_b$ in model $\mathcal{M}_b$. For instance, formula $\mathtt{X}\,\phi$ holds of state s in model $\mathcal{M}_b$ if ϕ holds in the second state of every run starting from s, while $\phi\ \mathtt{W}\ \psi$ holds in s if for every run starting in s, either ϕ holds in all states of the run, or ψ holds in some state of the run and ϕ holds in all previous states. Satisfaction of a formula ϕ in flow graph $\mathcal{G}$ with behavior $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$ is defined as satisfaction of ϕ 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 $\mathcal{S}$ satisfies a formula ϕ if the behavior $b(\mathcal{G}_p)$ of the flow graph $\mathcal{G}_p$ of every product $p \in \mathit{products}(\mathcal{S})$ satisfies ϕ.

[2]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 μ-calculus [25] with boxes and greatest fixed-points only. For details see again our previous work [19].

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 ψ and interface I consisting of provided and required methods, consider the class of all flow graphs with interface I satisfying ψ. A maximal flow graph for ψ and I is a flow graph $\mathcal{M}ax(\psi, I)$ 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 [19, 18].

Suppose a system with n components that are partitioned into two sets: The set of abstract components $\mathcal{G}_1, \ldots, \mathcal{G}_k$ specified with their local properties and interfaces $(\psi_1, I_1), \ldots, (\psi_k, I_k)$, and the set of concrete components $\mathcal{G}_{k+1}, \ldots, \mathcal{G}_n$. 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.

$$(1) \qquad \frac{\mathcal{G}_1 \models \psi_1 \;\; \cdots \;\; \mathcal{G}_k \models \psi_k \qquad \biguplus_{j=k+1,\ldots,n} \mathcal{G}_j \;\uplus\; \biguplus_{i=1,\ldots,k} \mathcal{M}ax(\psi_i, I_i) \models \phi}{\biguplus_{i=1,\ldots,n} \mathcal{G}_i \models \phi}$$

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 ϕ if (i) each specified (abstract) component $\mathcal{G}_i$ satisfies its respective local property ψ_i and (ii) the composition of the k maximal flow graphs $\mathcal{M}ax(\psi_i, I_i)$ with the flow graphs extracted from the code of the other components (concrete components) $\mathcal{G}_{k+1}, \ldots, \mathcal{G}_n$ satisfies ϕ.

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

[3]Our proof [19] is for global properties ϕ written in behavioral simulation logic and local properties ψ_i in structural simulation logic; here in the context of sLTL we use translations into the respective logic.

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 Φ, the top-level region of the SHVM is specified with Φ, and also every variation point VP of the SHVM is specified by a behavioral property ψ_{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 ϕ, perform the following two tasks:

(i) For every artifact name $a \in Art(M_C)$, extract the flow graph $\mathcal{G}_a$ from $Imp(a)$.

(ii) For all variation points VP_i with specification (ψ_{VP_i}, I_{VP_i}), construct the maximal flow graph $\mathcal{M}ax(\psi_{VP_i}, I_{VP_i})$. Then, compose the constructed graphs with the flow graphs of task (i), and model check the resulting flow graph against the region property ϕ, i.e.,

$$(2) \qquad \biguplus_{a \in Art(M_C)} \mathcal{G}_a \;\uplus\; \biguplus_{1 \le i \le n} \mathcal{M}ax(\psi_{VP_i}, I_{VP_i}) \;\models \phi$$

For properties given in sLTL, the behavior of $\mathcal{G}_{\mathcal{M}ax}$ 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 $\mathcal{S}$ be an SHVM with global property ϕ. If the verification procedure succeeds for $\mathcal{S}$, then $p \models \phi$ for all its products $p \in products(\mathcal{S})$.*

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 `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:

$$\varphi_{CD} \;=\; \mathit{sale} \rightarrow (\neg \mathit{payment}\ \mathtt{W}\ (\mathit{enterProd}\ \wedge\ r\ \wedge\ \mathtt{X}\ \mathit{sale}))$$

where the subformula $\mathit{enterProd}\ \wedge\ r\ \wedge\ \mathtt{X}\ \mathit{sale}$ captures a return from `enterProd` to `sale`.

