Compositional Feature-Oriented Systems*

Author's Version – September 12, 2019

Clemens Dubslaff

Technische Universität Dresden, Germany
clemens.dubslaff@tu-dresden.de

Abstract. Feature-oriented systems describe system variants through features as first-class abstractions of optional or incremental units of systems functionality. The choice how to treat modularity and composition in feature-oriented systems strongly influences their design and behavioral modeling. Popular paradigms for the composition of features are superimposition and parallel composition. We approach both in a unified formal way for programs in guarded command language by introducing *compositional feature-oriented systems (CFOSs)*. We show how both compositions relate to each other by providing transformations that preserve the behaviors of system variants. Family models of feature-oriented systems encapsulate all behaviors of system variants in a single model, prominently used in family-based analysis approaches. We introduce *family-ready* CFOSs that admit a family model and show by an annotative approach that every CFOS can be transformed into a family-ready one that has the same modularity and behaviors.

1 Introduction

Feature-oriented systems [29,18,2] excel in their concept of behavioral modularity [40,31] that is provided through features, i.e., first-class abstractions of an optional or incremental unit of functionality [40]. They are first and foremost used to model *software product lines (SPLs)* [16] where each software product corresponds to a combination of features. However, feature-oriented concepts have shown to be applicable in a wide range of areas, e.g., to model contexts [1] or heterogeneous (hardware) systems [22,21,5]. A central aspect within feature-oriented systems is the actual construction of a product from a given feature combination [30]. Annotative approaches, prominently applied in *featured transition systems (FTSs)* [14], incorporate all behaviors of any product in a single *family model* by annotating *feature guards* to behaviors. Behaviors are then effective in those products where the feature combination fulfills the feature guard. Family models are successful in the context of feature-oriented system

* The author has been supported by the DFG through the Cluster of Excellence EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy), the Collaborative Research Centers CRC 912 (HAEC) and TRR 248 (see https://perspicuous-computing.science, project ID 389792660), the Research Training Group RoSI (GRK 1907), Deutsche Telekom Stiftung, and the 5G Lab Germany.

analysis: a symbolic representation of the model in combination with a single analysis run may avoid the exponential blowup in the number of features that arises when analyzing every product one-by-one [15,21]. Opposed to annotative approaches, compositional approaches model the behaviors of features separately through *feature modules* [31,21] that are composed towards a product. The de-facto standard composition operators for feature modules are *superimposition* [8,33] and *parallel composition* [36], both syntactically defined on the chosen behavioral formalism for feature modules. Superimposition describes how a base behavior is changed when composing a feature module, a concept also apparent in delta-oriented modeling [39]. This approach is mainly used in the software-engineering domain and formalized, e.g., for feature-oriented variants of Java [3,7] and C++ [4]. But also for low-level programming languages such as guarded command languages [20], superimposition approaches have been applied for the analysis of feature-oriented systems [37,13]. On the other hand, parallel composition focuses on the interaction between composed feature modules via shared actions. Paramount in formal methods, it is not surprising that parallel-composition approaches are mainly used when it comes to the verification and analysis of feature-oriented systems [25,14,22]. Verification tools mostly rely on an input language based on guarded commands, which lead to fPromela [12] and ProFeat [9], feature-oriented extensions of the model-checker input languages of SPIN [27] and Prism [34], respectively.

Although both composition operators are widely used for the design and analysis of feature-oriented systems and there are common foundations in the case of guarded command languages, yet there does not exist any framework that covers both composition operators. We introduce *compositional feature-oriented systems (CFOSs)* that follow the usual two-level approach for feature-oriented systems comprising a feature model and behavioral model, but with the focus on compositional specification. In particular, we consider feature modules given in a featured variant of guarded command language as behavioral model, composed towards products specified in the feature model through parallel composition or superimposition. Provided the concept of CFOSs, we mainly answer the following research questions in this paper:

(RQ1) Is there an automated annotative approach for CFOSs that admits a family model and maintains behaviors, modularity, and locality?

(RQ2) Are there automated translations between CFOSs on superimposition and parallel composition that maintain behaviors, modularity, and locality?

Here, maintaining modularity is understood as preserving the feature model and the assignment of feature modules to features, while maintaining locality [31] ensures that variables over which the feature modules are specified do not change. Answering **(RQ1)** positively would provide a unified annotative and compositional approach to specify feature-oriented systems. For this, we introduce *family-ready* CFOSs where composing all feature modules yields a family model for the whole feature-oriented system but still allows for the compositional construction of single products. We show that any CFOS can be turned into a family-ready CFOS of polynomial size, mainly following a *lifting*

approach [38]. Family-ready CFOSs facilitate the specification of compositional *dynamic feature-oriented systems* [24,19,21], i.e., systems where feature combinations can change during runtime. Given a reconfiguration graph that describes the changes of feature combinations, the behavior of the dynamic (family-ready) CFOS is then provided by a simple product construction that resolves the feature guards in the composition of all features in the CFOS. The question **(RQ2)** addresses the relationship between superimposition and parallel composition and a positive answer would provide the foundations to use both formalisms interchangeably, e.g., analyzing feature-oriented systems specified with superimposition with tools that have an input language based on parallel composition or vice versa, generating programs based on superimposition out from verified CFOSs based on parallel composition. We show that any parallel-composition CFOS can be transferred into a superimposition CFOS that has exponential size and maintains behaviors, modularity, and locality. The converse is only possible not requiring locality and we present a transformation that turns any superimposition CFOSs into an exponentially-sized parallel-composition CFOS.

Further Related Work. In [23], a superimposition operator on interacting parallel processes (an extension of guarded command language) has been considered. While we consider superimposition on the same level as parallel composition, they developed a calculus where superimposition is applied on processes appearing in a fixed parallel composition.

Transformations akin to the one addressed in **(RQ1)** have been already addressed in the context of FTSs and their probabilistic counterparts. The lifted feature composition of [13] is a superimposition variant that incorporates case distinctions depending on possible feature combinations in the input language of NuSMV [11]. Their approach requires a modified composition other than the standard one in NuSMV and, applied to our guarded command language setting, would yield an exponential blowup while we present a polynomial translation. Also in [12] and [9], the lifting approach has been applied towards family models specified in fPromela and ProFeat, respectively. However, their semantics might introduce stutter steps and thus is not family-ready. Concerning dynamic feature-oriented systems, [24] and [19] describe feature switches through reconfiguration patterns and rules, respectively, while we follow the concept of reconfiguration graphs also used in [21]. However, [21] inherently requires the feature-oriented system specified to be family-ready while this is usually not the case during the development: feature modules do not a priori include information about all other features of the feature-oriented system to maintain reusability within similar but different systems.

2 Theoretical Foundations

In this section we introduce our formal framework used throughout the paper. Although not yet been considered like this in the literature, it mainly relies on standard concepts [20,21,14]. We denote by $\wp(X)$ the power set of a set X. Given

partial functions $f_i \colon X_i \rightharpoonup Y_i$ for $i \in \{1,2\}$ we define $f_1 \rtimes f_2 \colon X_1 \cup X_2 \rightharpoonup Y_1 \cup Y_2$ by $(f_1 \rtimes f_2)(x) = f_2(x)$ in case $f_2(x)$ is defined and $(f_1 \rtimes f_2)(x) = f_1(x)$ otherwise.

Interfaces. An *interface* $I = \langle Int, Ext \rangle$ with $Int \cap Ext = \varnothing$ characterizes *internal* and *external* elements through finite sets Int and Ext, respectively. If there is no chance of confusion we sometimes write I for the set $Int \cup Ext$. In case two interfaces $\langle X, X' \rangle$ and $\langle Y, Y' \rangle$ have disjoint internal elements, i.e., $X \cap Y = \varnothing$, they are *composable*. We define a composition operator $\oplus$ where $\langle X, X' \rangle \oplus \langle Y, Y' \rangle = \langle Z, Z' \backslash Z \rangle$ with $Z = X \cup Y$ and $Z' = X' \cup Y'$.

