Modeling and Model Checking Software Product Lines

Alexander Gruler[⋆], Martin Leucker, and Kathrin Scheidemann

Institut für Informatik · Technische Universität München · Germany

Abstract. Software product line engineering combines the individual developments of systems to the development of a family of systems consisting of common and variable assets. In this paper we introduce the process algebra PL-CCS as a product line extension of CCS and show how to model the overall behavior of an entire family within PL-CCS. PL-CCS models incorporate behavioral variability and allow the derivation of individual systems in a systematic way due to a semantics given in terms of multi-valued modal Kripke structures. Furthermore, we introduce multi-valued modal μ-calculus as a property specification language for system families specified in PL-CCS and show how model checking techniques operate on such structures. In our setting the result of model checking is no longer a simple *yes* or *no* answer but the set of systems of the product line that do meet the specified properties.

1 Introduction

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 [CN02]. Developing a set of related systems as a product line, rather than every system individually, enables the systematic exploitation of synergy effects. Product line concepts are, for example, widely used in automotive production, in order to allow for individuality and high customizability of vehicles, while at the same time keeping production cost low.

Modern vehicles are controlled by large sets of configurable electronic control units (ECUs) which communicate via buses and gateways. In other words, they are highly individual, distributed, embedded systems on wheels. Due to the prevalence of new technologies, the complexity of vehicle systems will increase further. In order to be able to guarantee high quality, security and safety requirements in future, the application of formal verification and analysis techniques get more and more essential. However, in order to be effectively applied in a product line context, formal modeling approaches as well as the applied formal verification techniques, need to efficiently cope with variability.

Despite their importance for the development of software intensive systems, formal modeling and especially verification techniques for product lines do not yet meet the industrial needs.

Kishi et al. in [KNK05] proposed to use traceability in a product model in order to automatically compose subsystem models of a configuration in order to model check that configuration. With this approach it is possible to automate the composition of

[⋆] This author was partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of project VEIA under grant 01ISF15A.

configurations, but it is still necessary to model check a model for each configuration separately.

Modular verification approaches like the one introduced by Li, Krishnamurthi and Fisler [LKF05] use compositional verification techniques in order to infer properties of an assembled system from the properties of its assets. Interfaces of feature-oriented modules contain constraints, similar to verification conditions that other modules must satisfy at composition time. In their approach, these conditions are automatically derived during feature verification. Configurations are verified individually, but results for partly integrated configurations and modules, which have already been verified, can of course be reused.

Larsen et al. defined a behavioral variability model for product line development based on modal I/O automata [LNW07], which are an extension of Larsen's and Thomsen's Kripke modal transition systems [LT91]. Their aim is not to verify product specific functional properties for configurations, but rather to verify the error free combinability of interfaces. Error free composition is characterized by the absence of deadlocks. It is not required that all possible configurations give an error free composition, but only that there exist configurations that can avoid errors under suitable use.

Kripke modal transition systems are used in [FUB07] to study the notion of *behavioral conformance* in the setting of software product lines.

In our approach we model the functional behavior of an entire product family in a single model which explicitly incorporates behavioral variability. Compared to other approaches which also contain the concept of variability in arbitrary development assets such as requirement specifications, design models or test models [PBvdL05], we have included the variability information into the behavioral model. In particular, the concept of variability is also considered in the semantical model. In contrast to other techniques, our model and the respective semantics allow the application of model checking techniques. By this, we can reduce typical questions from product-line engineering to model checking problems.

More specifically, this paper introduces PL-CCS as a variant of Milner's CCS [Mil95] designed to model the interaction of software components used in software product lines. While Milner's CCS is well-suited for describing the communication of (closed) software systems, it lacks support for defining a set of systems. We extend CCS by a variants operator $\oplus$, which allows to model alternative behavior, i.e. alternative processes, with the meaning that only one of the alternative processes will be existing in the final (running) system. The semantics of an PL-CCS system is defined in terms of a *labeled transition system for product lines* (PL-LTS), which is essentially a multi-modal Kripke structure extending Kripke modal transition systems [LT91]. Abstracting from the specification concept on top (in our case extended CCS), a PL-LTS assigns a semantics to the so far only vaguely defined notion of variability. Note that de Nicola et. al. [VN98] and Majster-Cederbaum [MC01] introduce as well an $\oplus$-operator which represents a form a variability. However, the approach is not tailored to product lines and does not study the question of verification.

The main benefit of the proposed modeling formalism is that it caters for automatic verification by model checking. We introduce the multi-valued modal μ-calculus as combination of Kozen's modal μ-calculus [Koz83] and multi-valued μ-calculus as defined by Grumberg and Shoham [SG05], yielding a property specification language suitable for specifying and checking properties of PL-CCS programs. More specifically, the result of model checking a property for a PL-CCS program is the *set* of configurations satisfying the property at hand and not only the answer if the property holds or not.

2 Product-Line CCS

In this section we introduce PL-CCS as an extension of Milner's process algebra CCS [Mil95]. PL-CCS is designed for modeling the behavior of an entire product line in a way especially suitable for automatic verification by model checking.

2.1 Product Lines

Before giving a formal approach to our notion of product lines, let us consider an example, and derive our formalism in an intuitive manner. Hereby, we will also fix the terminology we use in the rest of this paper, mostly following [CN02], where a *product line* is considered to be a set of 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.

As usual, we consider individual systems (products) that are built from subsystems in a compositional manner. Following CCS, we model the behavior of a system as well as its subsystems as processes. The parallel operator || known from CCS can be used to express that the behavior of the compound system is defined by the parallel execution of its subsystems. For example, a car C consists—amongst other things—of an engine E, a locking system LS, and an infotainment system IS, which operate in parallel. This is denoted as:

$$C \overset{def}{=} E \parallel LS \parallel IS$$

In the product lines we consider, subsystems may be realized by alternative *variants*. Entire subsystems may even be optional. For example, vehicles may be equipped with different locking systems, such as (i) a central locking system $LS_{central}$ controlling the locking of all doors, or alternatively (ii) a locking system $LS_{keyless}$, which allows remote keyless entry via a key fob. Such variants can be specified in PL-CCS using the binary *variants operator* $\oplus$. The usage of the variants operator can be understood as offering a set of possible "choices" realizing a variant. Thus, the *locking system* can be specified as follows:

$$LS \overset{def}{=} LS_{central} \oplus LS_{keyless}$$

As not all vehicles in the specified product line may be equipped with an infotainment system, we enrich PL-CCS with an *optional operator* $\langle \_ \rangle$, allowing the infotainment system to be declared as *optional*, written as:

$$C \overset{def}{=} E \parallel LS \parallel \langle IS \rangle$$

Both, the variants operator and the optional operator define a *variation point*. Given a PL-CCS model for an entire product line, individual systems can be derived by making decisions for all variation points. More precisely, choosing for every variants operator one variant and for every optional operator whether the optional subsystem is present or not, yields a specific *configuration*. The configuration then defines uniquely one *system*, also called *product*, that is derivable from the product line.

