# A Formal Approach to Software Product Families

Martin Leucker and Daniel Thoma

Institute for Software Engineering and Programming Languages
Universität zu Lübeck

**Abstract.** Software product line engineering deals with the combined development of a family of similar software systems. These systems provide a similar set of features and should therefore share a large number of common components. We study the user perspective of features and the engineering perspective of components and present a formal notion of features, component-based product families and their interaction. We then demonstrate using Milner's CCS how our formalism can be applied to extend an arbitrary modelling formalism with support for product lines. To verify that certain products indeed realize certain features, we propose $\mu$-calculus model-checking for multi-valued Kripke-structures. The model checking result in that case no longer is a simple truth-value, but a set of products, conforming to a certain property.

## 1 Introduction

The vast majority of electronic devices with which we interact is mainly controlled by software—in fact, software-intensive systems pervade our daily life. Typically, not only a single software-intensive system is constructed but rather a family of similar systems that share certain commonalities. Prominent examples of such families of software-intensive systems can be found in a multitude of different application domains, comprising embedded as well as business information systems. For example the model variants of the same model series of a car manufacturer, e.g. the variants of the *7-series BMW*, or the various variants of an operating system, e.g. the various editions of the operating system *Microsoft Windows 7*, constitute such families. Typical commonalities for such systems can be found for example in their (conceptual) functionality, their architectural component structure, or code. To enhance the efficiency of the software development and maintenance process, the integrated development of a family of software-intensive systems by explicitly making use of (reusing) their commonalities in a strategic and planned way seems a promising approach. This is the subject of software product family engineering.

Despite its obvious motivation, the way of constructing a family of systems by taking advantage of commonalities is not sufficiently explored—in particular with respect to its theoretical foundation. How can reuse based on commonalities between system variants take place in a systematic way? What are the fundamental concepts behind commonalities and differences of related systems, and

T. Margaria and B. Steffen (Eds.): ISoLA 2012, Part I, LNCS 7609, pp. 131–145, 2012.

how can we formally represent them? How can commonalities between family members be determined and even schematically computed? How can the relation between family members be modelled, and how are commonalities integrated into the construction of the individual family members? How can we verify correctness properties of a whole software product family instead of looking at the properties of each family member individually?

In this paper we address these questions from a formal point of view and provide an axiomatization of product family concepts using the language of algebraic specification [Wir90]. The axiomatization formalizes the key characteristics of any software product family, where the concept of commonality and the ability to compute the commonalities of an arbitrary subset of family members is the most important aspect for us.

The formal specification may be used as a guidance when defining explicit formalisms supporting the concept of software product families. In this paper, we recall (and slightly simplify) the account of [GLS08] which extends Milner's CCS by a variant operator yielding the product-line aware calculus PL-CCS. With the help of the specification, we can check that PL-CCS is indeed a reasonable product family extension of CCS.

Finally, to make this overview paper self-contained, we recall the model checking approach for PL-CCS that allows to check a whole family of systems with respect to $\mu$-calculus specifications.

## 2 Related Work

Most of the related approaches which deal with modelling of software product families are found in the area of *Feature Oriented Software Development* (FOSD) [CE00]. FOSD deals with the construction of variable software systems. A common specification technique for software product lines in FOSD are so-called *feature models* [KHNP90]. Feature models are used to model optional, mandatory and variable features, and in particular their dependencies. In that way a feature model allows to restrict the set of possible configurations of a product line, but in general it does not incorporate the information of how to construct the family members, nor does it allow to compute common parts of a given subset of family members. Thus, a feature model serves the same purpose as our dependency model, but does not represent a product family in our sense, i.e. as a construction blueprint that shows how the family members can actually be constructed from the common and variable parts, or how the members are related with respect to reusing common parts. Moreover, feature models usually lack a precise semantics which impedes to reason about features or feature combinations using formal methods.

To make these issues more precise, we recall the concept of features in the next section.

Regarding the algebraic treatment of software product families, there are some approaches which also unify common concepts, techniques and methods of feature-oriented approaches by providing an abstract, common, formal basis.

In this context, we consider especially the approaches [HKM06,HKM11,BO92] to be of interest.

The closest to our axiomatization of a software product family is an approach by Höfner et al. [HKM06,HKM11], introducing the notion of a *feature algebra*, and a *product family*, respectively, which describes the features of a family of products, and their typical operations from a semi-ring, algebraic perspective. The elements of a feature algebra are called product families. A product family corresponds to a set of products, where individual products are considered to be flat collections of features. In general, the structure of a feature algebra largely agrees with the structure of a software product family of type $Spf_\alpha$, as it can be built using the constructors (cf. Section 4) only. While Höfner et al. nicely characterize the structure of a product line from an algebraic point of view, they do not include operations that describe the manipulation or alteration of product families into their algebraic components. For example, Höfner et al. do not explicitly express the notion of configuration. In contrast, our approach defines functions that characterize how to manipulate and work with a product family, e.g. the functions selL and selR that formalize the act of configuring a product family, or the function is_mand that formalizes the notion of mandatory parts. In our opinion these additional operations are as essential as the basic constructors in order to formalize the notion of a product family.

The first work on verifying software product families via model checking is, to best of our knowledge, in [GLS08]. A slightly different verification approach is given in [CHSL11].

