Run-time Coordination of Reconfiguration Requests in Cloud Computing Systems

Salman Farhat, Simon Bliudze, Laurence Duchien, Olga Kouchnarenko

Run-time Coordination of Reconfiguration Requests in Cloud Computing Systems

Salman Farhat, Simon Bliudze, Laurence Duchien, Olga Kouchnarenko

RESEARCH REPORT
N° 9504
April 2023
Project-Teams Spirals

ISSN 0249-6399 ISRN INRIA/RR--9504--FR+ENG

Run-time Coordination of Reconfiguration Requests in Cloud Computing Systems

Salman Farhat*, Simon Bliudze†, Laurence Duchien‡,

Olga Kouchnarenko §

Project-Teams Spirals

Research Report n° 9504 — April 2023 — 20 pages

Abstract: Cloud applications and cyber-physical systems are becoming increasingly complex, requiring frequent reconfiguration to adapt to changing needs and requirements. Existing approaches compute new valid configurations either at design time, at runtime, or both. However, these approaches can lead to significant computational or validation overheads for each reconfiguration step. We propose a component-based approach that avoids computational and validation overheads using a representation of the set of valid configurations as a variability model. More precisely, our approach leverages feature models to automatically generate, in a component-based formalism called JavaBIP, run-time variability models that respect the feature model constraints. Produced run-time variability models enable control over application reconfiguration by executing reconfiguration requests in such a manner as to ensure the (partial) validity of all reachable configurations. We evaluate our approach on a simple web application deployed on the Heroku cloud platform. Experimental results show that the overheads induced by generated run-time models on systems involving up to 300 features are negligible, demonstrating the practical interest of our approach.

Key-words: Concurrent Component-based Systems, Variability Models, Self-Configuration, Dynamic Reconfiguration.

S. Bliudze was partially supported by ANR Investissments d'avenir (grant number ANR-16-IDEX-0004 ULNE).

O. Kouchnarenko was supported by the EIPHI Graduate School (grant number ANR-17-EURE-0002). This work was partially carried out while she was on a research leave at Inria Lille.

* Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France - salman.farhat@inria.fr
† Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France - simon.bliudze@inria.fr
‡ Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France - laurence.duchien@inria.fr
§ Université de Franche-Comté, CNRS, Institut FEMTO-ST, F-25000 Besançon, France - olga.kouchnarenko@femto-st.fr

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Coordination de l'exécution des demandes de reconfiguration dans les systèmes de cloud computing

Résumé : Les applications en nuage et les systèmes cyber-physiques deviennent de plus en plus complexes, nécessitant une reconfiguration fréquente pour s'adapter à l'évolution des besoins et des exigences. Les approches existantes calculent de nouvelles configurations valides lors de la conception du système, au moment de son exécution ou les deux. Cependant, ces approches peuvent entraîner des surcharges de calcul ou de validation importantes pour chaque étape de reconfiguration. Nous proposons une approche fondée sur un modèle à composants pour éviter les surcharges de calcul et de validation en utilisant une représentation de l'ensemble des configurations valides pour un système grâce aux modèles de variabilité. Plus précisément, notre approche exploite les modèles dits *feature models* pour générer automatiquement des modèles de variabilité dans un formalisme à composants, JavaBIP, qui respectent les mêmes contraintes. Ces modèles de variabilité permettent de contrôler la réconfiguration de l'application en temps d'exécution en exécutant les demandes de reconfiguration de manière à assurer la validité (partielle) de toutes les configurations atteignables. Afin d'évaluer l'intérêt de notre approche, nous l'avons expérimentée sur une application Web déployée sur la plateforme en nuage Heroku. Les surcharges induites par les modèles générés sur des systèmes comportant jusqu'à 300 features s'avèrent négligeables.

Mots-clés : Systèmes à base de composants concurrents, modèles de variabilité, auto-configuration, reconfiguration dynamique.

Contents

1 Introduction

Systems are increasingly required to be able to function continuously under tough circumstances, such as partial failures of subsystems, or changing user needs, while running without interruption and often unsupervised. Thus, after an initial configuration, reconfigurations are needed to keep the application compliant with the new needs and underlying platform constraints at run-time [22, 30]. Reconfigurations may modify the system architecture, and also the coordination between sub-parts of the system, notably w.r.t. failure events, components requests, or new requirements needed to be fulfilled.

Let us consider cloud applications, i.e., large concurrent software systems that are further constrained by the cloud platforms they run on [22]. In this context, a system's configuration is the set of resources that host an application, as well as the set of rules that define the system's sub-parts coordination and dependencies. The initial configuration aims to meet startup system needs and is not meant to last. Thus, in response to changing user requirements, systems must be reconfigured accordingly while ensuring that platform constraints are respected. Modeling and managing the variability and reconfigurations is an active software architecture research domain [30, 9, 16]. Feature modeling (FM) [19, 27] is a widely used approach to capture commonalities and variability across software systems that are part of a system family or a product line. Note that the problem of finding an assignment with only required constraints and XOR-groups is NP-hard [20]. Component-based systems (CBS), e.g., [13] for CBS supporting hierarchical architecture, allow building complex systems by composing components, which encapsulate data and code. In addition, some component models, e.g. Aeolus, Madeus, Concerto [12], JavaBIP [7], are executable, and allow run-time monitoring and control. We call them component-based run-time models.

Whatever is the approach to developing complex systems that are able to adapt to changing needs and demands–e.g., component-/agent-based systems, autonomous computing, emergent bio-inspired systems, etc.–analyzing and planning reconfigurations requires handling some metrics based on models [32, 23], and rules/policies [11, 22]. While using these approaches, computing either all valid configurations at design time, or an appropriate one at run-time, or both, induces computation and/or validation overheads for

each reconfiguration operation. This paper presents an approach to leverage variability models for acquiring a compact representation of a set of valid configurations of a system. It aims to automatically generate a formal executable model to safely perform reconfigurations in a scalable manner. To this end, we take advantage of feature models and component-based run-time models for enforcing safe-by-construction behaviour of concurrent component-based systems through the automatic derivation of executable models from requirements and safety constraints. Our framework, joining forces of features and components for safe reconfigurations (FeCo4Reco for short) with a lightweight effort, allows applying reconfigurations in a safe manner, with no overhead of either computing or validating the new configuration while having an executable model.

The FeCo4Reco process, shown in Fig 1, consists of three stages: 1) domain constraints are specified as a feature model, 2) the feature model is automatically transformed into a run-time Component-based Variability Model (CBVM) to make it run alongside the system, 3) the generated model is used by the deployers to set up initial configurations of the system, and to automatically monitor reconfiguration requests from the environment and safely execute them at run-time.

Outline and contributions. Section 2 presents the Heroku cloud running example used throughout the paper, and it also lists the research questions. Section 3 provides an overview of the underpinnings: feature models and component-based models. Motivated by driving the reconfiguration process without the need of pre-computing the possible configurations at design time, our *first* contribution is a component-based run-time variability model leveraging feature models and their underlying constraints.

Model transformation rules in Section 4, which are general enough for both feature models and component-based models, constitute the *second* contribution leading to a component-based variability model automatically generated with FeCo4Reco. Its main advantage consists of a compact encoding of all valid or partial-valid configurations, with partial-valid meaning that it can be transformed into a valid configuration by adding features. Being run-time, this model encodes reconfiguration operations while ensuring the safety property, saying that only partial-valid configurations can be reached as a result of any reconfigurations. Main properties related to reconfigurations are described in Section 5. Section 6 describes the implementation, and reports on experimental results on a non-trivial cloud example, with a discussion on the validity of the approach. They constitute the *third* practical contribution showing the interest of our approach in practice. Finally, Section 7 is dedicated to related work, and Section 8 concludes with future work directions.

This report is an extended version of our paper published at COORDINATION 2023 [15], including the proofs of all the theoretical results and a detailed presentation of the experimental setup.