Arithmetic Expressions and Constraints. Let Var be a finite set of *variables*, on which we define *evaluations* as functions $\eta \colon \mathit{Var} \to \mathbb{Z}$. The set of evaluations over Var is denoted by $\mathit{Eval}(\mathit{Var})$. Let z range over $\mathbb{Z}$ and v range over Var, then the set of *arithmetic expressions* $\mathbb{A}(\mathit{Var})$ is defined by the grammar $a ::= z \mid v \mid (a + a) \mid (a \cdot a)$. Variable evaluations are extended to arithmetic expressions by $\eta(z) = z$, $\eta(a_1 + a_2) = \eta(a_1) + \eta(a_2)$, and $\eta(\alpha_1 \cdot \alpha_2) = \eta(a_1) \cdot \eta(a_2)$. $\mathbb{C}(\mathit{Var})$ denotes the set of *constraints* over Var, i.e., terms of the form $(a \sim z)$ with $a \in \mathbb{A}(\mathit{Var})$, $\sim \in \{>, \geq, =, \leq, <, \neq\}$, and $z \in \mathbb{Z}$. For a given evaluation $\eta \in \mathit{Eval}(\mathit{Var})$ and constraint $(a \sim z) \in \mathbb{C}(\mathit{Var})$, we write $\eta \models (a \sim z)$ iff $\eta(a) \sim z$. Note that with integer-valued variables we can mimic Boolean variables $x \in \mathit{Var}$ by identifying x with $(x \geq 1)$ and $\neg x$ with $(x = 0)$.

Boolean Expressions. For a countable nonempty set X, we define *Boolean expressions* $\mathbb{B}(X)$ by the grammar $\psi ::= \mathtt{tt} \mid x \mid \neg\psi \mid \psi \wedge \psi$ where $x \in X$. We might denote by $\mathtt{ff} = \neg\mathtt{tt}$ and use well-known Boolean connectives such as disjunction $\vee$, implication $\rightarrow$, etc. from which a Boolean expression can be easily obtained using standard syntactic transformations. The *satisfaction relation* for Boolean expressions $\models \subseteq \wp(X) \times \mathbb{B}(X)$ is defined in the usual way, i.e., for any $Y \subseteq X$: $Y \models x$ iff $x \in Y$, $Y \models \neg\psi$ iff $Y \not\models \psi$, and $Y \models \psi_1 \wedge \psi_2$ iff $Y \models \psi_1$ and $Y \models \psi_2$. For an evaluation $\eta \in \mathit{Eval}(\mathit{Var})$ and $\psi \in \mathbb{B}(\mathbb{C}(\mathit{Var}))$, we write $\eta \models \psi$ iff $\{c \in \mathbb{C}(\mathit{Var}) : \eta \models c\} \models \psi$.

Transition Systems. A *transition system* is a tuple $\mathcal{T} = (S, \mathit{Act}, T, \iota)$ where S is a set of states, Act a finite set of actions, $T \subseteq S \times \mathit{Act} \times S$ is a transition relation, and $\iota \in S$ is an initial state. We usually write $s \xrightarrow{\alpha} s'$ in case $(s, \alpha, s') \in T$. For transition systems $\mathcal{T}_i = (S_i, \mathit{Act}_i, T_i, \iota_i)$ with $i \in \{1,2\}$ we denote by $\underline{S_i}$ the set of *reachable* states in $\mathcal{T}_i$, i.e., $\underline{S_i} \subseteq S_i$ is the smallest set for which $\iota_i \in \underline{S_i}$ and where for any $s \in \underline{S_i}$, $(s, \alpha, s') \in T_i$ we have $s' \in \underline{S_i}$. We call $\mathcal{T}_1$ and $\mathcal{T}_2$ *equivalent up to isomorphism*, denoted $\mathcal{T}_1 \cong \mathcal{T}_2$, if there is a bijection $\xi \colon \underline{S_1} \to \underline{S_2}$ such that $\xi(\iota_1) = \iota_2$ and $(s, \alpha, s') \in T_1$ iff $(\xi(s), \alpha, \xi(s')) \in T_2$.

2.1 Feature Models

Feature-oriented systems are usually specified by a *feature model* and a *featured behavioral model*. Given an abstract set of features F, a feature model $\mathcal{F}$ expresses variability in the system over valid combinations of features $\mathfrak{V}[\mathcal{F}] \subseteq \wp(F)$. The featured behavioral model describes the operational behaviors of features, i.e., their actual functionality.

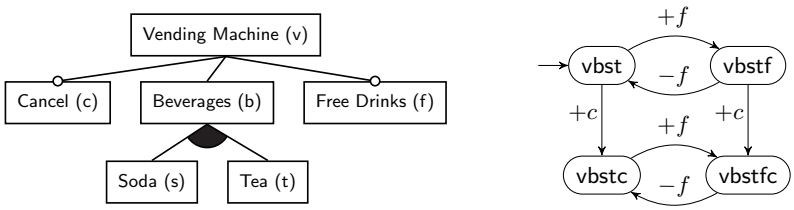

Fig. 1. Feature diagram (left) and reconfiguration graph (right) for a vending machine

Feature Diagrams. The de-facto standard feature model for static feature-oriented systems such as software product lines (SPL) is provided through *feature diagrams* [29]. They are tree-like hierarchical structures over nodes representing features. Figure 1 (left) depicts a feature diagram of a vending machine SPL [6,14] over features $F = \{\mathsf{v}, \mathsf{b}, \mathsf{s}, \mathsf{t}, \mathsf{f}, \mathsf{c}\}$. Feature v encapsulates the basic functionality of a vending machine, b has the functionality of providing drinks to the customer, either soda (feature s), or tea (feature t), or both. While usually drinks have to be paid, the optional feature f (indicated by the circle on top of the feature) adds the possibility to provide free drinks. When the second optional feature c is active, the customer can cancel any vending process, leading to a cash back. In our setting, it suffices to regard a feature diagram $\mathcal{D}$ as a compact representation of *valid feature combinations* $\mathfrak{V}[\mathcal{D}] \subseteq \wp(F)$. Within the vending machine SPL, there are 12 valid feature combinations.

Reconfiguration Graphs. *Dynamic feature-oriented systems* [24,26] allow to change feature combinations during system execution. A *reconfiguration graph* [19] (also called feature controller [21]) describes feature changes by a transition system $\mathcal{G} = \big(Loc \times \wp(F), Act, R, (\ell_0, I)\big)$ where Loc is a finite set of locations and $(\ell_0, I) \in Loc \times \wp(F)$ is an initial location with an initial feature combination. The set of valid feature combinations $\mathfrak{V}[\mathcal{G}]$ is the set of feature combinations reachable in $\mathcal{G}$. In Figure 1 (right) a reconfiguration graph for a vending machine SPL is depicted, where we abbreviate a node $(\ell, \{\mathsf{f}_1, \ldots, \mathsf{f}_n\})$ over the single location ℓ by $\mathsf{f}_1 \ldots \mathsf{f}_n$. The basic variant serves both soda and tea, and can be step-wise upgraded with features c and f, providing the functionality for cancel and free drinks, respectively. While a cancel upgrade cannot be reverted, it is possible to switch back from free to paid drinks.

2.2 Featured Transition Systems

As behavioral model for feature-oriented systems, [15] introduced the concept of *featured transition systems (FTSs)*. FTSs are transition systems whose transitions are amended with *feature guards*, i.e., Boolean expressions over the set of features. Formally, an FTS is a tuple $\mathsf{Fts} = (S, F, Act, T, \iota)$ where S is a set of states, F, and Act are finite sets features and actions, respectively, $T \subseteq S \times \mathbb{B}(F) \times Act \times S$ is a featured transition relation, and $\iota \in S$ is an initial