Note that CCS offers an operator $+$ for expressing *non-deterministic choice*. Although our variants operator $\oplus$ is to some extent similar to the CCS $+$, there are several important conceptual as well as formal differences between both. Thus, both operators are essential for PL-CCS (see also Rules 12 and 13 in Section 2.3).

2.2 PL-CCS – Syntax

In this section we introduce the syntax of PL-CCS programs, which allows us to define design models for software product lines.

Let Id be a finite set of *process identifiers* and Σ by a finite set of *input actions*. Usually, $P, Q, P_1, \dots$ range over process identifiers and $a, b, \dots$ range over input actions. As in CCS, let $\mathcal{A} = \Sigma \cup \bar{\Sigma} \cup \{\tau\}$ represent the set of *communication actions*, where $\tau \notin \Sigma \cup \bar{\Sigma}$ represents the *silent action*, and, $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$ is the set of *output actions*. Usually, $\alpha, \beta, \dots$ range over communication actions. By Nil, we denote the atomic *idle process.*

The set $\mathcal{P}$ of all *PL-CCS process expressions* (or short processes) is generated by the following grammar:

$$e ::= Q \mid Nil \mid \alpha.e \mid e + e \mid e \oplus e \mid e \parallel e \mid e[f] \mid e \backslash L \tag{1}$$

where $Q \in Id$ is a process identifier, $\alpha \in \mathcal{A}$ is an action, $L \subseteq \mathcal{A}$ is a set of action labels, and $f : \mathcal{A} \mapsto \mathcal{A}$ is a *renaming function*, i. e. a function respecting $f(\bar{a}) = \overline{f(a)}$ and $f(\tau) = \tau$.

Thus, syntactically, PL-CCS extends CCS [Mil95] only by the binary *variants operator* $\oplus$. An optional-operator $\langle \_ \rangle$ can be added to PL-CCS as the syntactical abbreviation:

$$\langle P \rangle := P \oplus Nil$$

In Section 2.3, where PL-CCS semantics is discussed, we will see that this abbreviation meets our intuition, allowing us to confine in this technical presentation of PL-CCS to the variants operator $\oplus$ only.

A *process definition* is an equation of the form $P \overset{def}{=} e$, 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 $(\mathcal{E}, P_1)$, where $\mathcal{E}$ 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 aside.

$$\begin{gathered} P_1 \overset{def}{=} e_1 \\ P_2 \overset{def}{=} e_2 \\ \vdots \\ P_n \overset{def}{=} e_n \end{gathered}$$

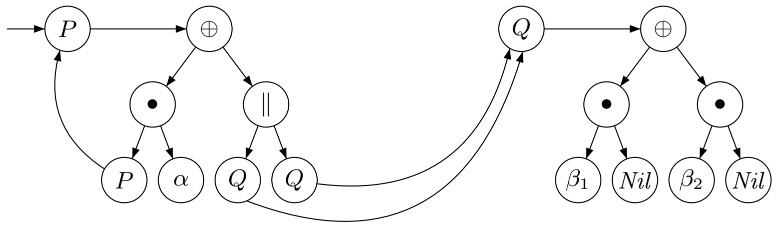


Fig. 1. A program dependency graph

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 $\|$ 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. For such well-formed PL-CCS programs we can define a *compositional* semantics (see section 2.3) which is exactly what we require from a product line approach. In the remainder of the section we successively introduce the syntactical restriction.

Definition 1 (complete PL-CCS program). *We call a PL-CCS program with the set of process definitions* $\{P_1 \overset{def}{=} e_1, \ldots, P_n \overset{def}{=} e_n\}$ complete*, if all process identifiers* P_i *on the left-hand sides of the defining equations are pairwise distinct and the defining equations* $e_1, \ldots, e_n$ *contain only process identifiers in the set* $\{P_1, \ldots, P_n\}$.

In the following, we consider only complete PL-CCS programs. Now, we turn towards the definition of a dependency graph of a PL-CCS program, which—similar to a control flow graph for programming languages—reflects the dependencies of process definitions in a program.

For a PL-CCS process e, let $pt(e)$ denote the parse tree of e defined in the usual manner as a tree labeled with operator symbols or process identifiers (in leafs). Given a complete PL-CCS program $\big(\{P_1 \overset{def}{=} e_1, \ldots, P_n \overset{def}{=} e_n\},\ P_1\big)$, we define its *program dependency graph* as the directed labeled graph given as follows: Its nodes comprise those for left-hand sides of the equations, labeled $P_1, \ldots, P_n$, together with the nodes of the parse trees for the right-hand sides of the equations. Its edges comprise the edges of the parse trees plus edges connecting left-hand sides of equations P_i to the roots of parse trees of the corresponding right-hand sides e_i. Additionally, we add edges from leafs of the parse trees labeled P_i to the node for the left-hand side of equation $P_i = e_i$. As an example, consider the following PL-CCS program whose program dependency graph is shown in Figure 1.

$$P \overset{def}{=} (\alpha.P) \oplus (Q \parallel Q) \qquad\qquad Q \overset{def}{=} \beta_1.Nil \oplus \beta_2.Nil$$

We call a node labeled Q *reachable* from a node labeled P if there exists a path from P to Q in its program dependency graph.

Intuitively, a program dependency graph reflects the dependencies between the process identifiers of a PL-CCS program with respect to its defining equations. A cycle in this graph that contains a node labeled by a parallel operator might represent a recursive process definition "spawning" an arbitrary number of copies of its own. If in such a context, the variants operator $\oplus$ comes into play, an unbounded number of configuration selections would be possible. We therefore consider in the following PL-CCS programs which forbid such a situation and thus are configurable within finitely many configuration selections.

