Safe Dynamic Reconfiguration of Applications with Features

Salman Farhat

INVENTEURS DU MONDE NUMÉRIQUE

Université de Lille

CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

MADIS Doctoral School
Speciality : Computer Science

Salman Farhat

University of Lille & CRIStAL & Inria
Project-team Spirals

Safe Dynamic Reconfiguration of Applications with Features

Thesis defended on July 11, 2024

Thesis Committee:

Mathieu Acher	Professor at the University of Rennes	Reviewer
Marius Bozga	CNRS Research Engineer at VERIMAG	Reviewer
Olga Kouchnarenko	Professor at the University of Franche-Comté	Examiner
Anne Etien	Professor at the University of Lille	Jury President / Examiner
Laurence Duchien	Professor at the University of Lille	Supervisor
Simon Bliudze	Inria Researcher at the Inria Center of the University of Lille	Co-supervisor

École Doctorale MADIS

Spécialité : Informatique

Salman Farhat

Université de Lille & CRIStAL & Inria

Équipe-Projet Spirals

Reconfiguration dynamique sûre des applications comportant des options fonctionnelles

Thèse soutenue le 11 Juillet 2024

Jury:

Mathieu Acher	Professeur à l'Université de Rennes	Rapporteur
Marius Bozga	Ingénieur de recherche CNRS au VERIMAG	Rapporteur
Olga Kouchnarenko	Professeure à l'Université de Franche-Comté	Examinatrice
Anne Etien	Professeure à l'Université de Lille	Présidente du jury / Examinatrice
Laurence Duchien	Professeure à l'Université de Lille	Directeur
Simon Bliudze	Chercheur Inria au Centre Inria de l'Université de Lille	Co-directeur

Acknowledgements

First of all, I want to thank my supervisors, Laurence Duchien and Simon Bliudze, for their guidance, support, and encouragement throughout my Ph.D. journey. I am grateful for their expertise, feedback, and continuous guidance. Thank you, Laurence, for the time you made available for me despite your many other responsibilities. Your motivation and kindness have been a source of inspiration for me. Simon, I appreciate your availability and the work environment that was always open for discussions and new ideas. Thank you for all the motivation you gave and for taking care of even the smallest details. I am grateful for the opportunity to work with you. This work wouldn't have been possible without your help and support. I am also thankful for the opportunity to work with Olga Kouchnarenko, who has been a great collaborator and an additional source of inspiration. Thank you, Olga, for all the meetings, discussions, and excellent work we've done together. Your availability, matching that of my supervisors, is deeply appreciated.

I would also like to thank the Spirals team members. Thank you, Lionel Seinturier, for being a great team leader. I'm grateful to Alexandre, Adrien, Clément, Trình, Daniel, Rémy, Hugo, Brel, Pierre, Sihem, Belkis, Walter, Jean-Luc, and all other Spirals members for our insightful discussions and for all the best wishes you've extended to me. Thank you, Naif, for all the laughs and good times we've shared. You've been a friend throughout this journey.

Next, I would like to thank my family, especially my father, Mortada Farhat, my mother, Imane Farhat, and my siblings, Latifa, Abbas, Yousef, Jaafar, Hasan, Salim, and Sarah, for their support. I appreciate every single call that they've made to support me through all these days, and I appreciate their presence in my life as they are the motivation. I am grateful for their encouragement and understanding throughout my studies.

I am also thankful to my friends in Lille. Lea, Malak, Eyad, Miled, Mehdi, Tamara, Ramy, Khaled, Raghda, Ali, Wassim, and Ramy: thank you for all our outings, our discussions, and your support. I am grateful for the memories we've made together. I would also like to thank my football friends, Hussein,

Yorgo, Wassim, Marc, and all the others, for the good times we've shared. You made Lille more enjoyable for me.

Finally, I would like to thank Diala for being more excited than me about my success, for her patience, and for her support. I am grateful for her presence.

I acknowledge and appreciate the contributions of all those who have supported me, both directly and indirectly, throughout this endeavor. Your impact on my academic and personal life has been invaluable. The completion of this PhD dissertation represents not only a personal milestone but also a collective achievement. I express my sincere gratitude to everyone who played a role in this journey, whether their contributions were visible or unseen.

Abstract

Cloud applications and cyber-physical systems require frequent reconfiguration at run-time to adapt to changing needs and requirements, highlighting the importance of dynamic reconfiguration capabilities. Additionally, the environment platforms can extend and modify their services at run-time, which necessitates a compositional approach to allow the modifications of the configurations. To manage the variability of large systems' architecture, feature models are widely used at design-time with several operators defined to allow their composition. Existing approaches compute new valid configurations either at design time, at runtime, or both, leading to significant computational or validation overheads for each reconfiguration step. In addition, building correct-by-construction formal models to handle application reconfigurations is a complex and error-prone task, and there is a need to make it automated as far as possible.

To address these challenges, we propose an approach that leverages feature models to automatically generate, in a component-based formalism called JavaBIP, component-based run-time variability models that respect the feature model constraints. These component-based run-time variability models are executable and can be used at runtime to enforce the variability constraints, that is, to ensure the (partial) validity of all reachable configurations.

As complex systems' architectures may evolve at run-time by acquiring new features and functionalities while respecting new constraints, we define 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.

For the experimental evaluation, we demonstrated the applicability of our approach in real-world scenarios by generating a run-time model based on the feature model of the Heroku cloud platform using our approach. This model is then used to deploy a real-world web application on the Heroku platform. Furthermore, we measured the time and memory overheads induced by the

generated run-time models on systems involving up to 300 features. The results show that the overheads are negligible, demonstrating the practical interest of our approach.

Résumé

Les applications en nuage et les systèmes cyber-physiques nécessitent une reconfiguration fréquente en cours d'exécution pour s'adapter à l'évolution des besoins et des exigences, ce qui souligne l'importance des capacités de reconfiguration dynamique. En outre, les plateformes d'environnement peuvent étendre et modifier leurs services en cours d'exécution, ce qui nécessite une approche compositionnelle pour permettre la modification des configurations. Pour gérer la variabilité de l'architecture des grands systèmes, les modèles de caractéristiques sont largement utilisés au moment de la conception, avec plusieurs opérateurs définis pour permettre leur composition. Les approches existantes calculent de nouvelles configurations valides soit au moment de la conception, soit au moment de l'exécution, soit les deux, ce qui entraîne d'importants frais généraux de calcul ou de validation pour chaque étape de reconfiguration. En outre, la construction de modèles formels corrects par construction pour gérer les reconfigurations d'applications est une tâche complexe et sujette aux erreurs, et il est nécessaire de l'automatiser autant que possible.

Pour relever ces défis, nous proposons une approche qui s'appuie sur les modèles de caractéristiques pour générer automatiquement, dans un formalisme basé sur les composants appelé JavaBIP, des modèles de variabilité d'exécution basés sur les composants qui respectent les contraintes du modèle de caractéristiques. Ces modèles de variabilité d'exécution basés sur les composants sont exécutables et peuvent être utilisés à l'exécution pour appliquer les contraintes de variabilité, c'est-à-dire pour garantir la validité (partielle) de toutes les configurations atteignables.

Comme les architectures des systèmes complexes peuvent évoluer à l'exécution en acquérant de nouvelles caractéristiques et fonctionnalités tout en respectant de nouvelles contraintes, nous définissons des opérateurs de composition pour les modèles de variabilité à l'exécution basés sur des composants qui non seulement encodent ces opérateurs de composition de modèles de caractéristiques, mais garantissent également une reconfiguration sûre à l'exécution. Pour prouver les propriétés de correction et de composition, nous proposons une nouvelle

équivalence $\mathcal{UP}$-bisimulation en plusieurs étapes et l'utilisons pour montrer que les modèles de variabilité d'exécution basés sur les composants préservent la sémantique des modèles de fonctionnalités composés.

Pour l'évaluation expérimentale, nous avons démontré l'applicabilité de notre approche dans des scénarios réels en générant un modèle d'exécution basé sur le modèle de caractéristiques de la plateforme cloud Heroku à l'aide de notre approche. Ce modèle est ensuite utilisé pour déployer une application web réelle sur la plateforme Heroku. En outre, nous avons mesuré les surcharges de temps et de mémoire induites par les modèles d'exécution générés sur des systèmes impliquant jusqu'à 300 fonctionnalités. Les résultats montrent que les surcharges sont négligeables, ce qui démontre l'intérêt pratique de notre approche.

Table of contents

List of figures xi

List of tables xiii

1 Introduction 1
- 1.1 Problem Statement . . . 1
- 1.2 Research Goals . . . 5
- 1.3 Contributions . . . 5
- 1.4 Tool Implementation . . . 7
- 1.5 Dissertation Outline . . . 7
- 1.6 List of Scientific Publications . . . 9

2 Background and Concepts 11
- 2.1 Cloud Computing . . . 11
  - 2.1.1 Cloud Computing Characteristics . . . 12
  - 2.1.2 Service Models . . . 13
  - 2.1.3 Deployment Models . . . 14
- 2.2 Software Product Lines Engineering . . . 15
  - 2.2.1 Variability . . . 15
  - 2.2.2 Software Product Line Engineering Process . . . 16
  - 2.2.3 Variability Model . . . 17
  - 2.2.4 Feature Model . . . 17
  - 2.2.5 Feature Diagram . . . 18
  - 2.2.6 Feature Model Logical Formula . . . 19
  - 2.2.7 Configuration Validity and Extensions to Feature Modeling 20
- 2.3 Dynamic Software Product Line . . . 21
- 2.4 The JavaBIP Framework . . . 22
- 2.5 Summary . . . 26

3 State of Art 27
- 3.1 Reconfiguration and Self-Adaptation . . . 27

3.2 MAPE-K Loop . . . 28
3.3 Approaches for Reconfiguration and Self-Adaptation . . . 29
3.3.1 DSPL-based Approaches . . . 30
3.3.2 Formal Component-based Approaches . . . 34
3.4 Compositionality and Composability . . . 37
3.4.1 Introduction to Compositionality and Composability . . . 37
3.4.2 Composition Approaches for Software Models . . . 39
3.5 Conclusion . . . 40

4 Automatic Generation of Component-based Run-time Variability Models 43
4.1 Background . . . 43
4.1.1 Feature Model Formalization . . . 44
4.1.2 Feature Model Notation . . . 44
4.2 Motivation . . . 45
4.2.1 Key Elements of the Heroku Cloud Platform . . . 46
4.3 Feco4Reco: A Framework . . . 49
4.3.1 Stage 1: Heroku Cloud Feature Model . . . 49
4.3.2 Stage 2: Transformation: Feature Model to Component-based Run-time Variability Model . . . 51
4.3.3 Stage 3: CBRTVM Integration . . . 62
4.4 Preserving Feature Model Semantics in CBRTVM . . . 62
4.5 Conclusion . . . 68

5 Composing Run-time Variability Models 69
5.1 Composition of Feature Models . . . 70
5.2 Composition of CBRTVMs . . . 71
5.2.1 Macros for Composition . . . 71
5.2.2 Composing **Requires** Macros . . . 73
5.2.3 Saturation Process for **Accepts** Macros . . . 74
5.2.4 Composition Operators on JavaBIP models . . . 74
5.3 A Bisimulation for Correctness and Compositionality Results . . . 75
5.3.1 Proof of Intersection Case . . . 81
5.3.2 Proof of Strict Intersection Case . . . 83
5.3.3 Proof of Union Case . . . 83
5.3.4 Congruence of the $\mathcal{UP}$-bisimulation . . . 85
5.4 Conclusion . . . 86

6 Practical and Experimental Validation **87**
6.1 Implementation of FeCo4Reco Transformation Process 87
6.2 Heruko Deployer Overview 89
6.2.1 Performance Evaluation 91
6.2.2 Safe Reconfiguraiton 91
6.3 Composition Operators Experimental Validation 94
6.3.1 Running Example 94
6.3.2 Experimental Setup and Evaluation 95
6.4 Conclusion 97

7 Conclusion and Perspective **99**
7.1 Summary of the Dissertation 99
7.2 Perspectives 100
7.2.1 Short-term Perspectives 101
7.2.2 Long-term Perspectives 101

Bibliography **103**

List of figures

1.1 Stages of the FeCo4Reco process. 7

2.1 Domain and application engineering. 16
2.2 An example feature diagram. 18
2.3 A JavaBIP component-based model. 25

4.1 Part of the Heroku cloud feature model. 50
4.2 Valid product selection. 51
4.3 The generation of the component-based run-time variability model involves a two-step transformation process. Steps 1 and 2 correspond to Subsections 4.3.2.1 and 4.3.2.2, respectively. . . . 51
4.4 Feature component FSM. 54
4.5 An example feature model. 56
4.6 Directed dependency graph generated from the feature model in Fig. 4.5. 57
4.7 Extracted SCCs from G_{FM}. 57
4.8 Part of the generated CBVM for the Heroku cloud FM: The behaviour of all the components is the same as shown in Fig. 4.4. For the sake of clarity, we shorten the names of the ports to the first letter. 61
4.9 Part of the CBRTVM generated for the feature model presented in Fig. 4.5. 66

5.1 Overview of Feature and JavaBIP models composition 70
5.2 Example of a dependency graph 73
5.3 76
5.4 76
5.5 Two feature models that have the same set of valid configurations. 76
5.6 77
5.7 77

6.1 Feature model metamodel. 88

6.2 JavaBIP metamodel. 89
6.3 Integration of the JavaBIP CBRTVM with a Cloud Computing System. 90
6.4 An interface for users to trigger requests. 90
6.5 Heroku deployer component. 91
6.6 The overview of Heroku Deployer integrated into CBRTVM. . . 92
6.7 Model overhead (average values for the generated FMs). 93
6.8 Using the generated CBRTVM to request the `Guru301` add-on to the `us` region knowing that `Guru301` requires `us`: success. . . 93
6.9 Using the generated CBRTVM to request the `Guru301` add-on to the `eu` region knowing that `Guru301` requires `us`: no add-ons. 93
6.10 Simplified Heroku Cloud feature model. 95
6.11 CloudWatch FM. 95
6.12 Observed configurations. 96

List of tables

3.1 MAPE-K loop table. A dash (–) in the table signifies that the corresponding phase is supported manually by developers or through tools made up at the design time. 38

4.1 Possible paths for performing reconfigurations. 48

6.1 Feature inclusion in configurations Φ_1, Φ_2, and Φ_3 94

Chapter 1

Introduction

The software system engineering life cycle [110, 128] typically involves several stages: analysis, design, development, testing, and finally deployment. In the analysis stage, the requirements and constraints of the system are identified and documented. During the design stage, the overall architecture and components of the system are planned and specified. The development stage involves the actual implementation of the system according to the design specifications. Testing is the process of verifying that the developed system meets the specified requirements and functions correctly. Finally, in the deployment stage, the system is installed and made operational in its intended environment.