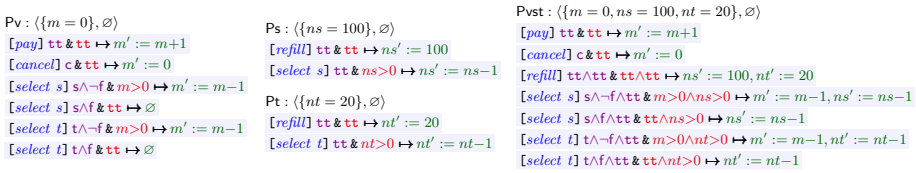

Fig. 2. Simple programs for the vending machine SPL

state. Given a feature combination $X \subseteq F$, Fts induces a transition system $\mathsf{Fts}(X) = (S, Act, T_X, \iota)$ where $(s, \alpha, s') \in T_X$ iff $(s, f, \alpha, s') \in T$ and $X \models f$. An FTS usually encodes all behaviors of valid feature combinations in a feature-oriented system, i.e., it is a *family model*. Family models facilitate the description of operational behaviors w.r.t. dynamic feature models and can be used for a family-based analysis [21]. The transition-system semantics of Fts and a reconfiguration graph $\mathcal{G} = \big(Loc \times \wp(F), Act', R, (\ell_0, I)\big)$ is defined as $\mathsf{Fts} \bowtie \mathcal{G} = \big(S \times Loc \times \wp(F), Act \cup Act', T_{\mathcal{G}}, (\iota, \ell_0, I)\big)$ where $T_{\mathcal{G}}$ is the smallest transition relation that satisfy the rules

$$\frac{(s, f, \alpha, s') \in T \quad (\ell, X) \in Loc \times \wp(F) \quad X \models f \quad \alpha \notin Act'}{(s, \ell, X) \xrightarrow{\alpha} (s', \ell, X)} \qquad \frac{s \in S \quad \big((\ell, X), \alpha, (\ell', X')\big) \in R \quad \alpha \notin Act}{(s, \ell, X) \xrightarrow{\alpha} (s, \ell', X')}$$

$$\frac{(s, f, \alpha, s') \in T \quad \big((\ell, X), \alpha, (\ell', X')\big) \in R \quad X \models f \quad \alpha \in Act \cap Act'}{(s, \ell, X) \xrightarrow{\alpha} (s', \ell', X')}$$

2.3 Featured Programs

Towards a compositional specification of FTSs, we rely on programs in a featured variant of Dijkstra's *guarded command language* [20], following a similar approach as within featured program graphs [12]. Let $V = \langle IntV, ExtV \rangle$ be a *variable interface* over *internal variables* $IntV$ and *external variables* $ExtV$, and F and Act finite nonempty sets of features and actions, respectively. We define $Cmd(V, F, Act)$ to be the set of *commands* $\langle f, g, \alpha, u \rangle$ where $f \in \mathbb{B}(F)$ is a feature guard, $g \in \mathbb{B}(\mathbb{C}(V))$ is a *guard*, $\alpha \in Act$ an action, and $u \colon IntV \rightharpoonup \mathbb{A}(V)$ is a partial function called *update*. For better readability, we usually write "$[\alpha]\ f \,\&\, g \mapsto u$" for "$\langle f, g, \alpha, u \rangle$" and denote updates u by sequences of the form $v_1 := a_1, \ldots, v_k := a_k$ where $v_i \in IntV$, $a_i \in \mathbb{A}(V)$, and $u(v_i) = a_i$ for all $i = 1, \ldots, k$.

Then, a *program* is a tuple $\mathsf{Prog} = (V, F, Act, C, \nu)$ where $C \subseteq Cmd(V, F, Act)$ is a finite set of commands and $\nu \in Eval(IntV)$ is an initial evaluation of internal variables. We assume w.l.o.g. that for every action in Act there is a command, i.e., for all $\alpha \in Act$ there are f, g, and u such that $[\alpha]\ f \,\&\, g \mapsto u \in C$, and that for each variable there is at least one command containing this variable. On the left of Figure 2 three programs Pv, Ps, and Pt are depicted, following the vending machine SPL example by implementing features v, s, and t, respectively. We indicated the initial variable evaluation in the variable interface and denoted updates (v, e) for some variable v and expression e by $v' := e$.

Following the concepts of delta-oriented formalisms [39], we define a *delta-program* as a pair (Prog, Δ) where Δ is a *modification* function

$$\Delta : Cmd\big(\langle ExtV, IntV\rangle, F, Act\big) \;\to\; \wp(Cmd\big(\langle V, \varnothing\rangle, F, Act\big)) \;.$$

We require that only finitely many commands involved in modifications, i.e., when D_Δ denotes the set of commands $c \in Cmd\big(\langle ExtV, IntV\rangle, F, Act\big)$ where $\Delta(c) \neq \{c\}$, we require D_Δ and $\Delta(c)$ to be finite for each $c \in D_\Delta$. The size of Δ is then defined as $|\Delta| = |D_\Delta| + \sum_{c\in D_\Delta} |\Delta(c)|$. We usually specify modification functions by only providing the finitely many modifications. Modification functions are naturally extended to sets of commands $E \subseteq Cmd\big(\langle ExtV, IntV\rangle, F, Act\big)$ by $\Delta(E) = \bigcup_{c\in E} \Delta(c)$. The *size* $|(\mathsf{Prog}, \Delta)|$ of a delta-program (Prog, Δ) is the number of commands in C and in modifications of Δ, i.e., $|(\mathsf{Prog}, \Delta)| = |C| + |\Delta|$.

Compositions. Let $\mathsf{Prog}_i = (V_i, F_i, Act_i, C_i, \nu_i)$ for $i \in \{1, 2\}$ be programs with $IntV_1 \cap IntV_2 = \varnothing$ and set $V = V_1 \oplus V_2$, $F = F_1 \cup F_2$, $Act = Act_1 \cup Act_2$, and $\nu = \nu_1 \rtimes \nu_2$. The *parallel composition* of Prog_1 and Prog_2 is defined as $\mathsf{Prog}_1 \| \mathsf{Prog}_2 = (V, F, Act, C, \nu)$ where C is the smallest set of commands satisfying the following rules:

$$(\text{int}_1)\ \frac{[\alpha]\ f_1 \,\&\, g_1 \mapsto u_1 \in C_1 \quad \alpha \notin Act_2}{[\alpha]\ f_1 \,\&\, g_1 \mapsto u_1 \in C} \qquad (\text{int}_2)\ \frac{[\alpha]\ f_2 \,\&\, g_2 \mapsto u_2 \in C_2 \quad \alpha \notin Act_1}{[\alpha]\ f_2 \,\&\, g_2 \mapsto u_2 \in C}$$

$$(\text{sync})\ \frac{[\alpha]\ f_1 \,\&\, g_1 \mapsto u_1 \in C_1 \quad [\alpha]\ f_2 \,\&\, g_2 \mapsto u_2 \in C_2}{[\alpha]\ f_1 \wedge f_2 \,\&\, g_1 \wedge g_2 \mapsto u_1 \rtimes u_2 \in C}$$

When $(\mathsf{Prog}_2, \Delta)$ is a delta-program with $\Delta(C_1) \subseteq Cmd\big(V, F, Act\big)$, the *superimposition of* Prog_1 *by* $(\mathsf{Prog}_2, \Delta)$ is defined as the program $\mathsf{Prog}_1 \bullet (\mathsf{Prog}_2, \Delta) = \big(V, F, Act, C_2 \cup \Delta(C_1), \nu\big)$. Intuitively, superimposition modifies the commands of Prog_1 according to Δ and adds the commands of Prog_2 to Prog_1. The result of a superimposition is a program, i.e., a delta-program with empty modifications. In this sense, superimposition also provides a composition on delta-programs by disregarding the modification sets of the first component and amending the empty modification set to the superimposition result.

Example 1. On the right of Figure 2, the parallel composition $\mathsf{Pvst} = \mathsf{Pv} \parallel \mathsf{Ps} \parallel \mathsf{Pt}$ is listed. Note that parallel composition is purely syntactic and select actions synchronize through rule (sync). The same result could be obtained through superimposition, i.e., $\mathsf{Pvst} = \mathsf{Pv} \bullet (\mathsf{Ps}', \Delta_\mathsf{s}) \bullet (\mathsf{Pt}', \Delta_\mathsf{t})$ where Ps' is as Ps but without the *select* s command (i.e., only the *refill* command) and Pt' is as Pt but with an empty set of commands. Further,

$$\begin{aligned}
\Delta_\mathsf{s} = \{&([\mathit{select}\ s]\ \mathtt{s}\wedge\neg\mathtt{f}\,\&\,m{>}0 \mapsto m' := m{-}1\,,\\
&\quad \{\,[\mathit{select}\ s]\ \mathtt{s}\wedge\neg\mathtt{f}\wedge\mathtt{tt}\,\&\,m{>}0\wedge ns{>}0 \mapsto m' := m{-}1, ns' := ns{-}1\,\}),\\
&([\mathit{select}\ s]\ \mathtt{s}\wedge\mathtt{f}\,\&\,\mathtt{tt} \mapsto \varnothing\,,\{\,[\mathit{select}\ s]\ \mathtt{s}\wedge\mathtt{f}\wedge\mathtt{tt}\,\&\,\mathtt{tt}\wedge ns{>}0 \mapsto ns' := ns{-}1\,\})\}\\
\Delta_\mathsf{t} = \{&([\mathit{select}\ t]\ \mathtt{t}\wedge\neg\mathtt{f}\,\&\,m{>}0 \mapsto m' := m{-}1\,,\\
&\quad \{\,[\mathit{select}\ t]\ \mathtt{t}\wedge\neg\mathtt{f}\wedge\mathtt{tt}\,\&\,m{>}0\wedge nt{>}0 \mapsto m' := m{-}1, nt' := nt{-}1\,\}),\\
&([\mathit{select}\ t]\ \mathtt{t}\wedge\mathtt{f}\,\&\,\mathtt{tt} \mapsto \varnothing\,,\{\,[\mathit{select}\ t]\ \mathtt{t}\wedge\mathtt{f}\wedge\mathtt{tt}\,\&\,\mathtt{tt}\wedge nt{>}0 \mapsto nt' := nt{-}1\,\})\\
&([\mathit{refill}]\ \mathtt{tt}\,\&\,\mathtt{tt} \mapsto ns' := 100\,,\{\,[\mathit{refill}]\ \mathtt{tt}\wedge\mathtt{tt}\,\&\,\mathtt{tt}\wedge\mathtt{tt} \mapsto ns' := 100, nt' := 20\,\})\}
\end{aligned}$$