## 3 Features

Intuitively, a product family subsumes a number of products with similar functionality. From an engineering perspective, organizing products in product families is beneficial, as it allows for a single development process, and eases the identification of common components. While product variants sometimes evolve over time for technical reasons, they are often specifically developed out of marketing concerns or to meet similar but different customer needs. In the latter case, a product family is first designed from an external, user perspective in terms of features without considering their technical structure. A feature in this context is the ability of a product to cover a certain use case or meet a certain customer need. Thus, it is frequently impossible to map features independently to certain technical properties.

An established method to design and structure the feature domain of a product family is the use of feature diagrams [KHNP90]. Feature diagrams do not describe the meaning of different features, since at that stage no common formalism to describe such properties and product behaviour is applicable. Instead they define the compositional structure and dependencies between features from a user perspective.

We use a product family for a fictional windscreen wiper system as running example. Figure 1 shows the corresponding feature diagram. The variants of our

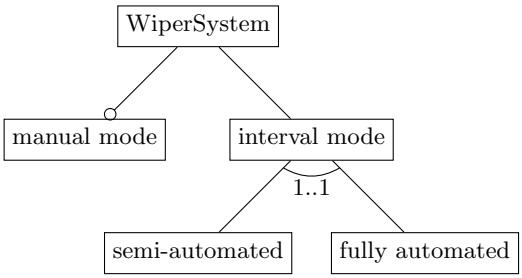


**Fig. 1.** Feature diagram for a product family of a windscreen wiper system

wiper systems may have a manual mode. They are required to have an interval mode controlled by a rain sensor. This mode can either be semi-automated and control only some operation modi of the wiper or fully automated. Thus, from our feature diagram we can derive four different feature combinations.

There are a lot of different variations of feature diagrams. All of them allow to express the compositional structure and optional, mandatory and alternative features. For most types the semantics can be given by translation into propositional logic with features as atomic propositions [SHT06].

Feature diagrams only describe possible combinations of features. To be able to express statements about products and their technical structure, we need to bind features to products.

**Definition 1.** $\mathcal{F} : \mathbb{P} \to 2^{\mathbb{F}}$ *is a feature function, mapping products* $p \in \mathbb{P}$ *to the features* $f \in \mathbb{F}$ *they have.*

Since feature diagrams only describe feature dependencies from a user or product designer perspective, some possible feature combinations might not actually be feasible, i.e. it is impossible for technological reasons to combine those features.

**Definition 2.** *The set* $F \subseteq \mathbb{F}$ *is a feasible feature combination if* $\exists p \in \mathbb{P} : F \subseteq \mathcal{F}(p)$.

Conversely, feasible feature combinations may not be possible with respect to a feature diagram as feasible combinations might be undesirable.

## 4 Specification of Product Lines

At a technological level, different feature combinations have to be realized by different products. To be able to manage complexity, products are usually described and built in a compositional manner. Consequently, sets of products are usually specified by introducing variation points at different levels of their compositional structure. A concrete product can be derived from such a description by selecting one alternative at each variation point. Widespread instances

of these concepts for software development are source code preprocessors and dependency injection frameworks.

Consider again the wiper system introduced above. To specify a certain wiper system, we would compose a wiper and a sensor component into one system.

$$\text{wiper} \| \text{sensor}$$

To realize the different feature combinations, we would use different variations of those components. To do so, we introduce variation points.

$$\text{wiper} := \text{wiper}_1 \oplus_1 \text{wiper}_2; \quad \text{wiper} := \text{sensor}_1 \oplus_2 \text{sensor}_2$$

To support product families in an arbitrary specification formalism, we introduce several generic operators. As we want to define product families following the compositional structure of the underlying formalism, we need an operator asset($A$), that converts an atomic product into a product family. To express shifting operators from products to product families in a generic way, we need an operator op $\circ$ arg, that applies an operator op (partially) to all products described by arg. In the case of the binary operator $\|$, we would write $(\| \circ A) \circ B$, to express that $\|$ is shifted to product families by applying it to all products described by $A$ using its first parameter. The resulting unary operator is then applied to each product from $B$.

Using these three operators, it is possible to lift the semantics of any product specification formalism to product families.

We can now add choice points in the same manner as in our example above. A choice operator $A \oplus_i B$ describes the product family, where a left choice for $i$ results in the products from $A$, and a right choice in the products from $B$. As the choice between left and right variants is bound to the index $i$, for every occurrence of an operator with the same index the same choice has to be made. It is thus possible to express dependencies between different choice points in a system.

It is usually the case that not all possible configurations of a product family describe a system that is technologically feasible. Thus, we introduce the empty product family $\bot$, containing no products. Using $\bot$, dependencies on choices may be expressed. For example, we could write $A \oplus_i \bot$ to express, that at some point in our product family specification, only a left choice may be made for $i$. To ease notation of these dependencies, we introduce a dependency operator $(i_1, L/R), \ldots, (i_k, L/R) \hookrightarrow A$, meaning $A$ requires left or right choices for certain $i_1, \ldots i_k$.

Using the operators described so far a product family can be completely described. To derive products from such descriptions we only need the operator conf($A$), returning all possible products annotated with the choices leading to them. For convenience we further introduce operators products($A$) and choices($A$), yielding the set of all products and choices, respectively.