In the past, software systems were designed to be static, meaning their configuration was fixed after deployment, and they were not expected to undergo significant changes during their operational lifetime. However, the advent of cloud computing and the increasing complexity of modern software environments have fundamentally altered this paradigm. Static systems face several limitations in these dynamic environments. First, they lack the flexibility to accommodate changes in requirements or operating conditions without significant effort and downtime. Second, they are unable to optimize their resource usage or scale their capacity in response to fluctuating demand, leading to inefficient resource utilization and potential performance bottlenecks. Third, they are susceptible to failures or degradation in the face of changing environmental conditions or component failures, which can compromise their reliability and availability [37].

1.1 Problem Statement

To address these limitations, systems deployed in dynamic environments are expected to reconfigure at runtime to stay compliant with new user needs and underlying platform constraints. For a system to reconfigure, it needs to

transition from its current configuration to a new configuration that aligns with the new requirements.

There are various possible events that may demand dynamic reconfiguration of the system at runtime. One factor is changes in user needs. As user demands may change, they may seek additional features that are not part of the current configuration of the system. For example, take a cloud-based online retailer where the system administrator has to add new monitoring and logging services. Specifically, they want to add a centralized log aggregation service that gathers and analyzes application logs, as well as a monitoring service that measures key performance indicators and provides warnings based on predetermined criteria. Where the monitoring service relies on the log aggregation service to obtain the aggregated logs for analysis.

In this scenario, it is important that the reconfiguration process activates the log aggregation service before the monitoring service. Attempting to activate the monitoring service first would result in errors or incorrect behavior, as it relies on the log aggregation service being already active and functioning properly. Thus, there is a need to enforce the system to transition through a safe reconfiguration path that respects the dependencies between services, so that the system can transition safely to the desired new configuration, avoiding inconsistencies or disruptions during the reconfiguration process.

This highlights the importance of an approach that allows **reconfiguration** not only from a **source configuration** to a **target configuration** in a **random order** but also enforces correct **activation** and **deactivation ordering**, respecting **dependencies** between **services**, to maintain **system integrity** and **consistency** throughout the **reconfiguration process**.

Configuration and Reconfiguration

In the context of distributed systems, such as component-based systems and microservices architectures deployed in the cloud, the system configuration refers to the components of the system deployed on cloud resources and their interconnections that make up the system at a given point in time. Reconfiguration, in this context, refers to the process of changing this system configuration by transitioning components from an inactive state to an active state or vice versa, effectively adding or removing them from the configuration. The activation and deactivation processes must be guided by synchronized actions between components to guarantee a safe reconfiguration.

The reconfiguration process can be executed either at runtime, which we call dynamic reconfiguration, or at downtime, which we call static reconfiguration.

The static reconfiguration process typically involves shutting down the system, making the necessary changes, and then restarting the system with the new configuration.

Dynamic reconfiguration allows the system to transition from one configuration to another target configuration at runtime. This capability is useful when continuous operation and availability are important, such as in cloud applications. By dynamically reconfiguring the system, components can be added and removed while the system remains operational, enabling it to respond to changing requirements without disrupting the whole system.

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, dynamic reconfigurations are needed to keep the application compliant with the new needs and underlying platform constraints at runtime [89, 141]. In this dissertation, only **dynamic reconfiguration** will be taken into account.

Whatever 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 [144, 90], and rules/policies [40, 89].

When employing these approaches, computing valid system configurations can be a computationally intensive task, especially for large-scale and highly configurable systems. There are multiple strategies that can be adopted, each with its own trade-offs. One approach is to compute all valid configurations at design time, which offloads the computational burden from the runtime environment, leading to better performance and responsiveness during operation. However, this is infeasible for systems with a large number of features, since the number of valid configurations can be exponential in the number of features, making it infeasible to enumerate all configurations [56, 133]. On the other hand, computing an appropriate configuration at run-time provides flexibility and adaptability, but may incur performance penalties due to the computational overhead involved.

Furthermore, formal approaches have been proposed to tackle reconfiguration and guarantee the properties of the system while applying dynamic reconfiguration. These approaches focus more on the safe execution of the reconfiguration plan while leaving behind the analysis and planning of the target configuration. Building these formal models correctly to guarantee safe-by-construction behavior is a challenging task that requires significant

effort and expertise. Constructing accurate formal models that capture all the relevant system properties, constraints, and behavioral aspects is complex and error-prone, especially for large-scale systems.

The problem we aim to tackle is enabling safe dynamic reconfiguration for large-scale software systems with numerous configuration options and a vast number of potential configurations. Ensuring safe reconfiguration is challenging, whereby safe means the reconfiguration process transitions the system only through (partial)-valid configurations. By partial-valid configuration, we refer to a configuration of the system at any given point during the reconfiguration process that adheres to domain constraints. Therefore, as the system transitions through a reconfiguration path involving multiple intermediate states, there is a need to ensure that the system transitions only through valid paths. Valid paths refer to sequences of intermediate states where these intermediate states are partial-valid configurations.

Furthermore, due to the high complexity of the systems and the numerous configurations of a highly configurable system, it is important to have an approach that allows satisfying the objectives at low cost, avoiding computational overhead in terms of memory and time.

Software Evolution

Software evolution [42, 97] is the continual development of system software to extend or modify 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. Note that this goes beyond reconfiguration which explores pre-defined configurations within the existing modeled domain.

To enable modeling a system as it evolves, there must be a means to integrate sub-models that model new functionalities 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 that models the modified configuration space.

For instance, consider a component-based model that represents the services of the Heroku Cloud platform [1]. Cloud platforms are dynamic and evolve over time [88], 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 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.

Therefore, as software systems evolve over time to meet changing requirements, there is a need for approaches that can be compositional so that new functionalities can be integrated to change the configuration space to support the evolution aspect.

1.2 Research Goals

In the above context, we aim to address three main research questions. The first research question we aim to respond to is how to allow the reconfiguration of software systems in a safe manner while avoiding additional overhead at run-time. We define safe in the sense that only partial-valid configurations can be reached as a result of any reconfigurations.

RQ1 How to enforce domain constraints during dynamic reconfiguration at low cost?

For the compositionality aspect of our approach, we aim to address two key research questions. The first question focuses on enabling compositionality within the approach, which is essential for supporting the evolution of software systems and meeting changing requirements. The second question addresses ensuring the correctness and consistency of the compositional approach, specifically in terms of enforcing domain constraints based on the semantics of the composition.

RQ2: How can we enable compositionality in our approach?

RQ3: How can we ensure that the compositional approach consistently enforces domain constraints based on the semantics of the composition?

1.3 Contributions

After presenting the goals and the research questions that this dissertation aims to answer, we provide an overview of the key contributions presented in this dissertation. The main contributions of our work are summarized as follows:

1. A component-based run-time variability model (CBRTVM) leverages feature models and their underlying constraints for enforcing safe-by-construction behavior of concurrent component-based systems.

We present an approach that leverages feature models to automatically generate an executable formal model that enforces safe reconfigurations at

runtime. To this end, we take advantage of feature models and component-based run-time models to enforce safe-by-construction behavior of concurrent component-based systems. Feature models, recognized for their efficacy in capturing domain variability, enable compact representations of valid system configurations. They are well-known and used to model the commonality and variability of complex environments such as cloud platforms [116]. This enables the creation of compact representations that encompass all valid configurations a system can have within its domain. On the other hand, component-based run-time models are generated automatically from feature models. The generated model is called a component-based run-time variability model (CBRTVM). This CBRTVM encodes reconfiguration operations while ensuring the safety property, stating that only (partial-)valid configurations can be reached as a result of any reconfigurations. The main properties related to reconfigurations are described in Chapter 4.

2. Model transformation rules that are general enough for both feature models and component-based models, enabling the automated generation of CBRTVMs using our FeCo4Reco (joining forces of features and components for safe reconfigurations) approach.

The FeCo4Reco framework enables the application of reconfigurations with minimal effort, without the overhead of computing or validating the new configuration, while providing a CBRTVM. FeCo4Reco transformation process consists of three stages, as illustrated in Figure 1.1. The initial stage involves the creation of a variability model (Feature Model) from domain constraints, such as a variability model of a cloud platform. In the second stage, this feature model is automatically transformed into a CBRTVM. The final stage involves the integration of this CBRTVM with the cloud platform, where all reconfiguration requests are sent to the CBRTVM and managed in a safe manner.

Furthermore, to facilitate compositionality of CBRTVMs, we present the following contributions:

3. Novel JavaBIP composition operators corresponding to feature model composition operators taken from the literature, rendering our approach compositional while preserving the safety of dynamic reconfiguration.

4. The notion of a multi-step $\mathcal{UP}$-bisimulation is defined, and the correctness and compositionality of the encoding are proved.

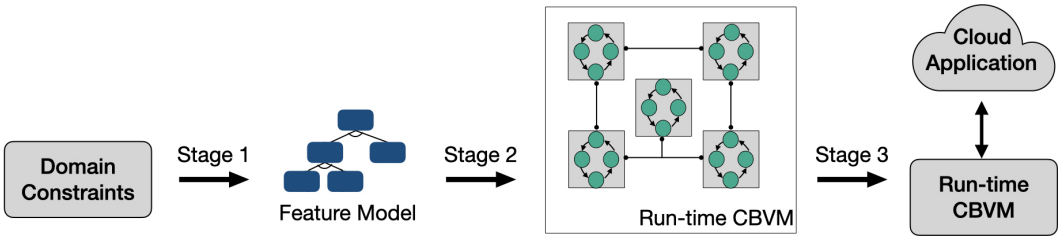

Figure 1.1: Stages of the FeCo4Reco process.

Finally, the fifth contribution is a practical implementation of the proposed approach, with experimental results using a non-trivial cloud example, showcasing the interest and applicability of our theoretical contributions (1-4) through validation and practical implementation.

1.4 Tool Implementation

We have implemented FeCo4Reco which enables the generation of CBRTVMs from feature models, as well as the composition of multiple CBRTVMs. Leveraging the ATLAS Transformation Language (ATL), our implementation provides a model transformation process that takes feature models as input conforming to feature model metamodel and produces CBRTVM conforming to the JavaBIP metamodel. Moreover, we have implemented a Java-based macro composer that facilitates the composition of CBRTVMs using union, intersection, and strict intersection operators.

1.5 Dissertation Outline

The dissertation is divided into seven chapters. The second chapter covers the Background and concepts that will be used throughout the dissertation. The third chapter discusses the state of the art. The fourth and fifth chapters present the theoretical contributions of this dissertation, the sixth chapter is the validation of our proposal, and the last chapter includes the conclusions and perspectives. Below, we present an overview of the individual chapters.

Chapter 2

Background and Concepts: This chapter provides the background information and the concepts that have been used in our contributions presented later in this dissertation. It includes an overview of fundamental principles in cloud computing, software product line engineering, feature models, and the JavaBIP framework.

Chapter 3

State of Art: This chapter presents the State of the Art, where we review and analyze current approaches in the literature that tackle self-adaptation and reconfiguration. It provides a comprehensive survey of existing methodologies, tools, and frameworks used for managing dynamic reconfiguration. Additionally, it presents approaches that support software evolution through composition.

Chapter 4

Automatic Generation of Component-based Run-time Variability Models: This chapter presents the first and second contributions of this dissertation, addressing RQ1. It introduces model transformation rules that are general enough for both feature models and component-based models. These rules enable the generation of the Component-based Run-time Variability Model from feature models, allowing for the automated creation of CBRTVMs using the FeCo4Reco approach.

Chapter 5

Composing Run-time Variability Models: This chapter presents the third and fourth contributions, addressing RQ2 and RQ3. It introduces composition operators for CBRTVMs, enabling the automated construction of component-based systems in a modular fashion while providing reusability, flexibility, and adaptability (third contribution). It defines three composition operators on JavaBIP models and studies their properties. Additionally, the notion of a multi-step $\mathcal{UP}$-bisimulation is defined, and the correctness and compositionality of the encoding are proved (fourth contribution).

Chapter 6

Practical and Experimental Validation: This chapter describes the practical implementation of the proposed approach and reports on experimental results using a non-trivial cloud example. It includes a discussion on the validity of the approach, constituting the fifth practical contribution that showcases the interest and applicability of our approach. This chapter also presents the practical implementation and experimental validation of the composition operators defined in Chapter 5.

Chapter 7

Conclusion and Perspective: The final chapter concludes the dissertation and outlines future work directions.

1.6 List of Scientific Publications

Some of the presented material is based on the following publications:

[72] Farhat Salman, Bliudze Simon, Duchien Laurence, and Kouchnarenko Olga. "**Toward run-time coordination of reconfiguration requests in cloud computing systems.**" International Conference on Coordination Languages and Models. Cham: Springer Nature Switzerland, 2023.

[70] Farhat Salman, Bliudze Simon, and Duchien Laurence. "**Safe Dynamic Reconfiguration of Concurrent Component-based Applications.**" 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C). IEEE, 2022.

[71] Farhat Salman, Bliudze Simon, Duchien Laurence, and Kouchnarenko Olga. "**Run-time coordination of reconfiguration requests in cloud computing systems.**" Diss. Inria, 2023.

[119] Farhat Salman, Bliudze Simon, Duchien Laurence, and Kouchnarenko Olga. "**Composing Run-time Variability Models.**" The European Conference on Software Architecture (ECSA 2024), 2024. *Under review.*

Chapter 2

Background and Concepts

In this chapter, we present the foundational concepts that form the basis for the contributions introduced in the following chapters. We begin by defining the key characteristics, service models, and deployment models of cloud computing. Building upon this foundation, we then explore the principles and practices of software product line engineering (SPLE). We define the core elements of SPLE, such as variability and feature models, and describe the complementary processes of domain engineering and application engineering. Furthermore, we discuss the extension of traditional SPLE into the realm of dynamic software product lines (DSPLs), which enable runtime adaptation and reconfiguration. Finally, we introduce the JavaBIP model and formalize its operational semantics, providing a formal foundation for coordinating the behavior of concurrent software components.

The objective of this chapter is not to present an in-depth description of all the existing approaches and technologies surrounding these concerns, but to give a brief introduction to these concerns, used throughout the dissertation. This introduction aims to provide a better understanding of the background and context in which our work takes place, as well as the terminology and concepts presented in the next chapters.