Semantics. A program where no external variable appears in any command intuitively behaves as follows. Starting in the initial variable evaluation, the evaluations of the internal variables are changed according to the updates of one of the *enabled* commands, i.e., commands where its guard and feature guard are satisfied in the current variable evaluation and feature combination. Formally, the FTS semantics of a program $\mathsf{Prog} = \big(\langle \mathit{IntV}, \mathit{ExtV}\rangle, F, \mathit{Act}, C, \nu\big)$ with $C \subseteq \mathit{Cmd}\big(\langle \mathit{IntV}, \varnothing\rangle, F, \mathit{Act}\big)$ is defined as $\mathsf{Fts}[\mathsf{Prog}] = \big(\mathit{Eval}(\mathit{IntV}), F, \mathit{Act}, T, \nu\big)$ where $(\eta, f, \alpha, \eta') \in T$ iff there is some $[\alpha]\ f \,\&\, g \mapsto u \in C$ with $\eta \models g$ and $\eta' = \eta/u$. Here, $\eta/u \in \mathit{Eval}(\mathit{IntV})$ formalizes the *effect* of an update u onto an evaluation η, i.e., $(\eta/u)(v) = \eta\big(u(v)\big)$ for all $v \in \mathit{IntV}$ where $u(v)$ is defined and $(\eta/u)(v) = \eta(v)$ otherwise.

3 Compositional Feature-oriented Systems

While an FTS is a monolithic behavioral model for feature-oriented systems, compositional approaches describe the behavior for each feature encapsulated in *feature modules* and how to combine feature modules towards a behavioral model for a specific feature combination.

Definition 1. *A* compositional feature-oriented system (CFOS) *is a tuple*

$$\mathsf{S} \;=\; \big(\; F,\, \mathcal{F},\, \mathfrak{M},\, \phi,\, \prec,\, \circ \;\big)$$

where F is a finite feature domain, $\mathcal{F}$ is a feature model over F, $\mathfrak{M}$ is a finite set of feature modules assigned to features through a function $\phi\colon F \to \mathfrak{M}$, $\prec\, \subseteq F \times F$ is a total order on F, and $\circ$ is a composition operation on feature modules.

In the following let S denote a CFOS as defined above and assume that for every feature $x \in F$ there is a valid feature combination $X \in \mathfrak{V}[\mathcal{F}]$ with $x \in X$. The *size* of S, denoted $|\mathsf{S}|$ is the sum of the sizes of the feature modules assigned to its features. For any nonempty feature combination $X \subseteq F$, we define the *product* $\mathsf{S}(X)$ recursively via

$$\begin{aligned} \mathsf{S}\big(\{x\}\big) &= \phi(x) && \text{for } x \in F \\ \mathsf{S}(X) &= \mathsf{S}\big(X \backslash \{x\}\big) \circ \phi(x) && \text{for } x = \max\nolimits_{\prec}(X) \end{aligned}$$

where $\max_{\prec}(X)$ stands for the maximal feature in X with respect to $\prec$.

While in Definition 1 we defined CFOS in a generic fashion, in the following we focus on feature modules whose specification relies on programs and delta-programs (see Section 2.2).

Definition 2. *A CFOS $\mathsf{S} = (F, \mathcal{F}, \mathfrak{M}, \phi, \prec, \circ)$ is called*

- $\|$-CFOS *when $\circ = \|$ and $\mathfrak{M}$ comprises pairwise composable programs, and*
- $\bullet$-CFOS *when $\circ = \bullet$ and $\mathfrak{M}$ comprises pairwise composable delta-programs.*

Example 2. Following the vending machine SPL of Figure 1, we define a $\|$-CFOS $\mathsf{S_v} = (F_\mathsf{v}, \mathcal{F}_\mathsf{v}, \mathfrak{M}_\mathsf{v}, \phi_\mathsf{v}, \prec_\mathsf{v}, \|)$ where $F_\mathsf{v} = \{\mathsf{v}, \mathsf{b}, \mathsf{s}, \mathsf{t}, \mathsf{f}, \mathsf{c}\}$, $\mathcal{F}_\mathsf{v}$ is one of the feature models of Figure 1, $\mathfrak{M}_\mathsf{v} = \{\mathsf{Pv}, \mathsf{Ps}, \mathsf{Pt}, \epsilon\}$ as given in Figure 2 and ϵ stands for an empty feature module, and $\phi_\mathsf{v}(\mathsf{v}) = \mathsf{Pv}$, $\phi_\mathsf{v}(\mathsf{s}) = \mathsf{Ps}$, $\phi_\mathsf{v}(\mathsf{t}) = \mathsf{Pt}$, and $\phi_\mathsf{v}(\mathsf{i}) = \epsilon$ for $\mathsf{i} \in \{\mathsf{b}, \mathsf{f}, \mathsf{c}\}$, and $\mathsf{v} \prec_\mathsf{v} \mathsf{b} \prec_\mathsf{v} \mathsf{s} \prec_\mathsf{v} \mathsf{t} \prec_\mathsf{v} \mathsf{f} \prec_\mathsf{v} \mathsf{c}$, Then, $\mathsf{S}\big(\{\mathsf{v}, \mathsf{b}, \mathsf{s}, \mathsf{t}\}\big) = \mathsf{Pvst}$ as specified in Figure 2.

We close this section with some technical definitions concerning CFOSs that are used throughout the paper.

Definition 3. *For a fixed feature domain F and total order $\prec \subseteq F \times F$, we define the* characteristic function $\chi\colon \wp(F) \to \mathbb{B}(F)$ *for any $X \subseteq F$ by*

$$\chi(X) \;=\; (x_1 \wedge x_2 \wedge \ldots \wedge x_m) \;\wedge\; (\neg y_1 \wedge \neg y_2 \wedge \ldots \wedge \neg y_n)$$

where $x_1 \cup x_2 \cup \ldots \cup x_m = X$, $y_1 \cup y_2 \cup \ldots \cup y_n = F \setminus X$, $x_1 \prec x_2 \prec \ldots \prec x_m$, and $y_1 \prec y_2 \prec \ldots \prec y_n$.

The characteristic function of a feature combination $X \subseteq F$ provides a uniquely defined Boolean expression that characterizes X, i.e., for all $Y \subseteq F$ we have $Y \models \chi(X)$ iff $X = Y$.

Definition 4. *Let $\mathsf{S}_i = (F, \mathcal{F}_i, \mathfrak{M}_i, \phi_i, \prec_i, \circ_i)$ be some $\circ_i$-CFOS with $\circ_i \in \{\|, \bullet\}$ for $i \in \{1, 2\}$ and $\mathfrak{V}[\mathcal{F}_1] = \mathfrak{V}[\mathcal{F}_2]$. In case for all $X \in \mathfrak{V}[\mathcal{F}_1]$*

- $\mathsf{Fts}\big[\mathsf{S}_1(X)\big](X) = \mathsf{Fts}\big[\mathsf{S}_2(X)\big](X)$ *we call* S_1 *and* S_2 product equivalent
- $\mathsf{Fts}\big[\mathsf{S}_1(X)\big](X) \cong \mathsf{Fts}\big[\mathsf{S}_2(X)\big](X)$ *we call* S_1 *and* S_2 behavioral equivalent

Intuitively, product equivalence also requires the same variable names and evaluations on both products, while behavioral equivalence only focuses on isomorphic behaviors. Clearly, product equivalence implies behavioral equivalence. In this paper we focus on these rather strong notions of equivalence as they can be guaranteed as such in our transformations we present in the next sections. However, other notions of equivalences, e.g., following the concept of *bisimulation* [36], could be imagined in the context of *labeled FTSs*.

