Calculating and Modeling Common Parts of Software Product Lines

Alexander Gruler · Martin Leucker · Kathrin Scheidemann
Institut für Informatik
Technische Universität München
Germany
{gruler,leucker,scheidem}@informatik.tu-muenchen.de

Abstract

This paper builds on product line CCS (PL-CCS), an algebraic approach to modeling the behavior of software product lines. The semantics of PL-CCS specifications is given in terms of labeled transition systems for individual products as well as for the entire product line and can be derived automatically. In this paper, we extend PL-CCS with a concept for specifying dependencies, show how to integrate it into a development methodology for product lines and validate its practical applicability by modeling a typical reactive system from the automotive domain. Most importantly, due to the algebraic nature of our model, we can derive calculation laws that allow to compute common parts of a product line. The application of the corresponding calculation rules is illustrated in detail with an example. By this, we obtain a formal foundation for restructuring product lines.

1. Introduction

Undoubtedly, modularity, a suitable composition mechanism, and the ability to derive (configure) individual systems from a product line model, are widely accepted to be the key concepts in the design of a software product line (SPL). These concepts are usually emphasized in common characterizations for SPLs, such as in [4]:

A software product line is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets.

To formally reason about SPLs, a formal framework for modeling SPLs is necessary. In [8], we have started with developing such a formal framework: Product Line CCS (PL-CCS). In PL-CCS, the definition above is implemented in a process algebra setting. In particular, for an SPL, it focuses on modeling its common parts, the points in which they vary, and the interrelation between the common and variable parts. A key feature of PL-CCS is that the representation of an entire product line is given within a single model, from which individual systems can easily be derived. The semantics of PL-CCS specifications is given by means of structural operational semantics (SOS) rules [17]. Given a PL-CCS specification (also called program), the SOS rules allow to derive a labeled transition system for individual products as well as for the entire product line automatically.

In general, a PL-CCS model has a large number of possible configurations. But not all combinatorially possible configurations are rational for various reasons. For example, due to various kinds of non-functional requirements (e.g., implementational, market decisions, etc.) certain configurations are not desired and should not be allowed. Therefore, in this paper, we enrich our model introduced in [8] with a concept that allows to characterize the reasonable configurations by restricting the set of all possible configurations. Such a restriction is commonly achieved by introducing various kinds of dependencies between model artifacts of the SPL model. We provide a simple, yet expressively powerful notion of dependencies based on propositional logic.

More importantly, we argue that the ability to restructure an SPL is as essential as the characteristics mentioned above—but usually not taken into account. For example, in order to apply an SPL model effectively in industrial development, it has to offer a restructuring mechanism in order to match the derivable systems with the set of existing core assets (components), from which the systems shall be build of. Intuitively, thinking of a product line as a hierarchical model containing several variation points, “restructuring” corresponds to moving the variation points throughout the hierarchical specification towards the leaves or towards the root. From another point of view, this means
to model individual systems using a higher or lower degree of common parts. Given a powerful restructuring concept, a designer may first model the functionality offered by an SPL independently from the set of existing software components. In a second step, the functionality may then be grouped into entities, using the restructuring mechanism, fitting the behavior of existing components. Additionally, the entities can be mapped easily to (COTS) components, providing the required behavior. Another situation which usually requires a restructuring of the SPL model is the subsequent adding of new features or functionality into an existing SPL.

We show that the algebraic nature of PL-CCS enables us to calculate a restructuring of our behavioral SPL model. We derive calculation laws that allow to compute common parts of a product line or any set of similar systems. In this paper, we cover the theory as far as necessary, but our focus is to demonstrate the respective concepts. In particular, the application of the corresponding calculation rules is illustrated in detail with an example taken from the automotive domain. Finally, we briefly highlight the embedding of our framework within an existing software product line development process [6].

**Contributions** We summarize our contributions:

- We recall an algebraic specification of the general concepts characterizing a product line: PL-CCS.
- We extend PL-CCS by a formal dependency model.
- We establish calculation rules for restructuring SPL programs and show their correctness.
- We illustrate our framework and especially the restructuring mechanism for computing common parts of an SPL with an industrial example taken from the automotive domain.
- We integrate our framework into an existing development process for SPLs.

**Related Work** Despite their importance for the development of software intensive systems, formal modeling techniques for product lines are rare.

