Compositional Algorithmic Verification of Software Product Lines*

Ina Schaefer[1], Dilian Gurov[2], and Siavash Soleimanifard[2]

[1] Technische Universität Braunschweig, Germany
i.schaefer@tu-braunschweig.de
[2] Royal Institute of Technology, Stockholm, Sweden
{dilian,siavashs}@csc.kth.se

Abstract. Software product line engineering allows large software systems to be developed and adapted for varying customer needs. The products of a software product line can be described by means of a *hierarchical variability model* specifying the commonalities and variabilities between the artifacts of the individual products. The number of products generated by a hierarchical model is exponential in its size, which poses a serious challenge to software product line analysis and verification. For an analysis technique to scale, the effort has to be linear in the size of the model rather than linear in the number of products it generates. Hence, efficient product line verification is only possible if *compositional* verification techniques are applied that allow the analysis of products to be *relativized* on the properties of their variation points. In this paper, we propose simple hierarchical variability models (SHVM) with explicit variation points as a novel way to describe a set of products consisting of sets of methods. SHVMs provide a trade–off between expressiveness and a clean and simple model suitable for compositional verification. We generalize a previously developed compositional technique and tool set for the automatic verification of control–flow based temporal safety properties to product lines defined by SHVMs, and prove soundness of the generalization. The desired property relativization is achieved by introducing variation point specifications. We evaluate the proposed technique on a number of test cases.

1 Introduction

System diversity is prevalent in modern software systems. Systems simultaneously exist in many different variants in order to adapt to their application context. Software product line engineering [23] aims at developing a set of systems variants with well-defined commonalities and variabilities by managed reuse in order to decrease time to market and to improve quality. During family engineering reusable core artifacts are developed, that are used to realize the actual products during application engineering.

* This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG) and the EU project FP7-231620 HATS.

The variability of the artifacts used for building a software product line can be described by a hierarchical variability model. In this model, on each level of hierarchy the commonalities of the product artifacts are specified in a common core, while the variabilities are represented by explicit variation points. Each variation point is associated with a set of variants that represent choices for realizing the variation points in different products. A variant can itself contain commonalities defined in a common core and variabilities specified by variation points introducing a new level of hierarchy.

Product line verification typically aims at establishing that *all* products of a product line satisfy a desired set of properties. The number of products defined by a hierarchical variability model is exponential in the size of the model. This explosion poses serious problems to ensuring the critical product requirements by static analysis or other formal verification techniques, and can render infeasible the verification of product lines by verifying all products individually. Formal verification techniques will only scale if their complexity is linear in the size of the hierarchical variability model rather than linear in the number of products. In order to achieve this scalability, these techniques have to be compositional, allowing to relativize the product properties towards properties of variation points.

In this paper, we generalize a previously developed compositional verification technique (and the corresponding tool set) for the automatic verification of control–flow based temporal safety properties [13, 15] to the compositional verification of hierarchically defined product lines. We propose simple hierarchical variability models (SHVM) as a novel way to specify the variability of product artifacts. SHVMs provide a clean and simple model facilitating compositional verification, while they are still sufficiently expressive for capturing variability. In this work, product artifacts consist of sets of public and private methods. In an SHVM, the artifact variability is defined by common core methods and explicit variation points on different hierarchical levels. The properties that can be handled fully automatically specify illegal sequences of method invocations, such as improper usage of API methods, in terms of temporal logic formulas, abstracting from the computed data. Compositionality, and the ability to relativize global SHVM properties on local assumptions for the core methods and the variation points, is achieved by means of maximal flow graphs that are derived algorithmically from the local assumptions. The flow graphs replace the assumptions when verifying global properties. The local specifications of core methods are verified by extracting flow graphs from the method implementations and model checking the induced behaviors against their specification.

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. [4] and Liu et al. [20] propose compositional verification techniques based on assume–guarantee style reasoning for product features. Other model checking approaches for product lines [8, 10, 18, 5] 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.

The paper is organized as follows. In Section 2, we present SHVMs to hierarchically represent product lines. In Section 3, we describe the foundations of our compositional verification technique. In Section 4, we present the compositional verification procedure for product lines and prove its soundness. In Section 5, we present tool support and an evaluation of the compositional verification technique. In Section 6, we review related work and conclude the paper in Section 7.

2 Hierarchical Variability Modelling

A *product* in the context of this work is defined by a set of methods. Products are not necessarily closed, i.e., they may still require additional methods such as API methods. A method m from a set of methods $Meth$ is understood as a method definition, consisting of a method name, the types of the return value and the parameters, and its implementation (method body). The methods of a product are partitioned into *public* and *private* methods. Public methods are visible to the outside of the product, while private methods are only visible within products and can be viewed as a means of implementing the public methods. For a product, the methods defined in the product are called *provided*, while the called methods that are not provided themselves are referred to as *required*.