4 Family-ready Systems

Family models for feature-oriented systems are appealing as they contain all behaviors of the system in a single model. Thus, they can help to avoid the construction of every product one-by-one, facilitate the specification of dynamic feature-oriented systems, enable a family-based analysis, and profit from concise symbolic representations exploiting shared behaviors between products. Towards a family model for some CFOS S, a naive annotative approach would be to amend each product $\mathsf{S}(X)$ for valid feature combinations X with feature guards $\chi(X)$[1] and join these modified products to a single model. For CFOSs

[1] Recall Definition 3 of $\chi(X)$, the characteristic Boolean expression of X.

relying on programs as defined in Section 2.3, the family model would then be the program that comprises exactly those commands $[\alpha]\ \chi(X) \wedge f \,\&\, g \mapsto u$ for which there is a valid feature combination X and a command $[\alpha]\ f \,\&\, g \mapsto u$ in the product $\mathsf{S}(X)$. This approach, however, leads to a monolithic model that discards all modularity and further requires to construct every product one-by-one beforehand.

In this section, we introduce the notion of *family-readiness*, capturing those CFOSs where the composition of all feature modules yields a family model for the feature-oriented system. Not all CFOSs are family-ready as feature modules only have to include information about their feature interactions with other required features and not their role within the whole feature-oriented system. This might be desired during system design, e.g., to ensure reusability of feature modules within other but similar CFOSs or simply for separating concerns. However, in later design steps, the benefits imposed by family-readiness prevail, e.g., when it comes to product deployment and analyzing all products of the CFOS. Furthermore, family-ready CFOSs unite the advantages from compositional and annotative approaches as issued in [30]. Addressing research question **(RQ1)** of the introduction, we provide automated translations of $\|$-CFOSs and $\bullet$-CFOSs into product equivalent family-ready $\|$-CFOSs and $\bullet$-CFOSs, respectively.

For the rest of this section, let us fix a $\circ$-CFOS $\mathsf{S} = (F, \mathcal{F}, \mathfrak{M}, \phi, \prec, \circ)$ for $\circ \in \{\|, \bullet\}$. Note that for any feature combination $X \subseteq F$, and in particular $X = F$, $\mathsf{S}(X)$ is defined as feature modules are pairwise composable. Furthermore, $\mathsf{S}(X)$ does not contain any command referring to external variables, providing that its FTS semantics $\mathsf{Fts}\big[\mathsf{S}(X)\big]$ is defined.

Definition 5. S *is* family-ready *if for all* $X \in \mathfrak{V}[\mathcal{F}]$

$$\mathsf{Fts}\big[\mathsf{S}(F)\big](X) \quad \cong \quad \mathsf{Fts}\big[\mathsf{S}(X)\big](X) \ .$$

Stated in words, a CFOS is family-ready if the composition of any valid feature combination X admits the same behavior as composing all feature modules and then performing a projection to feature combination X.

Example 3. Returning to the $\|$-CFOS S_v for the vending machine SPL provided in Example 2, we show that not every CFOS is a priori family-ready. $\mathsf{S}_\mathsf{v}(F) = \mathsf{Pvst}$ where Pvst is the program illustrated in Figure 2 on the right. However, for the feature combination $X = \{\mathsf{v}, \mathsf{b}, \mathsf{t}\}$, the transition system $\mathsf{Fts}\big[\mathsf{S}_\mathsf{v}(F)\big](X)$ contains behaviors refilling soda, represented by the command with action *refill* in Pvst, not present in the transition system $\mathsf{Fts}\big[\mathsf{S}_\mathsf{v}(X)\big](X)$.

The rest of this section is mainly devoted to the proof of the following theorem:

Theorem 1. *From any* $\circ$*-CFOS with* $\circ \in \{\|, \bullet\}$ *we can construct in polynomial time a product equivalent and family-ready* $\circ$*-CFOS.*

As the approach towards a family-ready system crucially depends on the composition operator and the corresponding feature module formalism, we sketch the proof for Theorem 1 separately for $\|$-CFOS and $\bullet$-CFOS. For both we propose

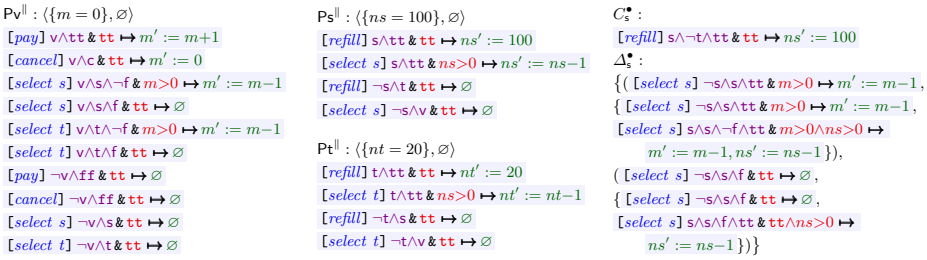

Fig. 3. Family-ready transformed feature modules for the vending machine SPL

local transformations that enrich feature modules with information about feature combinations in such a way that the modular structure of the system is maintained and the transformed model is family-ready.

4.1 Parallel Composition

Let S be a $\|$-CFOS and define the $\|$-CFOS $\mathsf{S}^{\|} = (F, \mathcal{F}, \mathfrak{M}^{\|}, \phi^{\|}, \prec, \|)$ through feature modules $\mathfrak{M}^{\|} = \{\mathsf{Prog}^{\|} : \mathsf{Prog} \in \mathfrak{M}\}$ where $\phi^{\|}(x) = (\phi(x))^{\|}$ for all $x \in F$. For any feature $x \in F$ we set $\phi(x) = (V_x, F_x, Act_x, C_x, \nu_x)$ and specify $(\phi(x))^{\|} = (V_x, F_x, Act_x, C_x^{\|}, \nu_x)$ by

$$C_x^{\|} = \{\ [\alpha]\ x \wedge f\ \&\ g \mapsto u : [\alpha]\ f\ \&\ g \mapsto u \in C_x\ \} \cup \tag{1}$$

$$\bigcup\nolimits_{\alpha \in Act_x} \{\ [\alpha]\ \neg x \wedge \bigvee\nolimits_{y \in F(\alpha)\setminus\{x\}} y\ \&\ \mathtt{tt} \mapsto \varnothing\ \} \tag{2}$$

Here, $F(\alpha)$ denotes the set of features whose modules contain an action $\alpha \in Act_x$, i.e., $y \in F(\alpha)$ iff $\alpha \in Act_y$. Intuitively, our transformation towards $C_x^{\|}$ can be justified as follows. Adding a feature guard x to every command in feature module $\phi(x)$ ensures that the command is only enabled in case the feature is active (1). Commands in (2) guarantee that if feature x is not active but another feature that has a synchronizing action, the feature module $\phi(x)$ does not block the actions of other feature modules.

Example 4. Figure 3 on the left shows feature modules of the family-ready CFOS $\mathsf{S}_{\mathsf{v}}^{\|}$ that arises from the CFOS S_{v} for the vending machine SPL defined in Example 2. As no other feature module contains a command with action *pay*, the command $[pay]\ \neg\mathtt{v} \wedge \mathtt{ff}\ \&\ \mathtt{tt} \mapsto$ is introduced for $\mathsf{Pv}^{\|}$ but never enabled[2]. The command $[refill]\ \neg\mathtt{s} \wedge \mathtt{t}\ \&\ \mathtt{tt} \mapsto \varnothing$ ensures that the *refill* action is not blocked in the family model when feature `s` is inactive and `t` is active.

The transformation can be performed in polynomial time in $|\mathsf{S}|$ as $F(\alpha)$ is computable in polynomial time and $|\mathsf{S}^{\|}| \leq |Act| \cdot |F| + |\mathsf{S}|$ where Act is the set of all actions in feature modules of S.

[2] Note that an empty disjunction always evaluates to `ff`.