A further common mechanism observed in product line development is the instantiation of components. Considering our wiper system example, a car might use separate systems to control front and rear wipers, which can be different

variants of the same product. Thus, we introduce a renaming operator $A[f]$, which renames all choice indices $i$ in $A$ by applying function $f$. Consider the description of the above wiper system. To compose two of them in one system allowing independent choices for each, we could write:

$$\text{wipersys} \| \text{ wipersys } [1/3, 2/4]$$

We give a formal definition of all those operators in Figure 2. We use higher order functions to define the operator $\circ$ and most signatures are defined using a type variable $\alpha$ known from polymorphic function types.

Given our formal notion of both the user and engineering perspective on product families, we are now able to precisely describe their connection.

**Definition 3.** *The technologically feasible configurations for a product famliy $P$ providing a set of features $F$ with respect to a feature function $\mathcal{F}$ is given by*

$$\mathcal{C}_{P,F,\mathcal{F}} = \{c \mid (c,p) \in \mathit{conf}(P), \mathcal{F}(p) \subseteq F\}$$

There usually is a multitude of possible product family specifications, where the same products can be derived using the same configuration. This observation warrants the following equivalence relation between product family specifications.

**Definition 4.** *Product family specifications $P$ and $Q$ are called configuration-equivalent*

$$P \equiv_{\mathrm{c}} Q \text{ iff } \mathit{conf}(P) \equiv \mathit{conf}(Q)$$

Using that equivalence and the axioms from Figure 2 we can prove several laws that facilitate restructuring product family specifications and identifying common parts in different variants.

The operator for lifting operators from an underlying formalism to product families $\circ$ is (left and right) distributive over the choice operator $\oplus_i$.

$$(P \circ Q) \oplus_i (P \circ R) \equiv_{\mathrm{c}} P \circ (Q \oplus_i R)$$
$$(P \circ R) \oplus_i (Q \circ R) \equiv_{\mathrm{c}} (P \oplus_i Q) \circ R$$

Thus all operators of an underlying formalism are distributive over the choice operators. We can therefore pull out common parts.

Choice operators with different index are distributive.

$$(P \oplus_j Q) \oplus_i (P \oplus_j R) \equiv_{\mathrm{c}} P \oplus_j (Q \oplus_i R) \text{ with } i \neq j$$
$$(P \oplus_j R) \oplus_i (Q \oplus_j R) \equiv_{\mathrm{c}} (P \oplus_i Q) \oplus_j R \text{ with } i \neq j$$

It is thus possible to change the way choices are nested and to pull out common choices.

Dependencies between choices can render certain parts of a specification inaccessible. When two dependent operators are directly nested, the following laws can be applied to simplify the specification.

$$P \oplus_i (Q \oplus_i R) \equiv_{\mathrm{c}} P \oplus_i R$$
$$(P \oplus_i Q) \oplus_i R \equiv_{\mathrm{c}} P \oplus_i R$$

**SPEC** Softwareproductfamily = {

**defines** $\mathsf{Spf}_\alpha$

**based_on** Bool, Nat, Set, HOFunc

**functions**

$\bot_\alpha$ : $\mathsf{Spf}_\alpha$

$\mathsf{asset}_\alpha$ : $\alpha \to \mathsf{Spf}_\alpha$

$\circ_{\alpha,\beta}$ : $\mathsf{Spf}_{\beta^\alpha} \times \mathsf{Spf}_\alpha \to \mathsf{Spf}_\beta$

$\oplus_\alpha$ : $\mathsf{Spf}_\alpha \times \mathbb{N} \times \mathsf{Spf}_\alpha \to \mathsf{Spf}_\alpha$

$[]_\alpha$ : $\mathsf{Spf}_\alpha \times (\mathbb{N}^\mathbb{N}) \times \mathbb{N} \to \mathsf{Spf}_\alpha$

$\hookrightarrow_\alpha$: $2^{\mathbb{N}\times\{L,R\}} \times \mathsf{Spf}_\alpha \to \mathsf{Spf}_\alpha$

$\mathsf{conf}_\alpha$ : $\mathsf{Spf}_\alpha \to 2^{2^{\mathbb{N}\times\{L,R\}}\times\alpha}$

$\mathsf{products}$ : $\mathsf{Spf}_\alpha \to 2^\alpha$

$\mathsf{choices}$ : $\mathsf{Spf}_\alpha \to 2^\mathbb{N}$

$\mathsf{comp}$ : $2^{\mathbb{N}\times\{L,R\}} \to 2^{2^{\mathbb{N}\times\{L,R\}}\times\alpha}$

$\mathsf{confd}_\alpha$ : $\{L,R\} \times \mathsf{Spf}_\alpha \to 2^{2^{\mathbb{N}\times\{L,R\}}\times\alpha}$

**axioms**

$\mathsf{comp}(C) = \bigwedge_{(i,d),(i,d')\in C} d = d'$

$\mathsf{confd}(d,P) = \{(\{(i,d)\} \cup c, p) \mid (c,p) \in \mathsf{conf}(P), \mathsf{comp}(\{(i,d)\} \cup c)\}$

$\mathsf{conf}(\bot) = \varnothing$

$\mathsf{conf}(\mathsf{asset}(a)) = (\varnothing, a)$

$\mathsf{conf}(\mathsf{apply}(F,P)) = \bigcup_{(c_1,f)\in\mathsf{conf}(F),(c_2,p)\in\mathsf{conf}(P),\mathsf{comp}(c_1\cup c_2)}(c_1 \cup c_2, f(p))$