2.1 Cloud Computing

Cloud computing refers to the on-demand delivery of computing services over the internet [138, 9]. Cloud computing is a concept that brings together several technologies used to deliver services. It aims to encourage companies to outsource the digital resources they store in their private data centers. These resources are then made available by third-party companies and accessed via the Internet.

The concept of cloud computing emerged at the beginning of the 2000s, and the changes that have led to its growth are numerous. First the rise of SaaS (Software as a Service), the product delivered by the cloud. Afterward, comes the concept of virtualization, which enables servers to be shared, facilitating production and increasing resource utilization. The concept of Cloud Computing was implemented in 2002 by Amazon, establishing Amazon Web Service, to web services to handle high traffic. Then, other companies such as Google and Microsoft have been offering similar services.

Several definitions have been proposed for cloud computing [136]. These definitions often highlight different aspects that are considered the main properties of cloud computing such as virtualization, pay-as-you-go, resource sharing, etc.

National Institute of Standards and Technology (NIST) defines cloud computing as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources. These resources include networks, servers, storage, applications, and services that can be rapidly provisioned and released with minimal management effort or service provider interaction [49].

Amazon Web Services (AWS) defines cloud computing as the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the internet with pay-as-you-go pricing [142].

Google Cloud (GCP) defines cloud computing as the delivery of computing services over the internet to offer faster innovation, flexible resources, and economies of scale. Users typically only pay for the cloud services they use, helping to lower operating costs, run infrastructure more efficiently, and scale as business needs change [27].

2.1.1 Cloud Computing Characteristics

Building upon these foundational definitions, cloud computing is distinguished by several key characteristics that define its value proposition and operational model [100]:

- On-demand self-service resource provisioning: Cloud computing enables users to provision and release computing resources, such as processing power, storage, and networking, without requiring human interaction with the cloud service provider [95].

- Broad network access: Cloud computing resources are accessible via standard networking protocols and can be reached through the Internet, allowing for widespread access [10, 87].

- Resource pooling: Cloud providers manage a shared pool of resources from which they allocate resources to consumers. While consumers cannot control the precise location of the resources, they may specify higher-level constraints, such as the data center or availability zone where the resources should be provisioned [117].

- Elasticity: Cloud computing allows for the rapid provisioning and release of resources, enabling users to scale their systems up or down based on demand [77, 8].

- Measured services: Resource usage is monitored, controlled, and recorded, providing both providers and consumers with insights into resource utilization. This information can be used to optimize resource allocation strategies and to charge users based on their actual resource consumption [60, 29].

2.1.2 Service Models

Computing resources of cloud computing are delivered as services, often referred to as XaaS (Anything as a Service) [64]. There are several known models, such as:

Infrastructure as a Service (IaaS): IaaS is the on-demand provision of infrastructure resources, most of which are located remotely in data centers. IaaS gives company administrators access to servers and their configurations [93]. The cost is directly linked to the occupancy rate. The advantage is that the customer has high flexibility, and total control of systems, enabling installation of all types of business software.

Platform as a Service (PaaS): PaaS is the on-demand access to a platform for developing, deploying, and managing applications without the complexity of building and maintaining the underlying infrastructure [127]. PaaS provides specialized development environments, including the necessary languages, tools and modules. The advantage is that these environments are hosted by a service provider based outside the company, which means that no infrastructure or maintenance staff are required, and you can focus on development.

Software as a Service (SaaS): provides access to fully developed and ready-to-use software applications over the internet [127]. Users simply use the software without needing to install, run and maintain applications locally.

There are more cloud service models, such as Storage as a Service [143], Database as a Service [75], Network as a Service [52], Function as a Service [124], and Security as a Service [137]. This versatility allows users to tailor their cloud solutions to meet diverse computing needs. The on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service that define cloud computing deliver significant flexibility, efficiency, and reliability benefits over traditional on-premises IT [95].

2.1.3 Deployment Models

There are four deployment models for accessing the services of a cloud computing environment:

- Public cloud: a service provider offering computing services offered by third-party vendors that the consumer can access and purchase the resource from the public cloud via the public internet. These can be free or on-demand, meaning that consumers pay for their CPU cycles, storage or bandwidth peruse.
- Private cloud: a proprietary data center that provides hosted services for a limited number of users.
- Community cloud: a cloud infrastructure provisioned for exclusive use by a specific community of users from different organizations who share common concerns, such as a collaborative mission, security requirements, or policy. The community cloud may be owned, managed, and operated by one or more organizations in the community, a third party, or a combination of them.
- Hybrid cloud: an infrastructure that contains links between a private cloud and a public cloud.

Now that we have introduced the cloud environment and its various services, it is clear that cloud computing offers a wide and numerous range of services. This implies that managing applications in the cloud can be a complex task, highlighting the need for tools to manage the inherent variability of cloud resources.

To address this requirement, the next subsection will introduce the concept of Software Product Line (SPL) engineering. SPL tools are recognized for their

capacity to represent the commonalities and variabilities of cloud services [39, 116, 111]. SPL promotes the development of a family of related software systems by identifying and exploiting the common aspects while encapsulating the variable elements. In the context of a Cloud platform, SPL can be particularly beneficial. Cloud environments are characterized by a multitude of services, configurations, and deployment options representing significant variability.

2.2 Software Product Lines Engineering

Traditional approaches to generating software products, such as the waterfall model [104], V-model [16], and incremental development [103], focus on constructing a single software system. These approaches employ multiple stages for the construction—requirements analysis, planning, implementation, testing, and deployment—each dedicated to distinct tasks and activities for constructing one software product. Software Product Lines (SPL) engineering, which will be discussed in more detail in this section and utilized later in our contributions, particularly in Chapter 4.

Unlike these traditional approaches, which focus on individual products, software product line engineering (SPLE) is a systematic approach for constructing a family of related software products [48, 98, 107, 132, 135, 145]. The essence of SPLE lies in its strategic concentration on establishing a set of software systems that share a managed set of features, seeking to fulfill the specific requirements of a particular market segment. This is achieved through the development of reusable core assets from which various products are generated via selective customization. An asset refers to any reusable resource that can be leveraged across different software products within a product line. These assets include software code, documentation, architectures, components, testing frameworks, etc [15, 48]. This enables the realization of a variety of distinct yet related products, essentially allowing for mass customization.

2.2.1 Variability

The variability of a software product line is reflected via the different ways in which the collection of reusable assets may be built to construct a software product. At the core of software product line engineering is variability management[109, 47, 14], which involves systematically modeling the commonalities and variabilities across the products in a family. Variability is commonly described through the concepts of variation points and variants. A variation point identifies a property that can differ among various products within the line, whereas a variant represents a specific option for that property at a given variation point.

2.2.2 Software Product Line Engineering Process

Software product line engineering involves two complementary processes: Domain Engineering and Application Engineering [140, 107]. The domain engineering process is responsible for creating reusable assets, while application engineering is the process of reusing those assets to build individual but similar software products.

These processes operate within two distinct spaces: the problem space and the solution space, as depicted in Figure 2.1.

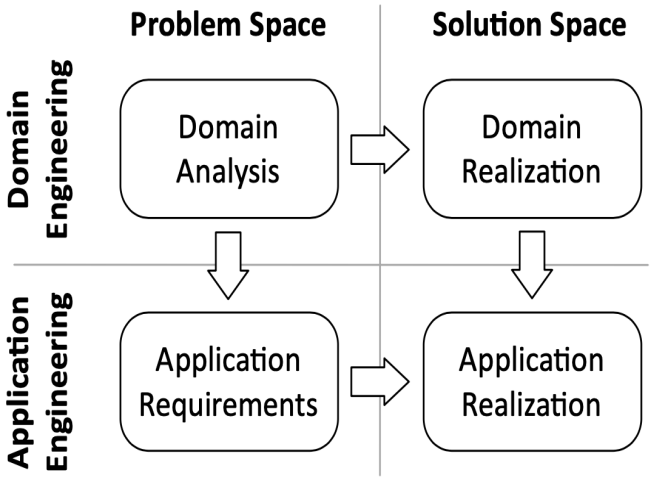

Figure 2.1: Domain and application engineering.

Domain Engineering

The domain engineering process is responsible for creating reusable assets that can be leveraged across multiple products within the same domain. In the problem space, it focuses on understanding and defining the variability requirements of the product line through Domain Analysis. This step involves identifying common and variable features, dependencies, and constraints across products, aiding in scoping the product line and comprehending the variability requirements. The solution space, on the other hand, is dedicated to realizing product line assets that address the identified variability requirements through Domain Realization, where reusable assets such as core components, libraries, and frameworks are designed and implemented.

Application Engineering

Application Engineering revolves around reusing the assets created during Domain Engineering to build individual but similar software products. This

process begins by identifying Application Requirements specific to each product and mapping them to the variability models and reusable assets. Subsequently, Application Realization involves configuring, composing, and assembling these assets to create the final product.

2.2.3 Variability Model

In the context of software product line engineering, a variability model [121, 46] serves as a systematic representation of the commonalities and variabilities across a family of related products within the same product line. It captures the aspects that remain consistent among different members of the product line (commonalities) and those that can be customized or varied to meet specific requirements (variabilities). The variability model provides a comprehensive overview of the potential variations in features, functionalities, or configurations that can be present in different instances of the software product line. It aids in managing and understanding the flexibility inherent in the product line, enabling efficient customization while maintaining a structured and organized development process.

There are several types of variability models used in software product line engineering. Feature models [82] represent variability using a tree-like structure of features and their relationships. Decision models [15, 121] capture variability as a set of decisions and their rationale. Orthogonal Variability Models (OVMs)[107, 109, 108] separate variability modeling from system artifacts using variation points and variants. Goal models [30, 11] represent variability in terms of stakeholder goals and their dependencies. Constraint-based models [80] express variability using logical constraints and rules. Each of these models has its strengths and suitability for different types of product lines and domains, and they may be combined to effectively capture and manage variability.

2.2.4 Feature Model

Feature models [82, 101, 134, 83] have become widely adopted for modeling variability in software product lines. A feature model is a compact representation of all the products within the SPL. Feature models are used to represent the commonality and variability of a domain in terms of features and their dependencies [25] and are visually represented by means of feature diagrams.

A feature was first defined as "user-visible aspects or characteristics of a domain" in [78]. Later, a feature is defined by [55] as a software artifact, such as part of code, a system component, etc. Feature models enable the effective management and organization of these features in a hierarchical structure.

Moreover, feature models can be transformed into propositional logic, allowing automated analysis and reasoning about the potential configurations

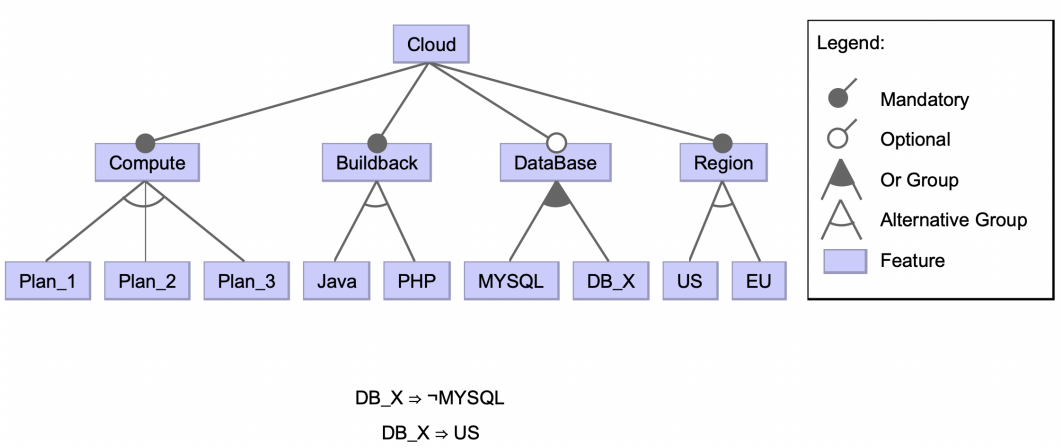

Figure 2.2: An example feature diagram.

within a software product line. This logical form enhances the usability of feature models. Specifically, the propositional logic evaluates to true when the configuration is valid, offering a solid foundation for evaluating the consistency of feature selections.

2.2.5 Feature Diagram

A feature diagram is a graphical representation of a feature model, visually depicting the hierarchical organization of features and their relationships. It provides an intuitive understandable view of the variability in a software product line. Figure 2.2 illustrates an example of a feature diagram. In this representation, features are graphically depicted as rectangles and arranged in a tree-like hierarchy. To express the variability, feature models employ several key concepts:

- **Mandatory and Optional Features**: Mandatory features are denoted by a black circle at the end of the connecting line, indicating that they are always included when their parent feature is selected. Optional features are denoted by an unfilled circle, indicating that they may or may not be included. In the example, `Cloud`, `Buildpack`, `Region`, and `Dyno` are mandatory features, while `Database` is an optional feature.

- **Feature groups**: Feature models support the concept of feature groups, which define the selection constraints among a set of sub-features. There are two types of feature groups:

  - **XOR-group**: In an XOR-group (also known as an alternative group), exactly one feature must be selected from the group whenever

the parent feature is selected. In the example, the sub-features of `Dyno (P1, P2,` and `P3)`, `Buildpack (Java` and `PHP),` and `Region (US` and `EU)` form an XOR-group, where exactly one of them must be selected.

– **OR-group**: In an OR-group, one or more features can be selected from the group whenever the parent feature is selected. In the example, the sub-features of `Databases` (`DB_X` and `MySQL`) form an OR-group, where either one or both features can be selected.

The combination of optional and mandatory sub-features can be seen as an AND-group, where all mandatory sub-features and any number of optional sub-features can be selected. For instance, under `Cloud`, `Region` and `Buildpack` and `Compute` are mandatory sub-features and `DataBase` is an optional feature, forming an AND-group.

- **Cross-tree constraints**: In addition to the main hierarchy, feature models allow the specification of cross-tree constraints to describe dependencies between arbitrary features. Two common types of cross-tree constraints are:

  – **Require**: If a feature A requires a feature B, the selection of feature A implies the selection of feature B. In the example, the selection of `MySQL` requires the selection of `US`.

  – **Exclude**: If a feature A excludes a feature B, the selection of feature A prohibits the selection of feature B, and vice versa. In the example, the selection of `MySQL` excludes the selection of `DB_X`.