Definition 2 (finitely configurable PL-CCS program). *We call a complete PL-CCS program* finitely configurable, *if its program dependency graph has no cycle containing a node labeled with $\|$ from which a node labeled with $\oplus$ is reachable.*

Consider Figure 1. While there is a cycle from P back to P from which a $\oplus$-operator is reachable, the program is finitely configurable as this cycle does not contain a node labeled $\|$. If instead $P \overset{def}{=} (\alpha.P) \parallel (Q \parallel Q)$, the program would not be finitely configurable, as the cycle from P to P would contain the parallel operator, and, still the $\oplus$-operator of the second equation is reachable.

Note that the definition of finitely configurable does not *characterize* the programs that are configurable within finitely many configuration selections, but is just a sufficient condition. However, as it is (already) undecidable whether a CCS program yields a finite or infinite state system, it is easy to see that it is also undecidable whether the transition system defined by a PL-CCS program would make use of only finitely many configuration selections. In the following, we therefore consider only on finitely configurable PL-CCS programs.
There is a further restriction we want to make. Consider the two independent systems P and R:

$$\begin{array}{lcl} P \overset{def}{=} Q \parallel Q & & \\ & \text{vs.} & R \overset{def}{=} Q_1 \oplus Q_2 \parallel Q_1 \oplus Q_2 \\ Q \overset{def}{=} Q_1 \oplus Q_2 & & \end{array}$$

When considering the left system P one might understand its meaning as follows: (i) P consists of two "instances" of the same variation Q. Hence, one selects once between Q_1 and Q_2 and follows this choice for any occurrence of Q in P. However, if we would specify P by expanding the definition of Q in the definition of P, we would get a system like R, which represents another intention: (ii) In R, we now have two (independent) variation points, which— though offering the same variants Q_1 and Q_2— might be configured differently from each other.

So far, the structural semantics rules, as we introduce them in Section 2.3, are only *compositional* for meaning (ii). Therefore—for the scope of this paper and to simplify the technical treatment—we only consider systems like R, where every variants operator can be configured independently from the configuration of other variation points. Note, that it is easy to extend our formalism to actually cope with both meanings, by introducing a second alternative operator with a suitable semantics for the case of P. However, as this would make the current presentation more technical, we refrain from giving this extension in this paper.

In order to allow only systems as in (ii), we consider only *fully expanded* PL-CCS programs:

Definition 3 (fully expanded PL-CCS program). *We call a complete and finitely configurable PL-CCS program* fully expanded, *if its program dependency graph satisfies the following: Removing all edges that are part of strongly connected components (yielding a possibly not connected graph), there is at most one path from every node to any $\oplus$ node.*

Note, that a finitely configurable PL-CCS program which is not fully expanded can be transformed into an equivalent fully expanded version.[1] The Definitions 1 to 3 allow us to characterize the set of *well-formed* PL-CCS programs, which will be the basis for the rest of this paper.

Definition 4 (well-formed PL-CCS program). *A PL-CCS program is* well-formed, *if it is complete, finitely configurable, and fully expanded.*

The rational for the syntactical restrictions leading to Definition 4 is that in a well-formed PL-CCS program we can easily label each variants operator with a unique natural number by parsing over the PL-CCS program and attaching a fresh number to every occurrence of a variants operator. This allows us to precisely define the concept of a variation point: We call a uniquely labeled variants operator with number $i \in \mathbb{N}$, denoted by $\oplus_i$, a *variation point*.

In practical applications, not all combinatorially possible configurations are meaningful or allowed for various non-functional reasons. For example, recall the example from Section 2.1: An OEM might always provide the more advanced keyless locking system whenever a premium infotainment system is selected. Thus, whenever the (optional) infotainment system is chosen, we have to select the keyless variant as well. Such non-functional dependencies between different subsystems are usually captured in a feature model. In their mathematical essence, feature models define a restricted set of configurations. The framework described in this paper does not include such a dependency model and has to be extended to cope with such restrictions. However, to keep the presentation simple, we defer the formal treatment of such feature dependencies to our future work.

2.3 Semantics of a PL-CCS Program

In the following, we define the semantics of a PL-CCS program. More precisely, we introduce three different semantics, the *flat semantics*, the *unfolded semantics*, and the *configured-transitions semantics*, and show how they are related. Basically, the first two semantics are only introduced to motivate and justify the final semantics, the configured-transitions semantics, which will be an appropriate basis for model checking described in Section 3.

[1] This sentence is not to be understood in a mathematical sense, as no semantics for non-fully expanded programs has and will be provided, which does not allow to define *equivalence* precisely.

Flat Semantics The *flat semantics* reflects the intuitive understanding of a PL-CCS program: Every PL-CCS program can be understood as the set of all (plain) CCS programs that can be derived by a total configuration of the PL-CCS program. More precisely, given a well-formed PL-CCS program, we choose for every variants operator either the process term on its left- or right-hand side and remove all the unselected terms together with the respective $\oplus$ symbols from the PL-CCS program. For every such configuration, this procedure results in a plain CCS program, which can be understood in the usual way, e.g. with the semantics described by Milner [Mil80].

Technically, given a well-formed PL-CCS program $Prog$ with $n \in \mathbb{N}$ variants operators, we label every variants operator $\oplus$ uniquely with a number in $\{1, \ldots, n\}$. The individual configuration selection made for the i^{th} $\oplus$-operator is stored in the i^{th} entry θ_i of the *configuration vector* $\theta \in \{R, L, ?\}^n$: The entry R represents the selection of the right process, L represents the selection of the left process, and $?$ represents the situation that none of the two alternatives has been selected so far. We call a configuration vector $\theta \in \{R, L, ?\}^n$ *fully configured* if $\forall i \in \{1, \ldots, n\} : \theta_i \neq \; ?$.

Given a well-formed PL-CCS program $Prog$ with n variants operators and a fully configured configuration vector $\theta \in \{R, L, ?\}^n$ we define a function

$$config : \mathcal{P} \times \{R, L, ?\}^n \rightarrow \mathcal{R}$$

where $\mathcal{R}$ is the set of CCS programs. The function $config$ reduces $Prog$ to a CCS program V, where V is constructed by removing all terms in $Prog$ which are not selected according to θ.

This allows us to define the flat semantics of a PL-CCS program $Prog$ as

$$[\![Prog]\!]_{Flat} = \{[\![V]\!]_{\text{CCS}} \;\;|\;\; \exists\theta : (config(Prog, \theta) = V)\} \tag{2}$$

where $[\![V]\!]_{CCS}$ denotes the conventional CCS semantics of the CCS program V as defined, e.g., in [Mil80] by means of SOS rules. Due to space limitations, we omit to present the original CCS-SOS rules but refer to the PL-CCS-SOS-rules given in Figure 2, which are of the same form as the original ones but additionally carry a configuration vector.