$\mathsf{conf}(P \oplus_i Q) = \mathsf{confd}(L,P) \cup \mathsf{confd}(R,Q)$

$\mathsf{conf}(P[f]) = \{(c', p) \mid (c,p) \in \mathsf{conf}(P), c' = \{(f(i),d) \mid (i,d) \in c\}, \mathsf{comp}(c')\}$

$\mathsf{products}(P) = \bigcup_{(c,p)\in\mathsf{conf}(P)}\{p\}$

$\mathsf{choices}(P) = \bigcup_{(c,p)\in\mathsf{conf}(P),(i,d)\in c}\{i\}$

$\varnothing \hookrightarrow P = \bot$

$(\{(i,L)\} \cup I) \hookrightarrow P = P \oplus_i (I \hookrightarrow P)$

$(\{(i,R)\} \cup I) \hookrightarrow P = (I \hookrightarrow P) \oplus_i P$

}

**Fig. 2.** Algebraic specification of a generic product line formalism

Since the configuration of product family specifications is defined inductively replacing a part by a configuration-equivalent expression yields a configuration-equivalent specification.

$$P \equiv_{\mathrm{c}} Q \text{ then } R \equiv_{\mathrm{c}} R[P/Q]$$

Note that in this case $[P/Q]$ refers to the syntactic replacement of a sub-expression.

The empty product family $\bot$ can be used to prohibit certain configurations. The laws involving $\bot$ facilitate the simplification of product family specifications in certain cases. It is possible to reduce expressions without any choices containing $\bot$.

$$\begin{aligned} P \circ \bot &\equiv_{\mathrm{c}} \bot \\ \bot \circ P &\equiv_{\mathrm{c}} \bot \\ \bot[f] &\equiv_{\mathrm{c}} \bot \end{aligned}$$

It is further possible to eliminate choices yielding $\bot$ for both the left and right choice.

$$\bot \oplus_i \bot \equiv_{\mathrm{c}} \bot$$

When similar components are used at multiple locations in a system, it often is beneficial to factor those components out into a single specification that can then be instantiated appropriately. Using the following laws, renamings of choice indices can be introduced bottom-up.

$$\begin{aligned} \mathrm{asset}(a)[f] &\equiv_{\mathrm{c}} \mathrm{asset}(a) \\ (P \circ Q)[f] &\equiv_{\mathrm{c}} P[f] \circ Q[f] \\ (P \oplus_i Q)[f] &\equiv_{\mathrm{c}} P[f] \oplus_{f(i)} Q[f] \end{aligned}$$

In doing so, identical sub-expressions using different indices can be defined over the same indices.

The laws discussed so far allow for refactorings of product family specifications that preserve the possible configurations of a product family. Often changes to the configurations are acceptable though, when they allow for more radical refactorings and the derivable products are still being preserved. That observation gives rise to the follow, more relaxed equivalence relation.

**Definition 5.** *Product family specifications $P$ and $Q$ are called product-equivalent*

$$P \equiv_{\mathrm{p}} Q \textit{ iff } \textit{products}\,(P) \equiv \textit{products}(Q)$$

Using this equivalence, we can prove some additional laws.

Obviously, two configuration-equivalent specifications are also product-equivalent.

$$P \equiv_{\mathrm{c}} Q \Rightarrow P \equiv_{\mathrm{p}} Q$$

Leaving out a top level renaming does not change the set of products.

$$P \equiv_{\mathrm{p}} P[f]$$

Choices, resulting in the same set of products, may be left out.

$$P \oplus_i P \equiv_{\mathrm{p}} P$$

While it is possible to apply the laws for configuration-equivalence on any subexpression, this is no longer the case for product-equivalence as there might be dependencies defined on certain configurations. It is still possible though when respecting some side conditions.

If $F \equiv_{\mathrm{p}} F[P/Q]$ then

$$F \oplus_i G \equiv_{\mathrm{p}} F[P/Q] \oplus_i G \text{ with } i \notin \mathrm{choices}(P, Q)$$
$$G \oplus_i F \equiv_{\mathrm{p}} G \oplus_i F[P/Q] \text{ with } i \notin \mathrm{choices}(P, Q)$$
$$F \circ G \equiv_{\mathrm{p}} F[P/Q] \circ G \text{ with } \mathrm{choices}(P, Q) \cap \mathrm{choices}(G) = \varnothing$$
$$G \circ F \equiv_{\mathrm{p}} G \circ F[P/Q] \text{ with } \mathrm{choices}(P, Q) \cap \mathrm{choices}(G) = \varnothing$$
$$F[f] \equiv_{\mathrm{p}} F[P/Q][f] \text{ with } i \in \mathrm{choices}(P, Q) \Rightarrow i = f(i)$$

## 5 PL-CCS

In the previous section, we have worked out an algebraic specification for the concept of product families. It is meant to serve as a meta model pointing out the fundamental ideas of any formalism having a notion of families.