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

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

$$\varphi_{EP} \;=\; \mathtt{G}\ \neg \mathit{payment}$$

- The interface of variation point `@Payment` is $I_P = (\{\texttt{payment}\}, \{\texttt{enterProd}\})$. 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:

$$\varphi_P \;=\; \mathtt{G}\ \neg \mathit{enterProd}$$

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

[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.

with the flow graph of method `sale`, and model check φ_{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 $\mathcal{F}$ we proceed as dictated by Definition 10. First we factor out the common implementations, if any, and then we proceed with the remainder $\mathcal{F}_V$. We identify the equivalence classes of the $C^*_{\mathcal{F}_V}$ relation using Union-Find structures. For each equivalence class $\mathcal{F}_i$ we identify the equivalence classes of the $N^*_{\mathcal{F}_i}$ relation. If there are no common implementations and each of the two equivalence relations has a single equivalence class, by Proposition 5 the family $\mathcal{F}$ is not simple and the program exits with an appropriate message. Otherwise, recursive calls are made on each of the equivalence classes of the $N^*_{\mathcal{F}_i}$ 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

[5]ProMoVer is available via the web interface `www.csc.kth.se/~siavashs/ProMoVer`

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].

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.

```
/**
* @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(){
...
```

Fig. 7. Annotations for variation point `@EnterProd` and its variant `Keyboard`

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

Table 1. Evaluation Results

Product Line	Depth	# Modules	# Products	t_{ind}[s]	t_{comp}[s]
`CD`	1	7	9	79	9
`CD/CH`	1	9	18	177	10
`CD/CT`	2	15	27	278	11
`CD/CH/CT`	2	17	54	652	12

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[6].

The results are summarized in Table 1 where `CD` denotes the product line of Example 8, `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 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

[6]The focus of the evaluation is on comparing the times required for verification, and not on the total times themselves.

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 formal-

ism, 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 $\bowtie$ 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 [50, 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. [7], Liu et al. [28], and Beek et al. [46] 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

[1] Agrawal R., T. Imielinski, A. Swami. Mining association rules between sets of items in large databases. In: Proceedings of the SIGMOD Conference, Washington, D.C., USA, 1993, 207–216.

[2] Alves V., C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, A. Rummler. An exploratory study of information retrieval techniques in domain analysis. In: Software Product Line Conference (SPLC), 2008, 67–76.

[3] Apel S., F. Janda, S. Trujillo, C. Kästner. Model Superimposition in Software Product Lines. In: International Conference on Model Transformation (ICMT), LNCS, Vol. **5563**, Springer, 2009, 4–19.

[4] Batory D., J. Neal Sarvela, A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transaction Software Engineering*, **30** (2004), No 6, 355–371.

[5] Bessling S., M. Huhn. Towards formal safety analysis in feature-oriented product line development. In: Jeremy Gibbons and Wendy MacCaull, editors, Foundations of Health Information Engineering and Systems, LNCS, Vol. **8315**, Springer Berlin Heidelberg, 2014, 217–235.

[6] Besson F., T. Jensen, D. Le Métayer, T. Thorn. Model checking security properties of control flow graphs. *J. of Computer Security*, **9** (2001), No 3, 217–250.

[7] Blundell C., K. Fisler, S. Krishnamurthi, P. Van Hentenryck. Parameterized Interfaces for Open System Verification of Product Lines. In: Proceedings of the 19th IEEE international conference on Automated software engineering, 2004, 258–267.

[8] Clarke D., M. Helvensteijn, I. Schaefer. Abstract delta modeling. Generative Programming and Component Engineering (GPCE), Springer, 2010.

[9] Classen A., P. Heymans, P.-Y. Schobbens, A. Legay, J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, 2010, 335–344.

[10] Czarnecki K., M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Generative Programming and Component Engineering (GPCE), LNCS, Vol. **3676**, Springer, 2005, 422–437.

[11] Czarnecki K., U. W. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[12] Czarnecki K., S. She, A. Wasowski. Sample spaces and feature models: There and back again. In: Proceedings of the Software Product Line Conference (SPLC), 2008, 22–31.

[13] Fantechi A., S. Gnesi. Formal Modeling for Product Families Engineering. In: Proceedings of the Software Product Line Conference (SPLC), IEEE, 2008, 193–202.

[14] Ganter B., R. Wille. Formal Concept Analysis: Mathematical Foundations. Springer, 1996.

[15] Gomaa H. Designing Software Product Lines with UML. Addison Wesley, 2004.

[16] Gruler A., M. Leucker, K. Scheidemann. Modeling and model checking software product lines. In: Formal Methods for Open Object-based Distributed Systems (FMOODS), LNCS, Vol. **5051**, Springer, 2008, 113–131.

[17] Grumberg O., D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, **16** (1994), No 3, 843–871.

[18] Gurov D., M. Huisman. Reducing behavioural to structural properties of programs with procedures. *Theoretical Computer Science*, **480** (2013), 69–103.