2 Motivating Example

This section describes the Heroku cloud [25] to motivate our approach and to illustrate its application. Heroku offers a range of API-controlled services, including dyno types, add-ons, buildpack, and regions, which provide developers with the means to create complex applications consisting of interacting pieces. For example, a typical web application may have a web component that is responsible for handling web

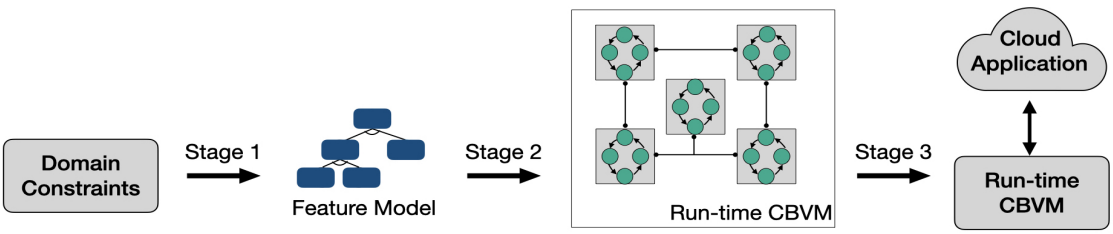

Figure 1: Stages of the FeCo4Reco process.

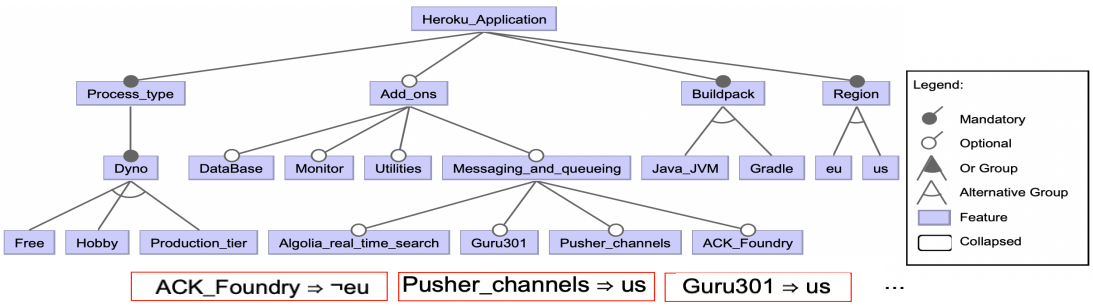

Figure 2: Part of the Heroku cloud feature model.

traffic. It may also have a queue (typically represented by an add-on on Heroku), and one or more workers that are responsible for taking some elements off of the queue and for processing them. Heroku permits building such architectures by allowing the user to configure the application using the process type, region, buildpack, and add-ons.

- Process type: All Heroku applications are launched and scaled using the container model on the Heroku platform. The Heroku containers, called Dynos, are virtualized Linux containers to run programs in isolation.
- Region: Applications on Heroku may be deployed in various geographic regions.
- Buildpack: Buildpacks convert the deployed code to an executable slug on a Dyno.
- Add-ons: Add-ons are additional services that can be attached to a Heroku application to provide extra functionalities such as data storage, monitoring, analytics, and data processing.

Figure 2 presents the Heroku cloud with services and constraints between these services. In addition to the mandatory components, optional functionalities, such as Heroku add-ons, are available. To help the developer, they are maintained by either a third-party provider or by Heroku. Add-ons are installed onto applications by using the Heroku service API interface. Furthermore, other constraints must be taken into account. For example, in Heroku, they express regional availability of services, inter-service dependencies, as well as architectural constraints. As a result, developers are expected to be Heroku experts in order to manage and control applications in a safe way while taking into account all the constraints on the context, in which the application is hosted.

Research Question – The main challenge is to allow the reconfiguration of software systems in a safe manner while avoiding additional overhead at run-time, the paper aims to address the following research question (RQ):

RQ How to enforce domain constraints during dynamic reconfiguration at low cost?

3 Background

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 [6]. A feature `f` could be a software artefact such as a part of code, a component, or a requirement. In Fig. 2 for the he Heroku cloud, features, graphically represented by rectangles, are organized in a tree-like hierarchy with multiple levels of increasing detail. 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 described dependencies between arbitrary features, e.g. selecting a feature *requires* the selection of another one, or that two features mutually *exclude* each other.

More precisely, let F be a set of features, and $Node$ the set of the nodes of a tree-like structure defined by the grammar of axiom $Node$:

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

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 3.1. *A feature model FM over a set of features F is a tuple $(root, \phi, \rho, \chi)$, where $root \in Node$, $\phi : \mu(root) \to F$ is an injective function associating features to nodes, and $\rho, \chi \subseteq F \times F$ are the* requires *and* excludes *relations, respectively, with χ being symmetric.*[1]

Given a feature $f \in F$ that appears in the FM, we denote by n_f the node corresponding to f, i.e. such that $\mu(n_f) = f$. Abusing notation, we also write $\pi(f) = f'$ iff $\pi(n_f) = n_{f'}$. Given an *AND*-node n, for each child mandatory node n' of n, i.e. such that $n = AND(\dots, \mathbf{mand}\ n', \dots)$, we write $mand(n')$.

Example 3.1. *Figure 2 shows a simplified example of the Heroku cloud feature model.* `Process_type`*,* `Region`*, and* `Buildback` *are mandatory features, whereas* `Add_ons` *are optional. The* `Process_type` *feature can be realized by using only one of the three alternative* `Dyno` *sub-features* `Free`*,* `Hobby`*, and* `Production_tier`*. On the contrary,* `Messaging_and_queuing` *can be implemented using any combination of the sub-features* `Algolia_real_time_search`*, ...,* `ACK_Foundry`*. In addition,* `ACK_Foundry` *and* `eu` *are mutually exclusive, and both* `Guru301` *and* `Pusher_channels` *require* `us`*.*

Definition 3.2. *Given a feature model $(root, \phi, \rho, \chi)$ 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 edges representing the parent, mandatory and requires relations:*

$$E = \{(f_1, f_2) \mid \pi(f_1) = f_2\} \cup \{(f_1, f_2) \mid \pi(f_2) = f_1 \wedge mand(f_2)\} \cup \rho\,.$$

The FM semantics is the set of its valid configurations [28].

The following definition allows for incremental design and development of real-world systems by considering consistent and well-formed configurations, even if they are not complete.

Definition 3.3. *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 $E(f) \subseteq \Phi$, i.e., the dependencies for each feature in the configuration are also included in the configuration.*

[1] In Fig. 2, 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 $\chi(f_1, f_2)$.

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 the respect of AND-node constraints);*

4. partial-valid*, if there exists a valid configuration* $\Phi' \supseteq \Phi$ *and it is free from internal conflict. A partial-valid configuration may not be saturated, meaning that some of the dependencies of its features are not included in the configuration.*

Saturated partial valid configurations are more restrictive than partially valid ones, as they require all the dependencies of the selected features to be included as well. This means that when building complex systems incrementally, we can ensure that each intermediate step includes the desired features with their necessary dependencies, resulting in consistent and well-formed configurations.

Assumption 1. *We assume that all considered feature models are such that any configuration free from internal conflict is partial-valid.*

3.2 JavaBIP Component-based Approach

A component is a software object, that encapsulates certain behaviours of a software element. The concept of component 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 [7], which is an open-source Java implementation of the BIP (Behaviour-Interaction-Priority) framework [4]. Given a set of *components* and a set of their *ports*, the component behaviour is defined by a finite state machine (FSM) with transitions labelled by ports. 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. In order to define allowed interactions, JavaBIP provides *requires* and *accepts* macros associated with enforceable ports and representing causal and acceptance constraints, respectively. This allows JavaBIP to provide a coordination layer that is powerful enough to model naturally and compositionally the constraints expressed in the feature model. Detailed presentation of these macros is provided in [7]. Intuitively, the *requires* macro specifies ports required for synchronization with the given port. For example, **requires(C1.p) = {C2.q, C3.r, C4.s}**[2] means that port `p` of component `C1` must be synchronized with at least one of the three ports: q, r, or s of components `C2`, `C3` and `C4`, respectively. The *accepts* macro lists all ports that are allowed to synchronize with the given port, thus allowing *optional* ports. For example, **accepts(C1.p) = {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. The operational semantics of a JavaBIP model is defined by a labelled transition system (LTS) $L = (Q, \Sigma, \rightarrow)$, where:

- Q is the cartesian product of the sets of component states,
- Σ is the set of allowed interactions (including singleton spontaneous ports),
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the maximal set of transitions such that the projection of each $(q, e, q') \in \rightarrow$ onto any component B is either $(q_B, \emptyset, q_B)$, for some state q_B of B, or a transition $(q_B, \{p\}, q'_B)$ with q_B, q'_B and p being two states and a port of B.