Note that feature constraints can be incorporated in the flat semantics by considering only appropriate configurations θ in Equation 2.

Unfolded Semantics Recall that in the flat semantics, a PL-CCS program gives rise to a *set* of transition systems, one for each fully configured configuration. In the unfolded semantics, the meaning of a PL-CCS program is defined by a *single* labeled transition system modeling the behavior of an entire product family. In particular, by combining the behavior of all derivable systems within *one* labeled transition system, it provides the basis for reducing effort in model checking, by considering commonalities between systems. Before defining the unfolded semantics we introduce a suitable transition system:

A *product-line transition system* (PL-LTS) with n variants operators is a tuple $(\mathcal{S}, \mathcal{A}, \mathit{\Delta}, \sigma)$, where $\mathcal{S}$ is a (countably, possibly infinite) set of states, $\mathcal{A}$ is a set of communication actions, and $\mathit{\Delta}$ is a finite set of transition relations of the form $\xrightarrow{\alpha,\, \nu} \subseteq \mathcal{S} \times \mathcal{S}$, where $\alpha \in \mathcal{A}$, $\nu \in \{R, L, ?\}^n$, and $\sigma \in \mathcal{S}$ is the start state.

$$\frac{P, \nu \xrightarrow{\alpha,\, \nu} P', \nu}{C, \nu \xrightarrow{\alpha,\, \nu} P', \nu}\ ,\ C \overset{\text{def}}{=} P \qquad (\textit{constant definition}) \tag{3}$$

$$\frac{}{\alpha.P, \nu \xrightarrow{\alpha,\, \nu} P, \nu}\ ,\ \text{for arbitrary } \nu \in \{R, L, ?\}^n \qquad (\textit{prefix}) \tag{4}$$

$$\frac{P, \nu \xrightarrow{\alpha,\, \nu} P', \nu}{P + Q, \nu \xrightarrow{\alpha,\, \nu} P', \nu} \qquad \textit{(non-deterministic choice (1))} \tag{5}$$

$$\frac{Q, \nu \xrightarrow{\alpha,\, \nu} Q', \nu}{P + Q, \nu \xrightarrow{\alpha,\, \nu} Q', \nu} \qquad \textit{(non-deterministic choice (2))} \tag{6}$$

$$\frac{P, \nu \xrightarrow{\alpha,\, \nu} P', \nu}{(P \parallel Q), \nu \xrightarrow{\alpha,\, \nu} (P' \parallel Q), \nu} \qquad \textit{(parallel composition (1))} \tag{7}$$

$$\frac{Q, \nu \xrightarrow{\alpha,\, \nu} Q', \nu}{(P \parallel Q), \nu \xrightarrow{\alpha,\, \nu} (P \parallel Q'), \nu} \qquad \textit{(parallel composition (2))} \tag{8}$$

$$\frac{P, \nu \xrightarrow{\alpha,\, \nu} P', \nu \qquad Q, \nu \xrightarrow{\bar{\alpha},\, \nu} Q', \nu}{(P \parallel Q), \nu \xrightarrow{\tau,\, \nu} (P' \parallel Q'), \nu} \qquad \textit{(parallel composition (3))} \tag{9}$$

$$\frac{P, \nu \xrightarrow{\alpha,\, \nu} P', \nu}{P[f], \nu \xrightarrow{f(\alpha),\, \nu} P'[f], \nu} \qquad \textit{(re-labeling)} \tag{10}$$

$$\frac{P, \nu \xrightarrow{\alpha,\, \nu} P', \nu}{(P \setminus L), \nu \xrightarrow{\alpha,\, \nu} (P' \setminus L), \nu}\ ,\ \alpha, \bar{\alpha} \notin L \qquad \textit{(restriction)} \tag{11}$$

Fig. 2. SOS rules for unfolded semantics, except of $\oplus$-operator

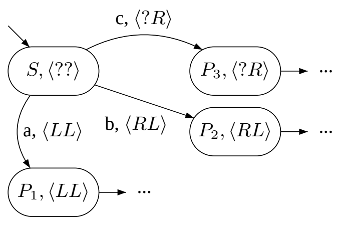
Thus, in a PL-LTS a transition from one state to another is labeled by an action α and an additional (partial) configuration vector ν. However, a transition $s \xrightarrow{\alpha,\, \nu} s'$ represents the set of all transitions $s \xrightarrow{\alpha,\, \nu'} s'$ with ν more general than ν':

Given two vectors $\nu, \nu' \in \{R, L, ?\}^n$, we call ν *more general* than ν', denoted by $\nu \sqsubseteq \nu'$, if $\forall i \in \{1, \ldots, n\} : \big((\nu_i = \ ?) \vee (\nu_i = \nu'_i)\big)$. We say that ν *characterizes* the set of configuration vectors $\{\nu' \mid \nu \sqsubseteq \nu'\}$.

Let us now elaborate on the *unfolded semantics* of PL-CCS programs. Similar as for CCS, we define the labeled transition relation by means of enriched 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 are enriched with a vector ν characterizing the configuration vectors for every transition.

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 given in Figure 2.

For example, rules (3) and (4) express that the execution of an action—specified either directly by action-prefixing as in (4) or indirectly by a constant definition as in



(a) PL-LTS

$$\dfrac{\dfrac{\dfrac{}{a.P_1, \langle LL\rangle \xrightarrow{a,\ \langle LL\rangle} P_1, \langle LL\rangle}}{a.P_1 \oplus_1 b.P_2, \langle ?L\rangle \xrightarrow{a,\ \langle LL\rangle} P_1, \langle LL\rangle}}{(a.P_1 \oplus_1 b.P_2) \oplus_2 c.P_3, \langle ??\rangle \xrightarrow{a,\ \langle LL\rangle} P_1, \langle LL\rangle}$$

(b) deduction

Fig. 3. PL-LTS for $S \overset{def}{=} (a.P_1 \oplus_1 b.P_2) \oplus_2 c.P_3$ and the deduction of transition $\xrightarrow{a,\langle LL\rangle}$.

rule (3)—can be performed without affecting the current configuration ν, i.e. any state $\alpha.P, \nu$ affords a transition labeled with the action α in every possible configuration ν.