[19] Gurov D., M. Huisman, C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, **206** (2008), No 7, 840–868.

[20] Gurov D., B. M. Østvold, I. Schaefer. A hierarchical variability model for software product lines. In: Post-proceedings of ISoLA 2011 Workshops, *CCIS*, **336** (2012), 181–199.

[21] Haber A., H. Rendel, B. Rumpe, I. Schaefer, F. van der Linden. Hierarchical variability modeling for software architectures. In: Proceedings of the Software Product Line Conference (SPLC), IEEE, 2011, 150–159.

[22] Haugen Ø., B. Møller-Pedersen, J. Oldevik, G. K. Olsen, A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In: Proceedings of the Software Product Line Conference (SPLC), IEEE, 2008, 139–148.

[23] Jiang T., B. Ravikumar. Minimal nfa problems are hard. *SIAM J. Comput.*, 22 (1993), No 6, 1117–1141.

[24] Kleene S. Representation of events in nerve nets and finite automata. Automata Studies, 1956.

[25] Kozen D. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27 (1983), 333–354.

[26] Lauenroth K., K. Pohl, S. Toehning. Model checking of domain artifacts in product line engineering. In: Automated Software Engineering (ASE), IEEE, 2009, 467–481.

[27] Leavens G., E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin. JML Reference Manual, February 2007. Department of Computer Science, Iowa State University.
`http://www.jmlspecs.org`

[28] Liu J., S. Basu, R. R. Lutz. Compositional model checking of software product lines using variation point obligations. Automated Software Engineering (ASE), **18** (2011), No 1, 39–76.

[29] Loesch F., E. Ploedereder. Optimization of variability in software product lines. In: Software Product Line Conference (SPLC), 2007, 151–162.

[30] Maier D. The Theory of Relational Databases. Computer Science Press, 1983.

[31] Meyer A., L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In: Proceedings of the 13th Annual Symposium on Switching and Automata Theory SWAT'72, IEEE Computer Society, 1972, 125–129.

[32] Millo J.-V., S. Ramesh, S. Krishna, G. Narwane. Compositional verification of software product lines. In: Integrated Formal Methods (Eds E. Johnsen, Luigia Petre), LNCS, Vol. **7940**, Springer Berlin Heidelberg, 2013, 109–123.

[33] Niu N., S. Easterbrook. Concept analysis for product line requirements. In:Proceedings of the ACM International Conference on Aspect-Oriented Software Development (AOSD), 2009, 137–148.

[34] Noda N., T. Kishi. Aspect-Oriented Modeling for Variability Management. In: Proceedings of the Software Product Line Conference (SPLC), IEEE, 2008, 213–222.

[35] Pérez J., J. Díaz, C. C. Soria, J. Garbajosa. Plastic Partial Components: A solution to support variability in architectural components. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA), 2009, 221–230.

[36] Pohl K., G. Böckle, F. J. van der Linden. Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, 2005.

[37] Requirement Elicitation, August 2009. Deliverable 5.1 of project FP7-231620 (HATS). http://www.hats-project.eu

[38] Ryssel U., J. Ploennigs, K. Kabitzsch. Automatic variation-point identification in function-block-based models. In: Generative Programming and Component Engineering (GPCE), New York, USA, 2010, ACM, 23–32.

[39] Schaefer I., D. Gurov, S. Soleimanifard. Compositional algorithmic verification of software product lines. In: Postproceedings of the Intlernational Symposium on Formal Methods for Components and Objects (FMCO 2010), Vol. **6957**, LNCS, Springer, 2011, 184–203.

[40] Schaefer I., R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, K. Villela. Software diversity: state of the art and perspectives. *Software Tools for Technology Transfer (STTT)*, **14**(2012), No 5, 477–495.

[41] Schwoon S. Model-Checking Pushdown Systems. PhD thesis, Technische Universität München, 2002.

[42] Snelting G. Reengineering of configurations based on mathematical concept analysis. *ACM Transaction on Software Engineering and Methodology*, **5** (1996), 146–189.

[43] Soleimanifard S., D. Gurov, M. Huisman. Procedure-modular specification and verification of temporal safety properties. *Software and System Modeling*, **14**(2015), No 1, 83–100.

[44] Stirling C. Modal and Temporal Logics of Processes. Springer, 2001.

[45] Stockmeyer L., A. Meyer. Word problems requiring exponential time: Preliminary report. In: Proceedings of the ACM Symposium on the Theory of Computing (STOC), 1973, 1–9.