Larsen et al. defined a behavioral variability model for product line development based on modal I/O automata [11], which are an extension of Larsen’s and Thomsen’s Kripke modal transition systems [12]. Their aim is not to model product specific functional properties for configurations, but rather to verify the error free combinability of interfaces. Kripke modal transition systems are used in [5] to study the notion of behavioral conformance in the setting of software product lines.

Modeling software systems in an algebraic manner has a long standing tradition. Starting from Milner’s [14] and Hoare’s [9] seminal contributions introducing CCS and, respectively, CSP, various authors have contributed enriched frameworks. Notably examples are the works by Bergstra et al. on process algebra [2], on Module Algebra [1], or on Network Algebra for Synchronous and Asynchronous Dataflow [3]. Variability issues are studied in a process algebra setting in [18] and [13]. However, we are not aware of any process algebra approach for modeling SPLs.

**Outline** We recall the essentials of PL-CCS in the next section. In Section 3, we introduce our formal dependency model. Algebraic laws for restructuring product lines are derived and extensively illustrated in Section 4. We briefly integrate our framework into an existing development process for SPLs in Section 5 and draw our conclusions in Section 6.

## 2. Product Line CCS

In this section we recall an algebraic model for an SPL which allows to model the behavior of all systems which are integrated within the SPL. Individual systems can then be derived from the product line model by configuring all variation points.

The advantages to combine the specification of similar systems in one compound model, the product line, are obvious: By this, we can explicitly model and investigate the commonalities and differences of single systems in relation to similar systems. In particular, we can adjust the set of derivable systems as well as individual systems, which are derivable, by restructuring the entire product line model. In the following, we recall such a model and sketch its semantics.

Our model termed PL-CCS, which stands for product line CCS, builds on Milner’s Calculus of Communicating Systems (CCS, [14]). However, CCS does not cater for the specification of variable parts, which is essential for product lines. Therefore, in [8], we have enriched CCS with a variants operator ⊕ which allows to specify variation points and corresponding variants. In the following, we briefly recall the essence of PL-CCS. As shown in detail in [8], PL-CCS is well suited for defining software product lines. In the subsequent sections, we establish algebraic laws for PL-CCS, that offer practical means for restructuring an SPL and hereby to compute common parts of an SPL. For the integration of the model into the development process we refer to Section 5.
2.1. Calculus of Communicating Systems

Before presenting PL-CCS, let us introduce Milner’s Calculus of Communicating Systems (CCS) in an intuitive fashion. A more precise account is given in terms of PL-CCS (Section 2.2), whose concepts cover those of CCS. For a comprehensive introduction to CCS, we refer to [15].

CCS is based on the notion of processes, which work in a parallel manner and may communicate with each other, synchronously. The behavior of processes is defined in an algebraic manner, that means by terms, which are formed using operators. The communication of a process, i.e. the interaction with its environment, is modeled by means of actions. A communication action can be combined with a process using the prefixing operator “·”. For example, let α be a communication action (input or output) and P be a process, then α.P is also a process, which performs (communicates) the action α and subsequently behaves like process P. For combining processes to form compound processes, CCS offers two operators: + and ∥. The ∥ operator represents parallel execution of several processes (with possible synchronous intercommunication). The + operator represents a non-deterministic choice, i.e. the term A + B represents a compound process which is initially capable of behaving either like A or B. The selection between A and B is made non-deterministically depending on the current communication action. However, choosing to perform an initial action of A means that A + B will become A, pre-empting the further behavior as defined by B, and vice versa. Intuitively, the initial action determines for a compound process A1 + · · · + An which of its sub-processes will be executed.

Complex systems are build up by defining equations. An entire system is specified by a set of (possibly recursive) equations with a distinguished starting process. Let us illustrate the idea taking an example from the automotive domain, modeling the behavior of a windscreen wiper FWs for the front window.

\[
\begin{align*}
FWs & \triangleq \text{off.FWs} + \text{intv.Intv} + \text{perm.Perm} \\
\text{Intv} & \triangleq \text{wipe.WaitL} + \text{intv2.Intv2} + \text{off.FWs} \\
\text{Wait} & \triangleq \text{wait.WaitL} \\
\text{Intv2} & \triangleq \text{wipe.Wait + intv.Intv} \\
\text{Perm} & \triangleq \text{perm.Perm} + \text{off.FWs}
\end{align*}
\]

Figure 1. Specification of a front screen wiper FWs in its standard version.