[2] We use a notation that is slightly different from that in [7] without change of meaning.

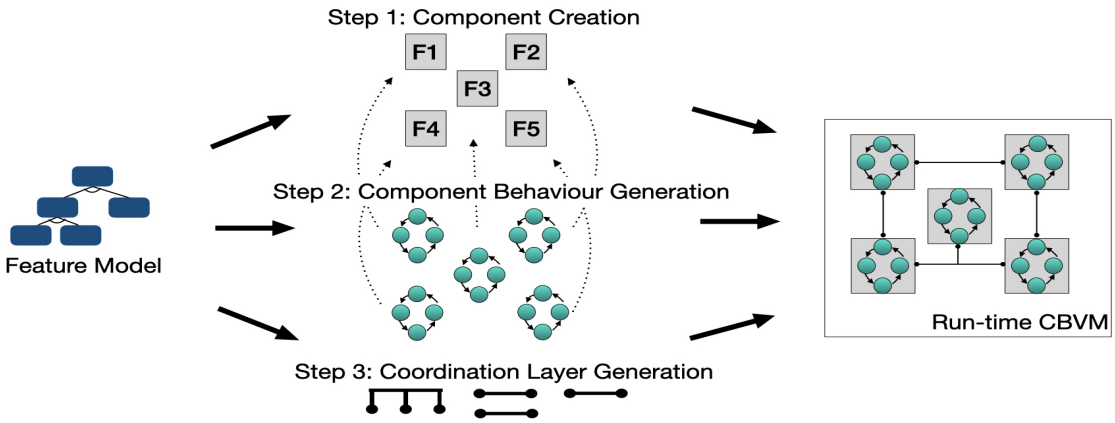

Figure 3: The generation of the component-based variability model involves a three-step transformation process, where Steps 1, 2, and 3 correspond to Subsections 4.1, 4.2, and 4.3, respectively.

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

Example 3.2. *Building on Example 3.1, Figure 5 on page 11 illustrates a JavaBIP model with five components. Graphically, enforceable and spontaneous transitions are shown by solid black and dashed green lines, respectively. Ports are shown as grey boxes on the sides of the components, and five connectors linking the ports define the possible interactions. Port* $activate_f$ *of the* `Algolia_read_time_search` *component can only be fired together with port* $selected_f$ *of the* `Messaging_and_queuing` *component. As there is no transition from* `init` *state of label* $selected_f$ *of the component* `Messaging_and_queuing`*, this prevents the* `Algolia_read_time_search` *component from entering the* `final` *state when the* `Algolia_read_time_search` *component is in* `S_f` *state. In this example, connector C1 is binary, while C5 defines an interaction involving five ports.*

4 Design and Transformation

This section describes a set of design rules for automatically generating a run-time component-based variability model using a feature model as input. Figure 3 presents the steps for the transformation of the encoding process. The process of encoding the feature model into a component-based variability model is done recursively. The process starts with the *root* node of the feature model and generates the components and their behavior. Subsequently, the generation of the coordination layer is performed based on the feature model constraints.

4.1 From Features to Components

Let us start by establishing a mapping between features and components. Given a feature, cf. Sect. 3, it is turned into a component. Let `f`$\in F$ and $n_f \in Node$, s.t. $f(n_f) =$`f`. To associate components with the nodes of the tree-like structure of root *root* whose nodes correspond to features, we define a function $\kappa : Node \rightarrow 2^{Comp}$ by:

$$\kappa(n_f) = \begin{cases} enc(n_f), & \text{if } n_f = leaf \\ \bigcup_{i=1}^{k} \kappa(Node_i) \cup enc(n_f), & \text{if } n_f = OR(Node_1, ..., Node_k) \vee \\ & n_f = XOR(Node_1, ..., Node_k) \vee \\ & n_f = AND([opt]Node_1, ..., [opt]Node_k) \end{cases} \quad (1)$$

Defined by induction on the node type, $\kappa(root)$ returns the set of components to be generated for the *root* node and all its descendent nodes. Then for a feature `f` on the n_f leaf node, $enc(n_f)$ encoding is called to generate a component of name `f`. For a compound feature, its encoding is called, and κ is recursively invoked on all the sub-nodes until the leafs are met.

4.2 Component Behaviour Generation

Once the set $\kappa(root)$ of components are determined, their behaviour is defined by finite state machines (FSMs), that are automatically generated. Each FSM has finite sets $S, T \subseteq S \times S$ of resp. states and transitions, with specific initial and final states in S. The corresponding FSM for a component `f` is generated, as illustrated in Fig. 4:

$$enc(n_f) = \text{FSM in Fig. 4} \tag{2}$$

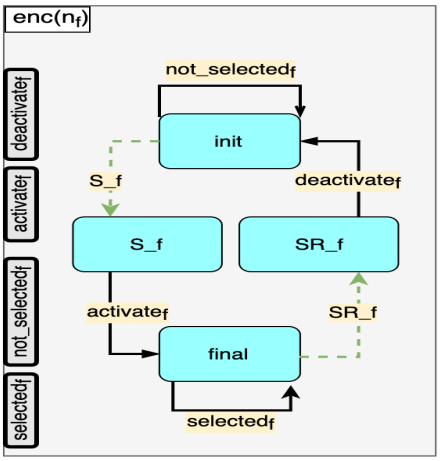

Figure 4: Feature component FSM.

States. In the FSM associated with component f, the states generated are: initial state `init`, where no feature is requested, and no feature is activated; intermediate states `S_f` and `SR_f`, to resp. start f or start reset f, while dealing with requests to activate f or deactivate f; and `final` a state, where feature f is activated.

Transitions. The FSM transitions are associated with either API functions, which require a component to perform actions (enforceable transitions), or event notifications, which allow reacting to external events from the environment (spontaneous transitions). Transitions correspond to the method invocations.

Example 4.1. *Figure 4 illustrates a FSM for feature f, with four states in blue and transitions among them. Transitions represented by dashed green arrows are spontaneous. For example, a spontaneous transition is performed to go to the intermediate state `S_f` when there is a reconfiguration request to activate f.*

4.3 Coordination Layer Generation

Once the individual behaviour of the generated components is defined, a coordination layer between components has to be fixed. Coordination is applied through interactions, which are sets of ports that define allowed synchronizations between components. These interactions are represented by connectors which are the structural representations of the interactions between the ports of the components. To construct this coordination layer, the dependency graph G_{FM} is built by Def. 3.2, and then it is used to compute strongly connected components (SCCs) to capture the set of features that are mutually dependent.

The macros for activating feature f are created based on the strongly connected component of f, denoted by SCC_f, the set of features that f depends on, via $E(f)$, both extracted from G_{FM}, and the set of features that are mutually exclusive with f, via $\chi(f)$. The activation macros of feature f are in Eq. 3 to 6.