A *product line* PL is defined as a set of method sets $PL \subseteq 2^{Meth}$ and can be represented by a *hierarchical variability model*, with the common methods of all products captured by a *core* set of methods separated into public and private methods. The differences between the products are represented by *variation points*. To each variation point, a set of *variants* is attached. The variants represent different possibilities to realize the variability described by this variation point. A variant can either comprise a set of core methods or be a hierarchical variability model itself, i.e., consisting of core set of methods and a set of variation points. A product is derived by resolving the variabilities, i.e., by selecting variants at the variation points on all levels of hierarchy. An example is given later in this section in Figure 1.

Hierarchical variability modeling captures the variability of the artifacts that are used to build the products, called solution space variability in [7]. In this work, hierarchical variability modeling describes the variability of the methods implementing single products. This is in contrast to problem space variablity [7] which is mainly represented in terms of product features. Product features denote a user-visible product functionality and are merely labels without inherent semantical meaning. The valid combinations of product features can be described by feature models [16] and correspond to the valid member products. The tree-hierarchy in feature models usually describes the sub/super-feature relationship between product features, while the hierarchy in hiearchical variability models refers to the commonality and variability of the solution space artifacts.

In this paper, we introduce a variant of the hierarchical variability modelling approach called *simple* hierarchical variability model (SHVM). An SHVM is a hierarchical variability model that requires exactly one variant to be selected at each variation point to obtain a product. In an SHVM, there is no means for defining constraints between different variants and variation points to represent that the selection of a variant at one variation point requires a specific variant at another variation point to be selected, thus restricting the number of derivable products. These simplifications constitute a trade–off between providing an expressive representation of product variability and a clean model that allows straight–forward application of the compositional verification procedure described in Section 4. This trade–off is discussed at the end of this section.

Definition 1 (Simple Hierarchical Variability Model). *A* simple hierarchical variability model (SHVM) $\mathcal{S}$ *is inductively defined as:*

(i) a ground model *consisting of a* core *set of methods* $M_C = (M_{pub}, M_{priv})$*, partitioned into public and private methods* $M_{pub}, M_{priv} \subseteq Meth$*, or*
(ii) a pair $(M_C, \{VP_1, \ldots, VP_N\})$*, where* M_C *is defined as above and where* $\{VP_1, \ldots, VP_N\}$ *is a non-empty set of* variation points*. A variation point* $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$ *is a non–empty set of SHVMs. The members of a variation point are called* variants*.*

The *variant interface* of a pair $(M_C, \{VP_1, \ldots, VP_N\})$ is defined as a pair of public required and public provided methods. The set of public provided methods is the union of all sets of public provided methods in the core methods and the variation points. The set of public required methods is the union of all sets of public methods required by the core methods and by the variation points without the methods provided by the core methods or another variation point.

We assume the following two *well–formedness* constraints on SHVMs. First, all variants attached to a variation point have to provide and require the same sets of public methods. This pair of public required and provided methods is called the *variation point interface.* The constraint guarantees that all variants offer the same functionality in terms of the provided public methods while the implementation of the public methods may differ in the variant's private methods. Second, in order to enforce that a derivable product does not contain several methods with the same name, it is required that the provided methods in each variation point interface are disjoint from each other and the core method set.

Example 1. As a running example throughout this paper, we consider a product line of cash desks that is a simplified version of the trading system product line case study proposed in [24]. The cash desks process purchases by retrieving the prices for all items to be purchased and calculating the total price. After the customer has paid, a receipt is printed and the stock is updated accordingly. The commonality of all cash desks is that every purchase is processed following the same process. However, the cash desks differ in the way how the items are entered. Some cash desks allow entering products using a keyboard, others only provide a scanner, and a third group provides both options which can be chosen

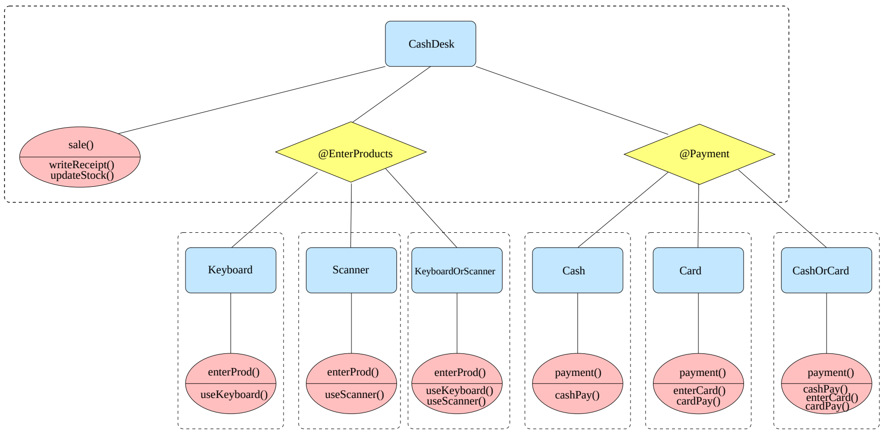

Fig. 1: The Cashdesk SHVM

by the cashier. 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. This set of cash desks is defined by an SHVM as follows:

$$\begin{aligned}\texttt{CashDesk} = (&(\{\texttt{sale}\}, \{\texttt{updateStock}, \texttt{writeReceipt}\}),\\ &\{@\texttt{EnterProducts}, @\texttt{Payment}\})\end{aligned}$$

$$\begin{aligned}\textit{where}\ @\texttt{EnterProducts} &= \{\texttt{Keyboard}, \texttt{Scanner}, \texttt{KeyboardOrScanner}\}\\ @\texttt{Payment} &= \{\texttt{Cash}, \texttt{Card}, \texttt{CashOrCard}\}\end{aligned}$$

$$\begin{aligned}\textit{and}\ \texttt{Keyboard} &= (\{\texttt{enterProd}\}, \{\texttt{useKeyboard}\})\\ \texttt{Scanner} &= (\{\texttt{enterProd}\}, \{\texttt{useScanner}\})\\ \texttt{KeyboardOrScanner} &= (\{\texttt{enterProd}\}, \{\texttt{useScanner}, \texttt{useKeyboard}\})\\ \texttt{Cash} &= (\{\texttt{payment}\}, \{\texttt{cashPay}\})\\ \texttt{Card} &= (\{\texttt{payment}\}, \{\texttt{enterCard}, \texttt{cardPay}\})\\ \texttt{CashOrCard} &= (\{\texttt{payment}\}, \{\texttt{cashPay}, \texttt{enterCard}, \texttt{cardPay}\})\end{aligned}$$

The common purchase process of all cash desks is modeled by the public core method `sale`. The private methods `updateStock` and `writeReceipt` represent internal details of the sale process. The two variation points `@EnterProducts` and `@Payment` represent the variabilities of the cash desks. The variation point `@EnterProducts` has the associated variants `Keyboard`, `Scanner` and `Keyboard-OrScanner` for entering product by keyboard, by scanner or providing both options. Both provide the public method `enterProd` that is internally realized by the different private methods `useKeyboard`, `useScanner` or their combination. Similarly, the variation point `@EnterProducts` has the associated variants `Cash`, `Card` and `CashOrCard` that provide the public method `payment` which is internally realized by different private methods in the respective variants.

An SVHM can be seen as a tri–partite directed *graph* having an SHVM–node as root, where SHVM–nodes have one core methods leaf child (split in public and private methods) and optional VP–node children that have two or more SHVM–node children. For the cashdesk example, a graphical presentation is shown in Figure 1. In the figure, SHVM-nodes are depicted by rounded boxes, core methods nodes by ovals, and VP–nodes by diamonds. The dotted rounded boxes depict what we call *modules* of the SHVM, defining the boundaries between SHVMs at different levels of hierarchy. The *size* of an SHVM is defined as the number of modules in its graph.

An SHVM *induces* a set of products P through all possible ways of resolving the variabilities of the SHVM. Variability resolution means to recursively select exactly one variant for each variation point. The set of products induced by a ground model containing only core methods is the singleton set comprising the set of core methods (and, thus, representing one product). The set of products induced by a variation point is the union of the product sets induced by its variants. Finally, the set of products induced by an SHVM with a non–empty set of variation points is the set of all products consisting of the core methods and of exactly one product from the set induced by each variation point.

Definition 2 (Variability Resolution). *Let $\mathcal{S}$ be an SHVM as defined above. The set $products(\mathcal{S}) \subseteq 2^{Meth}$ induced by $\mathcal{S}$ is inductively defined as follows:*

$$\begin{aligned} products(M_C) &= \{M_C\} \\ products(VP) &= \textstyle\bigcup_{\mathcal{S}\in VP} products(\mathcal{S}) \\ products(M_C, \{VP_1, \ldots, VP_N\}) &= \Big\{M_C \cup \textstyle\bigcup_{1\leq i\leq N} M_i \mid M_i \in products(VP_i)\Big\} \end{aligned}$$

Example 2. The SHVM defined in Example 1 induces the products:

$$products(\mathtt{CashDesk}) = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$$

where:

$$P_1 = \left\{\begin{array}{l}\mathtt{sale}, \mathtt{updateStock}, \mathtt{writeReceipt}, \mathtt{enterProd}_{\mathtt{Keyboard}}, \\ \mathtt{useKeyboard}, \mathtt{payment}_{\mathtt{Cash}}, \mathtt{cashPay}\end{array}\right\}$$

$$P_2 = \left\{\begin{array}{l}\mathtt{sale}, \mathtt{updateStock}, \mathtt{writeReceipt}, \mathtt{enterProd}_{\mathtt{Scanner}}, \\ \mathtt{useScanner}, \mathtt{payment}_{\mathtt{Cash}}, \mathtt{cashPay}\end{array}\right\}$$

$$P_3 = \left\{\begin{array}{l}\mathtt{sale}, \mathtt{updateStock}, \mathtt{writeReceipt}, \mathtt{enterProd}_{\mathtt{KeyboardOrScanner}}, \\ \mathtt{useKeyboard}, \mathtt{useScanner}, \mathtt{payment}_{\mathtt{Cash}}, \mathtt{cashPay}\end{array}\right\}$$