2.2.6 Feature Model Logical Formula

The logical representation of the feature model's constraints is as follows:

$$
\begin{aligned}
&\texttt{Cloud} \land \texttt{Buildpack} \land \texttt{Region} \land \texttt{Dyno} \\
&\land (\texttt{JVM} \oplus \texttt{Gradle}) \land (\texttt{US} \oplus \texttt{EU}) \land (\texttt{P1} \oplus \texttt{P2} \oplus \texttt{P3}) \\
&\land (\texttt{Database} \lor \neg\texttt{Database}) \land ((\texttt{DB\_X} \lor \texttt{MySQL}) \Rightarrow \texttt{Databases}) \\
&\land (\texttt{DB\_X} \Rightarrow \texttt{US}) \land (\texttt{DB\_X} \Rightarrow \neg\texttt{MySQL}) \\
&\land (\texttt{MySQL} \Rightarrow \neg\texttt{DB\_X})
\end{aligned}
$$

This logical formula captures the constraints and relationships defined in the feature model, allowing for the calculation of valid configurations and ensuring the consistency of feature selections.

2.2.7 Configuration Validity and Extensions to Feature Modeling

The number of the potential configurations of the feature model represented in Figure 2.2, before applying constraints, is computed as $2 \times 2 \times 3 \times 3 = 36$, accounting for choices in Compute, Buildpack, Database, and Region features. However, the application of cross-tree constraints reduces the total number of valid configurations to 30. This reduction emphasizes the utility of valid configurations in software product line engineering:

- They guarantee that software products derived from the feature model are consistent and adhere to specified constraints.
- They facilitate automated analysis and configuration management, ensuring only feasible product variants are realized.

A valid configuration example is:

$$\Phi = \{\texttt{Cloud}, \texttt{Buildpack, JVM}, \texttt{Region, US}, \texttt{Dyno, P1}, \texttt{Databases, MySQL}\}$$

This example adheres to all model constraints, illustrating the practical application of the feature model in defining valid product configurations.

While traditional feature models provide a solid foundation for representing variability in software product lines, researchers have proposed various extensions to enhance their expressiveness and capabilities [22]. Notable extensions include cardinality-based feature models [96, 113] that allow specifying cardinality constraints on feature groups for fine-grained control over feature selection, and the integration of attributes and attribute constraints [57] for representing quantitative and qualitative properties associated with features. Further extensions have been proposed to incorporate feature interactions [58], dependencies across multiple product lines (multi-product lines) [123], and temporal constraints [130, 31] for modeling dynamic and evolving systems. These extensions broaden the applicability of feature models across diverse domains, addressing the limitations of traditional models and used in more complex use cases.

The extensions to basic feature models offer significant potential for supporting dynamic adaptations and validations of software artifacts at run-time. For instance, cardinality-based feature models facilitate the dynamic adjustment of the active feature set based on cardinality constraints. This is useful in scenarios where requirements fluctuate, such as handling varying workloads

or adapting to changing environmental conditions. Attributed feature models, on the other hand, allow monitoring and validating attribute values of active features to satisfy attribute constraints. For example, in a cloud application, resources like CPU and memory can be dynamically allocated based on attribute constraints defined in the feature model, maintaining the required performance and quality of service levels. Nonetheless, in the context of this dissertation, we employ a basic feature model without the aforementioned extensions to represent variability.

2.3 Dynamic Software Product Line

A systematic approach to developing reconfigurable or adaptable systems, based on principles from SPLE [79], is called Dynamic Software Product Lines (DSPLs). DSPL approach leverages SPLE concepts to support the development of such systems. Classical software product lines and dynamic software product lines share some commonalities but differ in several key aspects [79].

In classical SPLs, variability management describes the different possible systems or products that can be derived from the product line. Business scoping identifies the common market or domain for the set of products. On the other hand, in DSPLs, variability management describes the different adaptations or runtime configurations that the system can take. Instead of business scoping, adaptability scoping identifies the range of adaptation and variability that the DSPL supports.

In addition, In dynamic software product line engineering, the development process is split into two main phases: design-time and runtime. The design-time phase is akin to the domain engineering phase in traditional software product line engineering, where the adaptation scope and variability are explicitly defined. During the runtime phase, which is analogous to the application engineering phase in SPLE but with the added capability of dynamic reconfiguration based on contextual variations, the system exploits the previously defined variability to perform safe adaptations as the context changes.

DSPL for Self-adaptation

There is no unified approach for building self-adaptive systems using Dynamic Software Product Lines (DSPLs). Different methodologies often concentrate on distinct facets of adaptive systems and employ varying techniques to address their respective challenges. DSPLs are often used to build systems with bounded adaptivity[24], in which dynamic variations are foreseen at design time. Nevertheless, there are works in the literature that support open adaptivity [24] by dynamically evolving the variability at runtime [32, 17]. Realizing

self-adaptation and dynamic reconfiguration capabilities in a DSPL requires additional mechanisms and techniques beyond variability modeling. These may include techniques such as goal-based reasoning [12], policy-based adaptation [91], and the use of SAT/CSP solvers [96, 23] for reasoning about feature models and constraints at runtime. Goal-based reasoning frameworks allow the system to reason about high-level goals and select appropriate adaptations to achieve those goals. Policy-based adaptation employs predefined rules or policies to govern adaptation decisions based on monitored events or context changes. Furthermore, SAT/CSP solvers are adopted for analyzing feature models, feature model optimization, and ensuring that dynamic reconfigurations satisfy the defined variability constraints.

In the next Chapter 3, various approaches that build upon the dynamic software product line (DSPL) concepts to facilitate runtime reconfiguration and self-adaptation in software systems will be explored.

Next, we will present the JavaBIP Framework [28]. JavaBIP Framework allows developers to think on a higher level of abstraction and clearly separate the functional and coordination aspects of the system behavior. It allows the coordination of existing concurrent software components.

2.4 The JavaBIP Framework

JavaBIP [28] is a Java implementation of the BIP framework. Behavior-Interaction-Priority (BIP) [18] is a component-based framework for the design of correct-by-construction systems. By superimposing three layers: behavior, interaction, and priority, it provides a simple yet effective framework for managing concurrent components. In the simplified version, there is no Priority. Hence, we denote it as BI(P). For the coordination of concurrent components, we make use of JavaBIP [28]. Only Behaviour and Interaction layers defined in BIP are relevant to our work. The macros defined in JavaBIP will be important for our contributions in Chapters 4 and 5.

Component Behaviour

The first layer (Behavior) presents atomic components with fixed activities considered ports, which are pairwise distinct. The components are modeled as Finite State Machines (FSMs), which have a finite number of states and a finite number of transitions between them, where transitions are labeled with ports. Ports form the interface of a component and are used to define its interactions with other components. A component is a software object that encapsulates certain behaviors 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.

JavaBIP allows three types of ports: *enforceable*, *spontaneous*, and *internal*. Enforceable ports represent actions controlled by the JavaBIP engine. They can be *synchronised*, i.e., executed together atomically. Spontaneous transitions are used to take into account changes in the environment, and therefore, they are not announced to the engine but rather executed after the detection of events in the environment of the component. Finally, internal transitions allow behavior specifications to update their state on the basis of internal information— when enabled, they are executed immediately. Spontaneous and internal transitions cannot be used for synchronization with other components [28].

Require and Accept Macros

The second layer defines component coordination by means of interaction models, that is, sets of interactions. Interactions are sets of ports that define allowed synchronizations between components. It is 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 [28]. 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 *require* macro specifies ports required for synchronization with the given port. For example, "$C1.p$ **Requires** $C2.q, C3.r, C4.s$"[1] means that port `p` of component `C1` must be synchronized with the three ports: q, r, and s of components `C2`, `C3` and `C4`, respectively.

$$p \quad \textbf{Requires} \quad a, \qquad \text{which formally means} \qquad p \Rightarrow \bigwedge_{q \in a} q \tag{2.1}$$

, where a is a set of ports, and P, is the set of all ports of all the JavaBIP components in the system, such that $a \subseteq P$.

A port that has a "**Requires** *true*" can be executed as a singleton as explained in Section 2.4.

If port x has a "**Requires** *true*", it means x can be executed as a singleton. On the other hand, if x has a "**Requires** *true*", then x will never be executed.

The *accept* macro defines that if a port p participates in an interaction, it must be accepted by all the participating ports in the considering interaction.

$$p \quad \textbf{Accept} \quad a, \qquad \text{which formally means} \qquad p \Rightarrow \bigwedge_{\substack{q \in P \setminus a \\ q \neq p}} \overline{q} \tag{2.2}$$

[1] We use a notation that is slightly different from that in [28] without a change of meaning.

The *accept* 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.

Definition 2.4.1. *(JavaBIP Model) Let* $CM = (\mathtt{C}, \rho, \alpha)$ *be a JavaBIP model, where:*

- $\mathtt{C}$ *is the set of* components*,*
- ρ *set of the* requires *macros, and*
- α *set of the* accepts *macros.*

Definition 2.4.2. *(JavaBIP Operational Semantics) Let* $JB = (\mathtt{C}, \rho, \alpha)$ *be a JavaBIP model. 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 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 $(q, e_1, q_1), (q_1, e_2, q_2), \ldots, (q_{n-1}, e_n, q') \in \rightarrow$.

Example 2.4.1. *A JavaBIP model is illustrated in Figure 2.3 with three components:* `Worker 1`*,* `Worker 2`*, and* `Lock`*. Graphically, enforceable transitions are shown by solid black lines, and spontaneous transitions are shown by dashed*

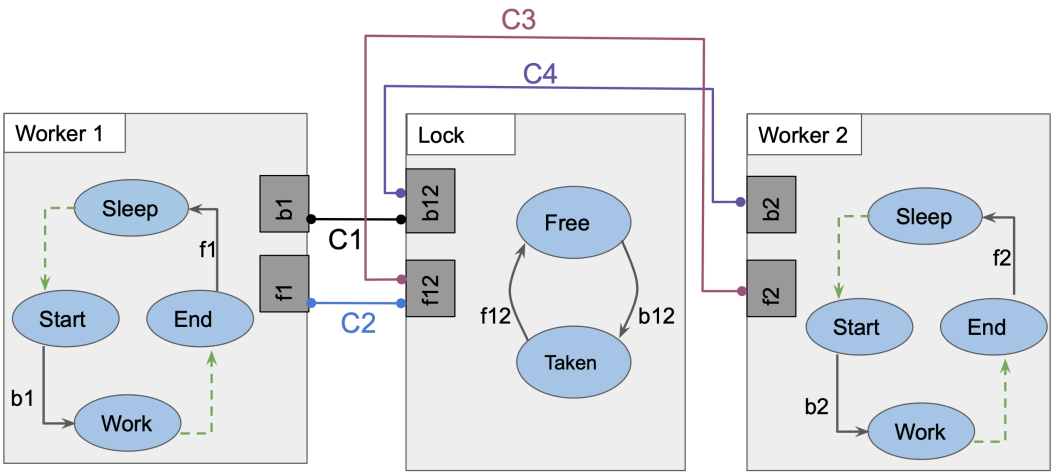

Figure 2.3: A JavaBIP component-based model.

green lines. Ports are shown as grey boxes on the sides of the components, and four connectors linking the ports define the possible interactions.

The green dashed transitions in the `Worker` *components are spontaneous transitions, and they are used to notify the components about environmental changes, i.e., when the user wants to activate a worker, it will inform the corresponding component, and the component will execute this spontaneous transition if it is in a state where this spontaneous transition is enabled.*

The model aims to ensure a safety property: preventing both workers from being in the state `Work` *simultaneously (mutual exclusion). To achieve this, the connectors are constructed as follows: the port* `b1` *of the* `Worker 1` *component can only be fired together with the port* `b12` *of the* `Lock` *component. Thus, once the* `Worker 1` *component has received the spontaneous activation notification from the state* `Sleep`*, it will move to the state* `Start`*. However, based on the connector* $C1$ *that connects* `b1` *with* `b12`*, the* `Worker 1` *and* `Lock` *components will move to the* `Work` *and* `Taken` *states, respectively. Now, assume* `Worker 2` *has received a spontaneous event and moves from state* `Sleep` *to state* `Start`*; it will not be possible to move from the* `Start` *state to the* `Work` *state, as the* `Lock` *component is in the state* `Taken`*. This means that the port* `b12` *is not enabled from this state, and by the connector* $C4$*, the port* `b2` *needs to be fired in synchronization with the port* `b12`*, which is not possible when the* `Lock` *component is in the* `Taken` *state. Thus, the mutual exclusion property is enforced, preventing both workers from being in the* `Work` *state simultaneously.*

Note that connectors in this example are binary, but connectors can define interactions involving more than two ports.

In this dissertation, we leverage the JavaBIP framework to generate formal component-based models, which are JavaBIP models that are correct-by-

construction to enforce domain constraints while executing the reconfiguration requests.

2.5 Summary

In this chapter, we have briefly introduced the principles and basic concepts that we will use throughout the dissertation. We began by explaining that cloud computing environments provide a wide array of configurable resources for applications, with both commonalities and variabilities present across these resources. To manage this variability in the configuration of cloud resources, we introduced feature models and explained their key role in the domain analysis phase of software product line engineering. We also discussed the extension of traditional software product lines into the realm of dynamic software product lines, which enable runtime adaptation and reconfiguration. Additionally, we presented the JavaBIP framework, detailing its key aspects

In the next chapter, we will present approaches proposed to address reconfiguration and self-adaptation and discuss their advantages and limitations.

Chapter 3

State of Art

Reconfiguration and self-adaptation are essential characteristics of modern software systems, particularly those deployed in dynamic contexts such as cloud platforms. These systems must be able to adapt to changing conditions, resource constraints, and evolving user requirements to keep compliant with user expectations.

This chapter aims to provide a comprehensive overview of the state of the art in self-adaptive systems and reconfiguration approaches proposed in the literature. We will explore various approaches, categorized into Dynamic Software Product Lines (DSPL) and formal component-based models, highlighting their key features and limitations. In addition, we survey approaches that tackle compositionality and composability, two closely related but distinct concepts in the context of system construction.

3.1 Reconfiguration and Self-Adaptation

Reconfiguration and self-adaptation of distributed software systems is a broad and active research topic [89, 53, 92, 102]. Modern software systems, especially those deployed in cloud environments, face the challenge of continuously adapting to changing requirements. This contrasts with traditional static systems that remain largely unchanged after the initial deployment phase. Reconfiguration and self-adaptation have emerged as two key, interrelated approaches for enabling software systems to modify their configuration in response to changing requirements.