Essential for the unfolded semantics is the treatment of the variants operator $\oplus$: Recall that it is a binary operator which allows to model a selection between two alternative processes where only one will be existing in the final system. Though looking similar to the ordinary CCS $+$-operator (which in a way also models a choice between alternatives), it has to be treated different, for two reasons: First, when a selection has been made, the same selection has to be taken when recursively revisiting the same $\oplus$-operator. Second, the choice has to be "made visibly" in the transition relation, to allow further reasoning on each configuration by model checking.

These two issues are captured by the following two SOS rules for the $\oplus$-operator:

$$\frac{P, \nu|_{i/L} \xrightarrow{\alpha,\ \nu'|_{i/L}} P', \nu'|_{i/L}}{P \oplus_i Q, \nu \xrightarrow{\alpha,\ \nu'|_{i/L}} P', \nu'|_{i/L}}\ ,\ \nu_i \neq R \qquad \textit{(configuration selection (1))} \qquad (12)$$

$$\frac{Q, \nu|_{i/R} \xrightarrow{\alpha,\ \nu'|_{i/R}} Q', \nu'|_{i/R}}{P \oplus_i Q, \nu \xrightarrow{\alpha,\ \nu'|_{i/R}} Q', \nu'|_{i/R}}\ ,\ \nu_i \neq L \qquad \textit{(configuration selection (2))} \qquad (13)$$

Here, $\nu|_{i/x}$ represents the updated vector ν where the entry at the i^{th} position is replaced by the value $x \in \{R, L\}$. All other entries keep their values, i.e. $\forall\, j \neq i : (\nu|_{i/x})_j = \nu_j$. Recall that ν_i yields the i^{th} element of the vector ν. Further note that the respective conditions of the alternative rules prevent the user from selecting a different alternative when re-entering the selection decision due to a recursive process present in CCS.

We define the *unfolded semantics* of a PL-CCS program *Prog*, denoted by $[\![Prog]\!]_{UF}$, as the PL-LTS obtained by applying the SOS rules to the main process identifier.

As an example, Figure 3(a) shows the PL-LTS when applying the *configuration selection* rules to the PL-CCS program starting with the main process definition $S \overset{def}{=}$

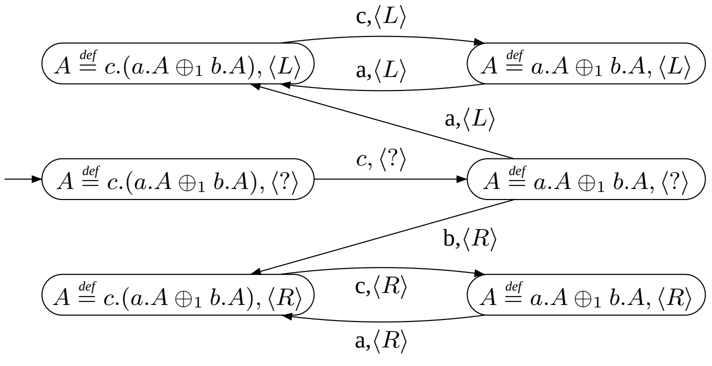

c,$\langle L\rangle$
$A \overset{def}{=} c.(a.A \oplus_1 b.A), \langle L\rangle$
a,$\langle L\rangle$
$A \overset{def}{=} a.A \oplus_1 b.A, \langle L\rangle$
a,$\langle L\rangle$
c, $\langle ?\rangle$
$A \overset{def}{=} c.(a.A \oplus_1 b.A), \langle ?\rangle$
$A \overset{def}{=} a.A \oplus_1 b.A, \langle ?\rangle$
b,$\langle R\rangle$
c,$\langle R\rangle$
$A \overset{def}{=} c.(a.A \oplus_1 b.A), \langle R\rangle$
$A \overset{def}{=} a.A \oplus_1 b.A, \langle R\rangle$
a,$\langle R\rangle$

Fig. 4. PL-LTS for the PL-CCS term $A \overset{def}{=} c.(a.A \oplus b.A)$

$(a.P_1 \oplus_1 b.P_2) \oplus_2 c.P_3$. Since the presence of $c.P_3$ in the final configuration only requires to select the right variant at the variation point $\oplus_2$, the corresponding transition to state $P_3, \langle ?R\rangle$ only fixes the second entry of the configuration vector to the value R while leaving any choice for the first entry (?). In contrast to that, the selection of either P_1 or P_2 requires to take two configuration decisions, reflected by the vectors $\langle LL\rangle$ and $\langle RL\rangle$ in the respective states $P_1, \langle LL\rangle$ and $P_2, \langle RL\rangle$. A corresponding deduction (applying twice Rule 12) for the selection of the variant P_1 is given in Figure 3(b). Note that the derivation shows that the semantics can require several configuration selections for deriving a single transition.

Figure 4 shows an example for the configuration selection rules for recursive process definitions. More specifically, the PL-LTS for the PL-CCS program $A \overset{def}{=} c.(a.A \oplus b.A)$ is shown. The state labels correspond to the process term together with the configuration under which they were reached. If the semantics would only depend on the current state's CCS-term (and not additionally on the configuration selected so far), the states at the left and the right column could not be told apart, since the process term is the same for all three states in one column. But since the unfolded semantics keeps track of which configuration was chosen so far, identical PL-CCS terms yield different states in the PL-LTS under different configurations. More precisely, this means that in the state labeled with $A \overset{def}{=} a.A \oplus_1 b.A, \langle L\rangle$ the semantics does not allow to have an outgoing transition labeled with $\xrightarrow{b,\langle R\rangle}$, since the dual configuration $\langle L\rangle$ has already been selected.

While the unfolded semantics is easily understood and does indeed represent the behavior of a PL-CCS program within a single transition system, the previous example leads one to suspect that the unfolded semantics yields non-compact transition systems. In the next section, we introduce a configured-transitions semantics, which is based on the unfolded semantics yet yields smaller transition systems.

