Composing Run-Time Variability Models

Salman Farhat, Simon Bliudze, Laurence Duchien, Olga Kouchnarenko

Composing Run-time Variability Models

Salman Farhat[2][0000−0002−4121−3139], Simon Bliudze[1][0000−0002−7900−5271]⋆,
Laurence Duchien[2][0000−0002−4517−5862], and Olga Kouchnarenko[3][0000−0003−1482−9015]⋆⋆

[1] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[2] Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[3] Université de Franche-Comté, CNRS, Institut FEMTO-ST, F-25000 Besançon, France
FirstName.LastName@inria.fr[1,2],Olga.Kouchnarenko@femto-st.fr[3]

Abstract. The sheer complexity of modern systems requires compositional approaches to variability modelling. To manage the variability of large systems' architecture, feature models are widely used at design-time, with several operators defined to allow their composition. However, complex systems' architectures may evolve at run-time by acquiring new features and functionalities while respecting new constraints. To address this challenge, this paper defines composition operators for component-based run-time variability models that not only encode these feature model composition operators, but also ensure safe run-time reconfiguration. To prove the correctness and compositionality properties, we propose a novel *multi-step $\mathcal{UP}$-bisimulation* equivalence and use it to show that the component-based run-time variability models preserve the semantics of the composed feature models. In addition, reachability results permit safe reconfiguration.

Keywords: System Architecture Evolution · Component-based Systems · Variability Models · Composition Operators · Multi-step Bisimulation

1 Introduction

Software evolution [10] is the continual development of system software to extend its own functionality over time by integrating new functionalities not originally modeled. Systems are expected to evolve over time, new functionalities can be introduced to the system that expands the configuration space that was previously modeled. To enable modeling a system as it evolves, there must be a means to integrate sub-models encapsulating new functionality to the original model. To support such an evolution, component models are expected to be composable in such a way as to be able to merge two separate models into one model that encapsulates the modified configuration space. For instance, consider a component-based model that represents the services of the Heroku Cloud platform [17]. Cloud platforms are dynamic and evolve over time [19], driven not only by enhancements from its core development team but also through contributions from its user community. Clients of Heroku Cloud, for instance, can develop and

⋆ S. Bliudze was partially supported by the ANR grant ANR-23-CE25-0012 (SmartCloud).
⋆⋆ O. Kouchnarenko was partially supported by the ANR-23-CE25-0004 grant (ADAPT) and the ANR-17-EURE-0002 grant (EIPHI Graduate School) during her research leave at Inria Lille.

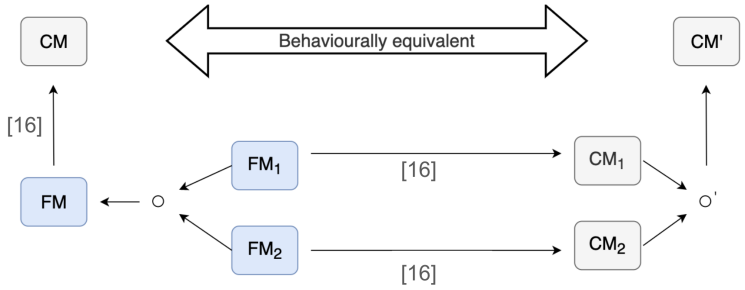


Fig. 1: Overview of Feature and JavaBIP models composition

introduce new services, such as advanced monitoring tools or supplementary management features, and subsequently offer these services on the Heroku marketplace. Then it becomes necessary to integrate these newly modeled services with the existing Heroku Cloud model, as the new services modify the configuration space of what was originally modeled. This integration ensures that the evolving Heroku Cloud is effectively modeled and adapted to accommodate the newly introduced services.

In software product line engineering domain, the composition of static variability models [11,22] like feature models (FMs) [7,18] is an active research area [20]. Various composition operators to compose FMs [1,2,3,4,9] have been proposed, and featured transition systems [12] are well-established models used for verification and validation purpose in different application domains. As highlighted in [12], it is challenging for these static models to support CPS or AI-intensive systems development. In the domain of component-based models, a recent survey [13] emphasizes the need for a suitable methodology to ensure the correctness of reconfigurations in component-based systems.

In this context, an automated model transformation approach has been developed for enforcing safe reconfiguration of software products by constructing, from feature models, executable component-based run-time variability models (CBRTVMs) [16]. These CBRTVMs, generated and executed in the JavaBIP framework [8], ensure that whenever a user requests the selection of a feature, all the required dependencies are selected at the same time (and similarly for feature deselection). If this is not possible, the operation is postponed without blocking other requests.

In this paper, we build upon that work, focusing on the compositionality of the approach (cf. Figure 1, where CM stand for component-based models, and FM for feature models). We consider three FM composition operators $\circ$ from [4] and define corresponding operators $\circ'$ over JavaBIP models. We introduce a novel notion of *multi-step $\mathcal{UP}$-bisimulation* to show that the composition of two CBRTVMs is behaviourally equivalent to the CBRTVM obtained from the composition of the corresponding FMs (Figure 1, at the top). Thus, we render the approach compositional while preserving the safety of dynamic reconfiguration.

Contributions and Outline This paper introduces composition operators for JavaBIP CBRTVMs, enabling an automated construction of component-based systems in a mod-

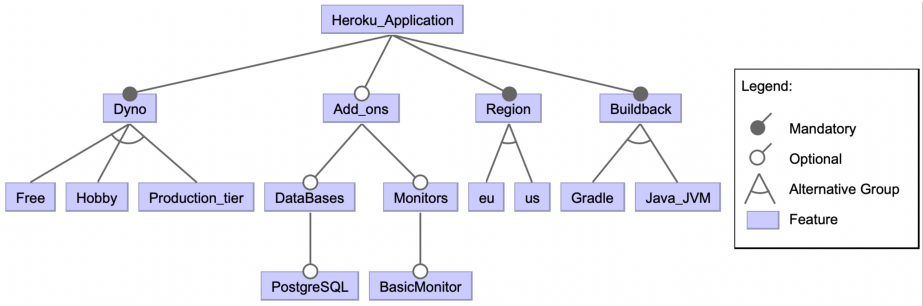


Fig. 2: Part of the Heroku Cloud feature model.

ular fashion while providing reusability, flexibility, and adaptability. In this context, the paper aims to address the following research question (RQ):

RQ1: How to encode Feature Model composition on CBRTVMs?
RQ2: How to define a behavioural equivalence to prove the correctness and compositionality of such an encoding?

The paper is organised as follows. Section 2 provides a motivating example. Background material is presented in Section 3. In Section 4, we address RQ1 by defining three composition operators on JavaBIP models and study their properties. In Section 5, we address RQ2 by defining the notion of a multi-step $\mathcal{UP}$-bisimulation and proving the correctness and compositionality of the encoding. Section 6, provides an experimental validation of our results. Section 7 discusses related work. Section 8 concludes the paper.

2 Motivating Example

Heroku Cloud is a platform-as-a-service provider, that offers a range of API-controlled services such as Dyno types, add-ons, and Regions [17]. Add-ons are supplementary functionalities encompassing services such as databases, monitoring, and messaging. We use a feature model representing a sub-set of Heroku services shown in Figure 2.