$$P_4 = \left\{\begin{array}{l}\mathtt{sale}, \mathtt{updateStock}, \mathtt{writeReceipt}, \mathtt{enterProd}_{\mathtt{Keyboard}}, \\ \mathtt{useKeyboard}, \mathtt{payment}_{\mathtt{Card}}, \mathtt{enterCard}, \mathtt{cardPay}\end{array}\right\}$$

$$P_5 = \left\{\begin{array}{l}\mathtt{sale}, \mathtt{updateStock}, \mathtt{writeReceipt}, \mathtt{enterProd}_{\mathtt{Scanner}}, \\ \mathtt{useScanner}, \mathtt{payment}_{\mathtt{Card}}, \mathtt{enterCard}, \mathtt{cardPay}\end{array}\right\}$$

$$P_6 = \left\{\begin{array}{l}\mathtt{sale}, \mathtt{updateStock}, \mathtt{writeReceipt}, \mathtt{enterProd}_{\mathtt{KeyboardOrScanner}}, \\ \mathtt{useKeyboard}, \mathtt{useScanner}, \mathtt{payment}_{\mathtt{Card}}, \mathtt{enterCard}, \mathtt{cardPay}\end{array}\right\}$$

$$P_7 = \left\{\begin{array}{l}\mathtt{sale}, \mathtt{updateStock}, \mathtt{writeReceipt}, \mathtt{enterProd}_{\mathtt{Keyboard}}, \\ \mathtt{useKeyboard}, \mathtt{payment}_{\mathtt{CashOrCard}}, \mathtt{cashPay}, \mathtt{enterCard}, \mathtt{cardPay}\end{array}\right\}$$

$$P_8 = \left\{\begin{array}{l}\texttt{sale}, \texttt{updateStock}, \texttt{writeReceipt}, \texttt{enterProd}_{\texttt{Scanner}}, \\ \texttt{useScanner}, \texttt{payment}_{\texttt{CashOrCard}}, \texttt{cashPay}, \texttt{enterCard}, \texttt{cardPay}\end{array}\right\}$$

$$P_9 = \left\{\begin{array}{l}\texttt{sale}, \texttt{updateStock}, \texttt{writeReceipt}, \texttt{enterProd}_{\texttt{KeyboardOrScanner}}, \\ \texttt{useKeyboard}, \texttt{useScanner}, \texttt{payment}_{\texttt{CashOrCard}}, \\ \texttt{cashPay}, \texttt{enterCard}, \texttt{cardPay}\end{array}\right\}$$

To disambiguate methods with the same name, but coming from different variants, we add as subscript the name of the parent SHVM–node, for instance, $\texttt{enterProd}_{\texttt{Keyboard}}$ refers to the method `enterProd` of the variant `Keybord`.

For a given SHVM, let AND and OR denote the maximal 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 $\frac{(OR\cdot AND)^{(ND+1)}-1}{OR\cdot AND-1}$. These bounds are obtained in a routine fashion by solving the corresponding recurrence relations. Notice that in SHVMs with a small nesting depth as in the example above, the exponential blow–up in the number of products is not observed: With branching factors of 3 and a nesting depth of 1, we have at most 9 products, but 7 modules. However, adding just another level of hierarchy, e.g., variability in the accepted type of cards, immediately results in an explosion (see Section 5).

SHVMs are a simplification of hierarchical variability modeling supporting a straight-forward application of compositional reasoning with the following consequences to the expressiveness for product variability. In SHVMs, exactly one variant has to be selected at every variation point. If a combination of variants (including optional variants) should be selectable, the combination has to be modeled as an additional variant associated to this variation point. In most cases (and also in the example in this section), combinations of variants require additional glue code for the cooperation of the different behaviors such that combinations have to be represented as separate variants anyway. SHVMs do not allow requires/excludes constraints between variants. These constraints restrict the set of possible products that can be derived from a hierarchical variability model. The removal of these constraints results in an SHVM which defines products that would not exist otherwise. The requirement that all variants associated to a variation point have the same interface restricts the method variability of the variants. This can be alleviated to some extend by adding required methods to the interface, although they are not called by the variant, and adding dummy implementations for provided methods.

3 A Framework for Compositional Verification

This section outlines the theoretical framework for verification of temporal safety properties upon which our compositional verification technique for product lines (described in the next section) is based. It relies on our earlier work on compositional verification (see e.g. [13, 12]).

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 3 (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, see *e.g.* [3].

Definition 4 (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 methods according to the above definition 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, and it is *open* otherwise. Flow graph *composition* is defined as the disjoint union $\uplus$ of their method graphs.

Example 3. Figure 2 shows a simple Java class and the (simplified) flow graph it induces. It consists of two method graphs, for method `even` and method `odd`, respectively. Entry nodes are depicted as usual by incoming edges without source. Its interface is $(\{\texttt{even}, \texttt{odd}\}, \emptyset)$, thus the flow graph is closed.