Let us elaborate on the correctness of the unfolded semantics in a sense made precise below. Therefore, recall that two transition systems are called bisimilar, denoted by $\approx$, when, starting at the initial states, every transition of one system can be simulated by one of the other system and vice versa (see [Mil95] for a precise definition). From a

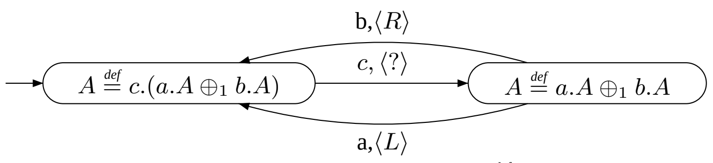

b,$\langle R\rangle$
c, $\langle ?\rangle$
$A \overset{def}{=} c.(a.A \oplus_1 b.A)$
$A \overset{def}{=} a.A \oplus_1 b.A$
a,$\langle L\rangle$

Fig. 5. Configured-transitions semantics for $A \overset{def}{=} c.(a.A \oplus b.A)$

PL-LTS, we obtain for a given configuration vector θ a labeled transition system by projecting to those states and transitions whose vector ν is more general, i.e. where $\nu \sqsubseteq \theta$, and discarding all other transitions. For a PL-CCS program with unfolded semantics $[\![Prog]\!]_{UF}$, let the transition system obtained in this way be denoted by $\Pi_\theta([\![Prog]\!]_{UF})$.

The following theorem states that (modulo bisimulation) the systems given in terms of the flat semantics and the unfolded semantics coincide.

Theorem 1 (Correctness of unfolded semantics). *Given a PL-CCS program $Prog$ and a configuration vector θ,*

$$[\![config(Prog, \theta)]\!]_{CCS} \approx \Pi_\theta([\![Prog]\!]_{UF})$$

Due to space limitations we omit the proof here and refer to an extended version of the paper [GLS08].

Configured-transitions Semantics In the following, we give a further semantics for a PL-CCS program which yields a smaller transition system and, at the same time, caters for model checking the entire product line as described in the next section. The idea is to identify states that have the same PL-CCS process term but only differ in the corresponding configuration vector.

Let $\overset{\alpha,\nu}{\Longrightarrow} \subseteq \mathcal{P} \times \mathcal{P}$ be defined by

$$P \overset{\alpha,\nu}{\Longrightarrow} P' \text{ iff there exists } \nu' \text{ with } P, \nu' \xrightarrow{\alpha,\ \nu} P', \nu$$

where $\alpha \in \mathcal{A}$ and $\nu, \nu' \in \{L, R, ?\}^n$ and $\xrightarrow{\alpha,\ \nu}$ is the relation defined in the previous section.

We define the *Configured-transitions semantics* of a PL-CCS program $Prog$, denoted by $[\![Prog]\!]_{CT}$, as the PL-LTS consisting of states reachable from the main process identifier wrt. $\overset{\alpha,\nu}{\Longrightarrow}$ and corresponding transition relations.

Figure 5 shows the transition system for the program $A \overset{def}{=} c.(a.A \oplus b.A)$. A comparison with Figure 4 showing the unfolded semantics for the same program shows that the configured-transitions semantics yields indeed smaller transition systems.

For any PL-CCS program $Prog$, every path in $[\![Prog]\!]_{UF}$ corresponds to one execution of one product of the family. This does no longer hold for the paths of $[\![Prog]\!]_{CT}$. For example, the path $cacb$ in the system shown in Figure 5 does not exist in any of the transition systems of $[\![A \overset{def}{=} c.(a.A \oplus b.A)]\!]_{Flat}$. However, the interesting property of the configured-transitions semantics is that for every configuration vector θ, the projection of $\Pi_\theta([\![Prog]\!]_{CT})$, similarly defined as for $[\![Prog]\!]_{UF}$, yields the same transition system (modulo isomorphism) as the one obtained when projecting $Prog$ wrt. θ and taking the CCS semantics:

Theorem 2 (Correctness of configured-transitions semantics). *Given a PL-CCS program $Prog$ and a configuration vector θ,*

$$[\![config(Prog, \theta)]\!]_{CCS} = \Pi_\theta([\![Prog]\!]_{CT})$$

A corresponding proof can be found in [GLS08].

3 Model Checking Product Lines

In this section, we introduce a *multi-valued modal version* of the μ-calculus suitable for specifying properties of individual configurations of a PL-CCS program. Furthermore, we sketch a game-based and therefore on-the-fly model checking approach for PL-CCS programs with respect to μ-calculus specifications.

We have chosen to develop our verification approach for specifications in the μ-calculus as it subsumes lineartime temporal logic as well as computation-tree logic as first shown in [EL86,Wol] 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 μ-calculus combines Kozen's modal μ-calculus [Koz83] and multi-valued μ-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 unlabeled diamond and box operators, by providing also action-labeled versions of these operators, which is essential to formulate properties of PL-CCS programs.[2]

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 $\neg x \sqsubseteq y$. 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.

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

Multi-valued modal μ-calculus Let $\mathcal{P}$ be a set of *propositional constants*, and $\mathcal{A}$ be a set of *action names*.[3]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 α-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 $\perp$ and a top $\top$ element, ordered in the expected manner. Value $\top$ then means that the property holds in the considered state while $\perp$ means that it does not hold. Similarly, $\mathcal{R}_\alpha(s, s') = \top$ reads as there is a corresponding a transition while $\mathcal{R}_\alpha(s, s') = \perp$ means there is no α-transition.

Let $\mathcal{V}$ be a set of propositional variables. Formulae of the *multi-valued modal μ-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 μ and ν are variable binders. We will also write η for either μ or ν. 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 subformula $\mathit{fp}(Z) = \eta Z.\psi$ of φ, where the set $\mathit{Sub}(\varphi)$ of *subformulae* of φ 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 φ 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 the meaning of free variables in φ, is an element of $\mathcal{L}^{\mathcal{S}}$ and is defined as shown in Figure 6. 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 ρ 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 α-successor satisfying φ. In a multi-valued version, we first consider the value of α-transitions and reduce it (meet it) with the value of φ in the successor state. As there might be different α-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 α-successor satisfying φ. Dually, $[\alpha]\varphi$ is

[3] 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.

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

Fig. 6. Semantics of mv-$\mathfrak{L}_\mu$ formulae

classically supposed to hold in states for which all α-successors satisfy φ. In a multi-valued version, we first consider the value of α-transitions and increase it (join it) with the value of φ in the successor state. As there might be several different α-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 α-successors satisfy φ.

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, φ and $\mathcal{S}$. According to [Tar55], least and greatest fixpoints of these functionals exist.