4.2 Superimposition

Let S be a $\bullet$-CFOS and define the $\bullet$-CFOS $\mathsf{S}^\bullet = (F, \mathcal{F}, \mathfrak{M}^\bullet, \phi^\bullet, \prec, \bullet)$ through feature modules $\mathfrak{M}^\bullet = \{\mathsf{Prog}^\bullet : \mathsf{Prog} \in \mathfrak{M}\}$ where $\phi^\bullet(x) = (\phi(x))^\bullet$ for all $x \in F$. Similar as within parallel composition, for any feature $x \in F$ we set $\phi(x) = (V_x, F_x, Act_x, C_x, \nu_x, \Delta_x)$ and specify $(\phi(x))^\bullet = (V_x, F_x, Act_x, C_x^\bullet, \nu_x, \Delta_x^\bullet)$. The naive solution to specify $C_x^\bullet$ and $\Delta_x^\bullet$ is to define for every command c appearing in C_x or Δ_x a feature guard $\sigma(c) \in \mathbb{B}(F)$ that stands for those valid feature combinations X where c appears in $\mathsf{S}(X)$, i.e., for all $X \in \mathfrak{V}[\mathcal{F}]$ we have $X \models \sigma(c)$ iff c appears in $\mathsf{S}(X)$. In this way, we encode those feature combinations explicitly in the feature guards where superimposed commands are effective:

$$C_x^\bullet = \left\{ \; [\alpha]\ \sigma(c) \wedge f \,\&\, g \mapsto u \;:\; c = [\alpha]\ f \,\&\, g \mapsto u \in C_x \; \right\} \tag{3}$$

$$\Delta_x^\bullet = \Big\{ \Big([\alpha]\ \sigma(c) \wedge f \,\&\, g \mapsto u, \{ [\alpha]\ \sigma(c) \wedge f \,\&\, g \mapsto u \} \cup \tag{4}$$

$$\{ [\underline{\alpha}]\ \sigma(\underline{c}) \wedge \underline{f} \,\&\, \underline{g} \mapsto \underline{u} : \underline{c} = [\underline{\alpha}]\ \underline{f} \,\&\, \underline{g} \mapsto \underline{u} \in \Delta_x(c) \} \Big) : \tag{5}$$

$$c = [\alpha]\ f \,\&\, g \mapsto u \; \Big\}$$

The rule (3) generates commands also in products, rule (4) preserves the commands that would be modified by later composed feature modules, and modifications are included by (5), adapted with fresh feature guards that include $\sigma(\cdot)$.

Example 5. On the right of Figure 3, the transformed feature module $(\mathsf{Ps}', \Delta_\mathsf{s})$ of Example 1 is depicted. Here, $\sigma(\cdot)$ is always either s or $\neg\mathsf{s}$.

The construction of $\sigma(c)$ for any command c can be achieved by stepwise composing feature modules and adjusting the combinations $\sigma(c)$. For this, not all products have to be explicitly constructed, avoiding an exponential blowup: a computed table for all commands appearing in any set of commands and modifications in the modules suffices, step-by-step adjusting the feature guards $\sigma(\cdot)$ until all feature modules have been processed. Note that $|\mathsf{S}^\bullet| \leq 2 \cdot |\mathsf{S}|$ as at most one command is added by (4) to each command modified, at most doubling the overall number of commands in S.

4.3 Dynamics and Family Models

Given a family-ready CFOS S, the product $\mathsf{S}(F)$ is a family model of the feature-oriented system, enabling an operational semantics for dynamic feature-oriented systems: when $\mathcal{G}$ is a reconfiguration graph over F, used as feature model in S, then $\mathsf{Fts}\big[\mathsf{S}(F)\big] \bowtie \mathcal{G}$ defines the transition-system semantics of S. However, this interpretation requires the construction of the FTS semantics of S, flattening the modular structure. Another approach also applied by the tool ProFeat [9] is to interpret feature variables as standard internal variables of a reconfiguration module (describing the reconfiguration graph) and then turn every featured program into a standard program where feature guards are conjoint with the command guards. In this way, a (non-featured) compositional system arises that

can be used, e.g., for verification purposes using standard methods. Note that this approach is applicable to both, superimposed and parallel composition CFOSs that are family-ready.

5 Between Composition Worlds

In this section we address research question **(RQ2)** posed in the introduction, i.e., whether the classes of $\|$-CFOSs and $\bullet$-CFOSs are expressively equivalent and whether there are automated transformations to turn a $\|$-CFOS into a product equivalent $\bullet$-CFOS and vice versa.

5.1 From Parallel Composition to Superimposition

Theorem 2. *For any $\|$-CFOS S over features F there is a $\bullet$-CFOS S' with $|\mathsf{S}'| \in \mathcal{O}\big(2^{|F|^2}\cdot|\mathsf{S}|^{|F|+1}\big)$ that is product equivalent to S.*

Let us sketch the proof of Theorem 2 assuming we have given a $\|$-CFOS $\mathsf{S} = (F, \mathcal{F}, \mathfrak{M}, \phi, \prec, \|)$. We define a $\bullet$-CFOS $\mathsf{S}' = (F, \mathcal{F}, \mathfrak{M}', \phi', \prec, \bullet)$ such that S and S' are product equivalent. Intuitively, the synchronization between commands in different feature modules of S has to be explicitly encoded into the sets of modifications in feature modules of S'.

To this end, we consider delta-programs $\mathfrak{M}' = \{\mathsf{Prog}' : \mathsf{Prog} \in \mathfrak{M}\}$ where $\phi'(x) = (\phi(x))'$ for all $x \in F$. For any feature $x \in F$ we set $\phi(x) = (V_x, F_x, Act_x, C_x, \nu_x)$, $V_x = \langle IntV_x, ExtV_x\rangle$, and specify $(\phi(x))' = (V'_x, F_x, Act_x, C'_x, \nu_x, \Delta'_x)$. Let $\downarrow x = \{y \in F : y \prec x\}$ and recall that $F(\alpha) = \{y \in F : \alpha \in Act_y\}$. We define $V'_x = \langle IntV_x, ExtV\rangle$ with $ExtV = \bigcup_{y\in\downarrow x} V_y \setminus IntV_x$. For the definition of C'_x, let $B(x,\alpha) \in \mathbb{B}(F)$ be recursively defined by $B(x,\alpha) = \mathtt{ff}$ if $\downarrow x \cap F(\alpha) = \varnothing$ and $B(x,\alpha) = B(y,\alpha) \vee x$ for $y = \max_\prec(\downarrow x \cap F(\alpha))$.

$$C'_x \;=\; \big\{\; [\alpha]\ \neg B(x,\alpha) \wedge f \,\&\, g \mapsto u \;:\; [\alpha]\ f \,\&\, g \mapsto u \in C_x \;\big\} \tag{6}$$

The ratio behind the adapted feature guard in (6) is that modifications in delta-programs can only modify *existing* commands during a composition and hence, there have to be initial commands in "$\prec$-minimal" features. All commands of "$\prec$-greater" features that would synchronize in case of parallel composition then only modify already composed commands towards a synchronized command. Without the added feature guard $\neg B(x,\alpha)$ in (6), commands of single features could be executable even they would synchronize with other features.

Following the composition order $\prec$ on features, we recursively define Δ'_x for each $x \in F$. In case $x = \min_\prec(F)$, we set $\Delta'_x = \varnothing$. Assume we have defined Δ'_y for all $y \prec x$ and define $\Gamma(X)$ for any $X \subseteq \downarrow x$ as the set of commands arising from superimposing the feature modules $(\phi(y))'$ for all $y \in X$ according to $\prec$. Formally, we set $\Gamma(\varnothing) = \varnothing$ and $\Gamma(X) = C'_y \cup \Delta'_y\big(\Gamma(X\setminus\{y\})\big)$ for $y = \max_\prec(X)$.

$$\begin{aligned}\Delta'_x \;=\; \bigcup_{X\subseteq\downarrow x} \big\{\; \big(\, [\alpha]\ f \,\&\, g \mapsto u\,, \{\, [\alpha]\ f \wedge \underline{f} \,\&\, g \wedge \underline{g} \mapsto u \times \underline{u} \;:\; \\ [\alpha]\ \underline{f} \,\&\, \underline{g} \mapsto \underline{u} \in C_x\}\big) : [\alpha]\ f \,\&\, g \mapsto u \in \Gamma(X), \alpha \in Act_x \;\big\}\end{aligned} \tag{7}$$

Intuitively, (7) makes the parallel composition of commands in $\phi(x)$ with commands in previously composed feature modules explicit. S' is constructible in exponential time in $|\mathsf{S}|$ and we have $|\mathsf{S}'| \leq |\mathsf{S}|^{|F|+1}$. In fact, one cannot hope to construct a product equivalent $\bullet$-CFOS that avoids an exponential blowup:

Proposition 1. *There is a sequence of $\|$-CFOS S_n, $n \in \mathbb{N}$, with $|\mathsf{S}_n| = 2 \cdot |\mathsf{S}_{n-1}|$ for which there is no $k \in \mathbb{N}$ where S_k has a product equivalent $\bullet$-CFOS S'_k with $|\mathsf{S}'_k| < 2^{|\mathsf{S}_k|-1}$.*

We sketch the proof of the above proposition. For $F_n = \{x_1, \ldots, x_n\}$, consider the $\|$-CFOS $\mathsf{S}_n = (F_n, \mathcal{F}_n, \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}, \phi_n, \prec_n, \|)$ with $\mathfrak{V}[\mathcal{F}_n] = \wp(F_n)$, $\phi_n(x_i) = \mathcal{M}_i$ and $x_i \prec_n x_j$ for all $i < j \in \{1, \ldots, n\}$. For $i \in \{1, \ldots, n\}$ we define $\mathcal{M}_i = (V_i, \varnothing, \{\alpha\}, C_i, \nu_i)$ where $V_i = \langle \{v_i, w_i\}, V_{i-1} \rangle$ with $V_0 = \varnothing$,

$$C_i = \{\, [\alpha]\ \mathtt{tt} \,\&\, \mathtt{tt} \mapsto v_i := 1\,,\ [\alpha]\ \mathtt{tt} \,\&\, \mathtt{tt} \mapsto w_i := 1 \,\},$$

and $\nu_i = \{(v_i, 0), (w_i, 0)\}$. Then $|\mathsf{S}_n| = 2 \cdot n$. Assume there is a $k \in \mathbb{N}$ such that there is a $\bullet$-CFOS S'_k that is product equivalent to S_k with $|\mathsf{S}'_k| < 2^{|\mathsf{S}_k|-1}$. Then in particular $\mathsf{Fts}\big[\mathsf{S}_k(F_k)\big](F_k) = \mathsf{Fts}\big[\mathsf{S}'_k(F_k)\big](F_k)$ and hence, all 2^k combinations of updates of v_i and w_i to 1 have to be captured by modifications. As furthermore there are $2^k - 1$ nonempty feature combinations, we have at least $2^k \cdot (2^k - 1)$ required modifications in S'_k. Hence, $|\mathsf{S}'_k| \geq 2^{2k} - 2^k \geq 2^{|\mathsf{S}_k|-1}$, contradicting the assumption $|\mathsf{S}'_k| < 2^{|\mathsf{S}_k|-1}$.

The above proposition is closely related to the well-known fact that performing parallel compositions might yield programs that have exponential size in the number of components.

5.2 From Superimposition to Parallel Composition

While for every $\|$-CFOS there is a product equivalent $\bullet$-CFOS, such a result for the converse direction cannot be expected.

Proposition 2. *There is a $\bullet$-CFOS for which there is no product equivalent $\|$-CFOS.*

Proof. Consider the $\bullet$-CFOS $\mathsf{S} = (F, \mathcal{F}, \{\mathcal{M}_a, \mathcal{M}_b\}, \phi, \prec, \bullet)$ where $F = \{a, b\}$, $\mathfrak{V}[\mathcal{F}] = \wp(F)$, $a \prec b$, and for $\tau \in F$ we have $\mathcal{M}_\tau = (V_\tau, F, \{\alpha\}, C_\tau, \nu_\tau, \Delta_\tau)$, $C_\tau = \{\, [\alpha]\ \mathtt{tt} \,\&\, \mathtt{tt} \mapsto v_\tau := 1 \,\}$ with $\nu_\tau = \{(v_\tau, 0)\}$ and $\phi(\tau) = \mathcal{M}_\tau$, $V_a = \langle \{v_a\}, \varnothing \rangle$ and $V_b = \langle \{v_b\}, \{v_a\} \rangle$, and $\Delta_a = \varnothing$ and

$$\Delta_b \;=\; \big\{\, (c = [\alpha]\ \mathtt{tt} \,\&\, \mathtt{tt} \mapsto v_a := 1\,, \{\underline{c} = [\alpha]\ \mathtt{tt} \,\&\, \mathtt{tt} \mapsto v_a := 2, v_b := 2\}) \,\big\}$$

Assume there is a $\|$-CFOS S' over feature modules $\mathcal{M}'_\tau$ for $\tau \in F$ that is product equivalent to S. Then, for any $\tau \in F$ variable v_τ has to be internal in $\mathcal{M}'_\tau = \mathsf{S}'(\{\tau\})$ as $\mathsf{Fts}[\mathcal{M}_\tau](\{\tau\}) = \mathsf{Fts}[\mathcal{M}'_\tau](\{\tau\})$. Consequently, $\mathcal{M}'_a$ has to contain at least one α-command updating v_a to 1 not enabled when b is active and at least one α-command updating v_a to 2 when b is active. Similarly, $\mathcal{M}'_b$ has to contain at least one α-command updating v_b to 1 and at least one α-command

updating v_b to 2 when a is active. Following the definition of parallel composition yields that the α-commands synchronize towards $\mathsf{S}'(F)$, leading to at least one α-command that updates v_a to 2 *and* v_b to 1. This commands yields a behavior in $\mathsf{S}'(F)$ not apparent in $\mathsf{S}(F)$. Hence, $\mathsf{Fts}[\mathsf{S}(F)](F) \neq \mathsf{Fts}[\mathsf{S}'(F)](F)$ and thus there is no such a S' product equivalent to S. □

However, by explicitly encoding the behaviors of some product of the $\bullet$-CFOS in a single feature module with copies of variables and guarding them with the corresponding feature combination, we obtain a behavioral equivalent $\|$-CFOS.

Theorem 3. *For any $\bullet$-CFOS S over features F there is a $\|$-CFOS S' with $|\mathsf{S}'| \in \mathcal{O}\big(|\mathsf{S}| \cdot 2^{|F|}\big)$ that is behavioral equivalent to S.*

We sketch the construction on which the proof of Theorem 3 is based on. Let $\mathsf{S} = (F, \mathcal{F}, \mathfrak{M}, \phi, \prec, \bullet)$ be a $\bullet$-CFOS and define a $\|$-CFOS $\mathsf{S}' = (F, \mathcal{F}, \mathfrak{M}', \phi', \prec, \|)$ through feature modules $\mathfrak{M}' = \{\mathsf{Prog}' : \mathsf{Prog} \in \mathfrak{M}\}$ where $\phi'(x) = (\phi(x))'$ for all $x \in F$. For any valid feature combination $X \in \mathfrak{V}[\mathcal{F}]$, let $\mathsf{S}(X) = (V_X, F_X, Act_X, C_X, \nu_X)$ with $V_X = \langle IntV_X, ExtV_X \rangle$ and define for all $x \in F$ a feature module $(\phi(x))' = (\langle IntV'_x, \varnothing \rangle, F, Act'_x, C'_x, \nu'_x)$. Let furthermore $\mathfrak{V}_x \subseteq \mathfrak{V}[\mathcal{F}]$ denote the set of all valid feature combinations containing $x \in F$, i.e., $\mathfrak{V}_x = \big\{X \in \mathfrak{V}[\mathcal{F}] : x \in X\big\}$, $\mathfrak{V}_x^{\prec} \subseteq \mathfrak{V}_x$ denote the set of all valid feature combinations where $x \in F$ is maximal, i.e., $\mathfrak{V}_x^{\prec} = \big\{X \in \mathfrak{V}_x : x = \max_{\prec}(X)\big\}$. Then, $IntV'_x = \bigcup_{X \in \mathfrak{V}_x^{\prec}} \big\{v_x : v \in IntV_X\big\}$ comprises copies of internal variables in feature modules of S, $Act'_x = \bigcup_{X \in \mathfrak{V}_x} Act_X$, $\nu'_x = \bigcup_{X \in \mathfrak{V}_x^{\prec}} \big\{(v_x, z) : (v, z) \in \nu_X\big\}$, and

$$C'_x = \bigcup_{X \in \mathfrak{V}_x^{\prec}} \big\{\, [\alpha]\ \chi(X) \wedge f \,\&\, [g]_x \mapsto [u]_x \,:\, [\alpha]\ f \,\&\, g \mapsto u \in C_X \big\} \ \cup \tag{8}$$

$$\big\{\, [\alpha]\ \textstyle\bigvee_{Y \in \mathfrak{V}_x \setminus \mathfrak{V}_x^{\prec}, \alpha \in Act_Y} \chi(Y) \,\&\, \mathtt{tt} \mapsto \varnothing \,:\, \alpha \in Act'_x \big\} \tag{9}$$