In this section, we present a concrete modelling formalism for product families. We enrich Milner's CCS by a variation operator. As the resulting calculus, which we call PL-CCS, is a model of the algebraic specification given in the previous section, it is a valid realization of a product family concept. The approach followed in this section is a slight simplification and extension of the account presented in [GLS08]. The syntax of PL-CCS is given as follows:

**Definition 6 (Syntax of PL-CCS)**

$$e ::= Q \mid \mathit{Nil} \mid \alpha.e \mid (e + e) \mid (e \parallel e) \mid (e)[f] \mid (e) \setminus L \mid \mu Q.e \mid (e \oplus_i e) \mid (e)[g]$$

Thus, we use a fixpoint-oriented account to CCS as in and enrich CCS by the variant operator $\oplus$, which may cater for additional renaming.

The semantics of PL-CCS may now be given in several, as we will show equivalent, ways. First, one might configure a PL-CCS specification in every possible way to obtain a set of CCS specifications, which may act as the, here called *flat*, semantics of a product family, which is basically a set of Kripke structures. To this end, we recall the definition of a Kripke structure and the semantics of a CCS process.

**Definition 7 (KS).** *A* Kripke structure $\mathcal{K}$ *is defined as*

$$\mathcal{K} = (\mathcal{S}, \mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}, L \subseteq \mathcal{S} \times \mathcal{P})$$

*where* $\mathcal{S}$ *is a set of states,* $\mathcal{R}$ *is a set of* $\mathcal{A}$*-labelled transitions, and* $L$ *labels states by its set of valid propositions.*

Next, we recall the definition of CCS. Due to space constraints, it is given according to Figure 3, ignoring the product label $\nu$ in the SOS-rules shown.

Now, we are ready to define the notion of a flat semantics for a PL-CCS family.

**Definition 8 (Flat Semantics of PL-CCS)**

$$[\![P]\!]_{flat} = \{(c, [\![p]\!]) \mid (c, p) \in \mathit{conf}(P)\}$$

Especially for verification purposes, it is, however, desirable, to provide a comprehensive semantics, which we do in terms of a *multi-valued Kripke structure*.

A *lattice* is a partially ordered set $(\mathcal{L}, \sqsubseteq)$ where for each $x, y \in \mathcal{L}$, there exists (i) a unique *greatest lower bound* (glb), which is called the *meet* of $x$ and $y$, and is denoted by $x \sqcap y$, and (ii) a unique *least upper bound* (lub), which is called the *join* of $x$ and $y$, and is denoted by $x \sqcup y$. The definitions of glb and lub extend to finite sets of elements $A \subseteq \mathcal{L}$ as expected, which are then denoted by $\bigsqcap A$ and $\bigsqcup A$, respectively. A lattice is called *finite* iff $\mathcal{L}$ is finite. Every finite lattice has a least element, called *bottom*, denoted by $\bot$, and a greatest element, called *top*, denoted by $\top$. A lattice is *distributive*, iff $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$, and, dually, $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$. In a *DeMorgan* lattice, every element $x$ has a unique *dual* element $\neg x$, such that $\neg\neg x = x$ and $x \sqsubseteq y$ implies $\neg y \sqsubseteq \neg x$. A complete distributive lattice is called *Boolean* iff the $x \sqcup \neg x = \top$ and $x \sqcap \neg x = \bot$.

While the developments to come do not require to have a Boolean lattice, we will apply them only to the Boolean lattices given by the powerset of possible configurations. In other words, given a set of possible configurations $N$, the lattice considered is $(2^N, \subseteq)$ where meet, join, and dual of elements, are given by intersection, union, and complement of sets, respectively.

**Definition 9 (MV-KS).** *A* multi-valued Kripke structure $\mathcal{K}$ *is defined as*

$$\mathcal{K} = (\mathcal{S}, \mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{L}, L : \mathcal{S} \times \mathcal{P} \to \mathcal{L})$$

*where* $\mathcal{S}$ *is a set of states,* $\mathcal{R}$ *is a set of* $\mathcal{A}$*-labelled transitions, denoting for which product the transition is possible, and* $L$ *identifies in which state which propositions hold for which product.*

Based on this notion, we provide the so-called *configured semantics* of a PL-CCS specification.

**Definition 10 (Configured Semantics of PL-CCS).** *The* configured semantics *of PL-CCS is given according to the SOS-rules shown in Figure 3.*

$$\frac{P \xrightarrow{\alpha,\nu} P'}{\mu Q.P \xrightarrow{\alpha,\nu} P'[Q/\mu Q.P]} \text{ (recursion)} \qquad \frac{P \xrightarrow{\alpha,\nu} P' \quad Q \xrightarrow{\bar{\alpha},\bar{\nu}} Q'}{(P \parallel Q) \xrightarrow{\tau,\nu\cap\bar{\nu}} (P' \parallel Q')} \text{ (par. comp. (3))}$$

$$\frac{}{\alpha.P \xrightarrow{\alpha,2^{\{R,L\}^n}} P} \text{ (prefix)} \qquad \frac{P \xrightarrow{\alpha,\nu} P'}{P[f] \xrightarrow{f(\alpha),\nu} P'[f]} \text{ (relabeling)}$$

$$\frac{P \xrightarrow{\alpha,\nu} P'}{P + Q \xrightarrow{\alpha,\nu} P'} \text{ (nondet. choice (1))} \qquad \frac{P \xrightarrow{\alpha,\nu} P'}{(P \setminus L) \xrightarrow{\alpha,\nu} (P' \setminus L)} \ , \ \alpha \notin L \text{ (restriction)}$$