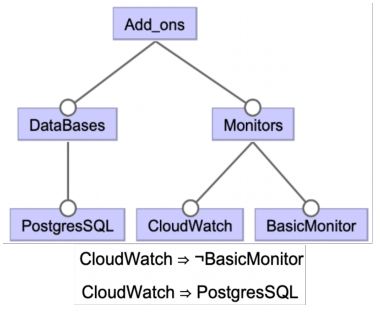


Fig. 3: CloudWatch FM

Suppose a new monitoring service, CloudWatch [24], is implemented and provided by Heroku. CloudWatch is an add-on service designed to monitor the performance of the computing units within the system. Its key functionality includes conducting comprehensive metric analyses, specifically focusing on CPU and memory usage metrics. This new service can be modelled in a separate feature model as presented in Figure 3.

To incorporate this service, the Heroku FM and the CloudWatch FM must be composed into one model that includes these new functionalities. Composition techniques, as outlined by Acher et al. [1], facilitate this process for static feature models.

This allows independent development of sub-feature models by different stakeholders and enables the reuse of existing models. In [16], the transformation process encodes static feature models in terms of component-based run-time variability models allowing safe reconfiguration at runtime. To allow a modular construction and composability of CBRTVMs, this paper defines composition operators for CBRTVMs that correctly encode the composition operators of feature models with the advantage of supporting configuration evolution at runtime.

3 Background

This section describes the preliminary notions related to feature models and their composition on the one hand, and to the JavaBIP component-based framework, on the other hand.

3.1 Feature Models

Introduced for product lines, feature models are used for representing the commonality and variability of features and of relationships among them [7]. A feature could be a software artifact such as a part of code, a component, or a requirement (cf. Figures 2 and 3). To express the variability of the system, feature models provide 1) a decomposition in sub-features, where a sub-feature may be mandatory (*black circle*), or optional (*unfilled circle*), 2) `XOR`-group or an `OR`-group. In a `XOR`-group, exactly one feature is selected, while in an `OR`-group, one or more features are selected, whenever the parent feature is selected. In addition, the combination of the optional and mandatory features is seen as an `AND`-group. In the main hierarchy, cross-tree constraints can be used to describe dependencies between arbitrary features, e.g. selecting a feature *requires* the selection of another one, or that two features mutually *exclude* each other.

We recall the formalisation of feature models that we have used in [16].

Definition 1 (Feature Diagram)**.** *A* feature diagram *is a tree-like structure conforming to the following grammar:*

$$Node ::= \texttt{OR}(Node_1, \ldots, Node_k) \mid \texttt{XOR}(Node_1, \ldots, Node_k) \mid \texttt{AND}([\textbf{mand}]Node_1, \ldots, [\textbf{mand}]Node_k) \mid leaf$$

We denote by $\pi \subseteq Node \times Node$ the parent *relation, i.e. a node n is a child of n' iff $\pi(n) = n'$. Let $\mu \subseteq Node \times Node$ be the reflexive and transitive closure of π^{-1}, i.e. $\mu(n)$ is the set of all descendants of $n \in Node$, including itself.*

Definition 2 (Feature model)**.** *A* feature model *over a set of features F is a tuple $FM = (root, \phi, \rho, \chi)$, where $root \in Node$ is a feature diagram, $\phi : \mu(root) \rightarrow F$ is a bijective function associating features to nodes, and $\rho, \chi \subseteq F \times F$ are the* requires *and* excludes *relations, respectively, with χ being symmetric.*[4]

For a feature $f \in F$ in the FM, we denote $n_f \stackrel{\text{def}}{=} \phi^{-1}(f)$, i.e. n_f is the node, such that $\phi(n_f) = f$. Abusing notation, we also write $\pi(f) = f'$ iff $\pi(n_f) = n_{f'}$. We use the predicate *mand* to denote the fact that a given node represents a mandatory feature.

[4] We write $f_1 \Rightarrow f_2$ iff $\rho(f_1, f_2)$ and $f_1 \Rightarrow \neg f_2$ (equivalently $f_2 \Rightarrow \neg f_1$) iff $\chi(f_1, f_2)$.

Definition 3 (Dependency Graph)**.** *Given an FM* $(root, \phi, \rho, \chi)$ *over* F*, its* dependency graph *is a directed graph* $G = (F, E)$*, where* F *is the set of features, and* $E \subseteq F \times F$ *is the set of directed edges representing the parent, mandatory and requires relations:*

$$E \stackrel{\text{def}}{=} \{(f_1, f_2) \,|\, \pi(f_1) = f_2\} \cup \{(f_1, f_2) \,|\, \pi(f_2) = f_1 \wedge mand(f_2)\} \cup \rho\,.$$

Definition 4 (Configuration Semantics)**.** *Let* $FM = (root, \phi, \rho, \chi)$ *be a feature model over a set of features* F *and let* (F, E) *be its dependency graph. A* configuration *is a set of features* $\Phi \subseteq F$*. We say that* Φ *is*

1. free from internal conflict *if, for any* $f_1, f_2 \in \Phi$*, holds* $(f_1, f_2) \notin \chi$*;*
2. saturated *if, for any* $f \in \Phi$*, holds* $E(f) \subseteq \Phi$*;*
3. valid *if it is saturated, free from internal conflict and respects structural constraints of* `XOR` *and* `OR` *nodes: exactly one (*`XOR`*) or at least one (*`OR`*) child feature selected, respectively (saturation implies that* `AND`*-node constraints are respected);*
4. partial-valid *if there exists a valid configuration* $\Phi' \supseteq \Phi$*.*

The FM semantics is the set of its valid configurations, denoted $[[FM]]$ [23].

Assumption 1. *We follow [16] in assuming that all considered feature models are such that any configuration free from internal conflict is partial-valid.*

3.2 Composition of Feature Models

In this paper, we adopt a denotational logic-based methodology for the composition of feature models, as outlined in [4]. This methodology encompasses the following steps:

1. The input feature models FM_1 and FM_2 are encoded as propositional formulae ϕ_{FM_1} and ϕ_{FM_2} respectively.
2. The composition operator is translated into a Boolean logic formula ϕ_c representing the composed feature model FM.
3. The feature diagram is then synthesized from ϕ_c.

We focus on three composition operators inspired by the ones in [4] and defined by the following Boolean formulae:

Intersection ($\cap$):	$\phi = \phi_{FM_1} \wedge \phi_{FM_2}$
Strict Intersection ($\dot{\cap}$):	$\phi = (\phi_{FM_1} \wedge not(F_2 \setminus F_1)) \wedge (\phi_{FM_2} \wedge not(F_1 \setminus F_2))$
Union ($\cup$):	$\phi = \phi_{FM_1} \vee \phi_{FM_2}$

where the set of features of the composed feature model is $F = F_1 \cup F_2$, and, for a given set of features $F' \subseteq F$, we define $not(F') \stackrel{\text{def}}{=} \bigwedge_{f \in F'} \neg f$.

Thus, a configuration $\Phi \subset F_1 \cup F_2$ is valid in $FM_1 \cap FM_2$ iff $\Phi \cap F_i$ is valid in FM_i for both $i = 1, 2$. It is valid in $FM_1 \,\dot{\cap}\, FM_2$, iff Φ is valid in FM_1 *and* FM_2. Finally, Φ is valid in $FM_1 \cup FM_2$ if the constraints for each feature in Φ are satisfied in either FM_1 or FM_2, i.e. holds $[[FM_1]] \cup [[FM_2]] \subseteq [[FM]]$ but not necessarily $[[FM_1]] \cup [[FM_2]] = [[FM]]$.