[46] Beek M., E. Vink. Towards Modular Verification of Software Product Lines with mCRL2. In: Part I of the Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Technologies for Mastering Change, LNCS, Vol. **8802**, Springer-Verlag New York, Inc. 2014, 368–385.

[47] Thüm T., S. Apel, C. Kästner, I. Schaefer, G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, **47** 2014, No 1, 1–45.

[48] Rob C. van Ommering. Software reuse in product populations. *IEEE Transaction on Software Engineering*, **31** (2005), No 7, 537–550.

[49] Völter M., I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: Proceedings of the Software Product Line Conference (SPLC), IEEE, 2007, 233–242.

[50] Von Rhein A., S. Apel, C. Kästner, T. Thüm, I. Schaefer. The pla model: on the combination of product-line analyses. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, 2013, 14–24.

[51] Ziadi T., L. Hélouët, J.-M. Jézéquel. Towards a UML Profile for Software Product Lines. In: Software Product Familiy Engineering (PFE), Vol. **3014**, LNCS, Springer, 2003, 129–139.

Appendix A. Proofs.

Proposition 1. *Let family $\mathcal{F}$ be simple. The following holds.*

(i) Let $a_i \in impls(\mathcal{F})$, and let $\mathcal{F}'$ be the projection of $\mathcal{F}$ on names $(\mathcal{F}) \setminus \{a\}$. a_i occurs in all products of $\mathcal{F}$, i.e., *$a_i \in \bigcap_{P \in \mathcal{F}} P$, iff $\mathcal{F} = \big\{\{a_i\}\big\} \bowtie \mathcal{F}'$. Then either $\mathcal{F}' = 1_{\mathcal{F}}$ and thus rule ($\mathcal{F}$1) applies, or else $\mathcal{F}'$ is simple and rule ($\mathcal{F}$2) 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., *$A_1 \times A_2 \subseteq \overline{C_{\mathcal{F}}}$, iff $\mathcal{F} = \mathcal{F}_1 \bowtie \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule ($\mathcal{F}$2) 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., *$\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_{\mathcal{F}}}$, iff $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule ($\mathcal{F}$3) applies in this case.*

P r o o f. The if parts of each case are immediate from Def. 3. The only-if parts are established by structural induction on the formation of $\mathcal{F}$.

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

(a) Let $\mathcal{F} = \big\{\{b_j\}\big\}$. Then $a_i = b_j$, and so $\mathcal{F} = \big\{\{a_i\}\big\} \bowtie 1_{\mathcal{F}}$.

(b) Let $\mathcal{F} = \mathcal{F}_1 \bowtie \mathcal{F}_2$ for simple $\mathcal{F}_1$ and $\mathcal{F}_2$ such that $\mathit{names}\,(\mathcal{F}_1) \cap \mathit{names}\,(\mathcal{F}_2) = \emptyset$. Assume w.l.o.g. that $a \in \mathit{names}\,(\mathcal{F}_1)$. Then $a_i \in \bigcap_{P \in \mathcal{F}_1} P$ and, by the induction hypothesis, $\mathcal{F}_1 = \big\{\{a_i\}\big\} \bowtie \mathcal{F}_1'$ where either $\mathcal{F}_1' = 1_{\mathcal{F}}$ or else $\mathcal{F}_1'$ is simple. In both cases, by the associativity of $\bowtie$, $\mathcal{F} = \big\{\{a_i\}\big\} \bowtie \mathcal{F}'$ for $\mathcal{F}' = \mathcal{F}_1' \bowtie \mathcal{F}_2$, and hence $\mathcal{F}'$ is simple.

(c) The case $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ for simple $\mathcal{F}_1$ and $\mathcal{F}_2$ such that $\mathit{names}\,(\mathcal{F}_1) = \mathit{names}\,(\mathcal{F}_2)$ and $\mathit{impls}(\mathcal{F}_1) \cap \mathit{impls}(\mathcal{F}_2) = \emptyset$ is not possible when $a_i \in \bigcap_{P \in \mathcal{F}} P$.

(ii) Let $\{A_1, A_2\}$ be a non-trivial partitioning of $\mathit{names}\,(\mathcal{F})$, let $\mathcal{F}_1$ and $\mathcal{F}_2$ be the projections of $\mathcal{F}$ on A_1 and A_2, respectively, and let $A_1 \times A_2 \subseteq \overline{C_{\mathcal{F}}}$. Again we consider three cases.