FWs is a simple version of a windscreen wiper for the front window. It offers three different operation modes, FWs, Intv, Perm. They represent the situations where the windscreen wiper is

(i) turned off but waiting in its initial mode (FWs),

(ii) operates with intermittent periodic stops (Intv),

(iii) and operates continuously at regular speed without intermittent breaks (Perm).

Figure 1 shows how such a behavior is specified in CCS. In the initial mode FWs, the operation modes are triggered by the actions off, intv, and perm, respectively. In Perm, the system executes constantly single wiper arm movements (action wipe) unless stopped by an off action. The interval mode (Intv) behaves similarly but adds a rest period between successive wiper arm movements, which is modeled by the output action wait. Additionally, it offers a second interval mode Intv2 with a shorter interval period, which can be activated only consecutively from the basic interval mode Intv. The interval control corresponds to the one typically found in cars with a turn-switch for the wiper functionality. In any mode, the off action sets the wiper back to its initial mode FWs.

2.2. Product Line CCS

As CCS does not facilitate the specification of variable parts—being an essential property of product lines—we have enriched CCS with a variants operator ⊕ which allows to specify variation points and corresponding variants in [8]. In the following, we briefly recall the essence of PL-CCS. In the next section, we enrich PL-CCS by a model for specifying dependencies of variants. Moreover, in the subsequent sections, we establish algebraic laws for PL-CCS that offer practical means for restructuring an SPL.

PL-CCS—Syntax In this section we introduce the syntax of PL-CCS programs. Let Id be a finite set of process identifiers and Σ be a finite set of input actions. Usually, P, Q, P1, . . . range over process identifiers and a, b, . . . range over input actions. As in CCS, let A = Σ ∪ Σ ∪ {τ} represents the set of communication actions, where τ ∈ Σ represents the silent action, and, Σ = {a | a ∈ Σ} is the set of output actions. Usually, α, β, . . . range over communication actions. By Nil, we denote the atomic idle process.

The set P of all PL-CCS process expressions (or short processes) is generated by the following grammar:

\[ e ::= Q | Nil | a.e | e + e | e ⊕ e | e \parallel e | e[f] | e\backslash L \]  

where Q ∈ Id is a process identifier, α ∈ A is an action, L ⊆ A is a set of action labels, and f : A ↦→ A is a renaming function, i.e. a function respecting f(a) = f(a) and f(τ) = τ.
Thus, syntactically, PL-CCS extends CCS [16] only by the binary variants operator $\oplus$ which allows to specify variation points. The usage of the variants operator can be understood as offering a set of possible (alternative) variants realizing a variation point. More precisely, in PL-CCS a single variation point models the direct choice between exactly two alternative variants, denominated the right and the left variant. Note, that this does not limit the expressiveness of $\oplus$ since multiple variants can easily be achieved by nesting an appropriate set of variants operators accordingly.

The variants operator allows to build up an entire product line by combing individual PL-CCS programs as possible variants. In particular, for the calculation of common parts, the considered individual system are initially combined as variants under a variation point in order to apply the restructuring rules.

Note that the concept of optional parts is a special case of alternative variation. Consequently, an optional-operator (\langle \rangle) can easily be added to PL-CCS as the syntactical abbreviation:

$$(P) := P \oplus Nil$$

A process definition is an equation of the form $P \equiv e$, where $P \in Id$ is a process identifier and $e \in \mathcal{P}$ is a PL-CCS process. We specify the behavior of an entire product family by a PL-CCS program: A PL-CCS program $Prog$ is a tuple $\langle E, P_1 \rangle$, where $E$ is a finite set of process definitions and $P_1 \in Id$ is the distinguished main process identifier of $Prog$. Typically, we denote a PL-CCS program by listing its equations, assuming that the left-hand side of the first equation is the main process identifier. Thus, we usually write only the set of defining equations as shown below.

\[
P_1 \equiv e_1 \\
P_2 \equiv e_2 \\
\vdots \\
P_n \equiv e_n
\]

Well-formed PL-CCS programs. Our goal is to model software product lines which require only an a priori finite number of decisions taken at variation points when deriving a specific system, which is the case for all product lines relevant in practice. So far, however, as in CCS, PL-CCS allows the creation of new processes by using the parallel operator $\parallel$ within recursive process definitions. In combination with our $\oplus$-operator this may potentially result in an unbounded number of variation points.