When synthesizing feature diagrams from Boolean formulas, it is important to note that a single Boolean formula corresponds to multiple possible feature model structures. Despite potential differences in dependency graphs, these varying structures encode the same set of valid configurations. In our work, we do not restrict the structure of the diagram for a given Boolean formula. Any algorithm can be used for synthesizing composed feature models, as long as the diagram is equivalent to the original formula.

3.3 JavaBIP Component-based Approach

A component is a software object that encapsulates certain behaviours of a software element. The concept of components is broad and may be used for component-based software systems, microservices, service-oriented applications, and so on. For the coordination of concurrent components, we make use of JavaBIP [8], which is an open-source Java implementation of the BIP (Behaviour-Interaction-Priority) framework [6].

In this context, the component behaviour is defined by a finite state machine (FSM) $(Q, P, \rightarrow)$, whose states (Q) are linked by transitions labelled by involved ports (P). JavaBIP allows two types of ports: *enforceable* and *spontaneous*. Enforceable ports represent actions controlled by the JavaBIP engine. They can be *synchronised*, i.e. executed together atomically. Spontaneous ports represent notifications that components receive about events that happen in their environment. They cannot be synchronised with other ports. An *interaction* is a set of ports—either one or several enforceable ports, or exactly one spontaneous port. To define allowed interactions, JavaBIP provides *requires* and *accepts* macros associated with enforceable ports and representing causal and acceptance constraints, respectively [8]. This allows JavaBIP to provide a coordination layer that is powerful enough to model—naturally and compositionally—the constraints expressed in the feature model. Intuitively, the requires macro specifies ports required for synchronization with the given port. For example, "$C1.p$ **Requires** $(C2.q; C3.r), C4.s$"[5] means that port `p` of component `C1` must be synchronized with at least one of the two ports q and r of components `C2` and `C3`, respectively, and with the s of component `C4`. The accepts macro lists all ports that are allowed to synchronize with the given port, thus allowing *optional* ports. For example, "$C1.p$ **Accepts** $C2.q, C3.r, C4.s, C5.t$" means that in addition to the ports listed by the requires macro, the port t of component `C5` is *also* allowed to synchronize with p despite not being required by it. Graphically, allowed interactions are defined by *connectors*. The behaviour specification of each component along with the set of requires and accepts macros are provided to the *JavaBIP engine*. The engine orchestrates the overall execution of the whole component-based system by deciding which component transitions must be executed at each cycle.

Let $JB = (\mathtt{C}, \rho, \alpha)$ be a JavaBIP model[6], where: `C` is the set of components, ρ is the set of the requires macros, and α is the set of the accepts macros. For a set of ports $a \subset \bigcup_{B \in C} P_B$, we write $a \models \rho, \alpha$ to denote that a satifies the conjunction of all the constraints in ρ and α seen as Boolean formulae (see [8] for detailed presentation). Note that, in particular, this implies the transitivity of the requires constraints.

[5] We use a notation that is slightly different from that in [8] without change of meaning.

[6] Throughout this paper, we use JB to denote arbitrary JavaBIP models, as opposed to CM, which is used to denote CBRTVMs, i.e. JavaBIP models generated from FMs.

The operational semantics of JB is defined by the labelled transition system (LTS) $L_{JB} = (Q, P, \rightarrow)$, where:

- $Q \stackrel{\text{def}}{=} \prod_{B \in C} Q_B$ is the Cartesian product of the sets of component states,
- $P \stackrel{\text{def}}{=} \bigcup_{B \in C} P_B$ is the set of all the enforceable and spontaneous ports in the system,
- $\rightarrow \subseteq Q \times 2^P \times Q$ is the set of transitions $q \xrightarrow{a} q'$, such that
  - either $a = \{p\}$ with $p \in P_B$ a spontaneous port of some component $B \in C$, (q_B, p, q'_B) a transition in B and $q_{B'} = q'_{B'}$, for all $B' \neq B$,
  - or all ports in a are enforceable, $a \models \rho, \alpha$, and, for any component $B \in C$, either $(q_B, a \cap P_B, q'_B)$ is a transition in B, or $a \cap P_B = \emptyset$ and $q_B = q'_B$.

A state q' is *reachable* from a state q if there exists a sequence of interactions $e_1, e_2, \ldots, e_n$ such that $(q, e_1, q_1), (q_1, e_2, q_2), \ldots, (q_{n-1}, e_n, q') \in \rightarrow$.

4 CBRTVMs and Their Composition

Throughout the paper, we use the term *Component-Based Run-Time Variability Model* (CBRTVM) to highlight the following facts: 1) the model in question is executable and can be *used at run time* to enforce the variability constraints, and 2) the set of valid configurations is never computed explicitly but is derived from components representing individual features. Thus, a JavaBIP implementation of a CBRTVM is a JavaBIP model with 1) one component per feature, encoding the feature life-cycle, and 2) a set of synchronisation macros encoding the dependencies among features. We build on such an implementation proposed in our previous paper [16].

4.1 Feature Model Encoding

For each feature f in a feature model FM, we define the corresponding component $enc(n_f)$ as shown in Figure 4. Each state represents either the presence or the absence of the corresponding feature in the configuration. The states `S_f` and `SR_f` represent the feature (de)activation having been requested but not yet realised. Transitions labeled by enforceable ports are shown as solid black arrows, whereas those labeled by spontaneous ports are shown as dashed green arrows. Below, ns_f, a_f, s_f, d_f denote, respectively, $not\_selected_f$, $activate_f$, $selected_f$ and $deactivate_f$. Let $L = (Q, P, \rightarrow)$ be the LTS of a JavaBIP model CM generated from FM. We define a mapping $\psi : Q \rightarrow 2^F$ that associates each state of the LTS to the configuration of the FM by putting $\psi(q) \stackrel{\text{def}}{=} \{f \in F \mid q_f \in \{f, SR\_f\}\}$.

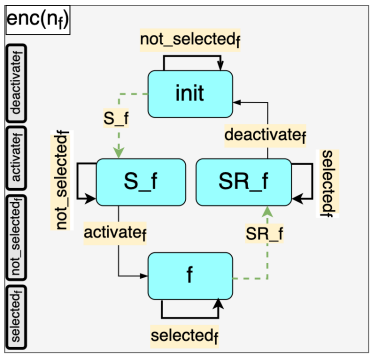


Fig. 4: FSM for a feature

Denote by SCC_f the strongly connected component of the dependency graph G_{FM} containing the feature f. For the feature f to be activated, 1) all features in SCC_f must be activated at the same time, 2) all *other* features that f depends on must either be activated at the same time or already selected, and 3) all features in conflict with f must not be selected. Thus, the synchronisation macros for the port a_f are defined by

$$\begin{aligned} &a_f \textbf{ Requires } (a_{f'})_{f' \in SCC_f \setminus \{f\}},\ (s_{f'}; a_{f'})_{f' \in E(f) \setminus SCC_f},\ (ns_{f'})_{f' \in \chi(f)}\,, \\ &a_f \textbf{ Accepts } (a_{f'})_{f' \in SCC_f \setminus \{f\}},\ (a_{f'})_{f' \in E^*(f) \setminus SCC_f},\ (s_{f'})_{f' \in E^*(f) \setminus SCC_f},\ (ns_{f'})_{f' \in \chi(f)}\,, \end{aligned} \tag{1}$$

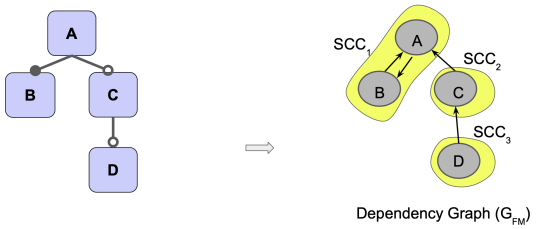


Fig. 5: Example of a feature model with its dependency graph (SCCs shown in yellow)

where E^* is the transitive closure of E. Notice that, due to cascading dependencies, there are more accepted ports than required ports.