Equation 3 states that firing port $activate_f$ requires firing three groups of ports at the same time: 1) *activate* ports of features in SCC_f except f, 2) *selected* ports of features that f depends on outside SCC_f, and 3) *not_selected* ports of features that f excludes.

$$\begin{aligned} requires(enc(n_f).activate_f) \stackrel{\text{def}}{=} & \left\{ enc(n_{f'}).activate_{f'} \;\middle|\; f' \in SCC_f \setminus \{f\} \right\} \\ & \cup \left\{ enc(n_{f'}).selected_{f'} \;\middle|\; f' \in E(f) \setminus SCC_f \right\} \cup \left\{ enc(n_{f'}).not\_selected_{f'} \;\middle|\; f' \in \chi(f) \right\} . \end{aligned} \tag{3}$$

Equation 4 states that the required ports of port $activate_f$ are also the accepted ones:

$$\begin{aligned} accepts(enc(n_f).activate_f) \stackrel{\text{def}}{=} & \left\{ enc(n_{f'}).activate_{f'} \;\middle|\; f' \in SCC_f \setminus \{f\} \right\} \cup \\ & \left\{ enc(n_{f'}).selected_{f'} \;\middle|\; f' \in E(f) \setminus SCC_f \right\} \cup \left\{ enc(n_{f'}).not\_selected_{f'} \;\middle|\; f' \in \chi(f) \right\} . \end{aligned} \tag{4}$$

Similarly, for every feature $f' \in E(f)$,

$$\begin{aligned} requires(enc(n_{f'}).selected_{f'}) &\stackrel{\text{def}}{=} \emptyset \\ accepts(enc(n_{f'}).selected_{f'}) &\stackrel{\text{def}}{=} \left\{ enc(n_{f''}).activate_{f''} \;\middle|\; f'' \in SCC_f \right\} \cup \\ \left\{ enc(n_{f''}).selected_{f''} \;\middle|\; f'' \in E(f) \setminus \{SCC_f, f'\} \right\} & \cup \left\{ enc(n_{f''}).not\_selected_{f''} \;\middle|\; f'' \in \chi(f) \right\} . \end{aligned} \tag{5}$$

For every feature $f' \in \chi(f)$,

$$\begin{aligned} requires(enc(n_{f'}).not\_selected_{f'}) &\stackrel{\text{def}}{=} \emptyset \\ accepts(enc(n_{f'}).not\_selected_{f'}) &\stackrel{\text{def}}{=} \left\{ enc(n_{f''}).activate_{f''} \;\middle|\; f'' \in SCC_f \right\} \cup \\ \left\{ enc(n_{f''}).selected_{f''} \;\middle|\; f'' \in E(f) \setminus SCC_f \right\} & \cup \left\{ enc(n_{f''}).not\_selected_{f''} \;\middle|\; f'' \in \chi(f) \setminus \{f'\} \right\} . \end{aligned} \tag{6}$$

Given the construction of the macros for activation, the corresponding deactivation connectors can be derived by reversing the activation process. In other words, the process of deactivating a feature f is symmetrical to the activation process, where the reverse operation of activation is deactivation, and *selected* becomes *not_selected* of $E^{-1}(f)$ set extracted from the transpose graph G^{-1}_{FM}. Notice that exclude constraints are not considered because they only affect the activation of features, not their deactivation.

$$\begin{aligned} requires(enc(n_f).deactivate_f) \stackrel{\text{def}}{=} & \left\{ enc(n_{f'}).deactivate_{f'} \;\middle|\; f' \in SCC_f \setminus \{f\} \right\} \\ & \cup \left\{ enc(n_{f'}).not\_selected_{f'} \;\middle|\; f' \in E^{-1}(f) \setminus SCC_f \right\} . \\ accepts(enc(n_f).deactivate_f) \stackrel{\text{def}}{=} & \left\{ enc(n_{f'}).deactivate_{f'} \;\middle|\; f' \in SCC_f \setminus \{f\} \right\} \\ & \cup \left\{ enc(n_{f'}).not\_selected_{f'} \;\middle|\; f' \in E^{-1}(f) \setminus SCC_f \right\} . \end{aligned} \tag{7}$$

For every feature $f' \in E^{-1}(f)$,

$$\begin{aligned} requires(enc(n_{f'}).not\_selected_{f'}) &\stackrel{\text{def}}{=} \emptyset \\ accepts(enc(n_{f'}).not\_selected_{f'}) &\stackrel{\text{def}}{=} \left\{ enc(n_{f''}).deactivate_{f''} \;\middle|\; f'' \in SCC_f \right\} \\ & \cup \left\{ enc(n_{f''}).not\_selected_{f''} \;\middle|\; f'' \in E^{-1}(f) \setminus \{SCC_f, f'\} \right\} . \end{aligned} \tag{8}$$

Example 4.2. *Based on the feature model presented in Fig. 5, the coordination layer macros were generated. To illustrate this step, let us consider* `Algolia_real_time_ search` *feature, which forms a singleton strongly connected component (SCC) in the dependency graph G generated from the feature model presented in Fig. 5. The SCC has only one dependency:* `Messaging_and_queuing` *is the parent of* `Algolia_real_time_ search` *feature. Moreover,* `Algolia_real_time_search` *is not mutually exclusive with any other features in the model. Using this information, the macro for the activation of* `Algolia_real_time_search` *feature is created as discussed in Sect.4.3, which is represented graphically by Connector C1. This connector synchronises* $activate_f$ *port of* `Algolia_real_time_search` *component with* $selected_f$ *port of its parent* `Messaging_and_queuing` *component. Intuitively, this ensures that the configuration with* `Algolia_real_time_search` *can be reached only when its dependencies are satisfied.*

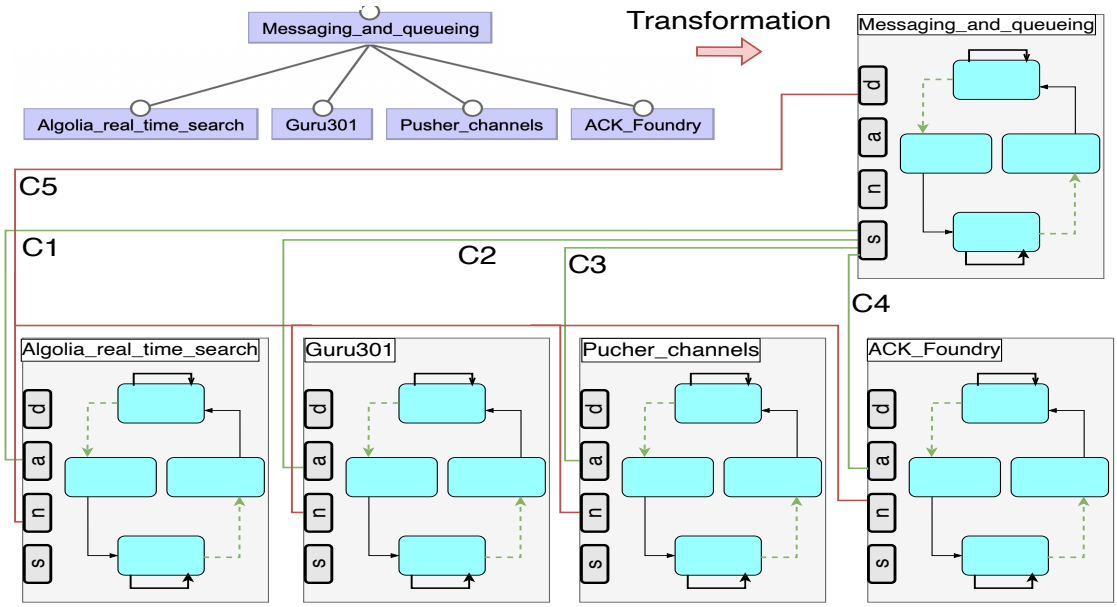

Figure 5: Part of the generated CBVM for the Heroku cloud FM: The behaviour of all the components is the same as shown in Fig. 4. For the sake of clarity, we shorten the names of the ports to the first letter.

Similarly, consider the deactivation of `Messaging_and_queuing` *feature, which forms a singleton strongly connected component (SCC) in the* G^{-1}*, and it has four dependencies with its sub-features. Using this information, the macro for the deactivation of* `Messaging_and_queuing` *feature is created as discussed in Sect. 4.3 which is represented graphically by Connector C5. Connector C5 synchronizes port* $deactivate_f$ *of component* `Messaging_and_queuing` *with all ports* $not\_selected_f$ *of its sub-features. Intuitively, this ensures that the parent can be deactivated only when all its sub-features are in the inactive states.*

After having performed all the steps, the encoding process presented in Fig. 3 terminates. Indeed, at every step, the designed rules deal with finite sets of features, constraints, nodes, components, and connectors. It is easy to establish that the FM semantics in terms of feature configurations [28] is preserved from the FM to the run-time CBVM by applying the encoding process, as the dependency graph issued from the feature model is used.