Approximants of mv-$\mathfrak{L}_\mu$ 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 α and any environment ρ, and $Z^\lambda := \bigsqcap_{\alpha<\lambda} Z^\alpha$ for a limit ordinal λ. 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 3 (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 $[\![\eta Z.\varphi]\!]_\rho(s) = x$ then $Z^\alpha(s) = x$.*

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

Theorem 4 (Correctness of Model Checking). *For all PL-CCS programs $Prog = (\mathcal{E}, P_1)$, every configuration vector ν, and formulae $\varphi \in mv$-$\mathfrak{L}_\mu$, we have*

$$[\![\mathit{config}(Prog,\nu)]\!]_{CCS} \models \varphi \text{ iff } \nu \in ([\![Prog]\!]_{CT} \models \varphi)(P_1)$$

The proof follows by structural induction on the formula.

While Theorem 3 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 context of multi-valued μ-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 μ-calculus, a slight adaption of the approach taken in [SG05] yields game-based approach for the full multi-valued modal μ-calculus.

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

4 Specification and Verification of a Sample Product-Line

Let us now demonstrate our approach on a simplified version of an industrial case study we have been working on. We consider a product line whose configurations realize different versions of a windscreen wiper system.

Specification At first, we specify the family of systems, using the formalism introduced in Section 2. The windscreen wiper systems that we specify in our family WipFam are each built of two subcomponents: a rain sensor, Sensor, and a windscreen wiper, Wiper. Both subcomponents can be realized by two variants, a high and a low one, respectively:

$$\mathit{WipFam} \overset{\text{def}}{=} \mathit{Sensor} \parallel \mathit{Wiper} \tag{E1}$$

$$\mathit{Sensor} \overset{\text{def}}{=} \mathit{SensL} \oplus_1 \mathit{SensH} \tag{E2}$$

$$\mathit{Wiper} \overset{\text{def}}{=} \mathit{WipL} \oplus_2 \mathit{WipH} \tag{E3}$$

The low variant SensL of the sensor is specified as follows:

$$\mathit{SensL} \overset{\text{def}}{=} \mathit{non}.\mathit{SensL} + \mathit{ltl}.\mathit{Raining} + \mathit{hvy}.\mathit{Raining} + \overline{\mathit{noRain}}.\mathit{SensL} \tag{E4}$$

$$\mathit{Raining} \overset{\text{def}}{=} \mathit{non}.\mathit{SensL} + \mathit{ltl}.\mathit{Raining} + \mathit{hvy}.\mathit{Raining} + \overline{\mathit{rain}}.\mathit{Raining} \tag{E5}$$

The low variant SensL only detects two different environmental conditions—dry and raining—even though the environment can stimulate the sensor with three different conditions: hvy for heavy rain, ltl for little rain and non for no rain. However, this sensor cannot differ between heavy and little rain, i. e. for this sensor, hvy and ltl have the same effect, as the sensor reaches a process $\mathit{Raining}$ and provides an action $\overline{\mathit{rain}}$, indicating solely the fact that it is raining (without precisely characterizing the intensity). As long as no rain has been detected, the sensor provides the action $\overline{\mathit{noRain}}$, respectively.

The high version of the sensor can distinguish between different degrees of rain intensity, i. e. SensH additionally differentiates heavy rain from little rain. Its PL-CCS specification is given in the following:

$$\mathit{SensH} \overset{\text{def}}{=} \mathit{non}.\mathit{SensH} + \mathit{ltl}.\mathit{Medium} + \mathit{hvy}.\mathit{Heavy} + \overline{\mathit{noRain}}.\mathit{SensH} \tag{E6}$$

$$\mathit{Medium} \overset{\text{def}}{=} \mathit{non}.\mathit{SensH} + \mathit{ltl}.\mathit{Medium} + \mathit{hvy}.\mathit{Heavy} + \overline{\mathit{rain}}.\mathit{Medium} \tag{E7}$$

$$\mathit{Heavy} \overset{\text{def}}{=} \mathit{non}.\mathit{SensH} + \mathit{ltl}.\mathit{Medium} + \mathit{hvy}.\mathit{Heavy} + \overline{\mathit{hvyRain}}.\mathit{Heavy} \tag{E8}$$

In this product line, the sensors can be arbitrarily combined with two variants of windscreen wipers, WipL and WipH. In particular, for this example we have no additional non-functional dependencies between the possible variants which would restrict the set of combinatorially possible configurations.

The low version WipL offers two operation modes: (i) a manual mode with perpetual wiper arm movement (action $\mathit{permWip}$), which has to be activated explicitly by the driver, (ii) and a semi-automatic interval mode in which the wiper arm moves at a lower frequency triggered by the rain sensor (via the action $rain$).

$$\begin{aligned} \mathit{WipL} &\overset{\mathit{def}}{=} \mathit{off}.\mathit{WipL} + \mathit{manualOn}.\mathit{Permanent} + \mathit{intvOn}.\mathit{Interval} && \text{(E9)}\\ \mathit{Interval} &\overset{\mathit{def}}{=} \mathit{noRain}.\mathit{Interval} + \mathit{intvOff}.\mathit{WipL} + \mathit{intvOn}.\mathit{Interval} && \text{(E10)}\\ &\quad + \mathit{rain}.\mathit{Wiping} + \mathit{hvyRain}.\mathit{Wiping} \\ \mathit{Wiping} &\overset{\mathit{def}}{=} \overline{\mathit{slowWip}}.\mathit{Interval} + \mathit{intvOn}.\mathit{Interval} && \text{(E11)}\\ \mathit{Permanent} &\overset{\mathit{def}}{=} \overline{\mathit{permWip}}.\mathit{Permanent} + \mathit{off}.\mathit{WipL} + \mathit{intvOn}.\mathit{Interval} && \text{(E12)} \end{aligned}$$

The high variant WipH can operate at two speeds: slow (action: $\overline{\mathit{slowWip}}$) and fast (action: $\overline{\mathit{fastWip}}$). Here, the wiper arm movement is fully controlled by the rain sensor and adjusts its frequency automatically to the current rain intensity.

$$\begin{aligned} \mathit{WipH} &\overset{\mathit{def}}{=} \mathit{off}.\mathit{WipH} + \mathit{intvOn}.\mathit{AutoIntv} && \text{(E13)}\\ \mathit{AutoIntv} &\overset{\mathit{def}}{=} \mathit{noRain}.\mathit{AutoIntv} + \mathit{intvOn}.\mathit{AutoIntv} + \mathit{rain}.\mathit{Slow} && \text{(E14)}\\ &\quad + \mathit{intvOff}.\mathit{WipH} + \mathit{hvyRain}.\mathit{Fast} \\ \mathit{Slow} &\overset{\mathit{def}}{=} \overline{\mathit{slowWip}}.\mathit{AutoIntv} + \mathit{intvOn}.\mathit{AutoIntv} && \text{(E15)}\\ \mathit{Fast} &\overset{\mathit{def}}{=} \overline{\mathit{fastWip}}.\mathit{AutoIntv} + \mathit{intvOn}.\mathit{AutoIntv} && \text{(E16)} \end{aligned}$$

The PL-CCS program specifying the entire product line WipFam is given by the equations E1–E16. The whole program WipFam is well-formed, which allows a unique numbering of all (two) variation points as shown by Equations E2 and E3.