Similarly, for the feature f to be deactivated, 1) all features in SCC_f must be deactivated at the same time and 2) all features that depend on f must either be deactivated at the same time or not yet selected. Thus, the synchronisation macros for the port d_f are defined by

$$\begin{aligned} &d_f \textbf{ Requires } (d_{f'})_{f' \in SCC_f \setminus \{f\}},\ (ns_{f'}; d_{f'})_{f' \in E^{-1}(f) \setminus SCC_f}\ , \\ &d_f \textbf{ Accepts } (d_{f'})_{f' \in SCC_f \setminus \{f\}},\ (d_{f'})_{f' \in \overline{(E^{-1})^*(f)} \setminus SCC_f}\ . \end{aligned} \tag{2}$$

Finally, the ports s_f and ns_f accept but do not require synchronisation with any other ports, i.e. their **Requires** macros are empty while their **Accepts** macros contain all enforceable ports in the system.

Example 1. Consider the dependency graph in Figure 5. For feature C, we have

$$a_C \textbf{ Requires } a_A; s_A \qquad \text{and} \qquad a_C \textbf{ Accepts } a_A, s_A, a_B, s_B\ ,$$

meaning that the port a_c can be synchronized with both a_A and a_B (allowing SCC_2 to be activated together with SCC_1), or with s_A and s_B (allowing SCC_2 to be activated after SCC_1). It is easy to see that, for instance, a_A and s_B cannot be enabled at the same time since activation of B would have required that of A.

Note. The encoding presented above is different from the one in [16]. Component behaviour in Figure 4 has additional looping transitions $\texttt{not\_selected}_f$ and $\texttt{selected}_f$ on states S_f and SR_f, respectively. Synchronisation macros (1) and (2) allow (de)activation of features at the same time as features in other SCCs, which we did not allow in [16]. However, these differences do not impact the theoretical results presented there. Specifically, any reachable state in the generated JavaBIP model corresponds to a saturated partial-valid configuration of the feature model. Conversely, if there exists a valid configuration in the feature model, it is guaranteed to be reachable in the JavaBIP model.

4.2 Composition Operators on Macros

Our approach is structural. Let CM_1 and CM_2 be two JavaBIP CBRTVMs. To compose them into CM' based on a composition operator $\circ'$, we take the union of their component sets, and compose the sets of their coordination macros. This section explains how these macros are composed for each of the composition operators presented in Section 3.2.

Composing Requires Macros Let us consider two sets of Requires macros, denoted ρ_1 and ρ_2. A new set ρ of Requires macros will be obtained in relation with operator $\circ' \in \{\cup, \cap, \dot{\cap}\}$.

Definition 5. *(Composition Operators) Let ρ_1 and ρ_2 be two sets of Requires macros. We define the following composition operators:*

- ***Intersection*** *($\cap$):*

$$\rho_1 \cap \rho_2 \stackrel{\text{def}}{=} \{x \textbf{ Requires } L_1 \ , \ L_2 \mid (x \textbf{ Requires } L_1) \in \rho_1 \textit{ and } (x \textbf{ Requires } L_2) \in \rho_2\} \cup \\ \{(x \textbf{ Requires } L) \in \rho_1 \mid x \in P_1 \setminus P_2\} \cup \{(x \textbf{ Requires } L) \in \rho_2 \mid x \in P_2 \setminus P_1\}$$

- ***Strict Intersection*** *($\dot{\cap}$): $\rho_1 \dot{\cap} \rho_2 \stackrel{\text{def}}{=} \overline{\rho_1} \cap \overline{\rho_2}$, where, for $i \in 1, 2$,*

$$\overline{\rho_i} \stackrel{\text{def}}{=} \rho_i \cup \{x \textbf{ Requires } \textit{false} \mid x \in P_{3-i} \setminus P_i\} .$$

  (A port that requires false will never be executed.)
- ***Union*** *($\cup$):*

$$\rho_1 \cup \rho_2 \stackrel{\text{def}}{=} \{x \textbf{ Requires } L_1 \ ; \ L_2 \mid (x \textbf{ Requires } L_1) \in \rho_1 \textit{ and } (x \textbf{ Requires } L_2) \in \rho_2\} \cup \\ \{x \textbf{ Requires } \textit{true} \mid x \in P_1 \setminus P_2\} \cup \{x \textbf{ Requires } \textit{true} \mid x \in P_2 \setminus P_1\}$$

  *(A port that has a "**Requires** true" constraint can be executed as a singleton.)*

Saturation Process for Accepts Macros The composition of Accepts macros is independent of the composition operator used. For two sets of macros α_1 and α_2, it is defined as the saturation of the set:

$$\{x \textbf{ Accepts } L_1 \ , \ L_2 \mid (x \textbf{ Accepts } L_1) \in \alpha_1 \textit{ and } (x \textbf{ Accepts } L_2) \in \alpha_2\} \\ \cup \{x \textbf{ Accepts } L_1 \mid x \in \mathtt{P}_1 \setminus \mathtt{P}_2\} \cup \{x \textbf{ Accepts } L_2 \mid x \in \mathtt{P}_2 \setminus \mathtt{P}_1\} .$$

Notice that, without saturation, ports required for interaction may be excluded from the Accepts macros. For instance, consider a scenario where port x requires port y (i.e. x **Requires** y), and port y requires port z (i.e. y **Requires** z). If the Accept macro for x only contains y (i.e. x **Accepts** y) after composition, then port z will be excluded (cf. Section 3.3). However, based on the Requires macros, x transitively requires z since y **Requires** z. To address this, saturation expands the right-hand side of each Accepts macro to include all ports required for interaction. In the example, it would add z to the Accepts macro for x, ensuring x accepts all necessary ports.

Let $\alpha = \{a_1, a_2, ..., a_n\}$ represent the set of Accepts macros, where each macro is denoted as $a_i : x_i$ **Accepts** L_i. We perform a saturation on α, which systematically iterates over each Accepts macro $a_i \in \alpha$, initializing the right-hand side rhs_i with L_i. It then expands rhs_i by conjoining additional ports from other Accepts macros that can interact with ports currently in rhs_i. This iteration continues until rhs_i stabilizes.

The resulting set α contains saturated Accepts macros, where the right-hand side of each macro encompasses all ports across composed interactions. This ensures the Accepts macros handle all relevant ports involved in potential interactions.