Since the CBVM is a JavaBIP model, it inherits the operational semantics of JavaBIP [7]. Notice that all interactions among enforceable ports correspond to either the activation of features (Eqs. (3–6)) or their deactivation (Eqs. (7) and (8)). Requesting individual feature activation or deactivation is done through notifications on spontaneous ports.

Proposition 4.1. *Given a run-time CBVM, for each interaction e allowed by Eqs. (3–8), exactly one of the sets* $\{f \in F \mid enc(n_f).activate_f \in e\}$ *and* $\{f \in F \mid enc(n_f).deactivate_f \in e\}$ *is not empty. Furthermore, that set is an SCC of the dependency graph.*

In other words, given a run-time CBVM, each interaction is either a feature activation or deactivation, which involves a strongly connected component in the dependency graph.

Proof. Follows trivially from Eqs. (3–8).

5 CBVM to Deal with Reconfigurations

By construction, the operational semantics of the run-time CBVM is represented by an LTS, whose states are implicitly described configurations with selected features, and whose transitions are labelled by interactions. Performing interactions leads to a configuration change, i.e. reconfigurations, and this section describes their properties.

Proposition 5.1. *Any configuration reachable in the run-time CBVM is a saturated partial-valid configuration.*

Proof. By induction. **Base case:** the empty configuration trivially respects all dependencies and, by Assumption 1, can be completed to a valid configuration. **Induction hypothesis:** if all configurations of size $\leq n$ reachable in the run-time CBVM are saturated partial-valid then that is also the case for configurations of size $n+1$. **Induction step:** Let Φ be a reachable configuration of size $n+1$. There is a reachable configuration $\Phi' \subsetneq \Phi$ and a transition $\Phi' \xrightarrow{e} \Phi$ with e and interaction allowed by Eqs. (3–6). Clearly, $|\Phi'| \leq n$. Hence, by the induction hypothesis, Φ' is a saturated partial-valid configuration. Let $C = \{f \in F \,|\, enc(n_f).activate_f \in e\}$. By Proposition 4.1 and the fact that $\Phi' \subsetneq \Phi$, C is an SCC of the dependency graph. By Eqs. (3–6), the dependencies of all features in C are satisfied. Hence Φ is saturated. Furthermore, also by Eqs. (3–6), the activation of C cannot violate any exclusion constraints. Hence Φ is free from internal conflict and, by Assumption 1, it is partial-valid.

Lemma 5.1. *Let $\Phi \subseteq F$ be a saturated partial-valid configuration. Let C be an SCC of the dependency graph. Then either $C \subseteq \Phi$ or $C \subseteq F \setminus \Phi$.*

Proof. Suppose that $C \subseteq F$ is an SCC, such that both $\Phi \cap C \neq \emptyset$ and $(F \setminus \Phi) \cap C \neq \emptyset$. Then there exists an edge $(f, f') \in C$, such that $f \in \Phi$ and $f' \notin \Phi$, contradicting the assumption that Φ is saturated. Indeed, by Def. 3.3, we have $f' \in E(f) \subseteq \Phi$.

Proposition 5.2. *Let $\Phi \subset \Phi'$ be two saturated partial-valid configurations. Assume Φ is the current configuration of the run-time CBVM. Then the operation of requesting the activation of all features in $\Phi' \setminus \Phi$ is confluent and terminates in the configuration Φ'.*

Proof. Observe that requesting the activation of features is performed by sending event notifications to spontaneous ports. Since components defined in Section 4.2 do not have conflicts among transitions labelled by spontaneous ports, such requests are fully independent.

Since Φ' is saturated partial-valid, it respects all dependencies and is free from internal conflict. By Lemma 5.1, there exist SCC $C_1, \ldots, C_k$, such that $\Phi' \setminus \Phi = \bigcup_{i=1}^{k} C_i$. We continue the proof by induction on k.

Base case: $k = 1$, i.e. $\Phi' \setminus \Phi$ is an SCC. By Eqs. (3–6), the features in $\Phi' \setminus \Phi$ can only be activated as one interaction. By observing the behaviour of components defined in Section 4.2 it is clear that this interaction can only be executed after the execution (in any order) of all spontaneous ports corresponding to the requests of these features. Thus, there is only one execution path possible and it leads to Φ'.

Induction hypothesis: If the statement of the proposition holds for all Φ and Φ' such that $\Phi' \setminus \Phi$ is the union of $k-1$ SCCs, then it also holds for those with k SCCs.

Induction step: By Eqs. (3–6), the activation of SCCs must respect a topological ordering of the DAG obtained by factoring the dependency graph by its SCCs. W.l.g. assume that $1, \ldots, k$ is a sub-sequence of one such ordering, i.e. features from an SCC C_i depend only on those from SCCs C_j with $j < i$. Then, for all $l \in [1, k]$, the configuration $\Phi \cup \bigcup_{i=1}^{l} C_l$ is saturated partial-valid. Applying the base case and the induction hypothesis to $\Phi \cup C_1$ and Φ' and noticing that we did not put any restrictions on the topological ordering concludes the proof.

Corollary 5.1. *For any reachable configuration in the run-time CBVM, there exists a reachable valid configuration.*

Proof. Let Φ be a reachable configuration. By Proposition 5.1, it is saturated partial-valid. Hence, there exists a valid configuration $\Phi' \supseteq \Phi$. Applying Proposition 5.2 to Φ and Φ' proves the corollary.

Corollary 5.2. *Any saturated partial-valid configuration is reachable in the run-time CBVM.*

Proof. Let Φ be a saturated partial-valid configuration. Applying Proposition 5.2 to $\emptyset$ and Φ shows that it is reachable.

Lemma 5.2. *Any synchronized activation of a set of features can be reversed by the corresponding synchronized deactivation of the same features.*

Proof. The $SCCs$ of G_{FM} and its transpose are identical by definition. The transposition of G_{FM} is used because deactivating a feature requires all dependent features to be inactive, which is the opposite of the activation process. This symmetry enables the symmetric derivation of macros for deactivation interactions, as shown in Eqs. 7–8 in Sect. 4.3.

Lemma 5.3. *Let Φ and Φ' be two saturated partial-valid configurations. Then $\Phi \cap \Phi'$ is a saturated partial-valid configuration.*

Proof. Consider any feature $f \in \Phi \cap \Phi'$. Since both Φ and Φ' are saturated partial-valid, we have $E(f) \subseteq \Phi$ and $E(f) \subseteq \cap\Phi'$ by Def 3.3. Thus, $E(f) \subseteq \Phi \cap \Phi'$, i.e. $\Phi \cap \Phi'$ is saturated. Furthermore, since Φ and Φ' do not violate any exclusion constraints then $\Phi \cap \Phi'$ does not violate any exclusion constraints. Hence $\Phi \cap \Phi'$ is free from internal conflict and, by Assumption 1, it is partial-valid.

Proposition 5.3. *Let Φ and Φ' be two saturated partial-valid configurations. Assume Φ is the current configuration in the run-time CBVM. Then the configuration Φ' can be reached by deactivating all and only those features in $\Phi \setminus \Phi'$, then activating all and only those features in $\Phi' \setminus \Phi$.*

Proof. By Lemma 5.3, $\Phi \cap \Phi'$ is a saturated partial-valid configuration. Hence, by Prop. 5.2, Φ can be reached from $\Phi \cap \Phi'$ by requesting the activation of all features in $\Phi \setminus (\Phi \cap \Phi') = \Phi \setminus \Phi'$. By Lemma 5.2, this implies that $\Phi \cap \Phi'$ can be reached from Φ by requesting the deactivation of all features in $\Phi \setminus \Phi'$. Similarly to the above, Φ' can be reached from $\Phi \cap \Phi'$ by requesting the activation of all features in $\Phi' \setminus (\Phi \cap \Phi') = \Phi' \setminus \Phi$.