$$\frac{Q \xrightarrow{\alpha,\nu} Q'}{P + Q \xrightarrow{\alpha,\nu} Q'} \text{ (nondet. choice (2))} \qquad \frac{P \xrightarrow{\alpha,\nu} P'}{P \oplus_i Q \xrightarrow{\alpha,\nu|_{i/L}} P'} \text{ (conf. sel. (1))}$$

$$\frac{P \xrightarrow{\alpha,\nu} P'}{(P \parallel Q) \xrightarrow{\alpha,\nu} (P' \parallel Q)} \text{ (par. comp. (1))} \qquad \frac{Q \xrightarrow{\alpha,\nu} Q'}{P \oplus_i Q \xrightarrow{\alpha,\nu|_{i/R}} Q'} \text{ (conf. sel. (2))}$$

$$\frac{Q \xrightarrow{\alpha,\nu} Q'}{(P \parallel Q) \xrightarrow{\alpha,\nu} (P \parallel Q')} \text{ (par. comp. (2))} \qquad \frac{P \xrightarrow{\alpha,\nu} P'}{P[g] \xrightarrow{\alpha,\nu[g]} P'[g]} \text{ (conf. relabeling)}$$

**Fig. 3.** The inference rules for the semantics of PL-CCS (and CCS when ignoring the second component of each transition label)

We conclude the introduction of PL-CCS stating that the flat semantics and the configured semantics are equivalent, in the following sense:

**Theorem 1 (Soundness of Configured Semantics)**

$$\{(c, p) \mid p = \Pi_c([\![P]\!]_{conf})\} = [\![P]\!]_{flat}$$

Here, $\Pi_c$ denotes the projection of a transition system to the respective configuration $c$, which is defined in the expected manner.

## 6 Model-Checking PL-CCS

In this section, we sketch a game-based and therefore on-the-fly model checking approach for PL-CCS programs with respect to $\mu$-calculus specifications.

We have chosen to develop our verification approach for specifications in the $\mu$-calculus as it subsumes linear-time temporal logic as well as computation-tree logic as first shown in [EL86,Wol83] and nicely summarized in [Dam94]. Therefore we can use our approach also in combination with these logics, and in particular have support for the language SALT [BLS06] used with our industrial partners.

Multi-valued modal $\mu$-calculus combines Kozen's modal $\mu$-calculus [Koz83] and multi-valued $\mu$-calculus as defined by Grumberg and Shoham [SG05] in a way suitable for specifying and checking properties of PL-CCS programs. More specifically, we extend the work of [SG05], which only supports unlabelled diamond and box operators, by providing also action-labelled versions of these operators, which is essential to formulate properties of PL-CCS programs.[1]

*Multi-valued modal $\mu$-calculus.* Let $\mathcal{P}$ be a set of *propositional constants*, and $\mathcal{A}$ be a set of *action names*.[2]A *multi-valued modal Kripke structure* (MMKS) is a tuple $\mathcal{T} = (\mathcal{S}, \{\mathcal{R}_\alpha(.,.) \mid \alpha \in \mathcal{A}\}, L)$ where $\mathcal{S}$ is a set of states, and $\mathcal{R}_\alpha(.,.) : \mathcal{S} \times \mathcal{S} \to \mathcal{L}$ for each $\alpha \in \mathcal{A}$ is a valuation function for each pair of states and action $\alpha \in \mathcal{A}$. Furthermore, $L : \mathcal{S} \to \mathcal{L}^{\mathcal{P}}$ is a function yielding for every state a function from $\mathcal{P}$ to $\mathcal{L}$, yielding a value for each state and proposition. For PL-CCS programs, the idea is that $\mathcal{R}_\alpha(s, s')$ denotes the set of configurations in which there is an $\alpha$-transition from state $s$ to $s'$. It is a simple matter to translate (on-the-fly) the transition system obtained via the configured-transitions semantics into a MMKS.

A Kripke structure in the usual sense can be regarded as a MMKS with values over the two element lattice consisting of a bottom $\bot$ and a top $\top$ element, ordered in the expected manner. Value $\top$ then means that the property holds in the considered state while $\bot$ means that it does not hold. Similarly, $\mathcal{R}_\alpha(s, s') = \top$ reads as there is a corresponding $\alpha$-transition while $\mathcal{R}_\alpha(s, s') = \bot$ means there is no $\alpha$-transition.

Let $\mathcal{V}$ be a set of propositional variables. Formulae of the *multi-valued modal $\mu$-calculus* in *positive normal form* are given by

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid q \mid \neg q \mid Z \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi$$

where $q \in \mathcal{P}$, $\alpha \in \mathcal{A}$, and $Z \in \mathcal{V}$. Let $mv\text{-}\mathfrak{L}_\mu$ denote the set of *closed* formulae generated by the above grammar, where the fixpoint quantifiers $\mu$ and $\nu$ are variable binders. We will also write $\eta$ for either $\mu$ or $\nu$. Furthermore we assume that formulae are well-named, i.e. no variable is bound more than once in any formula. Thus, every variable $Z$ *identifies* a unique sub-formula $fp(Z) = \eta Z.\psi$ of $\varphi$, where the set $Sub(\varphi)$ of *sub-formulae* of $\varphi$ is defined in the usual way.