4.3 Composition Operators on JavaBIP Models

Definition 6. *(Composition) Let* $CM_1 = (C_1, \rho_1, \alpha_1)$ *and* $CM_2 = (C_2, \rho_2, \alpha_2)$ *be JavaBIP models as defined in Section 3.3. Their composition by* $\circ' \in \{\cup, \cap, \dot{\cap}\}$ *is the JavaBIP model* $CM' = (C', \rho', \alpha')$ *where:*

- $C' = C_1 \cup C_2$ *is the union of their components,*
- ρ' *is the composed require macros such that* $\rho' = \rho_1 \circ' \rho_2$ *as defined in Section 4.2,*
- α' *is the saturated accept macros as defined in Section 4.2.*

Note that, when these composition operators are applied to CBRTVMs, the resulting models can be optimised. In the process of synthesizing a feature diagram from a Boolean formula in feature modelling [4], dead features can be identified and removed as they cannot be part of any valid configuration of FM. This goes beyond the scope of this paper, it is worth noting that the composed CBRTVM can be similarly optimised when a (sub)set of dead features is known. When composing the JavaBIP models $CM' = CM_1 \circ' CM_2$, the component set is defined as the union $C' = C_1 \cup C_2$ (see Section 4.3). Consequently, C' may contain components corresponding to dead features that may be excluded when synthesizing the composed feature model FM. To that end, CM' can also be modified by removing all such components corresponding to dead features. In addition, macros should be refined by removing ports associated with components corresponding to dead features. Ports that are on the left-hand side of a macro, e.g. p_f **Requires** L_1, should be removed. For ports that appear on the right-hand side of a macro, e.g. $p_{f'}$ **Requires** L_1 with $a_f \in L_1$, the list L_1 can be replaced by *false*, since at least one of the ports required to fire p'_f (the one corresponding to the dead feature) will never be enabled. For *accept* macros, ports linked to dead features are simply removed.

5 A Bisimulation for Correctness and Compositionality Results

Bisimulation is a binary relation commonly used in Concurrency Theory (e.g. [21]) to establish the behavioural equivalence between two transition systems: whenever one system can execute an action, the same action can be executed by the other from any equivalent state, and vice versa. Bisimilarity ensures that, not only the states reachable within the two systems are equivalent but so are the execution options at every moment. In this section, we propose the notion of multi-step $\mathcal{UP}$-bisimulation, which extends the concept of bisimulation by allowing transitions to match over multiple steps. The multi-step $\mathcal{UP}$-bisimulation is then used to show that the composed CBRTVMs preserve the semantics of the composed feature models (correctness of the encoding), and the equivalence is the congruence for the defined composition operators (compositionality of the encoding).

In the context of FMs, various structures can be synthesized from the same Boolean formula, leading to differing saturated partial-valid configurations. However, the set of valid configurations is the same. On their side, two CBRTVMs generated from a FM have the same set of valid configurations reachable from the initial configuration, but they may have different paths and intermediate states to reach the valid configuration. To deal with such a situation, we consider paths in the LTSs, rather than single transitions.

Definition 7 ($\mathcal{P}$-path)**.** *Let $L = (Q, P, \rightarrow)$, with $\rightarrow \subseteq Q \times 2^P \times Q$, be an LTS. Let $\mathcal{P}$ be a predicate on Q. A* $\mathcal{P}$-path *in L is a sequence of transitions $q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} \ldots \xrightarrow{l_k} q_{k+1}$, such that both $\mathcal{P}(q_1)$ and $\mathcal{P}(q_{k+1})$ hold. We write $q_1 \overset{u}{\underset{\mathcal{P}}{\Longrightarrow}} q_{k+1}$, with $u = \bigcup_{i=1}^{k} l_i$.*

Note that $\mathcal{P}$ may hold on some intermediate states of a $\mathcal{P}$-path. We now introduce the notion of the multi-step $\mathcal{UP}$-bisimulation. This allows us to compare behaviors of two LTSs w.r.t. states that satisfy a given predicate along the $\mathcal{P}$-paths where the sets of observable actions coincide. This way the multi-step $\mathcal{UP}$-bisimulation tolerates non-atomicity of interactions.

Definition 8 (Multi-step $\mathcal{UP}$-Bisimulation)**.** *Let $L_i = (Q_i, P_i, \rightarrow_i)$, with $i = 1, 2$ and $\rightarrow_i \subseteq Q_i \times 2^P \times Q_i$ be two LTSs. Let $\mathcal{P}$ be a predicate on $Q_1 \cup Q_2$. Let $\mathcal{U} \subseteq P_1 \cup P_2$ be a set of unobservable ports, such that $P_1 \setminus \mathcal{U} = P_2 \setminus \mathcal{U}$. A relation $R \subseteq Q_1 \times Q_2$ is a* multi-step $\mathcal{UP}$-bisimulation *if, for all $(q_1, q_2) \in R$, hold the following two conditions:*

- *for any $q_1 \overset{u_1}{\underset{\mathcal{P}}{\Longrightarrow}} q_1'$, there exists $q_2 \overset{u_2}{\underset{\mathcal{P}}{\Longrightarrow}} q_2'$, such that $(q_1', q_2') \in R$ and $u_1 \setminus \mathcal{U} = u_2 \setminus \mathcal{U}$,*
- *and symmetrically for any $q_2 \overset{u_2}{\underset{\mathcal{P}}{\Longrightarrow}} q_2'$ in L_2.*

Notice that, in the classical setting, when transition labels are singleton, i.e. $\rightarrow \subseteq Q \times \{\{p\} \mid p \in P\} \times Q$ and $P_1 = P_2$, multi-step $\mathcal{UP}$-bisimulation reduces to the classical bisimulation by taking $\mathcal{P} = \mathit{true}$ and $\mathcal{U} = \emptyset$.

Definition 9 (Multi-step $\mathcal{UP}$-bisimilarity)**.** *Given two JavaBIP models JB_1 and JB_2, a predicate $\mathcal{P}$ on their states and a set of unobservable ports $\mathcal{U}$, we say that they are* multi-step $\mathcal{UP}$-bisimilar*, denoted $JB_1 \simeq_{\mathcal{UP}} JB_2$, if there exists a multi-step $\mathcal{UP}$-bisimilation relating the initial states of their semantic LTSs.*

Multi-step $\mathcal{UP}$-bisimilarity allows us to speak of the equivalence of JavaBIP models in general and of CBRTVMs in particular.

Let FM_1 and FM_2 be two feature models, $\circ \in \{\cup, \cap, \hat{\cap}\}$, CM and CM' be the CBRTVMs derived as in Figure 1 with F and F' their respective sets of features. We are interested in comparing the configurations reached by the two models. Thus, we want to observe what features are (de)activated following given (de)activation requests.

Notice that, while $F \subseteq F' = F_1 \cup F_2$ by construction, it is possible that $F \subsetneq F'$, since dead features may be eliminated in $FM_1 \circ FM_2$.

In this context, we define the set of unobservable ports (cf. Figure 4) to be

$$\mathcal{U} \overset{\text{def}}{=} \{\texttt{selected}_f, \texttt{not\_selected}_f \mid f \in F'\} \cup \{\texttt{S}_f, \texttt{SR}_f \mid f \in F' \setminus F\}\,.$$

Since the notion of a "saturated partial-valid configuration" is specific to any given feature model, to establish equivalence of two CBRTVMs, we have to limit our consideration to valid configurations only. Thus we take $\mathcal{P}$ to be the predicate, such that $\mathcal{P}(q)$ evaluates to *true* exactly when either $q_f = \texttt{init}$, for all $f \in F$ or $\psi(q)$ is a valid configuration of the composed feature model $FM_1 \circ FM_2$ (cf. Figure 1).