Given a software architecture represented by a feature model, reconfigurations are operations applied to the architecture in response to either user requirements or external events in the system environment. The generated model is used to handle reconfiguration requests concurrently by controlling the application API through the methods associated with component transitions, as depicted in Fig.6. The run-time CBVM provides the capability to perform reconfigurations without the need to compute a path. The coordination layer, which is built based on the dependency graph of the feature model by Def. 3.2, ensures that the activation or deactivation of a feature occurs in the correct order and only if it is feasible to execute. Additionally, if the interaction of activation or deactivation of a feature is not possible from the current configuration, it will not be executed thus it will be on hold until it can be executed.

Example 5.1. *Building on Example 3.1, let us consider the scenario where we need to move the system from configuration* α_1 = {`Heroku_Application`, `Process_type`, `Dyno`, `Free`, `Region`, `us`, `Add_ons`, `Messaging_and_queuing`, `Guru301`} *to* α_2 = {`Heroku_Application`, `Process_type`, `Dyno`, `Free`, `Region`, `eu`} *by changing the region from* `us` *to* `eu` *and deactivating the* `Guru301` *service.*

Table 1: Possible paths for performing reconfigurations.

Path	Interaction 1	Interaction 2	Interaction 3	Validity
Path 1	Activate eu	Deactivate us	Deactivate Guru301	**Invalid**
Path 2	Activate eu	Deactivate Guru301	Deactivate us	**Invalid**
Path 3	Deactivate us	Deactivate Guru301	Activate eu	**Invalid**
Path 4	Deactivate us	Activate eu	Deactivate Guru301	**Invalid**
Path 5	Deactivate Guru301	Activate eu	Deactivate us	**Invalid**
Path 6	Deactivate Guru301	Deactivate us	Activate eu	**Valid**

Paths 1, 2 & 5 are invalid because the mutually exclusive `us` and `eu` regions are both activated at some point. Paths 3 & 4 are invalid because `Guru301` requires `us` but `us` is deactivated first.

There are six possible reconfiguration paths, as shown in Table 1, that can be taken to move the system from configuration α_1 to α_2. The run-time CBVM can receive the reconfiguration request in any order, however, not all reconfiguration paths are valid, as certain interactions can only occur in specific states. For instance, the only interaction possible from configuration α_1 is the deactivation of `Guru301` feature, as none of the other features depend on it. Once `Guru301` feature is deactivated, the `us` region can be deactivated since it requires synchronization with the "not_selected" port of `Guru301` component, which is already deactivated. Therefore, the interaction for deactivating the `us` region can be executed only from a state where `Guru301` is not active. Finally, the activation of the `eu` feature can only be executed from the state where the `us` region is not active since the `eu` feature is mutually exclusive with other regions, and its activation should be synchronized with the "not_selected" ports of other regions. Hence, the interaction for activating `eu` can be executed only from a state where the `us` region is not active.

Therefore, the only safe order of interactions is to first deactivate `Guru301`, then deactivate `us`, and finally activate `eu`. Any other order can take the system through a not-saturated partial valid intermediate configuration.

To conclude, notice that the run-time CBVM is only generated once without computing the set of valid configurations. Furthermore, it drives the reconfiguration process in a “lazy” manner, by postponing feature (de)activation until it can be safely executed. In particular, this means that we do not have to compute the reconfiguration plan.

6 FeCo4Reco Implementation and Experiments

On the implementation Our model transformation process has been implemented using the ATLAS Transformation Language (ATL) [18]. ATL is a domain-specific language for specifying model-to-model transformations. Starting from a source model that conforms to a source meta-model, allows the developer to produce a target model that conforms to a target meta-model [26]. In our approach, the generated model specification conforms to the JavaBIP meta-model [24]. The generated XML file is parsed using the DOM library in Java to generate the JavaBIP specification and the glue coordination. The reader can find our implementation on the Zenodo platform [1].

Figure 6 shows the FeCo4Reco architecture that enables stages 3 and 4 (Sect.1). The run-time CBVM consists of BIP Specs and the coordination glue. Each BIP Spec is run by a dedicated Executor (forming a JavaBIP module), which implements the FSM semantics and notifies the JavaBIP Engine about the enabled enforceable transitions. The engine uses the coordination glue to decide which components should take which transitions and notifies them accordingly. Component transitions that represent actual reconfiguration actions issue the corresponding HTTP requests through the feature APIs. Finally, reconfiguration requests are injected into the system in the form of spontaneous event notifications (cf. Sect. 3.2). This can be done by different means depending on the requirements of the Cloud comput-

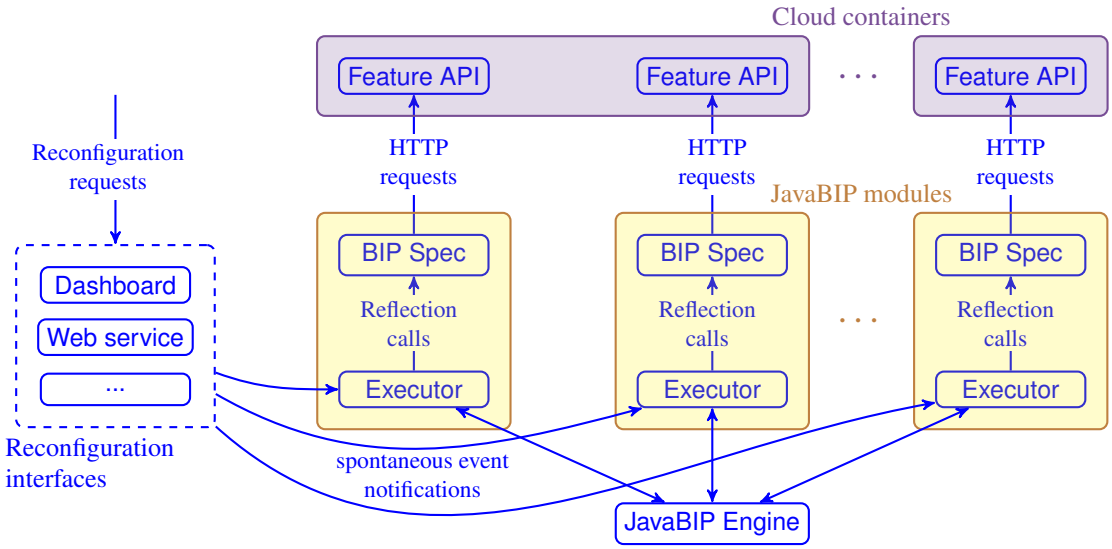

Figure 6: Integration of the JavaBIP Run-time CBVM with a Cloud Computing System.

ing system. For the purposes of this paper, we have designed a dashboard application with two buttons (Activate and Deactivate) for each feature as shown in Fig. 7.

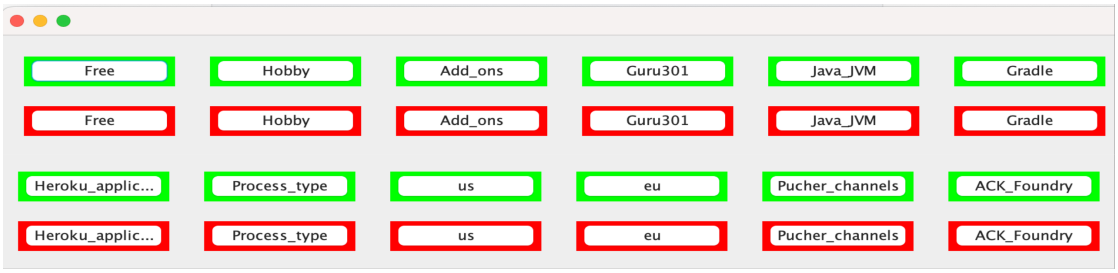

Figure 7: An interface for users to trigger requests.

On the overhead measures Given the number of features (100, 200, and 300) we have tested our approach on twenty randomly generated feature models for each value of this parameter. We have measured the overhead of the generated run-time CBVM over the system.

More precisely, we measured the overhead required to move the system from configuration α_1 to configuration α_2 using the generated CBVM, where α_1 is the current configuration, while α_2 is picked up at random. The CBVM overhead is shown in Fig. 8, both in terms of time (in milliseconds) and memory (in megabytes). Both time and memory overheads are clearly negligible for modern platforms and applications.