Here, $[g]_x$ and $[u]_x$ denote the guards and updates, respectively, where each variable v is syntactically replaced by v_x. Via (8) the behavior of $\mathsf{S}(X)$ is mimicked in the commands of the maximal feature in X, guarded by X (i.e., they are effective iff X is active). The second part (9) guarantees that features not maximal in a feature combination X do not block the behaviors encoded in its maximal feature, again guarded by X through $\chi(X)$. For any feature combination $X \in \mathfrak{V}[\mathcal{F}]$ the product $\mathsf{S}(X)$ has at most $|\mathsf{S}|$ commands, added to at most one feature module in S' by (8). The number of valid feature combinations is bounded by $2^{|F|}$ and for each of them, which leads to at most $|\mathsf{S}| \cdot 2^{|F|}$ commands in S' as a result of (8). Also the number of actions in any feature module is bounded by $|\mathsf{S}|$, leading to at most $|\mathsf{S}|$ commands for each feature module by (9). Consequently, $|\mathsf{S}'| \leq |\mathsf{S}| \cdot (2^{|F|} + |F|)$.

Remark on Locality and Dynamics. The transformation presented yields a family-ready CFOS S', but might violate *locality* [31], a central principle of feature-oriented systems that imposes commands and variables to be placed in those feature modules they belong to. Hence, S' is not suitable to be interpreted with a dynamic feature model as S' and $\mathsf{S}^{\bullet}$ (see Section 4.2) might not

be behavioral equivalent. Another approach towards a $\|$-CFOS that preserves locality could be to introduce multiple copies of actions splitting the updates across the feature modules. This could ensure that the feature modules are combined through synchronization with the same action. However, this requires to alter the meaning of actions and product equivalence to S is only obtained after projecting the copies of the actions to the original actions.

6 Concluding Remarks

We have presented compositional feature-oriented systems (CFOSs), a unified formalism for feature-oriented systems that are specified through feature modules in guarded command language, composed via superimposition or parallel composition. With providing transformations towards family-ready CFOSs, we connected annotative and compositional approaches for feature-oriented systems [30]. Our transformations between CFOSs with different kinds of composition operators connect feature-oriented software engineering (where superimposition is paramount) and the area of formal analysis of feature-oriented systems (mainly relying on parallel composition). As the concepts presented are quite generic in its nature, they could be applicable also to other kinds of feature-oriented programming paradigms than featured guarded command languages.

Extensions. For the sake of a clean presentation, we did not introduce the framework of CFOSs in full generality. However, many extensions can be imagined where the concepts take over immediately. Additional feature module granularity can be achieved by adopting delta-oriented concepts [39,19]. Given a set of components described through programs, each feature module could be described not by a single module but by composing multiple components. This enables the reuse of components in multiple feature modules and enables solutions for the optional feature problem [32]. Numeric features and multifeatures [17], can be included in a similar way as presented in [21], i.e., evaluating feature variables to non-negative integers and either treat them as attributes and cardinalities, respectively. When interpreted as cardinalities, also the product definition for CFOSs is affected, requiring to compose multiple instances of the feature. Furthermore, feature modules in CFOSs could also be probabilistic programs and delta-programs where updates in guarded commands are replaced by stochastic distributions over updates. As our transformations are completely specified on the syntactic level of guarded commands, our results would also take over to this probabilistic case. While we introduced superimposition by explicitly stating exact commands to be modified, also pattern-matching rules could be imagined that modify parts of a command after matching. Also this extension does not change much in the transformations, however one has to take care including sufficient feature-guard information into the superimposition patterns.

Further Work. An interesting direction that also motivated our work towards research question **(RQ2)** is the verification of programs from feature-oriented programming paradigms such as delta-oriented approaches with FEATHERWEIGHT JAVA [28,7]. Using known translations from JAVA to guarded com-

mand language [35] and then applying our transformation from superimposition CFOSs to parallel CFOSs paves the way to use standard verification tools that rely on guarded command languages as input [11,27,34]. We plan to investigate more clever transformations than the one presented here that could avoid large parts of the explicit encoding of superimposition into single feature modules. Another direction for which we would also rely on results presented in this paper is to include superimposition concepts into our tool ProFeat [10] to enable quantitative analysis of probabilistic superimposition CFOSs.

References

1. M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan, and J.-P. Rigault. Modeling context and dynamic adaptations with feature models. In *4th International Workshop Models@run.time at Models 2009 (MRT'09)*, page 10, 2009.
2. S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8:49–84, 2009.
3. S. Apel, C. Kästner, and C. Lengauer. Feature featherweight java: A calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 101–112, New York, NY, USA, 2008. ACM.
4. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering*, pages 125–140, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
5. C. Baier and C. Dubslaff. From verification to synthesis under cost-utility constraints. *ACM SIGLOG News*, 5(4):26–46, 2018.
6. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
7. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, Mar 2013.
8. K. M. Chandy and J. Misra. *A foundation of parallel program design*. Addison-Wesley, Reading, MA, 1988.
9. P. Chrszon, C. Dubslaff, S. Klüppelholz, and C. Baier. Family-based modeling and analysis for probabilistic systems - featuring ProFeat. In *Proc. of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 9633 of *Lecture Notes in Computer Science*, pages 287–304. Springer, 2016.
10. P. Chrszon, C. Dubslaff, S. Klüppelholz, and C. Baier. Profeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing*, 30(1):45–75, 2018.
11. R. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and O. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. pages 359–364. Springer, 2002.
12. A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with snip. *International Journal on Software Tools for Technology Transfer*, 14(5):589–612, Oct 2012.
13. A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming*, 80:416 – 439, 2014.

14. A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, 2013.
15. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proceedings of ICSE'2010*, pages 335–344. ACM, 2010.
16. P. Clements and L. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.
17. M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 472–481, Piscataway, NJ, USA, 2013. IEEE Press.
18. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., 2000.
19. F. Damiani and I. Schaefer. Dynamic delta-oriented programming. In *Proc. of the 15th Software Product Line Conference (SPLC), Volume 2*, pages 34:1–34:8. ACM, 2011.
20. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
21. C. Dubslaff, C. Baier, and S. Klüppelholz. Probabilistic model checking for feature-oriented systems. *Trans. Aspect-Oriented Software Development*, 12:180–220, 2015.
22. C. Dubslaff, S. Klüppelholz, and C. Baier. Probabilistic model checking for energy analysis in software product lines. In *13th International Conference on Modularity (MODULARITY)*, pages 169–180. ACM, 2014.
23. N. Francez and I. R. Forman. Superimposition for interacting processes. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90 Theories of Concurrency: Unification and Extension*, pages 230–245, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
24. H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. In *PFE*, pages 435–444, 2003.
25. A. Gruler, M. Leucker, and K. D. Scheidemann. Modeling and model checking software product lines. In *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008, Proceedings*, pages 113–131, 2008.
26. S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, April 2008.
27. G. J. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.
28. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.
29. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.
30. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 311–320, 2008.
31. C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC'11, pages 5:1–5:8, New York, NY, USA, 2011. ACM.

32. C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. S. Batory, and G. Saake. On the impact of the optional feature problem: analysis and case studies. In *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, pages 181–190, 2009.
33. S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):337–356, 1993.
34. M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591, 2011.
35. K. R. M. Leino, J. B. Saxe, and R. Stata. Checking java programs via guarded commands. In *Workshop on Object-oriented Technology*, pages 110–111. Springer-Verlag, 1999.
36. R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
37. M. Plath and M. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53 – 84, 2001.
38. H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 347–350, Washington, DC, USA, 2008. IEEE Computer Society.
39. I. Schaefer, A. Worret, and A. Poetzsch-Heffter. A model-based framework for automated product derivation. In *Proceedings of the 1st International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2009), collocated with the 13th International Software Product Line Conference (SPLC 2009), San Francisco, USA, August 24, 2009.*, 2009.
40. P. Zave. Feature-oriented description, formal methods, and dfc. In S. Gilmore and M. Ryan, editors, *Language Constructs for Describing Features*, pages 11–26, London, 2001. Springer London.