To prove the correctness of the composition operators' encodings, we have to show that, for any FM_1 and FM_2, holds $CM \simeq_{\mathcal{UP}} CM'$. Under Assumption 1, we can prove the following propositions:

Table 1: Feature inclusion in configurations Φ_1, Φ_2, and Φ_3

Configuration \ Features	Heroku_Application	Dyno	Free	Hobby	Production_tier	Add_ons	DataBases	PostgreSQL	Monitors	BasicMonitor	CloudWatch	Region	eu	us	BuildPack	Gradle	Java_JVM	Intersection	Strict Intersection	Union
Φ_1	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		*Invalid*	*Invalid*	*Valid*
Φ_2	✓	✓			✓	✓	✓	✓	✓		✓	✓	✓		✓	✓		*Valid*	*Invalid*	*Valid*
Φ_3	✓	✓			✓	✓	✓	✓	✓	✓		✓	✓		✓	✓		*Valid*	*Valid*	*Valid*

Proposition 1. *Let FM_1 and FM_2 be two feature models. Let $L = (Q, P, \rightarrow)$ and $L' = (Q', P', \rightarrow)$ be the semantic LTSs of the corresponding CBRTVMs CM and CM' as in Figure 1. Then $CM \simeq_{\mathcal{UP}} CM'$ with $\mathcal{P}$ and $\mathcal{U}$ defined as above.*

Sketch of the proof. The relation $R \subseteq Q \times Q'$, such that $(q, q') \in R$ iff $q_f = q'_f$, for all $f \in F$, and $q'_f \in \{\texttt{init}, \texttt{S\_f}\}$, for all $f \in F' \setminus F$, is a multi-step $\mathcal{UP}$-bisimulation relating the initial states $(\texttt{init})_{f \in F}$ and $(\texttt{init})_{f \in F'}$ of L and L', respectively. □

Multi-step $\mathcal{UP}$-bisimulation is not a congruence on JavaBIP models since it allows the breaking of interaction atomicity. However, it is a congruence on the sub-algebra of models generated from CBRTVMs by the composition operators defined in Section 4.

Proposition 2. *Let CM_1, CM_2, and CM_3 be composed from CBRTVMs. Let $\mathcal{P}$ be a predicate on the states of CM_1 and CM_2 and $\mathcal{U}$ a set of unobservable ports, such that $CM_1 \simeq_{\mathcal{UP}} CM_2$. For any $\circ' \in \{\cup, \cap, \dot{\cap}\}$, holds $CM_1 \circ' CM_3 \simeq_{\mathcal{U}'\mathcal{P}'} CM_2 \circ' CM_3$, with*

- *$\mathcal{P}'(q)$ = true iff $P(q_{12})$ = true and $\psi(q_3)$ is a valid configuration in CM_3, with q_{12} and q_3 the projections of q on the union of state spaces of CM_1 and CM_2, and on the state space of CM_3, respectively,*
- *$\mathcal{U}' \stackrel{\text{def}}{=} \mathcal{U} \cup \{\texttt{selected}_f, \texttt{not\_selected}_f \mid f \in F_3\}$, where F_3 is the set of features in CM_3.*

Sketch of the proof. The key observation is that any $\mathcal{P}$-path can be extended to a longer one in the semantic LTS of the same CBRTVM by firing unobservable (`not_`)`selected` ports after the firing of the corresponding (`de`)`activate` ports. □

These new results with the multi-step $\mathcal{UP}$-bisimulation used guarantee that for a given composition operator, a reachable state in the composed CBRTVM corresponds to a saturated partial-valid configuration in the composed feature model.

6 Experimental Validation

Integration of CloudWatch service For evaluation purposes, we use the Heroku Cloud FM (Figure 2) as FM_1. As FM_2, we take the CloudWatch FM (Figure 3) extended with all mandatory features from the Heroku Cloud FM alongside the first option

within these mandatory features, excluding *Dyno*, to which *Production_tier* is specifically appended. This setup allows us to better illustrate and experimentally validate the three composition operators studied in the paper.

Feature models (FMs) are taken as input and transformed into CBRTVMs. They are executable and can be *used at run time* to enforce the variability constraints. As explained in Section 4, the set of valid configurations is never computed explicitly but is derived from components representing individual features. Thus, CBRTVMs include JavaBIP component specifications along with synchronisation macros for coordination. We have developed a Java-based composer dealing with the macros that support three composition operators: union, intersection, and strict intersection. The component specifications sets of CBRTVMs to compose and a resulting macro file for a chosen composition operator are then packaged to create a composed CBRTVM.

Table 1 shows the configurations Φ_1, Φ_2, and Φ_3, and their validity in the composed FM across three distinct operators. We carried out our experiment by starting from an empty initial configuration and initiating spontaneous activation requests to the composed CBRTVM for all the features in configuration Φ_1 in random order. We repeated this process 50 times to observe the reached configurations for each case.

Figure 6 sums up the reachability outcomes across the operators used for the composition. As anticipated, the CBRTVM model aligned with theoretical results across all cases, transitioning to valid configurations while preventing invalid ones from being reached. Specifically, in the intersection case depicted in Figure 6, Φ_1 was never reached due to the exclusion constraint between *CloudWatch* and *BasicMonitor*. On the contrary, both Φ_2 or Φ_3 are reached depending on the activation sequence of *BasicMonitor* and *CloudWatch*, explaining the outcomes in Figure 6. In the strict intersection scenario, Φ_3 was the only valid configuration and was consistently reached, as Φ_1 and Φ_2 were never attained since they are not valid (cf. Table 1). Finally, in the union case, all configurations are reachable from the initial configuration, making Φ_1 attainable upon activating all features within it, as shown in Table 1. Φ_2 and Φ_3 in the union case are valid configurations however they are never reached (zero on the plot) since the activation of all features in Φ_1 is requested, and one has $\Phi_2 \subset \Phi_1$ and $\Phi_3 \subset \Phi_1$. However, if the activation of all features in only Φ_2 or Φ_3 were requested instead, then those configurations would be reached.

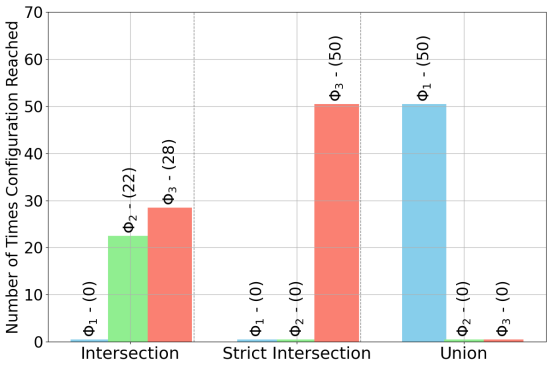
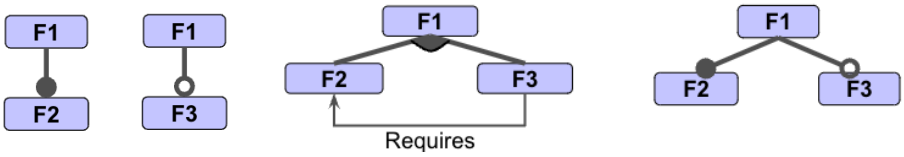


Fig. 6: Observed configurations.