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, $m_1\, \texttt{call}\, m_2$ for the invocation of method m_2 by method m_1 when method m_2 is provided by the program and $m_1\, \mathsf{call!}\, m_2$ when method m_2 is external, and $m_2\, \texttt{ret}\, m_1$ respectively $m_2\, \mathsf{ret?}\, m_1$ for the corresponding return from the call.

```
class Number {
   public static boolean even(int n){
      if (n == 0)
         return true;
      else
         return odd(n-1);
   }

   public static boolean odd(int n){
      if (n == 0)
         return false;
      else
         return even(n-1);

   }}
```

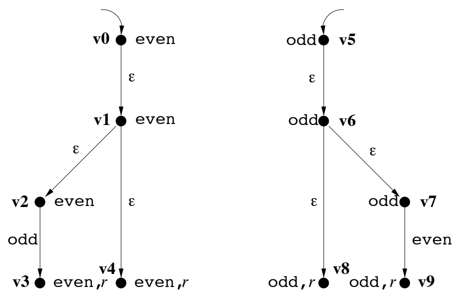

Fig. 2: A simple Java class and its flow graph

Definition 5 (Behavior). *Let $\mathcal{G} = (\mathcal{M}, E) : (I^+, I^-)$ be a flow graph such that $\mathcal{M} = (V, L, \to, A, \lambda)$. The behavior of $\mathcal{G}$ is defined as an initialized model $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \to_b, A_b, \lambda_b)$, 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, $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{m_1\ \mathsf{call!}\ m_2 \mid m_1 \in I^+, m_2 \in I^-\} \cup \{m_2\ \mathsf{ret?}\ m_1 \mid m_1 \in I^+, m_2 \in I^-\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$ and $\lambda_b((m, \sigma)) = m$, and $\to_b \subseteq S_b \times L_b \times S_b$ is defined by the following rules:*

$$\begin{array}{rll}
[\text{transfer}]\ (v,\sigma) \xrightarrow{\tau} (v',\sigma) & \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r \\
[\text{call}]\ (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2} (v_2, v_1' \cdot \sigma) & \text{if } m_1, m_2 \in I^+, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r, \\
 & \quad v_2 \models m_2, v_2 \in E \\
[\text{ret}]\ (v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1} (v_1, \sigma) & \text{if } m_1, m_2 \in I^+, v_2 \models m_2 \wedge r, v_1 \models m_1 \\
[\text{call!}]\ (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call!}\ m_2} (m_2, v_1' \cdot \sigma) & \text{if } m_1 \in I^+, m_2 \in I^-, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r \\
[\text{ret?}]\ (m_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret?}\ m_1} (v_1, \sigma) & \text{if } m_1 \in I^+, m_2 \in I^-, v_1 \models m_1
\end{array}$$

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 4. Consider the flow graph of Example 3. One example run through its (branching, infinite–state) behavior, from an initial to a final configuration, is:

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

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:

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

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, see [25].

We refine this program model to allow an explicit partitioning of method names into *public* and *private* ones, and introduce the notions of public interface and public behavior in order to abstract away from private methods which are used as a means of implementing the desired public behavior. On the flow graph level, such an abstraction is accomplished through inlining of private methods. For details the reader is referred to [13].

Specification The specification language for behavioral properties we use here is the safety–fragment of *Linear Temporal Logic* (LTL) that uses the weak version of until[3].

Definition 6 (Safety LTL). *The formulae 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 (see e.g. [28]) 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 on 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.

Compositional Verification 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.

[3] The theoretical underpinings 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 [17] with boxes and greatest fixed–points only (for details see again [13]).

Using maximal models for compositional verification was first proposed in [11] for finite–state systems, and was generalized for flow graphs in [13, 12].

The main principle of compositional verification based on maximal flow graphs can be presented, for a system that is partitioned into k sets of methods, as a proof rule with $k+1$ premises:

$$\frac{\mathcal{G}_1 \models \psi_1 \quad \cdots \quad \mathcal{G}_k \models \psi_k \qquad \biguplus_{i=1,\ldots,k} \mathcal{M}ax(\psi_i, I_i) \models \phi}{\biguplus_{i=1,\ldots,k} \mathcal{G}_i \models \phi} \tag{1}$$

The principle states that the composition of the sets of methods with the respective interfaces $\mathcal{G}_1 : I_1, ..., \mathcal{G}_k : I_k$ satisfies a global property ϕ if for some local properties ψ_i satisfied by the corresponding sets of methods $\mathcal{G}_i$, the composition of the maximal flow graphs for ψ_i and I_i satisfies property ϕ.

As we prove in [13], the rule is sound and complete when interfaces describe all provided and required methods, and is sound in the context of the private method abstraction mentioned earlier.

4 Compositional Verification of SHVMs

In this section we propose a compositional reasoning approach that is linear in the number of modules in the SHVM description of the product line, rather than linear in the number of generated products (which is exponential in the number of modules). This approach is an instantiation of the compositional verification principle presented above to SHVMs.