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

(b) Let $\mathcal{F} = \mathcal{F}_1' \bowtie \mathcal{F}_2'$ for simple $\mathcal{F}_1'$ and $\mathcal{F}_2'$ such that $\mathit{names}\,\big(\mathcal{F}_1'\big) \cap \mathit{names}\,\big(\mathcal{F}_2'\big) = \emptyset$. Let $A_1' = \mathit{names}\,\big(\mathcal{F}_1'\big)$ and $A_2' = \mathit{names}\,\big(\mathcal{F}_2'\big)$. If $A_1' = A_1$ then $A_2' = A_2$ and the result follows immediately. Otherwise, let $A_{1,1}' \stackrel{\mathsf{def}}{=} A_1' \cap A_1$, $A_{1,2}' \stackrel{\mathsf{def}}{=} A_1' \cap A_2$, $A_{2,1}' \stackrel{\mathsf{def}}{=} A_2' \cap A_1$ and $A_{2,2}' \stackrel{\mathsf{def}}{=} A_2' \cap A_2$. Then $\{A_{1,1}', A_{1,2}'\}$ and $\{A_{2,1}', A_{2,2}'\}$ are non-trivial partitionings of A_1' and A_2', respectively. Furthermore, $A_{1,1}' \times A_{1,2}' \subseteq \overline{C_{\mathcal{F}_1'}}$ and $A_{2,1}' \times A_{2,2}' \subseteq \overline{C_{\mathcal{F}_2'}}$. Then, by the induction hypothesis, $\mathcal{F}_1' = \mathcal{F}_{1,1}' \bowtie \mathcal{F}_{1,2}'$ and $\mathcal{F}_2' = \mathcal{F}_{2,1}' \bowtie \mathcal{F}_{2,2}'$ where, for all $i, j \in \{1, 2\}$, $\mathcal{F}_{i,j}$ is the projection of $\mathcal{F}_i$ on A_j and is simple. Then $\mathcal{F}_1 = \mathcal{F}_{1,1}' \bowtie \mathcal{F}_{2,1}'$ and $\mathcal{F}_2 = \mathcal{F}_{1,2}' \bowtie \mathcal{F}_{2,2}'$ are simple, and, by the associativity of $\bowtie$, $\mathcal{F} = \mathcal{F}_1 \bowtie \mathcal{F}_2$.

(c) The case $\mathcal{F} = \mathcal{F}_1' \cup \mathcal{F}_2'$ for simple $\mathcal{F}_1'$ and $\mathcal{F}_2'$ such that $\mathit{names}\,\big(\mathcal{F}_1'\big) = \mathit{names}\,\big(\mathcal{F}_2'\big)$ and $\mathit{impls}(\mathcal{F}_1') \cap \mathit{impls}(\mathcal{F}_2') = \emptyset$ is not possible when $\{A_1, A_2\}$ is non-trivial and $A_1 \times A_2 \subseteq \overline{C_{\mathcal{F}}}$.

(iii) Let $\{\mathcal{F}_1, \mathcal{F}_2\}$ be a non-trivial partitioning of $\mathcal{F}$ and let $\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_{\mathcal{F}}}$. Again we consider three cases.

(a) The case $\mathcal{F} = \big\{\{b_j\}\big\}$ is not possible when $\{\mathcal{F}_1, \mathcal{F}_2\}$ is non-trivial.

(b) The case $\mathcal{F} = \mathcal{F}'_1 \bowtie \mathcal{F}'_2$ for simple $\mathcal{F}'_1$ and $\mathcal{F}'_2$ such that $\mathit{names}\left(\mathcal{F}'_1\right) \cap \mathit{names}\left(\mathcal{F}'_2\right) = \emptyset$ is also not possible when $\{\mathcal{F}_1, \mathcal{F}_2\}$ is non-trivial and $\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_{\mathcal{F}}}$.