The semantics of a $mv\text{-}\mathfrak{L}_\mu$ formula is an element of $\mathcal{L}^{\mathcal{S}}$—the functions from $\mathcal{S}$ to $\mathcal{L}$, yielding for the formula at hand and a given state the *satisfaction value*. In our setting, this is the set of configurations for which the formula holds in the given state.

Then the *semantics* $[\![\varphi]\!]^{\mathcal{T}}_{\rho}$ of a $mv\text{-}\mathfrak{L}_\mu$ formula $\varphi$ with respect to a MMKS $\mathcal{T} = (\mathcal{S}, \{\mathcal{R}_\alpha(.,.) \mid \alpha \in \mathcal{A}\}, L)$ and an *environment* $\rho : \mathcal{V} \to \mathcal{L}^{\mathcal{S}}$, which explains

[1] Thus, strictly speaking, we define a multi-valued and multi-modal version of the $\mu$-calculus. However, we stick to a shorter name for simplicity.

[2] So far, for PL-CCS programs, we do not need support for propositional constants. As adding propositions only intricates the developments to come slightly, we show the more general account in the following.

$$
\begin{array}{rcl}
[\![true]\!]_\rho & := & \lambda s.\top \\
[\![false]\!]_\rho & := & \lambda s.\bot \\
[\![q]\!]_\rho & := & \lambda s.L(s)(q) \\
[\![\neg q]\!]_\rho & := & \lambda s.\overline{L(s)(q)} \\
[\![Z]\!]_\rho & := & \rho(Z)
\end{array}
\qquad
\begin{array}{l}
[\![\varphi \vee \psi]\!]_\rho := [\![\varphi]\!]_\rho \sqcup [\![\psi]\!]_\rho \\
[\![\varphi \wedge \psi]\!]_\rho := [\![\varphi]\!]_\rho \sqcap [\![\psi]\!]_\rho \\
[\![\langle\alpha\rangle\varphi]\!]_\rho := \lambda s. \bigsqcup\{\mathcal{R}_\alpha(s,s') \sqcap [\![\varphi]\!]_\rho(s')\} \\
[\![[\alpha]\varphi]\!]_\rho := \lambda s. \bigsqcap\{\neg\mathcal{R}_\alpha(s,s') \sqcup [\![\varphi]\!]_\rho(s')\} \\
[\![\mu Z.\varphi]\!]_\rho := \bigsqcap \{f \mid [\![\varphi]\!]_{\rho[Z\mapsto f]} \sqsubseteq f\} \\
[\![\nu Z.\varphi]\!]_\rho := \bigsqcup \{f \mid f \sqsubseteq [\![\varphi]\!]_{\rho[Z\mapsto f]}\}
\end{array}
$$

**Fig. 4.** Semantics of $mv\text{-}\mathfrak{L}_\mu$ formulae

the meaning of free variables in $\varphi$, is an element of $\mathcal{L}^\mathcal{S}$ and is defined as shown in Figure 4. We assume $\mathcal{T}$ to be fixed and do not mention it explicitly anymore. With $\rho[Z \mapsto f]$ we denote the environment that maps $Z$ to $f$ and agrees with $\rho$ on all other arguments. Later, when only closed formulae are considered, we will also drop the environment from the semantic brackets.

The semantics is defined in a standard manner. The only operators deserving a discussion are the $\Diamond$ and $\Box$-operators. Intuitively, $\langle\alpha\rangle\varphi$ is classically supposed to hold in states that have an $\alpha$-successor satisfying $\varphi$. In a multi-valued version, we first consider the value of $\alpha$-transitions and reduce it (meet it) with the value of $\varphi$ in the successor state. As there might be different $\alpha$-transitions to different successor states, we take the best value. For PL-CCS programs, this meets exactly our intuition: A configuration in state $s$ satisfies a formula $\langle\alpha\rangle\varphi$ if it has an $\alpha$-successor satisfying $\varphi$. Dually, $[\alpha]\varphi$ is classically supposed to hold in states for which all $\alpha$-successors satisfy $\varphi$. In a multi-valued version, we first consider the value of $\alpha$-transitions and increase it (join it) with the value of $\varphi$ in the successor state. As there might be several different $\alpha$-successor states, we take the worst value. Again, this meets our intuition for PL-CCS programs: A configuration in state $s$ satisfies a formula $[\alpha]\varphi$ if all $\alpha$-successors satisfy $\varphi$.

The functionals $\lambda f.[\![\varphi]\!]_{\rho[Z\mapsto f]} : \mathcal{L}^\mathcal{S} \to \mathcal{L}^\mathcal{S}$ are monotone wrt. $\sqsubseteq$ for any $Z, \varphi$ and $\mathcal{S}$. According to [Tar55], least and greatest fixpoints of these functionals exist.

*Approximants* of $mv\text{-}\mathfrak{L}_\mu$ formulae are defined in the usual way: if $fp(Z) = \mu Z.\varphi$ then $Z^0 := \lambda s.\bot$, $Z^{\alpha+1} := [\![\varphi]\!]_{\rho[Z\mapsto Z^\alpha]}$ for any ordinal $\alpha$ and any environment $\rho$, and $Z^\lambda := \bigsqcap_{\alpha<\lambda} Z^\alpha$ for a limit ordinal $\lambda$. Dually, if $fp(Z) = \nu Z.\varphi$ then $Z^0 := \lambda s.\top$, $Z^{\alpha+1} := [\![\varphi]\!]_{\rho[Z\mapsto Z^\alpha]}$, and $Z^\lambda := \bigsqcup_{\alpha<\lambda} Z^\alpha$.