To avoid this, we consider only a (syntactically) restricted subset of all PL-CCS programs. The syntactical restriction is achieved by three conditions: completeness, finitely configurable, and fully expanded, that are used to derive the notion of well-formed systems. The key property of well-formed system is that each variants operator $\oplus$ of a PL-CCS program can be indexed with a unique number $i \in \mathbb{N}$, resulting in operator $\oplus_i$. Thus, every finite PL-CCS program, i.e. one with a finite set of equations, has a finite number $n$ of variants operators. In essence, this restriction allows us to define a compositional semantics (see next paragraph) for well-formed PL-CCS programs, which is exactly what we require from a product line approach.

Configuring a product line, i.e. deriving individual systems, then boils down to choosing for each variation point $\oplus_i$ whether the left or right alternative is selected. Thus, a configuration can be given as a vector $\theta \in \{R, L\}^n$, where the $i^{th}$ entry holds the individual configuration of the $i^{th}$ variation point $\oplus_i$.

Partially configured systems—as e.g. relevant for model checking product lines [8]—may be given via a vector $\nu \in \{R, L, ?\}^n$, where the entry $?$ indicates that no selection for the respective variation point has been made so far. Certainly, the variants—and therefore the respective entries of $\theta$—are not independent. Our dependency model, as introduced in Section 3, addresses the issue of specifying dependencies between certain variants.

Recall the example of $FWs$ shown in Figure 1: the specification does not contain a variants operator and therefore describes a single system. If we combine $FWs$ with another—more comfortable—version of a front screen wiper $FWc$ (as we will do in detail in section 4.2), we obtain a (simple) product line $FWFam$ of front screen wipers, given as

$$FWFam \equiv FWs \oplus_1 FWc$$

Since the product line contains only one variation point, the configuration vector $\theta$ has only one entry. Choosing for example the standard variant $FWs$ means to select the left variant for the variation point $\oplus_1$, which would be reflected by the configuration $\theta = \langle L \rangle$.

A Labeled Transition System for SPLs The meaning of a PL-CCS program is defined by a single labeled transition system (LTS) modeling the behavior of an entire product family. In particular, combining the behavior of all derivable systems within one labeled transition system provides the basis for restructuring the system but also reduces the effort in model checking, by allowing to consider the commonalities between derivable systems. The LTS is a multimodal Kripke structure extending Kripke modal transition systems [12]. The semantics makes use of a suitable transition system which allows to model variable behavior:

A product-line transition system (PL-LTS) with $n$ variants operators is a tuple $\langle S, A, \Delta, \sigma \rangle$, where $S$ is a (countably, possibly infinite) set of states, $A$ is a set of communication actions, and $\Delta$ is a finite set of transition relations of the form $\alpha \cdot \nu \subseteq S \times S$, where $\alpha \in A$, $\nu \in \times \{R, L, ?\}^n$, and $\sigma \in S$ is the start state.
Thus, in a PL-LTS a transition from one state to another is labeled by an action $\alpha$ and an additional (partial) configuration vector $\nu$. However, a transition $s \xrightarrow{\alpha, \nu'} s'$ represents the set of all transitions $s \xrightarrow{\alpha, \nu'} s'$ with $\nu$ more general than $\nu'$: Given two vectors $\nu, \nu' \in \{R, L, ?\}^n$, we call $\nu$ more general than $\nu'$, denoted by $\nu \sqsubseteq \nu'$, if $\forall i \in \{1, \ldots, n\}$ : $(\nu_i = ?) \lor (\nu_i = \nu'_i)$. We say that $\nu$ characterizes the set of configuration vectors $\{\nu' \mid \nu \sqsubseteq \nu'\}$.

Let us now briefly sketch the idea of the semantics of PL-CCS programs (called unfolded semantics in [8]). Similar as for CCS, the labeled transition relation is defined by means of enriched structural operational semantics (SOS) rules. The states of the transition system are pairs of PL-CCS process expressions paired with a vector characterizing the configurations under which this state was reached. In order to keep track of the choices for the variants operators the original SOS rules (for CCS) are enriched with a vector $\nu$ characterizing the configuration vectors for every transition. By this, we basically label every transition with the information in which configurations the transition is present.