For every module $(M_C, \{VP_1, \ldots, VP_N\})$ in the SHVM, a specification has to be provided in order to allow for compositional verification. This comprises a specification for every public method $m \in M_{pub}$ by a public behavioral property ψ_m and a public interface $I_m = (I_m^+, I_m^-)$ declaring the names of the publicly provided and required methods, a specification for every variation point VP_i by a behavioral property ψ_{VP_i} and a public interface I_{VP_i}, and a specification of the SHVM node itself by a behavioral property ϕ and a public interface I. The SHVM nodes of variants attached to a variation point inherit the corresponding variation point specification. The lop–level SHVM is specified by the global product property that is to be verified. Our verification procedure for SHVMs is as follows.

Verification Procedure For every module $(M_C, \{VP_1, \ldots, VP_N\})$ of the SHVM, perform the following two independent tasks:

(i) For every public method $m \in M_{pub}$, extract the method graph $\mathcal{G}_m$ from the implementation of m, then inline the already extracted graphs of the private methods, and finally model check the resulting method graph $\mathcal{G}'_m$ against the specification ψ_m of m to establish $\mathcal{G}'_m \models \psi_m$. For the latter, we apply standard finite–state model checking.

(ii) For all public methods $m \in M_{pub}$ with specification (I_m, ψ_m), construct the maximal method graphs $\mathcal{M}ax(\psi_m, I_m)$, and for all variation points VP_i with specification (I_{VP_i}, ψ_{VP_i}), construct the maximal flow graphs $\mathcal{M}ax(\psi_{VP_i}, I_{VP_i})$. Then, compose the constructed graphs, resulting in flow graph $\mathcal{G}_{\mathcal{M}ax}$, and model check the latter against the SHVM property ϕ, i.e.,

$$\left(\biguplus_{m \in M_{pub}} \mathcal{M}ax(\psi_m, I_m) \uplus \biguplus_{1 \le i \le N} \mathcal{M}ax(\psi_{VP_i}, I_{VP_i}) \right) \models \phi \qquad (2)$$

For properties given in sLTL, we represent the behavior of $\mathcal{G}_{\mathcal{M}ax}$ as a PDS and use standard PDS model checking.

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

Theorem 1. *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 $p \in products(\mathcal{S})$.*

Proof. The proof is by induction on the structure of $\mathcal{S}$. For the base case, let $\mathcal{S}$ be a ground model, i.e., a core set of methods $M_C = (M_{pub}, M_{priv})$ with no variation points. Assume the verification procedure succeeds for $\mathcal{S}$. It has then established:

(i) $\mathcal{G}'_m \models \psi_m$ for all public methods $m \in M_{pub}$, and
(ii) $\biguplus_{m \in M_{pub}} \mathcal{M}ax(\psi_m, I_m) \models \phi$

From these, and by soundness of rule (1) refined for private method abstraction, it follows $M_C \models \phi$. Since $products(\mathcal{S}) = \{M_C\}$ in this case, we have $p \models \phi$ for all $p \in products(\mathcal{S})$.

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) $\mathcal{G}'_m \models \psi_m$ for all public methods $m \in M_{pub}$, and
(ii) $\left(\biguplus_{m \in M_{pub}} \mathcal{M}ax(\psi_m, I_m) \uplus \biguplus_{1 \le i \le N} \mathcal{M}ax(\psi_{VP_i}, I_{VP_i})\right) \models \phi$

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:

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

By Definition 2 we have $products(VP_i) = \bigcup_{1 \le j \le k_i} products(\mathcal{S}_{i,j})$, and hence:

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

Also by Definition 2, we know that every product p of $\mathcal{S}$ is the union of a core M_C and exactly one subproduct from every variation point. Due to (i), the public methods of M_C, after inlining the private ones, meet their respective specifications. Similarly, by (iii), all subproducts meet their respective specifications. Finally, by (ii) and from soundness of rule (1) refined for private method abstraction follows that $p \models \phi$. This concludes the proof. □

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

Example 5. To illustrate our compositional verification approach, we use the cashdesk product line described in Example 1. 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 methods 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 public core methods and variation points of the cashdesk SHVM. The specification of the `sale` method and the `@EnterProd` and `@Payment` variation points are as follows:

– The interface of method `sale` is $I_{sale} = (\{\mathtt{sale}\}, \{\mathtt{enterProd}, \mathtt{payment}\})$. In order to entail the global property, the local behavioral property that method `sale` (or, more precisely, its method graph as an open flow graph) has to satisfy is that it has to have invoked method `enterProd` and returned from the call before it can invoke method `payment`, after the return from which no more methods are invoked. Formally, this can be expressed by the sLTL formula:

$$\varphi_{sale} \;=\; \mathit{sale}\ \mathtt{W}'\ \mathit{enterProd}\ \mathtt{W}'\ \mathit{sale}\ \mathtt{W}'\ \mathit{payment}\ \mathtt{W}'\ (\mathtt{G}\ \mathit{sale})$$

where the derived temporal operator $\phi\ \mathtt{W}'\ \psi$ abbreviates $\phi \wedge (\phi\ \mathtt{W}\ \psi)$ and is by convention right–associative.