Reconfiguration refers to the ability to modify the system configuration after it has been deployed. This may involve actions such as adding or removing components and altering the connections between components. Self-adaptation, on the other hand, is the capability of a system to autonomously monitor its own state and operating environment, detect changes, decide how to respond,

and then execute the necessary reconfiguration actions. The main difference between reconfiguration and self-adaptation lies in the autonomous triggering of the reconfiguration process. Reconfiguration is often driven by external actors, such as system administrators or DevOps developers. They actively monitor the system and make strategic decisions about when and how to adjust the system configuration. In this case, the reconfiguration process is started and managed by human operators. In contrast, self-adaptation is an inherent capability of the system itself. The system autonomously monitors its own state and operating environment, and then decides on and executes the necessary reconfiguration actions without direct human intervention.

Software systems deployed in dynamic environments such as cloud platforms must reconfigure or self-adapt in response to evolving requirements and environmental changes. Evolving requirements arise as user requirements evolve, necessitating software systems to adapt by incorporating new features or removing existing ones. While reconfiguration and self-adaptation offer significant benefits, they also introduce new challenges and complexities, such as ensuring system consistency, managing dependencies between components, and verifying the correctness and stability of reconfiguration actions.

After discussing the importance of reconfiguration and self-adaptation in distributed software systems it is important to understand the underlying principles and concepts that enable these capabilities. One widely adopted conceptual model for self-adaptation is the MAPE-K loop, which provides a systematic approach to achieving autonomic behavior in software systems.

3.2 MAPE-K Loop

The MAPE-K adaptation loop [118], also known as the control loop, is a fundamental concept in the field of autonomic computing. It provides a systematic approach to achieving adaptability in software systems. The MAPE-K loop was originally proposed by researchers at IBM in a paper on the architectural blueprint for autonomic computing [86]. The MAPE-K loop consists of four main phases: Monitor, Analyze, Plan, and Execute. These phases work together to enable a system to continuously monitor its own state and operating environment, detect deviations from desired behaviors, and trigger appropriate adaptation actions. The four phases can be described as follows:

1. Monitor: The process for gathering, aggregating, and filtering obtained data from a managed resource for tracking.

2. Analysis: The process to analyze and identify the current system state based on the information collected by the monitor process. If the system

is not compliant with the expected requirements and specifications, then an adaptation should occur.

3. Plan: The process of planning the target configuration. This involves defining a desired configuration in which the system behaves as expected.

4. Execute: The process of executing the reconfiguration plan to transition the system from its current configuration to the target configuration. At this stage, the managed system carries out the reconfiguration plan.

The K represents the knowledge step which is the knowledge base used by the system. This includes information about the system behaviour and contextual information.

The MAPE-K loop operates as a continuous cycle, where the Knowledge component is updated based on the results of the Execute phase, and the updated knowledge is then used by the Monitor, Analyze, and Plan phases in the next iteration of the loop. This closed-loop feedback mechanism enables the system to continuously adapt and evolve in response to changes in its environment and operational conditions.

3.3 Approaches for Reconfiguration and Self-Adaptation

This section presents an overview of various approaches proposed in the literature that tackle reconfiguration and self-adaptation in software systems. We survey the major approaches, categorized into Dynamic Software Product Line (DSPL) approaches and formal component-based approaches. For each category, we highlight the key features, strengths, and limitations of the respective approaches, providing a good understanding of the state of the art in this domain.

We present a selection of representative works from the DSPL and formal component-based approaches. The aim is to highlight the diversity of techniques and methodologies that have been proposed in the literature to tackle the challenges of runtime reconfiguration and self-adaptation.

For the DSPL-based approaches, we have selected works that illustrate the different ways variability modeling and DSPL concepts are used to enable runtime reconfiguration and self-adaptation. Some approaches employ variability models at design-time, while others leverage them at run-time. The works also showcase the use of different reasoning techniques, such as SAT/CSP solvers, to derive valid configurations during adaptation. Overall, the chosen DSPL approaches highlight the diversity in how the DSPL paradigm is extended and applied to facilitate self-adaptation across various domains and contexts.

The formal component-based approaches presented cover a diverse range of techniques, such as flat and hierarchical component models that cater to different architectural styles. Some approaches focus on only deployment and others on both deployment and reconfiguration. The selected works also exhibit varied execution semantics, including parallel execution of reconfiguration actions. Importantly, the formal foundations underlying these approaches draw from various mathematical domains, such as process algebras, architecture description languages, and automata-based formalisms, providing different reasoning capabilities for specifying, analyzing, and verifying self-adaptive and reconfigurable behaviors.

3.3.1 DSPL-based Approaches

As mentioned in Sect 2.3, DSPL techniques make use of variability models from SPLs to reflect the multiple run-time configurations a self-adaptive system might adopt. These variability models operate as a formalized definition of the system's adaptation space - they describe the set of configurations that are valid. By relying on such models during adaptation, DSPL procedures guarantee the system only reconfigures itself into safe states when adjusting to changes in its needs.

A fundamental feature of the DSPL paradigm is its separation between variability modeling in the problem space and the implementation of system artifacts in the solution space. As the adaptation needs change, the variability model can be updated independently, and new artifacts can be assembled together at run-time to realize the desired new system configuration. Due to the characteristics of explicit variability management and effective separation of concerns, DSPL-based techniques have been extensively investigated and used in the area of self-adaptive and reconfigurable systems throughout the last years.

Next, we present the DSPL approaches and compare them based on different criteria. Firstly, the variability modeling approach is used to model domain constraints and variability. Secondly, the reasoning and analysis techniques are employed to reason about and determine valid configurations, including SAT solvers and constraint solvers. Finally, adaptation execution and coordination, evaluate how each approach handles the execution of the reconfiguration process, such as parallel or linear execution, and what safety properties are tested during the execution phase.

Sossa et al. [129, 130] propose extending variability models with temporal constraints and reconfiguration operations to model the variability and adaptation lifecycle of cloud computing systems. The extended static variability model

with the temporal constraints is then used to generate a transition system [41] that takes into account the temporal constraints between the configurations (e.g. a database service can be upgraded but cannot be downgraded). In this system, states represent valid system configurations, and transitions signify changes from one state to another. Key advantages of this approach include explicitly modeling temporal adaptation constraints.

Morin et al.[99] introduce a DSPL approach for supporting dynamic adaptation. This approach involves monitoring for system and environmental changes to trigger the adaptation process. Goal-based reasoning evaluates these changes against system objectives, employing feature models extended with goals for decision-making. A Configuration Invariant Checker, employing Constraint Satisfaction Problems (CSP), validates modifications against the goal-oriented feature model, ensuring system requirements. A key advantage of this approach is the explicit consideration of system goals during the reconfiguration process. Furthermore, the use of CSP for configuration validation provides guarantees regarding the validity of the target configurations.

Acher et al.[3] explore the application of feature models for reconfiguring Dynamic Adaptive Systems (DAS), focusing on the dual variability of both the software and its operating environment within a Dynamic Software Product Line framework. The approach models the system and its environment as two distinct but interconnected SPLs, where dependency constraints facilitate adaptation in response to environmental changes. They support the analysis which is conducted through the evaluation of possible configurations and their impacts, using the information provided by the FMs. In addition, the planning involves determining the appropriate target configuration, guided by the dependency constraints between the system and environment FMs. This model-driven approach, similar to prior work [73], uses feature models instead of architecture/aspect models to capture variability, but still enables design-time validation of adaptation rules/dependencies and using feature models at run-time to reason about using SAT solvers and apply adaptations based on environment changes. A key advantage of the approach is the explicit modeling of the dual variability of both the software and its operating environment.

SALOON [112, 115, 116] Quinton et al. introduced SALOON, a software product lines-based approach that streamlines the selection and configuration of cloud environments for application deployment. By employing extended feature models with cardinalities and attributes [114], SALOON automates the identification of a suitable cloud environment that provides all the required functionalities to satisfy the application requirements, as well as the cloud services that meet those specific application requirements. This is achieved through a

Cloud Knowledge Model, which abstractly represents the functionalities across different cloud providers, enabling a high-level, requirement-driven selection process. Once an optimal cloud environment is chosen, SALOON generates executable configuration scripts, facilitating the automatic setup of the cloud environment according to the defined requirements. The key advantages of this approach are that it allows the automatic selection of a suitable cloud environment and enables the selection of cloud services that conform to the application requirements.

Pfannemüller et al. [105] proposed a Dynamic Software Product Line approach that uses context feature models to represent system variability incorporating context variability in the same model. The context feature model allows specifying constraints between context and system features to model how reconfigurations should occur based on different contexts. They transform the feature model into Boolean formulas in Conjunctive Normal Form (CNF), enabling SAT solvers to efficiently analyze and identify valid configurations at runtime, supporting dynamic adaptation based on the current context. The key advantage of this approach is that it provides a unified context feature model that captures both system capabilities and context information and leverages efficient SAT solvers on this unified model to reason on runtime reconfigurations, taking into account constraints between both system and context.

Cordy et al. [50] introduce adaptive featured transition systems (A-FTS) to model dynamically adaptive systems based on the feature model. This approach represents the system as a set of static programs (configurations) with transitions between them, indicating system adaptations in response to environmental changes. Rules/Policies determine how the system should react in different situations and how to adapt. The key advantage of the adaptive featured transition systems (A-FTS) approach is that it provides a formal and systematic way to model and reason about dynamically adaptive systems.

Cetina et al. [41] proposes a dynamic software product line approach for developing autonomic systems by reusing variability models at run-time. Variability models like feature models are used to capture the different configurations a system can take at design-time. At run-time, the same feature models are leveraged as the knowledge base driving autonomic capabilities like self-configuration. The approach involves monitoring contextual changes and translating those into activation/deactivation of features based on predefined resolutions (rules/policies).

Conclusion and Discussion The DSPL approaches uses variability models to model the variability of a domain such as feature models, but employ different

extensions to capture additional adaptation concerns. Sossa et al. [129, 130] extend variability models with temporal constraints to model the temporal dependencies between configurations. Acher et al.[3] model the dual variability of both the software system and its operating environment. Morin et al.[99] uses goal-based feature models. Pfannemüller et al. [105] propose context feature models that unify system capabilities and context information in the same model, allowing constraints between context and system features. SALOON [112, 115, 116] uses extended feature models with cardinalities and attributes to abstractly represent cloud environment functionalities. These extensions help account for aspects like temporal adaptation paths, environmental contexts, deployment contexts, and non-functional properties in the variability models. Cordy et al. [50] and Cetina et al. [41] use the basic feature model as a variability model.

Different techniques have been proposed for the sake of the analysis of these models at runtime. Sossa et al.[129, 130] and Cordy et al.[50] generate a transition system from the variability models and constraints to model the system behavior. Morin et al.[99] and SALOON[112, 115, 116] use Constraint Satisfaction Problems (CSP) over goal-oriented feature models. Acher et al.[3] and Pfannemüller et al.[105] translate feature models to Boolean constraints and leverage SAT solvers for reasoning. Finally, Cetina et al. [41] use rules and policies-based reasoning that are made at design-time.

Most of these DSPL approaches primarily focus on the planning phase of the MAPE-K adaptation loop, determining valid target configurations. However, they generally lack support for the execution phase, i.e., realizing the reconfiguration process to transition the system to the desired configuration. Additionally, while some approaches like Sossa et al.[129, 130] and Cordy et al.[50] transform feature models into transition systems, they don't provide support for the execution of the reconfiguration process as the transition models only show the transition between possible valid configurations with no explicit listing on the order of the execution of the reconfiguration process.

While the DSPL-based approaches offer strong mechanisms to manage reconfiguration via variability modeling, they have limitations in modeling how the reconfiguration process from the source configuration to the target configuration can be performed and in which order. The DSPL-based approaches can determine the validity of target configurations but do not model the coordination required during the reconfiguration process. To address this limitation, formal component-based approaches offer complementary benefits. Formal component-based models can capture the detailed deployment and reconfiguration lifecycles of individual system components. These formal foun-

dations enable rigorous reasoning about the reachability of target configurations, ensuring the correctness of the overall system during adaptation.

3.3.2 Formal Component-based Approaches

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.

To compare these formal component-based approaches, we consider three main criteria. First, the component specification, including how components and their behaviors are defined. Second, the support for deployment and reconfiguration, such as modeling the deployment and the reconfiguration process, and sequential and parallel execution of actions. Third, the formalism and correctness guarantees, encompass well-defined formal semantics, support for formal verification and analysis, and correctness guarantees for reachable states.

Aeolus [62, 61] is a formal component model designed specifically for distributed, cloud-based software systems. It allows declaring components with automata-based lifecycles, and dependencies. Aeolus components represent deployable resources of the cloud, where each component represents a software package, considered as a resource that provides and requires different functionalities, and may be created or destroyed [63]. The model enables reasoning about the achievability of target configurations and the automatic synthesis of deployment plans. Aeolus uses tools such as a planning tool [90] or Zephyrus tool [61]. Tools allow the user to compute a valid configuration satisfying a high-level specification [61]. These tools can compute valid deployment sequences that avoid conflicts and respect dependencies. The formal model Aeolus ensures deployment correctness. The key advantages of the Aeolus model are that it provides a formal yet expressive foundation for modeling the complex requirements of distributed cloud systems, and it has many tools integrated that use it as a model to reason about deployment plans.

Madeus [44] is a formal component-based deployment model designed for distributed software systems. Madues focuses on modeling and coordinating the deployment of distributed software systems. It was proposed as an extension of the Aeolus model. Madeus represents each software component by an internal Petri net-like structure to capture its detailed deployment lifecycle. Components expose ports and connections between ports establish dependencies. The key advantage of Madeus is the ability to express parallelism and coordination during deployment, going beyond sequential actions. The internal nets allow

concurrency within components. Dependencies enable parallel deployment of independent elements. This increases deployment speed and efficiency compared to sequential models. The key advantage is the ability to express parallelism and coordination during deployment, going beyond sequential actions.

Concerto [45, 43] is a formal model designed for efficient reconfiguration of component-based distributed systems. It represents software components via customizable state machines that declare lifecycle and behaviors. Components expose ports that enable coordination. Concerto allows parallel execution of reconfiguration actions both within and across components. Dependencies between ports synchronize component evolutions. The key advantage of Concerto is the ability to maximize parallelism during reconfigurations through fine-grained modeling. Concerto optimizes parallelism while ensuring correctness through its formal semantics. Components in Concerto react asynchronously to behavior change requests. The model coordinates component evolutions based on port connections and usage. The key advantage is the ability to maximize parallelism during reconfigurations on the internal behaviour of the components and between components.