On safe reconfiguration To illustrate that the reconfigurations are performed safely we deployed a simple web application onto the Heroku cloud with the run-time CBVM generated from the Heroku cloud FM (Fig. 2). The CBVM is running on a local server using Tomcat 9. It intercepts (re)-configuration requests via APIs using HTTP request methods. The CBVM acts on the received requests to control migrating the system along a valid path to the desired configuration. A simple configuration for the web application is dyno free (free container), region (eu), and build-pack (Java jvm). Consider two reconfiguration scenarios

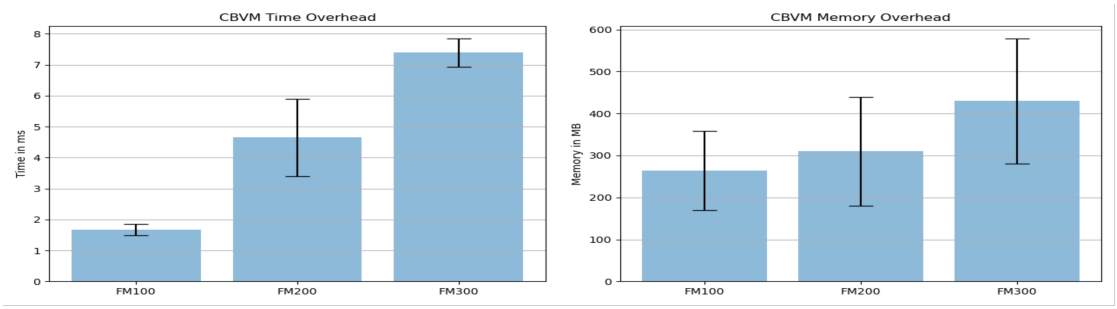

Figure 8: Model overhead (average values for the generated FMs).

in relation to Example 3.1:

1. Adding add-on `Guru301` service to the application hosted in the `us` region

2. Adding add-on `Guru301` service to the application hosted in the `eu` region

Scenario 1: In this case, the `Guru301` service is successfully added to the web application. Indeed, the web application is hosted in the `us` region. Thus, the CBVM transition for activating `Guru301` is triggered synchronously with the selection of the `us` region. The final configuration is shown in Fig. 9.
Scenario 2: In this case, the `Guru301` service is not added to the web application. Upon receiving the request, the run-time CBVM postponed the activation of `Guru301` service until the application region becomes `us`. The final configuration is shown in Fig. 10.

Furthermore, Figures 9 and 10 show the final configurations for two scenarios described in Sect.6 obtained with the generated run-time CBVM. Notice that the traces of our real collaborative activities have been removed to allow the blind review process.

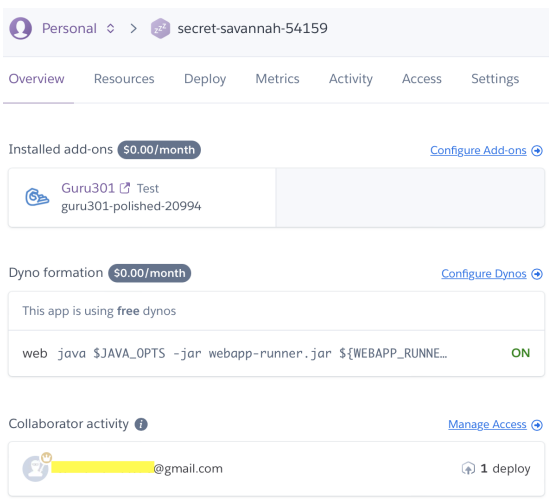

Figure 9: Using the generated CBVM to request the `Guru301` add-on to the `us` region knowing that `Guru301` requires `us`: success.

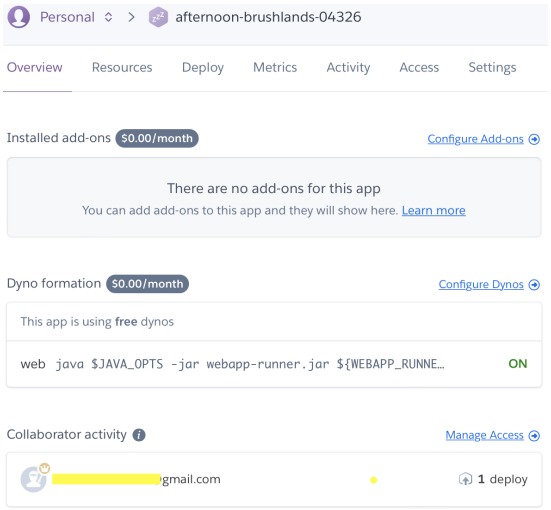

Figure 10: Using the generated CBVM to request the `Guru301` add-on to the `eu` region knowing that `Guru301` requires `us`: no add-ons.

This experiment illustrates that, as expected, the run-time CBVM enforces safe reconfiguration of the cloud application.

7 Related Work

The last two decades have seen various forms of systems evolution. In computer science, it is often related to system architectures, where *dynamic reconfiguration* and *self-adaptation* remain very active research axes for self-adaptive systems [30, 2]. Model-based approaches and service-oriented techniques make use of models together with different artefacts, such as rules, policies, objectives, to model and to deal with system evolution, as e.g., in the framework of self-adaptive systems [31, 30].

Model-based approaches to reconfiguration use system models to analyze information and calculate a reconfiguration plan/self-adaptation plan to keep the system compliant with the user expectation and context constraints. Following [21], the approaches in [22] either compute the set of all possible configurations in advance at design time, or a new valid configuration at run-time. As indicated in Sect.1, this induces computational or validation overheads, whenever a reconfiguration is needed. Indeed, when all valid configurations are precomputed at design time, they must be stored explicitly, e.g. for determining the appropriate choice at run-time. This is problematic since the set of configurations is exponential in the number of features, or in the size of logic formulae, but particularly so for distributed systems, where a copy of the list of features or subformulae has to be stored at every node. Alternatively, at run-time, the new configuration must be computed and validated thus inducing a computational overhead. In [10] the authors advocate the re-use of variability and commonalities at run-time for autonomic computing. In FAMA [5] the engine, Autonomic Reconfigurator, uses the associated resolution to query the run-time models about necessary architectural modifications defined by conditions at design time, in order to generate a reconfiguration plan. Our proposals further leverage variability models to exploit both features and components within component-based run-time variability models.

Component-based approaches allow describing system architectures made up of components that define the life-cycle of the system parts. Connectors control the relationships between the components and establish interactions between them. Typically, Madeus and Concerto [12] define component-based models focusing on modelling and coordinating the life-cycle of interacting parts of a system. Concerto provides ports that represent the ability of a component to provide services to other components (e.g. service, piece of data) during its life cycle. Unlike Madeus, Concerto is equipped with a reconfiguration language that allows the system administrator to modify the architecture by executing reconfiguration scripts (controlling the system by inserting new reconfiguration actions).

The recent survey [2] analyses approaches, methodologies, or design patterns for managing runtime variability through software reconfigurations in Dynamic SPL for self-adaptive systems. The authors note that many existing approaches compensate between memory and time, as, e.g., [26, 29].

Our approach exploits the advantages of both features and components to generate run-time component-based variability models that ensure, by construction, safety properties through reconfigurations. In [29], FM has been extended with relative cardinalities and then used for checking temporal logic formulae, in order to support the dynamic evolution of microservices applications. Starting from an extended FM, instead of generating a whole transition system to check the reconfiguration path against propositional and temporal constraints, in the present paper, we generate a run-time CBVM for handling reconfiguration safely. However, dealing with liveness temporal properties remains a future work direction.