– 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 calls the `payment` method, neither directly nor via a call to one of its non-public methods. Formally, this property can be expressed by the formula[4]:

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

– 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 calls the `enterProd` method:

$$\varphi_{P} \;=\; \texttt{G}\,\neg \mathit{enterProd}$$

The variants `Keyboard`, `Scanner`, `KeyboardOrScanner`, `Cash`, `Card` and `CashOrCard` inherit the specification of their SHVM node from the respective variation point specification. The specification of the public methods `enterProd` and `payment` is similar to the specification of the `@EnterProd` and `@Payment` variation points.

Next, we have to verify that all public methods satisfy their behavioral property. For the `sale` method, we have to inline the private methods `writeReceipt` and `updateStock` to obtain the method graph of the sale method, Then we check that the method graph satisfies the property φ_{sale} by finite-state model checking. Similarly, we verify the `enterProd` and `payment` methods defined in the variants `Keyboard`, `Scanner`, `KeybordOrScanner`, `Cash`, `Card` and `CashOrCard`.

Finally, we have to establish that all SHVMs satisfy their SHVM specification. For the top–level SHVM, we construct the maximal models for the specifications of the variation points `@EnterProducts` and `@Payment` and for the public method φ_{sale}, and model check φ_{CD} against the composition of these maximal models. The properties of the variants `Keyboard`, `Scanner`, `KeyboardOrScanner`, `Cash`, `Card` and `CashOrCard` are easy to verify because each of them contains only one public method. A maximal model for the specification of this public method is constructed and checked against the inherited variation point property.

5 Tool Support and Evaluation

ProMoVer [26] 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 public interfaces. All interfaces, local and global properties are provided to the tool as assertions in the form of program annotations. ProMoVer accepts a JML–like syntax for

[4] This and the following property would trivialise if we specified the set of required methods to be empty. For now, however, our tool does not check interfaces.

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

```
/**
* @variation_point:
*   EnterProd
* @variation_point_interface:
*   provided enterProd
* @variation_point_ltl_prop:
*   G ! payment
* @variants:
*     Keyboard,Scanner,
*     KeyboardOrScanner
*/
```