Fractal [34, 35, 126] is a hierarchical component model for constructing configurable software systems. Fractal allows the construction of component architectures where components can be recursively nested. Each Fractal component has a membrane that contains controller objects that implement reconfiguration capabilities through the use of its reflective API. This API allows developers to programmatically manipulate the component architecture. For example, to add a component, a developer might use the API to instantiate the component, configure it, and then insert it into the component hierarchy. These controllers can adapt a component's sub-components, bindings and lifecycle, enabling adaptation. The key advantage is the use of membranes with controller objects that implement reconfiguration capabilities through a reflective API

TOSCA [26, 131, 20] (Topology and Orchestration Specification for Cloud Applications) aims to make deploying and managing cloud-based applications portable and automated. TOSCA allows describing multi-component application architectures and their management procedures in a standardized format. The core idea is to model an application as a topology graph of components represented as nodes, connected by relationships. The TOSCA model is packaged together with the artifact files needed to implement the application, such as VM images, software packages, and scripts. A TOSCA runtime like OpenTOSCA [26] processes this model and the artifacts bundle (called CSAR) to deploy the modeled application topology and manage it. OpenTOSCA first parses the model to create node and relationship instances, then invokes their

operations and artifacts, such as running VM install scripts, assigning IPs, and configuring ports. Pre-defined plans can further orchestrate these operations to execute lifecycle stages like deploy, scale, or upgrade. The key advantage is making deploying and managing cloud-based applications automated across different cloud environments.

SOFA [36, 106] SOFA (SOFtware Appliances) is a formal component-based model that provides a structured way to represent and manage software components within a hierarchical architecture. Components are defined as templates with well-defined interfaces, implementation artifacts, and associated management procedures, which can be nested to reflect the relationships and dependencies between different parts of the software. Connections among interfaces are represented by bindings, and physical interconnections are made through connectors, which can bind more than two interfaces. Each component has interfaces, controllers, and an internal architecture employing microcomponents - objects with various functionality. Invocations on component interfaces are passed through the appropriate microcomponent chains and then to the component's subcomponents. Components able to create other components at run-time are marked as factories, and the produced components are described as dynamic components. The key advantage is providing a structured and systematic way to represent and manage software components within a hierarchical architecture.

Dr-BIP [67, 68] El Ballouli et al. introduces the Dr-BIP (Dynamic Reconfigurable BIP) framework, which extends the BIP (Behavior, Interaction, Priority) framework [18] for modeling self-configuring systems. This model-based approach, combined with a component and connector architectural style, provides a formal and structured way to describe dynamic changes within a system. The use of architectural motifs in Dr-BIP serves as the basis for structuring the system and coordinating its reconfiguration at run-time. These motifs define sets of components that evolve according to specific interaction and reconfiguration rules, allowing the system to adapt dynamically to changes. The key advantage is using architectural motifs to provide a formal and structured way to describe dynamic changes and coordinate the reconfiguration of self-configuring systems.

Conclusion and Discussion Component-based approaches employ different formalisms to model component behaviors and interactions. Aeolus [62, 61] and Dr-BIP [67, 68] use standard automata, TOSCA [26, 131, 20] represents application architectures as topology graphs of nodes (components) connected by relationships, Madeus [44] adopts Petri nets, Concerto [45, 43] employs a modified form of Petri nets, Fractal [34, 35, 126] has hierarchical components

with membranes and controllers, and SOFA [36, 106] represents components as hierarchical templates with interfaces and artifacts. Regarding deployment and reconfiguration support, Madeus [44] solely enables deployment in a parallel manner, Aeolus [62, 61] supports both reconfiguration and deployment but relies on external tools for reconfiguration, Concerto [45, 43] tailors its formalism for efficient parallel reconfiguration across and within components, Fractal [34, 35, 126] supports dynamic architectures through its API, TOSCA [26, 131, 20] provides standardized workflows for cloud application deployment and management, and SOFA [36, 106] allows dynamic component creation through factories. All these component models are manually constructed, an error-prone and complex task, especially for complex systems. The manual effort required to accurately define components, constraints, and dependencies can lead to potential errors in the formal models, particularly for complex systems.

Formal component-based approaches offer rigorous semantics for specifying component behaviors, enabling analysis and verification of reconfiguration actions. These models facilitate the automated execution of reconfigurations, ensuring correctness and consistency, if the formal model was constructed correctly. The key advantages of these approaches are the ability to synthesize deployment plans by construction, express parallelism and coordination during deployment, enable parallel execution of reconfiguration actions, and provide a structured way to represent and manage software components within a hierarchical architecture. However, constructing a model relies on human expertise to define the components and constraints, which can be an error-prone task, especially for complex systems such as cloud applications.

3.4 Compositionality and Composability

The development of complex, component-based software systems and systems of systems has become increasingly important in modern software engineering. These systems are often composed of multiple subsystems that need to be integrated and coordinated to achieve the desired functionality. In this context, the concepts of compositionality and composability are important for ensuring the correctness, flexibility, and adaptability of the overall system.

3.4.1 Introduction to Compositionality and Composability

Compositionality and composability are two related but distinct concepts in the context of component-based software engineering and system construction [125].

Approach\keys	***Monitor***	***Analysis***	***Plan***	***Execute***
DSPL Approaches				
Sossa [130]			x	
Morin et al. [99]	-	x	x	
Acher et al. [3]		x	x	
SALOON [116]		x	x	
Pfannemüller et al. [105]		x	x	
Cordy et al. [50]	-	-	x	
Cetina et al. [41]	-	-	x	
Component-based Approaches				
Fractal [33]			-	x
Aeolus [62]			-	x
Madues [44]				x
Concerto [45]			-	x
TOSCA [26]			-	x
SOFA [36]			-	x
Dr-BIP [67, 68]		-	-	x

Table 3.1: MAPE-K loop table. A dash (–) in the table signifies that the corresponding phase is supported manually by developers or through tools made up at the design time.

Compositionality is concerned with the preservation of component properties when they are integrated into a larger system. If a system exhibits compositionality, it means that the overall system behavior can be deduced from the behaviors of its constituent components and their composition rules [125].

The reason why compositionality is an important notion in the context of software development and model integration is that it permits modular reasoning about system behavior. As software systems expand and new components are added, compositionality guarantees that the properties of the current components are kept, and the behavior of the entire system can be expected based on the composition of its parts. This is essential to controlling the complexity of growing systems and maintaining their validity when new features are implemented.

On the other hand, composability is the ability to combine components in a meaningful way, such that the resulting system exhibits the desired behavior and properties [74, 125]. Composability is important for enabling the integration of new functionalities into existing software systems, as it provides a way to incorporate new components without disrupting the overall system behavior. This is a key requirement for supporting the evolution of software systems over

time, as new features and capabilities need to be added while preserving the existing functionality.

3.4.2 Composition Approaches for Software Models

In component-based software engineering, compositionality and composability are important concepts because they allow complex systems to be built from separate components while maintaining desired properties and behaviours. In this context, various approaches have been proposed to support the composition of software systems.

Composition of Feature Models

We will present the work upon which we build and extend, focusing on compositional aspects. Earlier work by Acher et al. [4, 6] and Carbonnel et al. [38] proposed techniques for composing FMs using predefined operators such as union and intersection. To respect the semantics of these operators, specific rules for merging common features during composition were defined. Building on this work, Acher et al. [5, 7] introduced more advanced techniques to enable composing FMs under arbitrary user-defined operators. One approach encodes input FMs as Boolean formulas [19, 59] 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, we contribute with equivalent composition operators designed for composing run-time JavaBIP models.

In Featured Transitions Systems (FTS) [51], 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 [65] 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.

Composition of Aspect Models

An early description of the distinction between positive and negative variability can be found in [139], where authors combine model-driven and aspect-oriented software development to support both variability types. In [139] features are separated in models and composed by aspect-oriented composition techniques on model level. Positive variability refers to the ability to add new functionality or features to a system, while negative variability refers to the ability to remove or disable existing functionality. Aspect-oriented techniques enable the explicit expression and modularization of variability on model, code, and template levels. Differently from [139], our approach is only model-driven when supporting composition and reconfiguration to allow both variability types.

Composition of Component-based Models

In the domain of component-based models, a recent survey [54] 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 the development and management stages (cf. [54], 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. [13] propose a formal framework for the 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, [13] shows that their intersection composition $A_1 \cap A_2$ enforces the conjunction $\phi_1 \wedge \phi_2$. Our approach to CBRTVMs composition also aims to facilitate incremental 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, including union, and strict intersection.

3.5 Conclusion

In this chapter, we provided an overview of the approaches proposed in the literature to tackle reconfiguration and self-adaptation. These approaches were categorized into dynamic software product lines approaches and formal component-based approaches. As motivated by [41, 66], who emphasize the necessity of integrating different approaches to achieving effective reconfiguration,

our approach uses software product line tools and component-based models together to achieve reconfiguration.

Dynamic software product line approaches, leveraging the principles of software product line engineering, focus on managing variability at run-time to support dynamic adaptation. These approaches are effective for efficiently managing a large space of potential configurations. The dynamic software product line approach presented either computes the set of all possible configurations in advance at design time, or a new valid configuration at run-time. 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, but particularly so for distributed systems, where a copy of the list has to be stored at every node. Alternatively, at run-time, the new configuration must be computed and validated thus inducing a computational overhead.

Most dynamic software product line approaches presented in the related work do not state their respective time overheads. However, results from [129, 130, 21] show that reasoning on FM that only validates configuration compliance with the FM takes around 10^3 ms (or more) for 300 features. In terms of memory, storing all valid configurations for a feature model with n features would require an amount of memory on the order of 2^{n-1} bytes. For example, storing configurations for a feature model with 300 features would require memory on the order of 10^{88} bytes, which is impractical. Storing all valid configurations is only feasible for small feature models with a limited number of features.

Furthermore, the dynamic software product line approaches, as illustrated in Table 3.1, do not tackle the execution phase. This limitation is expected, as these feature models are only employed at run-time to validate or extract a valid target configuration, without specifying how to transition to that target configuration. Therefore, there is a need to extend these approaches not only to reasoning on the validity of the target configuration but also to the execution path required to reach it.

The presented formal component-based approaches, offer rigorous semantics for specifying component behaviors and interactions, enabling analysis and verification of reconfiguration actions. These models facilitate the automated execution of reconfigurations, ensuring correctness and consistency, if the formal model was constructed correctly. However, constructing a model relies on human expertise to define the components and constraints which is an error-prone task, especially for complex systems such as cloud applications.

In contrast to both approaches, we introduce an approach that lies in the generation of a correct-by-construction Component-based Run-time Variability Model (CBRTVM) that enforces domain constraints for dynamic reconfiguration while leveraging software product line engineering tools to capture domain variability. Unlike many works using software product line techniques and tools, our approach goes beyond employing static variability models at run-time, e.g., [69], and avoids the overhead of computing or validating new target configurations explicitly.

To summarize, we propose an approach that leverages both software product line engineering tools and formal component-based models to facilitate safe dynamic reconfiguration. The use of software product line tools is to benefit from the domain variability and automatically generate a component-based run-time variability model. The generated CBRTVM in JavaBIP introduces monitoring and controlling of the application behavior by dealing with reconfiguration requests while ensuring the (partial) validity of all reachable configurations.

We also presented approaches that tackle compositionality and composability. To our knowledge, the closest work done in this domain to compose two models is by Attie et al. [13] in the context of component-based systems. We are not aware of other approaches that perform composition in the same manner as our proposed method.

Chapter 4

Automatic Generation of Component-based Run-time Variability Models

In this chapter, we propose Feco4Reco [72, 70] an approach for deriving executable component-based run-time variability models (CBRTVM) from feature models. The goal is to tackle the research question RQ1: How to enforce domain constraints during dynamic reconfiguration at low cost?

Our approach leverages principles from software product lines (SPL) and formal component-based models to enforce domain constraints during safe dynamic reconfiguration. Specifically, we use feature models from SPL to compactly represent the set of valid configurations by capturing domain constraints. We then provide automated model transformation rules to derive executable CBRTVMs from these feature models.

The generated CBRTVM encodes all the constraints and dependencies specified in the feature model, enabling non-expert developers to safely apply reconfigurations. 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. By leveraging the CBRTVM, developers without expertise in the hosting context can perform reconfigurations while adhering to the constraints and dependencies inherent to the platform, as these are already enforced within the generated model.

4.1 Background

In this section, we introduce the formalization of the models that form the foundation for our contributions. We introduce a formal notation for feature models and their semantics in terms of valid configurations. To facilitate incre-

mental system design and development, we introduce the notion of saturated partial-valid configurations, which ensure consistency and well-formedness of a configuration at an intermediate step through reconfiguration.

4.1.1 Feature Model Formalization

Definition 4.1.1. *(Feature Diagram) 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 $Node$:*

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

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 4.1.2. *(Feature Model) 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]

4.1.2 Feature Model Notation

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 $n = AND(\ldots, \textbf{mand}\ n', \ldots)$, we write $mand(n')$.

Definition 4.1.3. *(Feature Model Dependency Graph) 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 directed edges representing the parent, mandatory and requires relations:*

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

The FM semantics is the set of its valid configurations [122].

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.

[1] In Fig. 4.1, 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 4.1.4. *(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, *which means that for any $f \in \Phi$, it holds that $E(f) \subseteq \Phi$;*
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 the respect of AND-node constraints);*
4. partial-valid *if there exists a valid configuration $\Phi' \supseteq \Phi$.*

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

4.2 Motivation

Cloud platforms like Heroku exemplify the key characteristics of cloud computing described in Chapter 2. Heroku enables developers to build, run, and scale applications without managing the underlying infrastructure. Heroku relies on the infrastructure provided by cloud service providers such as Amazon Web Services (AWS) [142] and Google Cloud Platform (GCP) [27] to host and run applications deployed on its platform. Applications on Heroku are hosted in virtualized containers and virtual servers that are provisioned on top of the infrastructure provided by these cloud providers.

Heroku provides a flexible cloud platform-as-a-service (PaaS) for deploying and running applications. The platform comprises essential elements such as Dynos (lightweight Linux containers), geographic regions for hosting applications closer to users, buildpacks for automating application setup, and a rich ecosystem of add-on services that can be easily integrated with applications [76, 85].

The deployment and reconfiguration of applications in the Heroku cloud environment are governed by specific constraints and dependencies. For instance,

certain add-on services may be exclusive to specific regions, and Heroku only permits hosting resources in a single region simultaneously. Additionally, numerous other constraints and dependencies exist between services in the cloud environment. Consequently, reconfiguration must be performed in a manner that respects and adheres to these interdependencies.

To ensure a safe reconfiguration, a systematic approach is required to identify and follow a valid reconfiguration path that respects the constraints and dependencies.

4.2.1 Key Elements of the Heroku Cloud Platform

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 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 dynos, regions, buildpacks, and add-ons.