Except for the variants operator $\oplus$, the (original) CCS rules do not influence the construction of the vectors attached to the transitions and are therefore only adjusted in order to be capable of dealing with vectors. The respective rules are omitted for space reasons but are given in [8].

A sketch of an example for a PL-LTS obtained for a family defined by $P + (Q \oplus, S)$ is shown in Figure 3(a). Here, $\nu|_{i,j,X}$ represents the substitution of the $i^j$ entry of vector $\nu$ by the variants identifier $X \in \{L, R, ?\}$.

### 3. Modeling Dependencies in PL-CCS

In general, a PL-CCS model has a large number of possible configurations. But not all combinatorially possible configurations are rational for various reasons:

- Due to various kinds of non-functional reasons (e.g. implementational, market decisions, etc.) certain configurations are not desired and should not be allowed.
- Due to the hierarchical structure of our PL-CCS terms, some configurations are deceptive. Consider for example the simple term $((A \oplus_1 B) \oplus_2 C)$ and its four possible configurations:

<table>
<thead>
<tr>
<th>configuration</th>
<th>yields system</th>
<th>deceptive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(L, L)$</td>
<td>A</td>
<td>no</td>
</tr>
<tr>
<td>$(L, R)$</td>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>$(R, L)$</td>
<td>B</td>
<td>no</td>
</tr>
<tr>
<td>$(R, R)$</td>
<td>C</td>
<td>yes</td>
</tr>
</tbody>
</table>

Out of the four combinatorially possible configurations, the configurations $(L, R)$ and $(R, R)$ are deceptive, since they make a selection for $\oplus_1$ even though the whole term $A \oplus_1 B$—and thus neither of the variants $A$ or $B$—cannot be included in the resulting system because the right variant is chosen for the second variation point $\oplus_2$. Such deceptive configurations should be prohibited by appropriate dependencies.

Therefore, in this section, we enrich our model introduced in [8] with a concept that allows to characterize the reasonable configurations by restricting the set of all possible configurations. Such a restriction is commonly achieved by introducing various kinds of dependencies between model artifacts of the SPL model. In the following, we introduce a single, simple, but expressively powerful notion of dependencies based on propositional logic.

In general, a dependency is a relation between variants. This reflects the natural understanding in the context of product lines: The selection of a certain variant of one variation point determines also the choice for the variants of other variation points. Against the background of product line engineering, this is the most general form of defining dependencies, since the existence or absence of a variant is the fundamental statement that is expressed by a product line model.

A dependency model $DEP$ is given as a formula in propositional logic. More precisely, $DEP$ is a formula over atomic propositions which represent the existence of single variants. In order to model individual variants, we define for every variation point $\oplus_i$ two atomic propositions $L_i$ and $R_i$ with the meaning that its left or right variant is selected to be present in the resulting system, respectively. For a product line with $n$ variation points we define $AP = \{R_i \mid i \in \{1, \ldots, n\}\} \cup \{L_i \mid i \in \{1, \ldots, n\}\}$ to be the set of all such atomic propositions.

In general, $DEP$ can be an arbitrary propositional formula, but usually it will consist of an arbitrary conjunction or disjunction of sub-formulas representing the dependencies of an individual variant. An example for a very simple, though typical $DEP$-formula is:

$$DEP \equiv (R_4 \Rightarrow R_{17}) \land (R_2 \Rightarrow L_{42}) \land (L_2 \Rightarrow \neg L_8)$$

It states that whenever $R_4$, $R_2$ or $L_2$ is selected, the variants $R_{17}, L_{42}$ or $\neg L_8$ of the respective variation points $17$, $42$, and $8$, also have to be selected.

By allowing (the full) propositional logic with the described atomic propositions in order to model the dependencies, we can express all kind of imaginable dependencies, e.g. also more complex ones such as:

$$((R_4 \lor L_{42}) \land ((R_4 \land L_2) \Rightarrow (R_{17} \lor R_{18}))$$

It states that a valid configuration has to select at least one of the variants $R_4$ or $L_{42}$ in any case. Additionally, if both
variants \( R_4 \) and \( L_2 \) are selected simultaneously, also \( R_{17} \) or \( R_{18} \) has to be selected.