Illustration of $\mathcal{P}$-path equivalence Let us illustrate the multi-step transition matching, where a sequence of transitions in one model corresponds either to a single transition or to a sequence of transitions in the other model. Consider the feature models *FM1* and *FM2* shown in Figures 7a and 7b. Following the process in Figure 1, we have



(a) FM1 (b) FM2 (c) (d)

Fig. 7: Two feature models (a,b) and two options for their intersection (c,d)

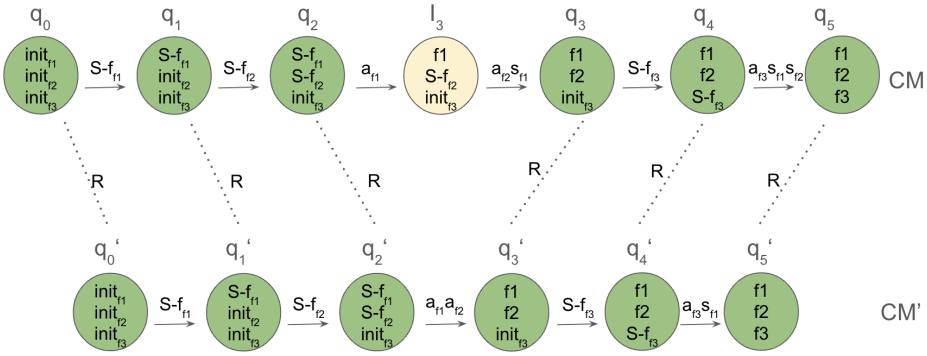


Fig. 8: Execution traces of CM and CM', with CM generated from the FM in Figure 7c.

automatically generated the CBRTVMs for these two feature models. We then used the macros composer for the intersection operator to generate the model CM'.

The behaviour of CM' must be compared with that of a CBRTVM CM generated for a feature model representing $FM1 \cap FM2$. However, as described in [4], several feature models can be used to represent $FM1 \cap FM2$, among which, in particular, any of the two FMs shown in Figures 7c and 7d. It is easy to see that these resulting feature models have the same set of valid configurations despite their different structures.

The CBRTVM generated for the FM in Figure 7d behaves exactly as CM'. Since our goal, here, is to illustrate the equivalence of different paths, we focus on the FM in Figure 7c. Let CM be the generated CBRTVM corresponding to this FM.

Consider the execution traces associated with a specific sequence of reconfiguration requests: activation of the feature $F1$, followed by $F2$, then $F3$. Figure 8 shows how both models respond to this sequence of requests starting from the initial states q_0 and q'_0 of CM and CM' that are related by R as defined in Section 5. The sets of ports used as labels for the two paths in Figure 8 coincide up to the unobservable ports s_{f1} and s_{f2}.

From their initial states q_0 and q'_0, upon reception of the activation request for $F1$ (through the spontaneous port $S\text{-}f_{f_1}$), both CM and CM' transition to states q_1 and q'_1, respectively. These target states are linked by the relation R. Then, when the activation request for $F2$ is received, both models transition to q_2 and q'_2, where $(q_2, q'_2) \in R$. Afterwards, CM moves to the state I_3 (in yellow in Figure 8), which corresponds to a saturated partial-valid configuration that is not a valid one. Hence, $\mathcal{P}(I_3) = \mathit{false}$, meaning that a $\mathcal{P}$-path cannot terminate in I_3. Hence, according to Definition 8, this

state does not need to be related to any state in CM'. However, the next transition in CM leads to the state q_3, which corresponds to a valid configuration. On its side, from q'_2 CM' performs the transition $a_{f1}a_{f2}$. Finally, following the reception of the request for the activation of $F3$, the two $\mathcal{P}$-paths in Figure 8 reach their terminal states $(q_5, q'_5) \in R$. Notice that the last transition in the top trace involves two unobservable ports, s_{f1} and s_{f2}, as opposed to the one in the bottom trace, which only involves s_{f1}. This reflects the fact that $F3$ has an explicit dependency on both features $F1$ and $F2$ in the FM of Figure 7c, but only on $F1$ in $FM2$.

7 Related Work

In the context of static variability models based on feature models [1,3,9], the techniques for composing FMs using predefined operators such as union and intersection have been studied. To respect the semantics of these operators, specific rules for merging common features during composition were defined. In addition, the work in [2,4] introduced more advanced techniques to enable composing FMs under arbitrary user-defined operators. The approach encodes input FMs as Boolean formulas and translates the composition operator into a Boolean formula over encoded models to obtain the composed model formula. The resulting feature model diagram can then be synthesized from the boolean formula. Another approach relates features through constraints in a separate view model aggregated with inputs. In line with this research (cf. Figure 1), we contribute with composition operators designed for composing run-time component models. While existing feature model composition focuses on static variability models, our framework provides automated support for dealing with configuration evolution in a modular and a compositional fashion.

A comparison of variability modeling and decision modeling approaches can be found in [14]. Both approaches focus on variability modeling but from different points of view, and both provide model derivation support. Our composition approach is FM-oriented as hierarchy, (de)composition, as well as configuration constraints, are essential in FM. Furthermore, our framework provides automated support for dealing with configuration workflows, which is essential in decision modelling approaches.

In Featured Transitions Systems (FTS) [12], each transition is annotated with a combination of features to determine the variants that can execute it. As they were initially thought in the static setting where all the features and their relationships could be specified in advance and not allowed to change, FTS do not support run-time adaptation, e.g., of CPS or AI-intensive systems with new features, constraints and functionalities. In [15], the composition of features is tackled by both superimposition and parallel composition, which are the most used in variability-intensive systems engineering. The authors introduce compositional feature-oriented systems (CFOSs) as a unified formal way for programs in a guarded command language. Unlike FTS-based verification and validation, our compositional approach allows mixing design-time and run-time techniques, and thus by some means they support operations over FTS such as FTS merge.

An early description of the distinction between positive and negative variability can be found in [25], where authors combine model-driven and aspect-oriented software development to support both variability types. In [25], features are separated in models and composed by aspect-oriented composition techniques on model level. Aspect-oriented techniques enable the explicit expression and modularization of variability on model,

code, and template level. Differently from [25], our approach is only model-driven when supporting composition and reconfiguration to allow both variability types.

In the domain of component-based models, a recent survey [13] emphasizes that a suitable methodology to ensure the correctness of reconfigurations in component-based systems is still needed. We believe that the present work contributes to this active research topic, at both development and management stages (cf. [13], Fig. 1).

Component-based models are compositional by their intrinsic nature. Nevertheless, adding composition operators for building complex component-based systems is of interest, both theoretical and practical, namely because of safety properties to ensure or to preserve by construction. In this domain, Attie et al. [5] propose a formal framework for compositional construction of software architectures by introducing an associative, commutative intersection composition operator for architectures. If architectures A_1 and A_2 enforce safety properties ϕ_1 and ϕ_2 respectively, [5] shows that their intersection composition $A_1 \cap A_2$ enforces the conjunction $\phi_1 \wedge \phi_2$. Our approach to CBMs composition also aims to facilitate incremental software system construction. However, in our approach the composition is directly performed on the syntactic representation of coordination constraints rather than by encoding interaction models into Boolean formulas. In addition, we introduce new composition operators beyond intersection.