Verification From our example system family WipFam, we can derive four different individual systems, as we can combine the subsystem variants arbitrarily. Having specified the family in PL-CCS, we can now apply the model checking approach described in Section 3, in order to verify functional properties for configurations in the system family.

Thinking of a relevant property, for instance, one could possibly be interested in verifying for a windscreen wiping system whether or not a driver is always able to switch to automatic windscreen wiping mode. (Property 1, formalized in Equation 14). Another property could demand the windscreen wiper to wipe fast, once it is raining heavily (Property 2, formalized in Equation 15).

$$\mu X.\langle . \rangle X \vee \langle \mathit{intvOn} \rangle \mathit{true} \tag{14}$$

$$\nu Y.[.]Y \wedge (\neg \langle \mathit{intvOff} \rangle \mathit{true} \vee [\mathit{hvy}]\langle \overline{\mathit{fastWip}} \rangle \mathit{true}) \tag{15}$$

In our example, Property 1 holds for the set of all possible configurations $\langle L, L\rangle$, $\langle R, L\rangle$, $\langle L, R\rangle$,and $\langle R, R\rangle$, which can be denoted by the single vector $\langle ?, ?\rangle$. However, Property 2 is only satisfied in the configuration, in which the high variants of both subsystems are used, i. e. the result of applying the proposed model checking algorithm is the set containing the single configuration vector $\langle R, R\rangle$. Intuitively, it is easy to see why: As the low version of the windscreen wiper does not provide a fast wiping mode, it never provides the output action $\overline{\mathit{fastWip}}$. In consequence, the wind screen wiper can never wipe fast if the low version is used. However, even if the high version of the windscreen wiper is used, but combined with the low version of the rain sensor, the property is still not satisfied. The sensor is not able to provide the output action $\overline{hvyRain}$, which would trigger the wiper to wipe fast. Using our product line specific model checking approach, we are able to identify the configurations which do and do not satisfy a verified property—and which we so far motivated only illustratively—in an automatic way.

5 Conclusion

In this paper, we propose a process algebra approach to software product lines that allows automatic analysis and verification by means of model checking. We introduced PL-CCS as a variant of Milner's CCS designed to model the overall behavior of similar software systems developed as a software product line. Its semantics can conveniently be defined in terms of multi-valued modal Kripke structures. Furthermore, we introduced multi-valued modal μ-calculus as a property specification language for systems formulated in PL-CCS. Model checking then allows to verify either an entire software product line, or, to point out which variants of the product line do not meet given correctness properties. We are currently working on algebraic properties of PL-CCS, on the integration of a dependency model for modeling feature constraints, as well as on an implementation of the proposed model checking approach.

Acknowledgement: We thank Mila Mejster-Cederbaum for valuable comments on an earlier draft of this paper.

References

[BLS06] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT—structured assertion language for temporal logic. In *Proceedings of the 8th International Conference on Formal Engineering Methods*, volume 4260 of *Lecture Notes in Computer Science*, September 2006.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines. Practices and Patterns.* Addison Wesley, 2002.

[Dam94] Mads Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, April 1994.

[EJS93] "E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla". On model-checking for fragments of mu-calculus. In C. Courcoubetis, editor, *Proc. 5th International Computer-Aided Verification Conference*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 1993.

[EL86] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Symposium on Logic in Computer Science (LICS '86)*, pages 267–278, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[FUB07] Dario Fischbein, Sebatian Uchitel, and Victor Braberman. A foundation for behavioural conformance in software product line architectures. In *Proceedings of the 2nd Workshop on the Role of Software Architecture for Testing and Analysis*, 2007.

[GLS08] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modelling and Model Checking Software Product Lines. Technical Report TUM-I0806, Technische Universität München, February 2008.

[KNK05] Tomoji Kishi, Natsuko Noda, and Takuya Katayama. Design verification for product line development. In J. Henk Obbink and Klaus Pohl, editors, *9th International Conference on Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 150–161. Springer, September 2005.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[LKF05] Harry C. Li, Shriram Krishnamurthi, and Kathi Fisler. Modular verification of open features using three-valued model checking. *Automated Software Engineering*, 2005.

[LNW07] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O automata for interface and product line theories. In Rocco De Nicola, editor, *16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79. Springer, April 2007.

[LT91] Kim Guldstrand Larsen and Bent Thomsen. Partial specifications and compositional verification. *Theor. Comput. Sci.*, 88(1):15–32, 1991.

[MC01] Mila E. Majster-Cederbaum. Underspecification for a simple process algebra of recursive processes. *Theor. Comput. Sci.*, 266(1-2):935–950, 2001.

[Mil80] Robin Milner. *A Calculus for Communicating Processes*, volume 92 of *LNCS*. Springer, 1980.

[Mil95] Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin Heidelberg New York, 2005.

[SG05] Sharon Shoham and Orna Grumberg. Multi-valued model checking games. In Doron Peled and Yih-Kuen Tsay, editors, *3rd International Symposium on Automated Technology for Verification and Analysis*, volume 3707 of *Lecture Notes in Computer Science*, pages 354–369. Springer, October 2005.

[Sti95] Colin Stirling. Local model checking games. In Insup Lee and Scott A. Smolka, editors, *6th International Conference on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 1–11. Springer, August 1995.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific J.Math.*, 5:285–309, 1955.

[VN98] Simone Veglioni and Rocco De Nicola. Possible worlds for process algebras. In Davide Sangiorgi and Robert de Simone, editors, *9th International Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 179–193. Springer, September 1998.

[Wol] Pierre Wolper. A translation from full branching time temporal logic to one letter propositional dynamic logic with looping. unpublished manuscript.