The advantages of expressing the dependencies as a formula in propositional logic are evident: (i) Logic offers the full expressiveness which is sufficient to embody all kinds of possible dependency relations. (ii) A logical formula can easily be fed into a model checker or theorem prover.

Since only either \( L \) or \( R \) can be selected for each variation point we additionally require a duality for the variants of any variation point \( i \): \( L_i \Rightarrow \neg R_i \) and \( R_i \Rightarrow \neg L_i \). These duality rules are included to every model \( DEP \) by default. Moreover, these dualities allow to eliminate every negation in \( DEP \). Together with the following tautologies 2 to 6, we can transform every \( DEP \) formula into a normal form of the kind:

\[
DEP \equiv \bigwedge_{i=1}^{m} (A_i \Rightarrow \bigvee_{j=1}^{n_i} B_{i,j})
\]

where \( A_i, B_{i,j} \in AP \) are atomic propositions.

\[
A \Rightarrow B \equiv (\neg A \lor B) \quad (2)
\]

\[
A \Rightarrow (B \land C) \equiv (A \Rightarrow B) \land (A \Rightarrow C) \quad (3)
\]

\[
A \Rightarrow (B \lor C) \equiv \neg A \lor B \lor C \quad (4)
\]

\[
(A \land B) \Rightarrow C \equiv A \Rightarrow (\neg B \lor C) \quad (5)
\]

\[
(A \lor B) \Rightarrow C \equiv (A \Rightarrow B) \land (A \Rightarrow C) \quad (6)
\]

The normal form reflects a natural understanding of dependency: The \( A_i \) are all those variants whose selection influences the remaining configuration in exactly the way as described by the respective conjunctions \( \bigvee_{j=1}^{n_i} B_{i,j} \).

The normal form offers several benefits: The transformation into the normal form directly supports the dependency methodology, since it allows to specify dependencies in any (initial) way while being able to transform them in the normal form afterwards. Moreover, minimizing the normal form using standard minimization techniques for propositional logic [10] helps to determine the set of all individual variants ranging over \( AP \) which are involved into a dependency relation at all.

Finally, we establish the connection between a (general) PL-CCS model and a dependency relation. In combination with a dependency relation, a PL-CCS model becomes applicable for modeling realistic product lines. In particular, a realistic SPL is a tuple consisting of a PL-CCS model and a corresponding dependency model.

A given configuration \( \theta \) of a product line \( PL \) is said to be valid according to a given dependency model \( DEP \), if it fulfills the restrictions imposed by \( DEP \). If we denote a configuration vector in a logical equivalent way as the set of atomic propositions in \( AP \) matching the entries of the vector, we can express this fact logically by requiring that \( \theta \) (denoted as a set of logical propositions) is a model (in the logical sense, i.e. an assignment for which the formula evaluates to true) for \( DEP \):

\[ \theta \models DEP \]

4. Calculating Restructurings with PL-CCS

In this section, we establish algebraic laws for PL-CCS that can be applied to restructure PL-CCS specifications. To illustrate our concepts, we extend the wiper example shown in Section 2. Recall that the standard wiper introduced in Figure 1 offers a plain interval mode. Let us now introduce a more comfortable version \( FWc \) of a windscreen wiper for the front window that offers similar functionality as \( FWs \), but includes an automatically adjusting wiper arm speed for the interval mode matching the current rain intensity. Its specification is given in Figure 2.

\[
\begin{align*}
FWc & \equiv \text{off} \cdot FWs + \text{intv} \cdot \text{IntvC} + \text{perm} \cdot \text{Perm} \\
\text{IntvC} & \equiv \text{wip} \cdot \text{WaitL} + \text{hvyRn} \cdot \text{Fast} + \text{off} \cdot FWc \\
\text{WaitL} & \equiv \text{wait} \cdot \text{Wait} \\
\text{Fast} & \equiv \text{wip} \cdot \text{wait} \cdot \text{Fast} + \text{off} \cdot FWc + \text{litRn} \cdot \text{IntvC} + \text{perm} \cdot \text{Perm} \\
\text{Perm} & \equiv \text{wip} \cdot \text{Perm} + \text{off} \cdot FWc
\end{align*}
\]

Figure 2. Specification of a front screen wiper \( FWc \) with comfort features.