- Dynos: Heroku applications run on lightweight, isolated Linux containers called Dynos. These containers provide a secure and scalable runtime environment for applications. Dynos come in different sizes and configurations, each with its own set of resources (CPU, memory, and storage). Developers can choose from various Dyno types, such as Free, Hobby, and Production tier, depending on their application requirements and budget.

- Regions: Heroku offers a global infrastructure with 16 geographic regions spread across the world. This allows developers to deploy their applications in locations that are closest to their target clients, minimizing latency and ensuring optimal performance.

- Buildpacks: Heroku uses buildpacks to automate the process of compiling and configuring application source code into executable slugs that run on Dynos. Buildpacks are pre-configured scripts that detect the programming language and framework used by an application and set up the necessary dependencies and runtime environment. Heroku supports a wide range of buildpacks for popular languages such as Ruby, Node.js, Java, Python, and Go.

- Add-ons: One of the key strengths of the Heroku platform is its extensive ecosystem of add-ons. Heroku offers over 150 add-on services that can

be easily integrated into applications to extend their functionality and capabilities. These add-ons cover a wide range of categories, including databases (e.g., PostgreSQL, MongoDB), monitoring and logging (e.g., New Relic, Papertrail), application performance management (e.g., Scout, Librato), security (e.g., Okta, Sqreen), messaging (e.g., SendGrid, Twilio), search (e.g., Algolia, Elasticsearch), and analytics (e.g., Mixpanel, Segment). In addition to the pre-built add-ons available in the Heroku marketplace, Customers can also create their own custom add-ons and publish them in the Heroku Elements marketplace. This empowers developers and businesses to build and share specialized services.

The deployment and reconfiguration of applications in the Heroku cloud environment are subject to specific constraints. For example:

1. **Dependency on Region Availability:** Certain add-on services, such as $Guru301$, are exclusively available in specific regions, e.g., the US region. Thus, deallocating resources from the US region can disrupt service dependencies and lead to failures.

2. **Exclusive Hosting Region:** Heroku allows hosting resources in only one region at a time. When migrating an application from one region to another, it is essential to consider that resources cannot be simultaneously deployed in both regions. Initiating allocation of the EU region while resources remain deployed in the US region, for instance, will result in a failure.

4.2.1.1 Valid Reconfiguration Path

Suppose there is an application running on a Hobby dyno within the US region, and it is utilizing the Guru301 add-on service, which is exclusive to US regions. If there is a need to migrate this application to the EU region and detach the Guru301 add-on from the application, a reconfiguration process becomes necessary. For migration, there is a need to allocate resources in the EU region and deallocate them from the US region, and simply detach the Guru301 from the application. The desired new configuration Φ' would be $\Phi' = \{Hobby, EU\}$, while the current configuration Φ is $\Phi = \{Hobby, US, Guru301\}$.

To achieve a safe reconfiguration to the desired target configuration of $Hobby$ dyno in the EU region, three actions must be taken:

1. Allocate the resources in the EU region.

2. Deallocate the resources from the US region.

Table 4.1: Possible paths for performing reconfigurations.

Path	Interaction 1	Interaction 2	Interaction 3	Validity
Path 1	Allocate EU	Deallocate US	Deactivate Guru301	**Invalid**
Path 2	Allocate EU	Deactivate Guru301	Deactivate us	**Invalid**
Path 3	Deactivate us	Deactivate Guru301	Allocate EU	**Invalid**
Path 4	Deactivate us	Allocate EU	Deactivate Guru301	**Invalid**
Path 5	Deactivate Guru301	Allocate EU	Deallocate US	**Invalid**
Path 6	Deactivate Guru301	Deallocate US	Allocate 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.

3. Deactivate the *Guru*301 add-on service.

There are six possible paths to reach this desired target configuration from the current configuration Φ=$\{Hobby, US, Guru301)$ as presented in Table 4.1. Each of these paths involves a sequence of activation and deactivation steps, and the choice of path may have implications for dependencies and constraints between resources. A naive approach of simply deactivating the US region first would result in an error. This is because *Guru*301 depends on US region availability, so deallocation of the rescources from the US region would break this dependency and cause *Guru*301 to fail. Similarly, attempting to allocate the resources to the EU region while resources are still deployed in US would also fail. Heroku only allows resources to be hosted in one region at a time, so resources in the EU cannot be allocated simultaneously with US. Thus, not all the paths can be taken to reach the target configuration.

The only valid reconfiguration path as presented in Table 4.1 is:

1. First, deactivate the *Guru*301 add-on while keeping the resources in the US region. This removes the dependency on the US-only *Guru*301 service.

2. Next, the resources deployed in the US region can be safely deallocated, as the *Guru*301 service is already deactivated and no other service is reliant on it.

3. Finally, the resources can be allocated in the EU region to complete the migration to the desired target configuration $\Phi = \{Hobby, EU\}$

The proposed ordering adheres to the constraints and dependency management requirements, ensuring that reconfiguration actions are executed without

violating any constraints. Failure to comply with these constraints can lead to disruptions in application continuity. Consequently, a systematic coordination approach is necessary to facilitate seamless reconfiguration.

Although Path 6 is the only valid reconfiguration path in this specific example, it is important to note that in other cases, there could be several potential paths for reconfiguring from the source to the target configurations while still adhering to the cloud constraints. This highlights the importance of adopting an approach to the reconfiguration process, ensuring that the path chosen for the reconfiguration of the system respects cloud constraints. Furthermore, our proposed model does not restrict or invalidate any of these valid paths. Instead, it allows for the execution of any valid reconfiguration path.

4.3 Feco4Reco: A Framework

We propose Feco4Reco [72, 70] 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 as discussed in Chapter 3.

The FeCo4Reco process, shown in Fig 1.1, consists of three stages: 1) domain constraints are specified as a feature model, 2) the feature model is automatically transformed into a Component-based Run-time Variability Model (CBRTVM) to make it run alongside the system, 3) the generated CBRTVM 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 runtime.

4.3.1 Stage 1: Heroku Cloud Feature Model

The Heroku cloud feature model captures the configurable service options and dependencies in the Heroku platform. It includes the mandatory features *Dyno*, *Region*, and *Buildpack*, representing core aspects of deploying applications. The optional features *Add_ons* allow attaching additional services to extend functionality, such as managed databases, monitoring, messaging, queuing, and security services.

As shown in Fig. 4.1, the feature diagram has a tree-like structure with *Heroku_Application* as the root feature, presenting the Heroku cloud with services and constraints. In addition to mandatory features, optional features

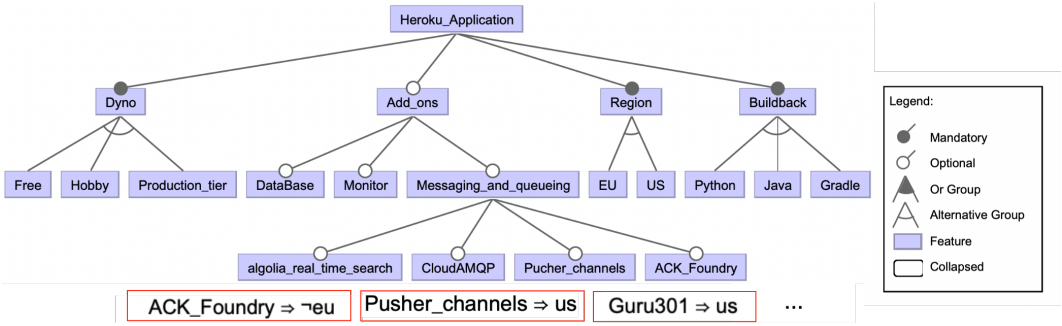

Figure 4.1: Part of the Heroku cloud feature model.

like Heroku add-ons, maintained by third-party providers or Heroku, are available and can be installed onto applications using the Heroku service API interface. Other constraints to consider include regional availability of services, inter-service dependencies, and architectural constraints. Consequently, developers require expertise in Heroku to manage and control applications safely while accounting for all constraints on the hosting context.

Example 4.3.1. *Figure 4.2 presents a highlighted valid product configuration* $\Phi = \{Heroku_Application, Free\ Dyno, EU\ Region, Java\ JVM\}$ *within the Heroku cloud feature model. This configuration represents a specific selection of Heroku platform features that can be used to deploy an application, and it satisfies the criteria for a valid configuration as defined in Def 4.1.4. The green rectangle encompasses the chosen features, which include the base* $Heroku\_Application$ *feature, the* $Free\ Dyno$*,* $EU\ Region$ *and* $Java\ JVM$ *for the buildpack of the application. This configuration indicates that the selected Heroku cloud product will have a* $Free$ *dyno type deployed in the* EU *region with a* $Java\ JVM$ *buildpack. The feature model visualization effectively captures the valid combinations of features and constraints, allowing users or administrators to explore and select desired configurations for their Heroku-based applications.*

In our approach, the feature model is constructed at design time by experienced developers familiar with the target cloud platform. This one-time effort ensures that the feature model accurately captures the configurable service options, dependencies, and constraints of the underlying cloud platform. The feature model is realized based on the specific constraints and capabilities of the cloud platform, and it will be transformed into a Component-Based Run-Time Variability Model (CBRTVM) as detailed in Stage 2. While the example presented uses the Heroku cloud feature model, our approach is not limited to any particular cloud platform and can be applied to other cloud environments as well.

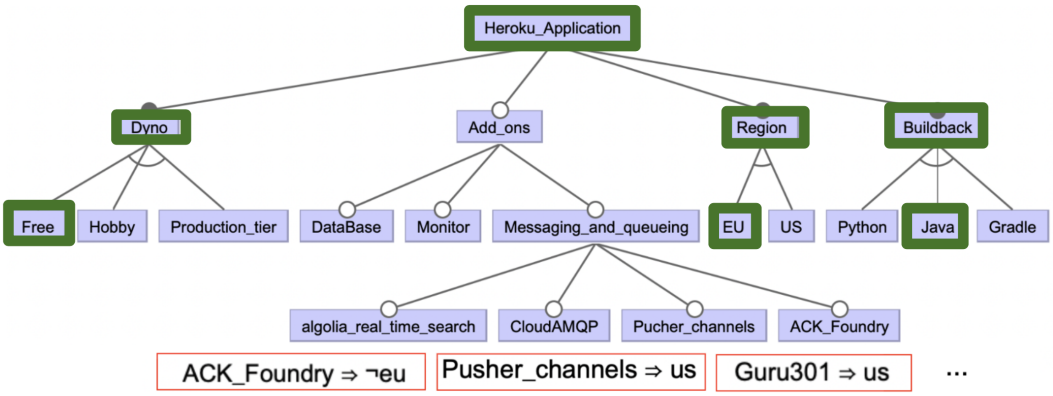

Figure 4.2: Valid product selection.

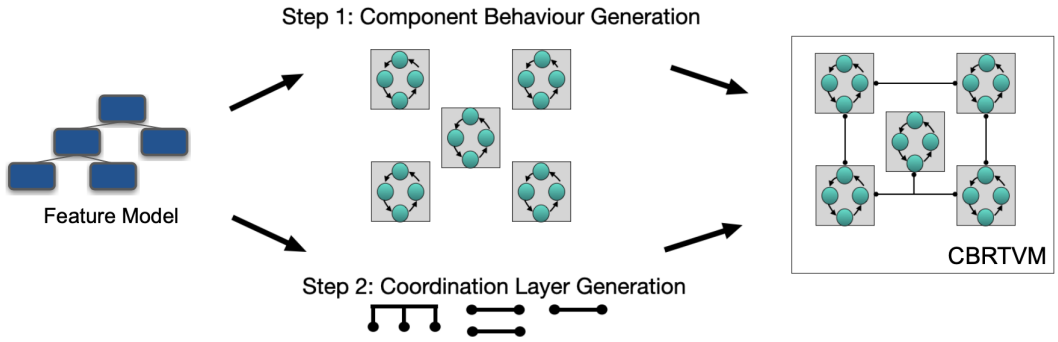

Figure 4.3: The generation of the component-based run-time variability model involves a two-step transformation process. Steps 1 and 2 correspond to Subsections 4.3.2.1 and 4.3.2.2, respectively.

4.3.2 Stage 2: Transformation: Feature Model to Component-based Run-time Variability Model

This section describes a set of design rules for automatically generating a component-based run-time variability model using a feature model as input. Figure 4.3 presents the steps for the transformation of the encoding process. The encoding of the feature model into a CBRTVM is achieved recursively. Initiated by the *root* node of the feature model, each feature triggers the generation of a corresponding component along with its associated behavior. Consequently, for every feature in the model, a component is generated. Following the generation of components, the coordination layer is constructed in accordance with the constraints specified by the feature model.

4.3.2.1 Step 1: From Features to Components

To start, we initiate the process by establishing a mapping between features and components. Building upon the concepts discussed in Sect. 4.1, we undertake the transformation of a given feature into a corresponding 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 the *root* whose nodes correspond to features, we define a function $\kappa : Node \rightarrow 2^{Comp}$ by:

$$\kappa(n_f) = \begin{cases} \{enc(n_f)\}, & \text{if } n_f = leaf \\ \bigcup_{i=1}^{k} \kappa(Node_i) \cup enc(n_f), & \text{if } n_f = OR(Node_1, ..., Node_k) \vee \\ & n_f = XOR(Node_1, ..., Node_k) \vee \\ & n_f = AND([opt]Node_1, ..., [opt]Node_k) \end{cases} \tag{4.1}$$

This recursive algorithm (Algo 1), captured by the function $\kappa(root)$, is designed to guide the systematic process of generating components necessary for the main *root* node and its descendant nodes within a hierarchical structure. The algorithm initiates at the *root* node and then proceeds into a recursive exploration through all subsequent sub-nodes until it ultimately reaches the leaf nodes. The algorithm operates in two modes: leaf node handling and compound feature handling. For leaf nodes containing a singular feature like `f`, the algorithm employs the $enc(n_f)$ encoding operation to create a corresponding component labeled `f`. When encountering compound features, the same encoding operation generates a component labeled `f`. Additionally, the algorithm invokes itself, κ, for all sub-nodes within the compound feature, recursively generating components until leaf nodes are reached.