(c) Let $\mathcal{F} = \mathcal{F}'_1 \cup \mathcal{F}'_2$ for simple $\mathcal{F}'_1$ and $\mathcal{F}'_2$ such that $\mathit{names}\left(\mathcal{F}'_1\right) = \mathit{names}\left(\mathcal{F}'_2\right)$ and $\mathit{impls}(\mathcal{F}'_1) \cap \mathit{impls}(\mathcal{F}'_2) = \emptyset$. If $\mathcal{F}'_1 = \mathcal{F}_1$ then $\mathcal{F}'_2 = \mathcal{F}_2$ and the result follows immediately. Otherwise, let $\mathcal{F}'_{1,1} \stackrel{\mathsf{def}}{=} \mathcal{F}'_1 \cap \mathcal{F}_1$, $\mathcal{F}'_{1,2} \stackrel{\mathsf{def}}{=} \mathcal{F}'_1 \cap \mathcal{F}_2$, $\mathcal{F}'_{2,1} \stackrel{\mathsf{def}}{=} \mathcal{F}'_2 \cap \mathcal{F}_1$ and $\mathcal{F}'_{2,2} \stackrel{\mathsf{def}}{=} \mathcal{F}'_2 \cap \mathcal{F}_2$. Then $\{\mathcal{F}'_{1,1}, \mathcal{F}'_{1,2}\}$ and $\{\mathcal{F}'_{2,1}, \mathcal{F}'_{2,2}\}$ are non-trivial partitionings of $\mathcal{F}'_1$ and $\mathcal{F}'_2$, respectively. Furthermore, $\mathcal{F}'_{1,1} \times \mathcal{F}'_{1,2} \subseteq \overline{N_{\mathcal{F}'_1}}$ and $\mathcal{F}'_{2,1} \times \mathcal{F}'_{2,2} \subseteq \overline{N_{\mathcal{F}'_2}}$. Then, by the induction hypothesis, $\mathcal{F}'_{1,1}$, $\mathcal{F}'_{1,2}$, $\mathcal{F}'_{2,1}$ and $\mathcal{F}'_{2,2}$ are all simple, and hence $\mathcal{F}_1 = \mathcal{F}'_{1,1} \cup \mathcal{F}'_{2,1}$ and $\mathcal{F}_2 = \mathcal{F}'_{1,2} \cup \mathcal{F}'_{2,2}$ are simple, too. Furthermore, since $\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_{\mathcal{F}}}$ implies $\mathit{impls}(\mathcal{F}_1) \cap \mathit{impls}(\mathcal{F}_2) = \emptyset$, rule $(\mathcal{F}3)$ applies.

This concludes the proof. □

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

P r o o f. We show $sd'(\mathcal{S}) = |\mathit{impls}(\mathcal{S})|$ by structural induction. First, let $\mathcal{S}$ be a ground model with common set M_C. We have:

$$\begin{aligned} & sd'(M_C) & \\ = {} & |M_C| & \{\text{Def. 6}\} \\ = {} & |\mathit{impls}(M_C)| & \{\text{Def. 5}\} \end{aligned}$$

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

$$\begin{aligned} & sd'((M_C, \{VP_1, \ldots, VP_n\})) & \\ = {} & |M_C| + \Sigma_{1 \leq i \leq n} \Sigma_{1 \leq j \leq k_i} sd'(\mathcal{S}_{i,j}) & \{\text{Def. 6}\} \\ = {} & |M_C| + \Sigma_{1 \leq i \leq n} \Sigma_{1 \leq j \leq k_i} |\mathit{impls}(\mathcal{S}_{i,j})| & \{\text{Ind. hyp.}\} \\ = {} & |M_C \cup \textstyle\bigcup_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} \mathit{impls}(\mathcal{S}_{i,j})| & \{\text{Def. 7}\} \\ = {} & |\mathit{impls}((M_C, \{VP_1, \ldots, VP_n\}))| & \{\text{Def. 5}\} \end{aligned}$$

This concludes the proof. □

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 induced by the SHVM is bound by* $OR^{\frac{AND\cdot(AND^{ND}-1)}{AND-1}}$ *and is thus exponential in the size of the SHVM, which is bound by* $\dfrac{(OR\cdot AND)^{(ND+1)}-1}{OR\cdot AND-1}$.

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.

$$T(0) = T_0$$
$$T(n) = OR \cdot T(n-1)^{AND}$$
$$T(0) = T_0$$
$$T(n) = OR \cdot AND \cdot T(n-1) + 1$$

□

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

Proof. By structural induction. First, let $\mathcal{S}$ be a well-formed ground model with common artifact implementations M_C. In that case $family(M_C)$ 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 $\mathcal{S}$ be a well-formed variability model $(M_C, \{VP_1, \ldots, VP_n\})$ with variation points $VP_i = \{\mathcal{S}_{i,j} \mid 1 \le j \le k_i\}$. As the induction hypothesis, assume the result holds for all $\mathcal{S}_{i,j}$. Since $\mathcal{S}$ is well-formed, so are all $\mathcal{S}_{i,j}$ by Definition 7, and hence, by the induction hypothesis, all families $family(\mathcal{S}_{i,j})$ are simple. For every variation point VP_i we have

$$family(VP_i) = \bigcup_{1\le j\le k_i} family(\mathcal{S}_{i,j})$$

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

$$family(\mathcal{S}) = \{M_C\} \bowtie \prod_{1\le i\le n} family(VP_i)$$

by Definition 8. Further, by well-formedness constraint ($\mathcal{S}2$) of Definition 7, we have that $\mathit{names}(M_C) \cap \mathit{names}(VP_i) = \emptyset$ for all i, and $\mathit{names}(VP_{i_1}) \cap \mathit{names}(VP_{i_2}) = \emptyset$ whenever $i_1 \neq i_2$. Now, M_C is simple due to well-formedness constraint ($\mathcal{S}1$) of Definition 7 (see base case), and since all $\mathit{family}(VP_i)$ are simple, by formation rule ($\mathcal{F}2$) of Definition 3, $\mathit{family}(\mathcal{S})$ is also simple. □