8 Conclusion and Future work

This paper introduces composition operators for CBRTVMs, enabling automated construction of component-based systems. It provides reusability by allowing CBRTVM models to be reused across systems and instances through composition, flexibility by enabling models to be composed according to specific user needs through various operators, and adaptability by facilitating the incremental addition of functionalities.

In a broader software engineering perspective, we have automatically related the composition of feature models, which are well-established variability models for software evolution, with the composition of component-based models. We have also provided new composition operators for the CBRTVMs, that preserve the FM semantics. These contributions push FM variability and their composition, which are available at the design and development stages, into safe runtime reconfiguration that is automatically ensured at the management stage.

In future work, we plan to develop a more generalized composition approach that goes beyond the predefined operators currently examined. By allowing for user-defined composition, we intend to provide greater flexibility in constructing CBRTVMs.

To prove the correctness and compositionality of our approach, we have introduced a novel multi-step $\mathcal{UP}$-bisimulation relation. We have shown that it is a congruence on the sub-algebra of JavaBIP models generated by CBRTVMs but not on JavaBIP models in general. In future work, we are planning to study and characterise the maximal sub-algebra of models for which multi-step $\mathcal{UP}$-bisimulation is a congruence.

Acknowledgements The authors are very grateful to the SEFM 2024 PC chairs and to the anonymous reviewers for the excellent organisation of the selection process and for the very constructive reviews contributing to the final version of this paper.

Disclosure of Interests The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Acher, M., Collet, P., Lahire, P., France, R.: Composing feature models. In: International Conference on Software Language Engineering. pp. 62–81. Springer (2009)
2. Acher, M., Collet, P., Lahire, P., France, R.: Comparing approaches to implement feature model composition. In: Modelling Foundations and Applications: 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings 6. pp. 3–19. Springer (2010)
3. Acher, M., Collet, P., Lahire, P., France, R.: Managing variability in workflow with feature model composition operators. In: Software Composition: 9th International Conference, SC 2010, Malaga, Spain, July 1-2, 2010. Proceedings 9. pp. 17–33. Springer (2010)
4. Acher, M., Combemale, B., Collet, P., Barais, O., Lahire, P., France, R.B.: Composing your compositions of variability models. In: Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29–October 4, 2013. Proceedings 16. pp. 352–369. Springer (2013)
5. Attie, P., Baranov, E., Bliudze, S., Jaber, M., Sifakis, J.: A general framework for architecture composability. Formal Aspects of Computing **28**(2), 207–231 (2016)
6. Basu, A., Bensalem, B., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE software **28**(3), 41–48 (2011)
7. Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wąsowski, A.: A survey of variability modeling in industrial practice. In: Proc. of the 7th Int. Wshop on Variability Modelling of Software-intensive Systems. pp. 1–8 (2013)
8. Bliudze, S., Mavridou, A., Szymanek, R., Zolotukhina, A.: Exogenous coordination of concurrent software components with JavaBIP. Software: Practice and Experience **47**(11), 1801–1836 (2017)
9. Carbonnel, J., Huchard, M., Miralles, A., Nebut, C.: Feature model composition assisted by formal concept analysis. In: Int. Conf. on Evaluation of Novel Approaches to Software Engineering. vol. 2, pp. 27–37. SciTePress (2017)
10. Chapin, N., Hale, J.E., Khan, K.M., Ramil, J.F., Tan, W.G.: Types of software evolution and software maintenance. Journal of software maintenance and evolution: Research and Practice **13**(1), 3–30 (2001)
11. Chen, L., Babar, M.A.: A systematic review of evaluation of variability management approaches in software product lines. Inf. Softw. Technol. **53**(4), 344–362 (2011). https://doi.org/10.1016/J.INFSOF.2010.12.006
12. Cordy, M., Devroey, X., Legay, A., Perrouin, G., Classen, A., Heymans, P., Schobbens, P., Raskin, J.: A decade of featured transition systems. In: ter Beek, M.H., Fantechi, A., Semini, L. (eds.) From Software Engineering to Formal Methods and Tools, and Back. LNCS, vol. 11865, pp. 285–312. Springer (2019). https://doi.org/10.1007/978-3-030-30985-5_18
13. Coullon, H., Henrio, L., Loulergue, F., Robillard, S.: Component-based distributed software reconfiguration: A verification-oriented survey. ACM Comput. Surv. **56**(1), 2:1–2:37 (2024). https://doi.org/10.1145/3595376, https://doi.org/10.1145/3595376
14. Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wasowski, A.: Cool features and tough decisions: a comparison of variability modeling approaches. In: Eisenecker, U.W., Apel, S., Gnesi, S. (eds.) 6th Int. Wshop on Variability Modelling of Software-Intensive Systems, Germany, 2012, Proc. pp. 173–182. ACM (2012). https://doi.org/10.1145/2110147.2110167
15. Dubslaff, C.: Compositional feature-oriented systems. In: Ölveczky, P.C., Salaün, G. (eds.) Software Engineering and Formal Methods - 17th Int. Conf. SEFM 2019, Proc. LNCS, vol. 11724, pp. 162–180. Springer (2019). https://doi.org/10.1007/978-3-030-30446-1_9
16. Farhat, S., Bliudze, S., Duchien, L., Kouchnarenko, O.: Toward run-time coordination of reconfiguration requests in cloud computing systems. In: Jongmans, S., Lopes, A. (eds.)

Coordination Models and Languages - 25th IFIP WG 6.1 Int. Conf. COORDINATION 2023, Part of the 18th Int. Federated Conf. DisCoTec'23, Portugal, June, 2023, Proc. LNCS, vol. 13908, pp. 271–291. Springer (2023)
17. Heroku cloud application platform, https://www.heroku.com, accessed on 22/02/2024
18. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., Carnegie-Mellon Univ Pittsburgh, Pa, Software Engineering Inst (1990)
19. Kratzke, N.: A brief history of cloud application architectures. Applied Sciences **8**(8), 1368 (2018)
20. Nešić, D., Krüger, J., Stănciulescu, u., Berger, T.: Principles of feature modeling. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 62–73. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338906.3338974, https://doi.org/10.1145/3338906.3338974
21. Sangiorgi, D.: Introduction to bisimulation and coinduction. Cambridge University Press (2011)
22. Schmid, K., Rabiser, R., Grünbacher, P.: A comparison of decision modeling approaches in product lines. In: Proceedings of the 5th International Workshop on Variability Modeling of Software-Intensive Systems. pp. 119–126 (2011)
23. Schobbens, P., Heymans, P., Trigaux, J.: Feature diagrams: A survey and a formal semantics. In: 14th IEEE Int. Conf. RE'06. pp. 136–145. IEEE Computer Society (2006)
24. Stephen, A., Benedict, S., Kumar, R.A.: Monitoring IaaS using various cloud monitors. Cluster Computing **22**(Suppl. 5), 12459–12471 (2019)
25. Voelter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: 11th Int. Conf. SPLC 2007. pp. 233–242. IEEE Computer Society (2007). https://doi.org/10.1109/SPLINE.2007.23