The novelty of our approach lies in the generation of a correct-by-construction runtime CBVM that enforces domain constraints for dynamic reconfiguration while leveraging SPL tools to capture domain variability. Unlike many works using SPL techniques and tools, our approach goes beyond employing static variability models at run-time, e.g. [14], and avoids the overhead of computing or validating new target configurations explicitly. Furthermore, our approach differs from other by-construction reconfiguration techniques, such as [29], which require computing the global LTS to derive the reconfiguration paths. Instead, we encode the feature model constraints into a JavaBIP run-time CBVM. In turn, the

JavaBIP engine relies on Binary Decision Diagrams (BDDs) [3] to efficiently encode the various operational and coordination constraints of the system and to compute the possible interactions from the current CBVM state [7, 17]. In particular, permanent constraints, which encode information about the behavior, glue, and data wires of the components, are encoded only once at the initialization of the JavaBIP system. The only constraints that are recomputed at each cycle are the temporary ones that encode the current states of the components. This allows us to strike the right balance between, on one hand, precomputing the global reconfiguration LTS—which involves a very long computation at the initialisation phase and potentially requires a huge amount of memory to store the result—and, on the other hand, verifying all the constraints at run-time—which eliminates the initialisation phase and the memory overhead at the cost of a run time overhead orders of magnitude larger than that observed with our approach.

8 Conclusion and Future work

This paper describes an automated approach for enforcing by construction the safe reconfiguration behaviour of software products through the automatic derivation of executable, run-time component-based variability models (CBVMs) from feature models. The run-time CBVMs, control the application behaviour by handling reconfiguration requests and executing them so as to ensure the saturated partial validity of all reachable configurations without having to compute, nor validate them at run-time. Our approach ensures the preservation of feature model semantics and constraint consistency in the generated models as established in Sect. 5. Additionally, the feasibility and effectiveness of our approach are demonstrated through an evaluation of the overhead induced by the run-time CBVM on applications containing up to 300 features. The experimental results show the interest of our approach for handling reconfiguration requests with low overhead over the application. Furthermore, the successful integration of our approach into a real-world case scenario with the Heroku cloud demonstrates the feasibility of our approach for practical applications. Therefore, our approach provides a solution to the research question formulated in Sect. 2.

The key threat to the validity of our work lies in Assumption 1. We expect that assumption to hold for a large proportion of realistic feature models. Efficiently verifying or enforcing that assumption in the general case is hard [20]. However, we can put in place additional heuristics based on the propagation of exclusion constraints to further increase the proportion of feature models that satisfy the assumption.

As a future work direction, we will explore stronger heuristics to enforce Assumption 1. Furthermore, we intend to generalise our approach to constraints among features formulated in terms of arbitrary Boolean formulas. As shown in [8], that will require extending JavaBIP with priority models. To streamline the user experience and minimise the necessity to wait for user input, we also plan to extend the feature model with default values. Additionally, we aim to incorporate temporal constraints over features, e.g. excluding the possibility of downgrading the database plan [29] and plan to investigate the analytical treatment of the space complexity of the BDDs used in our approach.

References

[1] *Toward Run-time Coordination of Reconfiguration Requests in Cloud Computing Systems*. Zenodo, March 2023.

[2] Oscar Aguayo and Samuel Sepúlveda. Variability management in dynamic software product lines for self-adaptive systems—a systematic mapping. *Applied Science*, 12(20):10240, 2022.

[3] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[4] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE software*, 28(3):41–48, 2011.

[5] David Benavides, Pablo Trinidad, Antonio Ruiz-Cortés, and Sergio Segura. Fama. In *Systems and software variability management*, pages 163–171. Springer, 2013.

[6] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proc. of the 7th Int. Wshop on Variability Modelling of Software-intensive Systems*, pages 1–8, 2013.

[7] Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. Exogenous coordination of concurrent software components with JavaBIP. *Software: Practice and Experience*, 47(11):1801–1836, 2017.

[8] Simon Bliudze and Joseph Sifakis. Synthesizing glue operators from glue constraints for the construction of component-based systems. In Sven Apel and Ethan Jackson, editors, *10th International Conference on Software Composition*, volume 6708 of *LNCS*, pages 51–67. Springer, 2011.

[9] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A classification of dynamic reconfiguration in component and connector architecture description languages. In *4th Int. Wshop ModComp*, volume 1, 2017.

[10] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, 91:3–23, 2014.

[11] Carlos Cetina, Joan Fons, and Vicente Pelechano. Applying software product lines to build autonomic pervasive systems. In *2008 12th Int. SPL Conf.*, pages 117–126. IEEE, 2008.

[12] Maverick Chardet, Hélène Coullon, and Simon Robillard. Toward safe and efficient reconfiguration with concerto. *Science of Computer Programming*, 203:102582, 2021.

[13] Ivica Crnkovic, Michel Chaudron, Séverine Sentilles, and Aneta Vulgarakis. A classification framework for component models. *Software Engineering Research and Practice in Sweden*, page 3, 2007.

[14] Sina Entekhabi, Ahmet Serkan Karataş, and Halit Oğuztüzün. Dynamic constraint satisfaction algorithm for online feature model reconfiguration. In *Int. Conf. on Control Engineering and Information Technology (CEIT)*, pages 1–7, 2018.

[15] Salman Farhat, Simon Bliudze, Laurence Duchien, and Olga Kouchnarenko. Toward run-time coordination of reconfiguration requests in cloud computing systems. In Antónia Lopes and Sung-Shik T. Q. Jongmans, editors, *Proc. of the 25th International Conference on Coordination Models and Languages (COORDINATION 2023)*, LNCS. Springer, June 2023. To appear.

[16] H. Gomaa and M. Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *Proc. 4th Working IEEE/IFIP Conf. WICSA 2004*, pages 79–88, 2004.

[17] Mohamad Jaber, Ananda Basu, and Simon Bliudze. Symbolic implementation of connectors in BIP. In Filippo Bonchi, Davide Grohmann, Paola Spoletini, and Emilio Tuosto, editors, *Proceedings 2nd Interaction and Concurrency Experience: Structured Interactions, ICE 2009, Bologna, Italy, 31st August 2009*, volume 12 of *EPTCS*, pages 41–55, 2009.

[18] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[19] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh, Pa, Software Engineering Inst, 1990.

[20] Oliver Kautz. The complexities of the satisfiability checking problems of feature diagram sublanguages. *Software and Systems Modeling*, pages 1–17, 2022.

[21] Jeffrey O. Kephart. Research challenges of autonomic computing. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th Int. Conf. ICSE*, pages 15–22. ACM, 2005.

[22] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17:184–206, 2015.

[23] Tudor A Lascu, Jacopo Mauro, and Gianluigi Zavattaro. A planning tool supporting the deployment of cloud applications. In *2013 IEEE 25th Int. Conf. on Tools with Artificial Intelligence*, pages 213–220. IEEE, 2013.

[24] Anastasia Mavridou, Joseph Sifakis, and Janos Sztipanovits. DesignBIP: A design studio for modeling and generating systems with BIP. *arXiv preprint arXiv:1805.09919*, 2018.

[25] Neil Middleton and Richard Schneeman. *Heroku: up and running: effortless application deployment and scaling*. " O'Reilly Media, Inc.", 2013.

[26] Clément Quinton, Daniel Romero, and Laurence Duchien. Saloon: a platform for selecting and configuring cloud environments. *Software: Practice and Experience*, 46(1):55–78, 2016.

[27] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives, 2012.

[28] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *14th IEEE Int. Conf. RE'06*, pages 136–145. IEEE Computer Society, 2006.

[29] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Extending dynamic software product lines with temporal constraints. In *2017 IEEE/ACM 12th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 129–139. IEEE, 2017.

[30] Danny Weyns. Software engineering of self-adaptive systems. In Sungdeok Cha, Richard N. Taylor, and Kyo Chul Kang, editors, *Handbook of Software Engineering*, pages 399–443. Springer, 2019.

[31] Zhuoqun Yang, Zhi Li, Zhi Jin, and Yunchuan Chen. A systematic literature review of requirements modeling and analysis for self-adaptive systems. In Camille Salinesi and Inge van de Weerd, editors, *Proc. 20th Int. Conf. REFSQ 2014*, volume 8396 of *LNCS*, pages 55–71. Springer, 2014.

[32] Ji Zhang and Betty HC Cheng. Model-based development of dynamically adaptive software. In *Proc. of the 28th Int. Conf. on Software engineering*, pages 371–380, 2006.

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399