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

Proof. For every i, by Proposition 1, $\mathcal{F}_i$ is simple. Furthermore, by condition ($\mathcal{C}2$), all names of $\mathcal{F}_i$ are correlated, and hence, by Proposition 1, $\mathcal{F}_i$ is not formed by rule ($\mathcal{F}2$). Since $\mathcal{F}$ is non-core, $\mathcal{F}_i$ is also non-core and is therefore formed by ($\mathcal{F}3$). Hence, again by Proposition 1, there are at least two equivalence classes of $\mathit{impls}(\mathcal{F}_i)$ w.r.t. implementation sharing $N^*_{\mathcal{F}_i}$, and thus $k_i \geq 2$.

That all $\mathcal{F}_{i,j}$ 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 $\mathcal{F}_{i,j}$ are strictly smaller is enforced through the formation rules for simple families from Definition 3: rule ($\mathcal{F}1$) requires the existence of a shared artifact implementation, rule ($\mathcal{F}2$) requires at least two equivalence classes on names, and rule ($\mathcal{F}3$) requires at least two equivalence classes on implementations, and thus the decomposition into canonical form is never trivial. □

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

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

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

$$\begin{aligned} & \mathit{shvm}(\{P\} \bowtie \textstyle\prod_{1\leq i\leq n} \bigcup_{1\leq j\leq k_i} \mathcal{F}_{i,j}) \\ = {} & (P, \{VP_1, \ldots, VP_n\}) && \{\text{Def. 10}\} \\ & \text{where } VP_i = \{\mathit{shvm}(\mathcal{F}_{i,j}) \mid 1 \leq j \leq k_i\} \end{aligned}$$

By Proposition 5, all $\mathcal{F}_{i,j}$ are simple and strictly smaller than $\mathcal{F}$ and hence, by the induction hypothesis, all $\mathit{shvm}(\mathcal{F}_{i,j})$ are well-formed variability models. Now, since $\mathcal{F}$ is in canonical form, conditions ($\mathcal{C}1$) to ($\mathcal{C}3$) hold, ensuring the well-formedness constraints ($\mathcal{S}1$) to ($\mathcal{S}3$), respectively, and hence also $\mathit{shvm}(\mathcal{F})$ is a well-formed variability model. □

Lemma 1. *For every simple family $\mathcal{F}$ we have:*

$$\mathit{family}(\mathit{shvm}(\mathcal{F})) = \mathcal{F}$$

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

$$\begin{aligned} & \mathit{family}(\mathit{shvm}(\{P\})) & \\ = \ & \mathit{family}(P) & \{\text{Def. 10}\} \\ = \ & \{P\} & \{\text{Def. 8}\} \end{aligned}$$

Next, let $\mathcal{F}$ 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 $\mathcal{F}$, and thus, by Proposition 5, for all $\mathcal{F}_{i,j}$. We have:

$$\begin{aligned} & \mathit{family}(\mathit{shvm}(\{P\} \bowtie \textstyle\prod_{1\le i\le n} \bigcup_{1\le j\le k_i} \mathcal{F}_{i,j})) & \\ = \ & \mathit{family}((P, \{VP_1, \ldots, VP_n\}))) & \{\text{Def. 10}\} \\ & \text{where } VP_i = \{\mathit{shvm}(\mathcal{F}_{i,j}) \mid 1 \le j \le k_i\} & \\ = \ & \{P\} \bowtie \textstyle\prod_{1\le i\le n} \bigcup_{1\le j\le k_i} \mathit{family}(\mathit{shvm}(\mathcal{F}_{i,j})) & \{\text{Def. 8}\} \\ = \ & \{P\} \bowtie \textstyle\prod_{1\le i\le n} \bigcup_{1\le j\le k_i} \mathcal{F}_{i,j} & \{\text{Ind. hyp.}\} \end{aligned}$$