Both versions, \( FWs \) and \( FWc \) shall be used as possible variants in a family of front wipers. Therefore we combine them as alternative variants under the same variation point:

\[
FWFam \equiv FWs \oplus FWc
\]

Before combining both processes under the same variation point, we have to ensure that the appearing process identifiers do not overlap. In this example, we therefore index every (clashing) identifier of the specification of \( FWc \) with the subscript \( c \). Note, that identical action identifiers represent identical actions, i.e. actions will not be renamed, since they are unique for all systems in the context of the entire product line.

4.1. Algebraic Laws

In order to compute the common parts of alternative processes, we observe the following algebraic laws. Similar to laws in various calculi, which lay the ground to e.g. solve equations, these laws define calculation rules and allow to restructure a PL-CCS model for a product line. In this section we will describe the theoretical foundations and proofs
for the laws. For an illustrative application we refer to Section 4.2.

In order to establish the connection between mandatory and variable parts, we observe several distributive laws:

**Theorem 4.1** (distributivity of non-deterministic choice). The non-deterministic composition with a process \( P \) distributes over the alternative selection of processes:

\[
P + (Q \oplus S) = (P + Q) \oplus (P + S)
\]  
(7)

This defines a left-distributivity. Notice that since \(+\) is commutative, this implies also right-distributivity and hence full distributivity of \(+\) over \(\oplus\).

**Proof.** Proof in analogue to the proof of Theorem 4.1: Both sides of the equation yield the same labeled transition system (modulo bisimulation).

**Theorem 4.3** (distributivity of action prefixing). Action prefixing distributes over the alternative selection of processes.

\[
a.P \oplus a.Q = a.(P \oplus Q)
\]  
(9)

**Proof.** Applying the SOS rules for the variants operator \(\oplus\) to both sides of the theorem directly yields identical labeled transition systems.

**Relevance of the distributive laws for product line engineering** From an algebraic perspective, the distributive laws make a rather unspectacular statement about the connection of different operators. From a product line perspective, the distributive laws describe an essential concept how to restructure the parts of an SPL. An outstanding property of the distributive laws is that they represent the connection between mandatory (not varying) and variable parts. More precisely, if we read both Theorems 4.1 and 4.2 from left to right, they express the formal relationship how mandatory parts can be distributed over variation points. Applied in the other direction from right to left, the theorems describe how identical (in terms of bisimulation) parts can be extracted from two alternative variants of the same variation point. This equals the extraction of the common part of two variants. We will illustrate these rules in Section 4.2.

Note, that the application of the distributive laws does not change the number of variation points nor does it change the assignment between a variant and its representation label \( R \) or \( L \) in the configuration, e. g. consider equation 7. If, without loss of generality (wlog), the right variant is selected, i.e. \( \theta_i = R \), the resulting system will always be \( P + S \), no matter which side of Equation 7 we use to perform the configuration (derive the system). In particular, choosing the right variant, i.e. setting \( \theta_i = R \), will always result in choosing process \( S \), independent of the application of the distributive law.
The process identifier $B$ side of the variation point, while identifiers). Additionally, every substitution is only substituted on the left-hand side of the term on which it appears and the right-hand side of the term on which it appears. However, the variants operator can be made commutative if we adjust the respective entry $\theta_i$ of the configuration together with respective propositions in the dependency model whenever we apply the commutativity law. For example for a chosen configuration $\theta_k = R$ for a variation point $P \oplus_k Q$, we get $\theta_k = L$ when changing to $Q \oplus_k P$ according to the commutativity law.

4.2. Example

For the simple product line $FWFam$ we can now compute common parts by applying the introduced algebraic laws. We start with checking for well-formedness and by numbering all appearing variants operators uniquely. For this example, we only have one variants operator, which is assigned number 1, resulting in $\oplus_1$.