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

/**@core: Keyboard
* @local_interface:
*   required
* @local_ltl_prop: G ! payment
*/
public int enterProd(){
...
```

Fig. 3: Annotations for variation point `@EnterProd` and its variant `Keyboard`

annotations (cf. [19]) as special comments called *pragmas*. To simplify the specification of local properties, ProMoVer provides a facility for extracting local properties from source code. Further, it 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 refered to [27].

We have adapted ProMoVer for verifying properties of SHVMs according to the compositionality principle described in Section 4. For this adaptation, we have extended the annotation language to support the definition of core methods, 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 analysed is represented by annotations. The product property, the variation point properties and the specifications of the public core methods are also provided by annotations. Figure 3 shows in the left column the annotation for the `@EnterProd` variation point, while the annotations for its `Keyboard` variant with core method `enterProd` are shown in the right column. ProMoVer fully automatically extracts the SHVM modules and the corresponding flow graphs from the annotated source code and performs the associated model checking tasks.

For evaluating our compositional verification approach, we considered the verification of the safety property explained in Example 5 for different versions of the trading system product line [24]. 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 1

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

Table 1: Evaluation Results

and extended it by an optional coupon handling functionality within the `sale` method, and a variation point for accepting different card types as a hierarchical refinement of variant `Card`. For each product line, we compared the time required to verify all induced products individually with the time for compositional verification. The experiments were performed on a SUN SPARC machine[6].

The results are summarized in Table 1 where `CD` denotes the product line of Example 1, `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 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 by an SHVM satisfy a property such that verification procedure fails, while the property is satisfied by the products defined by an hierarchical variablity model containing variant constraints. In this case, an additional check of the excluded products would be required.

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

6 Related Work

The existing approaches to represent product line variability on the artifact level can be classified into three main directions [30]. First, annotative approaches consider one model representing all products of a product line. Variant annotations, e.g., using UML stereotypes [31, 9], presence conditions [6], or separate variability representations, such as orthogonal variability models [23], define which parts of the model have to be removed to derive the model of a concrete product. Second, compositional approaches [2, 30, 21, 1] associate product fragments with product features which are composed for particular feature configurations. Third, transformational approaches, such as [14], represent variability by rules determining how modelling elements of a base model have to be replaced for a particular product model.

In this paper, we pursue an alternative approach to model the variability of a software product line by hierarchical variability modelling in SHVMs. Similar approaches are only pursued for modeling the variability of components contained in a software architecture. Plastic partial components [22] capture component variability by extending partially defined components with variation points and associated variants. However, variants cannot contain variable components, and thus the model is not fully hierarchical. In the Koala component model [29], component variability is defined by designated linguistic concepts, called diversity interfaces and switches, but these are fixed in a given component architecture.

Most approaches for 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. Existing model checking techniques are adapted to deal with optional behavior defined by variant annotations. For instance, in [8], modal transition systems are extended by variability operators from deontic logic. In [10], the process calculus CCS is extended with a variant operator to represent a family of processes. In [18], transitions of I/O-automata are related to variants. In [5], product families are modeled by transition systems where transitions are labelled with features, so that state reachability modulo a set of features can be computed.

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. [4] and Liu et al. [20] propose techniques for compositional verification of product features and are the only existing compositional verification techniques for product families in the literature so far. 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. However, 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.

7 Conclusion

We present a novel hierarchical variablility model for software product lines, in which the variability of products in terms of sets of public and private methods is specified by defining common core methods and variation points at different hierarchical levels. The model allows to adapt a previously developed method and tool set for compositional verification of procedural programs such that the exponential blow–up required for verifying all products individually is avoided: The number of verification tasks resulting from our method is linear in the size of the variablity model rather than in the number of products. This is achieved by the introduction of variation point specifications on which product properties are relativized, and the construction of 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 variablity model and providing the necessary specifications.

Our first experiments with the tool show a dramatic gain in performance even for models with a low hierarchical depth. In future work, we plan to extend our hierarchical variability model with optional variants and constraints between variants in order to facilitate the direct verification of more expressive hierarchical variability models.

Acknowledgement We thank Afshin Amighi for his help with flow graph extraction, and Björn Terelius for his help with obtaining the analytical bounds.

References

1. S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model Superimposition in Software Product Lines. In *International Conference on Model Transformation (ICMT '09)*, volume 5563 of *LNCS*, pages 4–19. Springer, 2009.
2. D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.
3. F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *J. of Computer Security*, 9(3):217–250, 2001.
4. C. Blundell, K. Fisler, S. Krishnamurthi, and P. van Hentenryck. Parameterized Interfaces for Open System Verification of Product Lines. In *Automated Software Engineering (ASE '04)*, pages 258–267. IEEE, 2004.
5. A. Classen, P. Heymans, P. Schobbens, A. Legay, and J. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *International Conference on Software Engineering (ICSE '10)*, pages 335–344. IEEE, 2010.
6. Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Generative Programming and Component Engineering (GPCE '05)*, volume 3676 of *LNCS*, pages 422 – 437. Springer, 2005.

7. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.
8. A. Fantechi and S. Gnesi. Formal Modeling for Product Families Engineering. In *Software Product Line Conference (SPLC '08)*, pages 193–202. IEEE, 2008.
9. H. Gomaa. *Designing Software Product Lines with UML.* Addison Wesley, 2004.
10. A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *Formal Methods for Open Object-based Distributed Systems (FMOODS '08)*, volume 5051 of *LNCS*, pages 113–131. Springer, 2008.
11. O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
12. D. Gurov and M. Huisman. Reducing behavioural to structural properties of programs with procedures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, volume 5403 of *LNCS*, pages 136–150. Springer, 2009.
13. D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7):840–868, 2008.
14. Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Software Product Line Conference (SPLC '08)*, pages 139–148. IEEE, 2008.
15. M. Huisman and D. Gurov. CVPP: A tool set for compositonal verification of control-flow safety properties. In *Formal Verification of Object–Oriented Software (FoVeOOS '10)*, volume 6528 of *LNCS*, pages 107–121. Springer, 2010.
16. K. Kang, J. Lee, and P. Donohoe. Feature-Oriented Project Line Engineering. *IEEE Software*, 19(4), 2002.
17. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
18. K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *Automated Software Engineering (ASE '09)*, pages 269–280. IEEE, 2009.
19. G.. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, February 2007. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.
20. Jing Liu, Samik Basu, and Robyn R. Lutz. Compositional model checking of software product lines using variation point obligations. *Autom. Softw. Eng.*, 18(1):39–76, 2011.
21. N. Noda and T. Kishi. Aspect-Oriented Modeling for Variability Management. In *Software Product Line Conference (SPLC '08)*, pages 213–222. IEEE, 2008.
22. Jennifer Pérez, Jessica Díaz, Cristóbal Costa Soria, and Juan Garbajosa. Plastic Partial Components: A solution to support variability in architectural components. In *WICSA/ECSA*, pages 221–230, 2009.
23. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques.* Springer, 2005.
24. Requirement Elicitation, August 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
25. S. Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, Technische Universität München, 2002.
26. S. Soleimanifard, D. Gurov, and M. Huisman. Procedure–modular verification of control flow safety properties. In *Workshop on Formal Techniques for Java Programs (FTfJP '10)*, 2010.
27. S. Soleimanifard, D. Gurov, and M. Huisman. Promover: Modular verification of temporal safety properties. In *Software Engineering and Formal Methods (SEFM '11)*, 2011. To appear.

28. C. Stirling. *Modal and Temporal Logics of Processes.* Springer, 2001.
29. R.. van Ommering. Software reuse in product populations. *IEEE Trans. Software Eng.*, 31(7):537–550, 2005.
30. M. Völter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Software Product Line Conference (SPLC '07)*, pages 233–242. IEEE, 2007.
31. T. Ziadi, L. Hélouët, and J. Jézéquel. Towards a UML Profile for Software Product Lines. In *Software Product Familiy Engineering (PFE '03)*, volume 3014 of *LNCS*, pages 129–139. Springer, 2003.