This concludes the proof of the lemma. □

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

$$\mathit{shvm}(\mathit{family}(\mathcal{S})) = \mathcal{S}$$

Proof. By structural induction. First, let $\mathcal{S}$ be a ground model with common set M_C. We have:

$$\begin{aligned} & \mathit{shvm}(\mathit{family}(M_C)) & \\ = \ & \mathit{shvm}(\{M_C\}) & \{\text{Def. 8}\} \\ = \ & M_C & \{\text{Def. 10}\} \end{aligned}$$

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

$$\mathit{shvm}(\mathit{family}((M_C, \{VP_1, \ldots, VP_n\})))$$

$$
\begin{aligned}
&= \mathit{shvm}(\{M_C\} \bowtie \textstyle\prod_{1\le i\le n} \bigcup_{1\le j\le k_i} \mathit{family}(\mathcal{S}_{i,j})) && \{\text{Def. 8}\}\\
&= (M_C, \{VP'_1, \ldots, VP'_n\}) && \{\text{Def. 10}\}\\
&\quad \text{where } VP'_i = \{\mathit{shvm}(\mathit{family}(\mathcal{S}_{i,j})) \mid 1 \le j \le k_i\}\\
&= (M_C, \{VP'_1, \ldots, VP'_n\}) && \{\text{Ind. hyp.}\}\\
&\quad \text{where } VP'_i = \{\mathcal{S}_{i,j} \mid 1 \le j \le k_i\}\\
&= (M_C, \{VP_1, \ldots, VP_n\}) && \{\text{Def. } \mathcal{S}\}
\end{aligned}
$$

To justify the second step above we need to show that

$$\{M_C\} \bowtie \textstyle\prod_{1\le i\le n} \bigcup_{1\le j\le k_i} \mathit{family}(\mathcal{S}_{i,j})$$

is in canonical form. This is established as follows, using that $\mathcal{S}$ is simple. First, the restriction that variation points have at least two variants and the constraint ($\mathcal{S}3$) guarantee that just the artifact implementations in M_C and no other artifact implementations are shared by all products of $\mathcal{S}$, and thus condition ($\mathcal{C}1$) is satisfied.

Next, constraint ($\mathcal{S}2$) 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 ($\mathcal{S}3$) guarantee that artifact names implemented by the same variation point must be correlated, and thus condition ($\mathcal{C}2$) is satisfied.

And finally, constraint ($\mathcal{S}3$) 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 ($\mathcal{C}3$) is satisfied. This concludes the proof of the lemma. □

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

P r o o f. The proof is by induction on the structure of $\mathcal{S}$. For the base case, let $\mathcal{S}$ be a ground model with common set M_C. Assume the verification procedure succeeds for $\mathcal{S}$. It has then established: $\biguplus_{a\in Art(M_C)} \mathcal{G}_a \models \phi$. From this, and by soundness of rule (1), it follows that $M_C \models \phi$. Since $\mathit{products}(\mathcal{S}) = \{M_C\}$ in this case, we have $p \models \phi$ for all $p \in \mathit{products}(\mathcal{S})$.

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

$$(i) \quad \biguplus_{a \in Art(M_C)} \mathcal{G}_a \uplus \biguplus_{1 \leq i \leq n} \mathcal{M}ax(\psi_{VP_i}, I_{VP_i}) \models \phi$$

By the assumption, the verification procedure has also succeeded for all $\mathcal{S}_{i,j}$. 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:

$$\forall i : 1 \leq i \leq n. \, \forall j : 1 \leq j \leq k_i. \, \forall p \in \mathit{products}(\mathcal{S}_{i,j}). \, p \models \psi_{VP_i}$$

By Definition 8 we have $\mathit{products}(VP_i) = \bigcup_{1 \leq j \leq k_i} \mathit{products}(\mathcal{S}_{i,j})$, and hence:

$$(ii) \quad \forall i : 1 \leq i \leq n. \, \forall p \in \mathit{products}(VP_i). \, p \models \psi_{VP_i}$$

Also by Definition 8, we know that every product p of $\mathcal{S}$ 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 (1) follows that $p \models \phi$. This concludes the proof. □

Siavash Soleimanifard
Dilian Gurov
KTH Royal Institute of Technology
Stockholm, Sweden
e-mails: {siavashs,dilian}@csc.kth.se

Ina Schaefer
Technical University of Braunschweig
Braunschweig, Germany
e-mail: i.schaefer@tu-braunschweig.de

Bjarte M. Østvold
Norwegian Computing Center
Oslo, Norway
e-mail: bjarte@nr.no

Minko Markov
University of Sofia
Sofia, Bulgaria
e-mail: minkom@fmi.uni-sofia.bg

Received June 23, 2015
Final Accepted October 8, 2015