A typical calculation starting with the main process equation of the family as given in Figure 4. Note that, sim-
ilar to applying the usual calculation rules when solving an equation in calculus, we only apply the defined algebraic rules in order to compute the desired common parts. In line (1), the defining equation for the family $FWFam$ is listed. Substituting the definitions for $FWs$ and $FWc$, we obtain line (2). Recall that overlapping process identifiers of system $FWc$ have been indexed by the subscript $c$. As $FWs$ is “on the left-hand side” of $\oplus$, we can substitute $FWs$ by $FWs \oplus FWc$, according to Theorem 4.4. Similarly, also $FWc$ can be substituted by $FWs \oplus FWc$, on the right-hand side. This yields line (3). Now, we may apply the distributive law for $+$ (Theorem 4.1) and factor out $\text{off.}(FWs \oplus FWc)$, obtaining line (4). Line (5) is obtained by replacing $Perm$ and $Perm_\_c$ by their defining equations, in which we have, similarly as for line (2), substituted $FWs$ and $FWc$ by $FWs \oplus FWc$, using Theorem 4.4. As $Perm$ and $Perm_\_c$ in the substituted form are defined in the same manner (bisimilar), we may resort to $Perm$. Using commutativity of $+$ and again Theorem 4.1 yields line (6). Finally, using the distributive law for action prefixing (Theorem 4.3), we obtain line (7).

2. $\text{perm.}(\text{wipe}.Perm + \text{off.}(FWFam))$

The restructured version of the product line can be derived from the result of the calculation, i.e. line (7) of Figure 4: Since line (7) equals line (1) we have:

$$FWFam \equiv \text{off.}(FWs \oplus FWc) + \text{perm.}(\text{wipe}.Perm + \text{off.}(FWs \oplus FWc)) + \text{intv.}\left(\text{Intv} \oplus \text{IntvC}\right)$$

For the common parts we substitute $FWs \oplus FWc$ by $FWFam$. Together with the process definitions of the variable parts $\text{Intv}$ and $\text{IntvC}$ we obtain the restructured version of the product line as shown in Figure 5.

5. Methodological Integration into the Development Process

In this section we show how the introduced PL-CCS model can be integrated into a development process for software product lines. SPLs can be modeled at various abstraction layers. Usually, a SPL is modeled at the level of mandatory or optional features, which are first and foremost any observable properties, comprising functional as well as non-functional ones.

In [6] we have introduced a layered framework which allows to model a SPL as a chain of different models at consecutive abstraction layers. The layered framework starts by formalizing requirements in a so-called Service Diagram and leads to a so-called Service Network which represents a logical architecture of the system. The Service Diagram formalizes the user-observable functionality which is offered by the system by means of stream-processing functions. In contrast, the Service Network gives an architectural view by modeling the system as a network of logical, asynchronously communicating, totally defined components. However, it still abstracts completely from any hardware or platform-specific implementation details. But since the behavior of all components is already totally defined, it is quite easy to derive an actual implementation from a Service Network.

Both service models are formally defined and have a clear semantics which is based on the notion of a service [7]. This lays the foundation for a seamless, formal transition between both models. However, the transition can not be performed automatically since the Service Diagram does not contain any structural information (number and kind of components).

Similar to the Service Diagram, the algebraic model introduced in this paper also allows to model the behavior of a family of systems as a product line. In fact, the behavior of a system can be specified in a mixed style using both models. But in contrast to the Service Diagram, due to the
6. Conclusion

We have presented a practical approach to model a SPL in a formal manner which allows to compute the common parts of a product line, in a well-defined way. Our framework lays the foundations of an algebraic approach to software product lines. We enriched our initial work on PL-CCS tailored to model checking [8] by a formal dependency model. Moreover, we have worked out important calculation laws useful for restructuring PL-CCS programs. The rules especially allow to derive common parts of an entire PL-CCS program. We have shown that our approach is applicable by giving an illustrative industrial example taken from the automotive domain. With the provided operational semantics we can easily understand the entire product line as well as derivable systems as a labeled transition system. Again, this allows to easily match individual components of our (algebraic) model with (implementation) artifacts, such as existing assets. Finally, we briefly sketched how our framework integrates into an existing development process for SPLs.

Our approach has a interesting surplus: Using our dependency model we can express non-functional requirements formally. Since the calculation laws do not alter the number of variation points, restructuring a PL-CCS program automatically respects these non-functional requirements.

The ability to compute commonalities in a SPL lays the foundation for the definition of a wide range of metrics which now can be based onto a formal model and consequently can make quantitative statements. Such metrics are essential, in particular for product line engineering, e.g. for prognosticating the benefit of introducing a SPL based on a certain set of existing assets, measuring the success of a product line, and so on. The development of metrics is part of our future work.

References