**Theorem 2 (Computation of Fixpoints, [Tar55]).** *For all MMKS $\mathcal{T}$ with state set $\mathcal{S}$ there is an $\alpha \in \mathbb{O}\mathrm{rd}$ s.t. for all $s \in \mathcal{S}$ we have: if $[\![\eta Z.\varphi]\!](s) = x$ then $Z^\alpha(s) = x$.*

The following theorem states that the multi-valued modal semantics of the $\mu$-calculus is indeed suitable for checking the different configurations of a PL-CCS program.

**Theorem 3 (Correctness of Model Checking).** *For all PL-CCS programs $P$ and formulae $\varphi \in mv\text{-}\mathfrak{L}_\mu$, we have*

$$(c, \mathcal{K}) \in [\![P]\!]_{flat} \text{ with } \mathcal{K} \models \varphi \text{ iff } c \in ([\![P]\!]_{conf} \models \varphi)$$

The proof follows by structural induction on the formula.

While Theorem 2 also implies a way for computing the satisfaction value of an $mv$-$\mathfrak{L}_\mu$-formula and a given MMKS, this naive fixpoint computation is typically expensive. Game-based approaches originating from the work by [EJS93] and [Sti95] allow model checking in a so-called *on-the-fly* or *local* fashion. In the context of multi-valued $\mu$-calculus, the game-based setting becomes technically more involved, as described in detail in [SG05]. Nevertheless, the essence of the game-based approach of computing a satisfaction value based on the so-called *game graph* is similar. For the multi-valued modal $\mu$-calculus, a slight adaptation of the approach taken in [SG05] yields a game-based approach for the full multi-valued modal $\mu$-calculus. Furthermore abstraction-techniques like those presented in [CGLT09] may be applied.

Due to space limitations, we skip details of the game-based model checking approach for the multi-valued modal $\mu$-calculus.

## 7 Conclusion

In this paper, we have presented a formal foundation for product families, both from a feature as well as a technical perspective and their connection. Based on that foundation we have shown several equivalence laws, that allow for save transformations between different product family specifications. Hence, they facilitate reliable refactorings.

We then applied our formal framework to the well-established, parallel specification formalism, CCS to derive Product-Line-CCS. We have further shown, how PL-CSS can be used to model product lines and efficiently apply model checking to verify properties of a whole product family at once.

We believe this combination of reliable refactorings and verifiable properties yields a robust, formal framework to develop software product families in a safe manner.

## References

BLS06. Bauer, A., Leucker, M., Streit, J.: SALT—Structured Assertion Language for Temporal Logic. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 757–775. Springer, Heidelberg (2006)

BO92. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engineering and Methodology 1(4), 355–398 (1992)

CE00. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)

CGLT09. Campetelli, A., Gruler, A., Leucker, M., Thoma, D.: *Don't Know* for Multi-valued Systems. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 289–305. Springer, Heidelberg (2009)

CHSL11. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A.: Symbolic model checking of software product lines. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 321–330. ACM, New York (2011)

Dam94. Dam, M.: CTL* and ECTL* as fragments of the modal $\mu$-calculus. Theoretical Computer Science 126(1), 77–96 (1994)

EJS93. Emerson, E.A., Jutla, C.S., Sistla, A.P.: On Model-Checking for Fragments of $\mu$-Calculus. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 385–396. Springer, Heidelberg (1993)

EL86. Emerson, E.A., Lei, C.L.: Efficient model checking in fragments of the propositional $\mu$-calculus. In: LICS 1986: Proceedings of the 1st Annual Symposium on Logic in Computer Science, pp. 267–278. IEEE Computer Society Press, Washington, D.C., USA (1986)

GLS08. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and Model Checking Software Product Lines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 113–131. Springer, Heidelberg (2008)

HKM06. Höfner, P., Khedri, R., Möller, B.: Feature Algebra. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 300–315. Springer, Heidelberg (2006)

HKM11. Höfner, P., Khedri, R., Möller An, B.: algebra of product families. Software and Systems Modeling 10, 161–182 (2011) 10.1007/s10270-009-0127-2

KHNP90. Sholom, G., Cohen Kyo, C., Kang, J.A., Hess, W.E.: Novak, and A. Spencer Peterson. Feature oriented design analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21-ESD-90/TR-222, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)

Koz83. Kozen, D.: Results on the propositional $\mu$-calculus. Theoretical Computer Science 27, 333–354 (1983)

SG05. Shoham, S., Grumberg, O.: Multi-valued Model Checking Games. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 354–369. Springer, Heidelberg (2005)

SHT06. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C.: Feature diagrams: A survey and a formal semantics. In: 14th IEEE International Requirements Engineering Conference RE 2006, pp. 139–148 (2006)

Sti95. Stirling, C.: Local Model Checking Games. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 1–11. Springer, Heidelberg (1995)

Tar55. Tarski, A.: A lattice-theoretical fixpoint theorem and its application. Pacific Journal of Mathematics 5, 285–309 (1955)

Wir90. Wirsing, M.: Algebraic specification. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B), pp. 675–788 (1990)

Wol83. Wolper, P.: A translation from full branching time temporal logic to one letter propositional dynamic logic with looping (unpublished manuscript)