Algorithm 1 Function $\kappa(n_f)$

```
1: function κ(n_f)
Require: n_f: Node type
Ensure: Components generated for n_f and its descendants
2:     if n_f is a leaf node then
3:         return enc(n_f)
4:     end if
5:     components ← []
6:     if n_f is a compound feature node then
7:         subNodes ← extractSubNodes(n_f)
8:         for subNode ∈ subNodes do
9:             components.addAll(κ(subNode))
10:        end for
11:        components.addAll(enc(n_f))
12:    end if
13:    return components
14: end function
```

Component Behaviour Generation Upon establishing the set $\kappa(root)$ of components, their behaviors are defined through the automated generation of finite state machines. Each finite state machine is composed of a finite set S representing states and a subset $T \subseteq S \times S$ signifying transitions. The transition set T is defined within the Cartesian product of states, specifically $S \times S$. In addition, a designated initial state `init` within S is specified. The individual component, referred to as `f`, is visually illustrated in Figure 4.4. This figure encapsulates the behavior of the component within its corresponding finite state machine (FSM).

$$enc(n_f) = \text{FSM in Fig. 4.4} \tag{4.2}$$

States. The generated Finite State Machine associated with component f, the states are:

- Initial State (*init*): This state signifies the absence of both feature activation and feature request.
- Intermediate States:
  - Start Feature State (S_f): a request for the activation of feature `f` has been initiated; however, the actual activation has not yet been carried out.
  - Start Reset Feature State (SR_f): a request for the deactivation of feature `f` has been initiated; however, the actual deactivation has not yet been carried out.

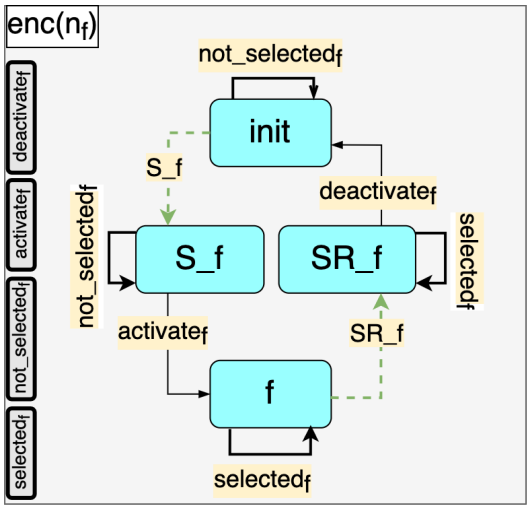

Figure 4.4: Feature component FSM.

- State (f): This state represents the successful activation of feature f.

Transitions. The transitions within the FSM are categorized into two types of ports:

- Spontaneous Transitions: These transitions are represented by green dashed arrows and are used to receive notifications from the external environment. Their purpose is to enable the component to automatically react to external triggers, such as a user activating or deactivating a feature. When the component receives a notification, if it is in a state where the spontaneous transition can be triggered, it will execute the transition, resulting in a state change.

- Enforceable Transitions: These transitions correspond to the execution of API functions that mandate a component to perform specific actions. Enforceable transitions represent actions controlled by the JavaBIP engine. Enforceable transitions are used for coordination between components to manage dependencies on when a feature can be activated safely and when it can be deactivated.

Example 4.3.2. *Figure 4.4 serves as a visual representation of the FSM corresponding to the feature f. The FSM structure encompasses four distinct states, explicitly denoted as* `init`, `S_f`, `SR_f`, *and* `f`, *highlighted in blue. Transitions represented by dashed green arrows are spontaneous transitions (`S_f` and `SR_f`) which are used to request the activation and the deactivation of feature f. For example, when there is a need to activate feature f, a spontaneous call for f activation can be made. Assume the component is initially in the* `init` *state.*

If the component receives a spontaneous event `S_f` *(start feature f) from the external environment (typically triggered by the system administrator), and the component has a transition labeled* `S_f` *from the* `init` *state, then the component will execute this spontaneous transition. This will cause the component to transition from the* `init` *state to the* `S_f` *state.*

The `S_f` *state is an intermediate state (Start feature f) and is not the active state where the activation API is called for activating feature f. From the* `S_f` *state, an enforceable transition can be executed which is associated with the activation API once executed to initiate the actual activation of feature f. This enforceable transition, labeled* `activate_f`*, will trigger the API call for the activation. The execution of this enforceable transition needs to be coordinated with other components according to the coordination layer, which will be discussed in the next subsection.*

Similarly, when there is a request to deactivate feature f, a spontaneous transition `SR_f` *will be executed, taking the component from the* `f` *state to the* `SR_f` *state (Start deactivation of feature f). From here, another enforceable transition will be executed to finalize the deactivation of feature f.*

4.3.2.2 Step 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 graphically represented by connectors.

Prerequisites for Coordination Layer Generation

To construct this coordination layer in our approach, two essential prerequisites are needed:

1. **Construction of the Feature Model Dependency Graph G_{FM}:** The construction of the dependency graph G_{FM} is carried out in accordance with Def. 4.1.3. The dependency graph G_{FM} represents the dependencies between features in the feature model.

2. **Computation of the Strongly Connected Components (SCCs):** The subsequent step involves computing the SCCs from the dependency graph G_{FM}. These SCCs constitute sets of features with interdependencies, embodying features that are mutually reliant.

A key characteristic of Strongly Connected Components (SCCs) is their ability to identify sets of features that have mutual, bidirectional dependencies.

Example 4.3.3. *Consider the case where feature A requires feature B, and feature B also requires feature A. In this scenario, features A and B would*

form an SCC, as they have a mutual dependency on each other. This means that if feature A is selected, feature B must also be selected, and vice versa. There cannot be a valid configuration that includes only A or only B due to their interdependency.

On the other hand, if there is a one-way dependency, such as feature A requiring feature B, but not the other way around, then feature B could be included in a configuration without feature A however feature A to be in the configuration it requires feature B in the configuration. This is because the dependency is unidirectional, rather than a mutual, interdependent relationship captured by an SCC.

For this reason, and to capture the dependencies and interdependencies between the features, we compute the SCCs that will be used to create the coordination layer in our approach.

In the context of the feature model depicted in Figure 4.5, the subsequent generation of the directed graph, is presented in Figure 4.6. For the illustration purposes, in this graph representation:

- Blue arrows depict *require* constraints, where the selection of one feature necessitates the inclusion of another feature due to a dependency.
- Green arrows indicate mandatory child features that must be included if their respective parent feature is selected.
- Pink arrows represent parent-child hierarchical relationships among features in the feature model.

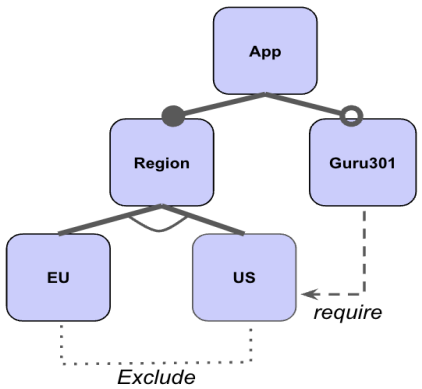

Figure 4.5: An example feature model.

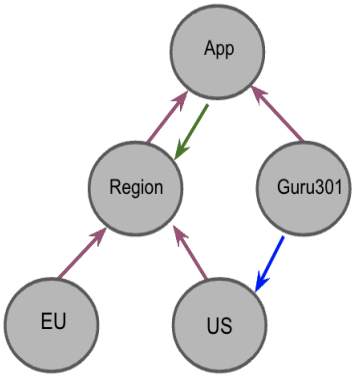

Figure 4.6: Directed dependency graph generated from the feature model in Fig. 4.5.

Subsequent to the graph generation, the computation of the SCCs from the constructed directed graph G_{FM} ensues, as depicted in Figure 4.7. For example, consider SCC_1, containing both the *App* and *Region* features. The interdependence between these features mandates their simultaneous activation and deactivation. Activating solely the *App* feature, without considering the presence of *Region*, would not respect the underlying dependencies. In addition, SCC_4 encompasses the *Guru*301 feature, which relies on the *App* and *US* features for its activation, which means that the *Guru*301 feature can be activated after the activation of *App* and *US* features.

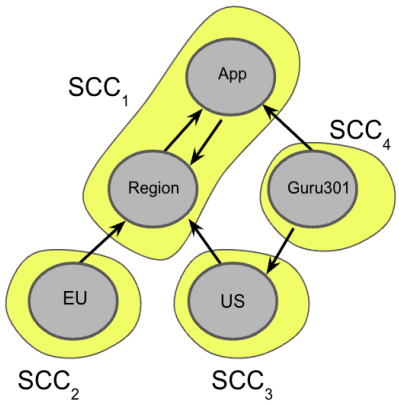

Figure 4.7: Extracted SCCs from G_{FM}.

In the following subsections, we will present the require and accept macros for the enforceable ports of the CBRTVM components. These macros specify the coordination between the components.

Each component corresponding to a feature f has four enforceable ports: `selected`, `not_selected`, `activate`, and `deactivate` as presented in Fig 4.4. We will show how the require and accept macros are constructed for these ports. For the `activate` ports, we will refer to the require and accept macros as the activation macros. Similarly, the require and accept macros for the `deactivate` ports will be called the deactivation macros'.

Activation Macros Generation

Before delving into the activation macros, it is important to mention that in our work, we treat XOR-groups (also known as alternative groups) in feature models as OR-groups with additional mutual exclusion constraints added between the child features. Although an XOR-group allows only one of its child features to be selected, it is semantically represented as an OR-group with the constraint that the child features are mutually exclusive to each other. To enforce this mutual exclusion, we extend the feature model by adding pairwise exclude constraints between all the child features of the XOR-group.

The activation macros for feature f are formulated by considering its relationships within the feature model structure. These macros are created using three key components:

- **Strongly Connected Component (SCC_f):** This represents a group of interdependent features, including f. For the activate port of the component corresponding to feature f, the require macro will include all the activate ports of the features in $SCC_f \setminus f$. This ensures that all the features in the same strongly connected component are activated together due to their interdependencies.

- **Dependency Set ($E(f)$):** This set encompasses the features that f relies on for its proper functionality. The require macro for the activate port of the component corresponding to f will include the `selected` ports of all components that correspond to features in $E(f) \setminus SCC_f$. This ensures that the activate port cannot be executed unless all the features that f depends on are already active.

- **Mutual Exclusion Set ($\chi(f)$):** This set consists of features that cannot be active at the same time as f due to the *exclude* constraints. The require macro for the activate port of the component corresponding to f will include the `not_selected` ports of all the components that correspond to features in $\chi(f)$. This ensures that the activate port cannot be executed if any of the features that are mutually exclusive with f are active.

The activation macros are defined in equations 4.3 to 4.6.

Equation 4.3 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.

$$requires\Big(enc(n_f).activate_f\Big) \stackrel{\text{def}}{=} \{enc(n_{f'}).activate_{f'} \mid f' \in SCC_f \setminus \{f\}\} \cup \{enc(n_{f'}).selected_{f'} \mid f' \in E(f) \setminus SCC_f\} \cup \{enc(n_{f'}).not\_selected_{f'} \mid f' \in \chi(f)\} \ . \tag{4.3}$$

Equation 4.4 states that the required ports of port $activate_f$ are also the accepted ones:

$$accepts\Big(enc(n_f).activate_f\Big) \stackrel{\text{def}}{=} \{enc(n_{f'}).activate_{f'} \mid f' \in SCC_f \setminus \{f\}\} \cup \{enc(n_{f'}).selected_{f'} \mid f' \in E(f) \setminus SCC_f\} \cup \{enc(n_{f'}).not\_selected_{f'} \mid f' \in \chi(f)\} \ . \tag{4.4}$$

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

$$requires\Big(enc(n_{f'}).selected_{f'}\Big) \stackrel{\text{def}}{=} \emptyset$$
$$accepts\Big(enc(n_{f'}).selected_{f'}\Big) \stackrel{\text{def}}{=} \{enc(n_{f''}).activate_{f''} \mid f'' \in SCC_f\} \cup \{enc(n_{f''}).selected_{f''} \mid f'' \in E(f) \setminus \{SCC_f, f'\}\} \cup \{enc(n_{f''}).not\_selected_{f''} \mid f'' \in \chi(f)\} \ . \tag{4.5}$$

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

$$requires\Big(enc(n_{f'}).not\_selected_{f'}\Big) \stackrel{\text{def}}{=} \emptyset$$
$$accepts\Big(enc(n_{f'}).not\_selected_{f'}\Big) \stackrel{\text{def}}{=} \{enc(n_{f''}).activate_{f''} \mid f'' \in SCC_f\} \cup \{enc(n_{f''}).selected_{f''} \mid f'' \in E(f) \setminus SCC_f\} \cup \{enc(n_{f''}).not\_selected_{f''} \mid f'' \in \chi(f) \setminus \{f'\}\} \ . \tag{4.6}$$

The creation of these activation macros ensures that the dependencies, exclusions, and interdependencies defined in the feature model are enforced during the execution of the `activate` ports. Furthermore, in Section 4.4, we present the theoretical results that show the composed CBRTVM models are correct by construction and reconfigurations are carried out in a safe manner such that only partial-valid configurations can be reached as a result of any reconfiguration.

Deactivation Macros Generation

Given the construction of the macros for activation, the corresponding deactivation connectors can be derived by reversing the activation. 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}.

$$\begin{aligned} requires\Big(enc(n_f).deactivate_f\Big) &\stackrel{\text{def}}{=} \{enc(n_{f'}).deactivate_{f'} \mid f' \in SCC_f \setminus \{f\}\} \\ &\cup \Big\{enc(n_{f'}).not\_selected_{f'} \mid f' \in E^{-1}(f) \setminus SCC_f\Big\} . \\ accepts\Big(enc(n_f).deactivate_f\Big) &\stackrel{\text{def}}{=} \{enc(n_{f'}).deactivate_{f'} \mid f' \in SCC_f \setminus \{f\}\} \\ &\cup \Big\{enc(n_{f'}).not\_selected_{f'} \mid f' \in E^{-1}(f) \setminus SCC_f\Big\} . \end{aligned} \tag{4.7}$$

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

$$\begin{aligned} requires\Big(enc(n_{f'}).not\_selected_{f'}\Big) &\stackrel{\text{def}}{=} \emptyset \\ accepts\Big(enc(n_{f'}).not\_selected_{f'}\Big) &\stackrel{\text{def}}{=} \{enc(n_{f''}).deactivate_{f''} \mid f'' \in SCC_f\} \\ &\cup \Big\{enc(n_{f''}).not\_selected_{f''} \mid f'' \in E^{-1}(f) \setminus \{SCC_f, f'\}\Big\} . \end{aligned} \tag{4.8}$$

Notice that the exclude constraints are not considered because they only affect the activation of features, not their deactivation. The exclude constraints ensure that certain features cannot be active at the same time. However, when a feature is being deactivated, the exclude constraints do not play a role, as the deactivation of a feature does not depend on the activation state of the features it excludes.

Example 4.3.4. *Based on the feature model presented in Fig. 4.8, 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. 4.8. 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.4.3.2.2, 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,*

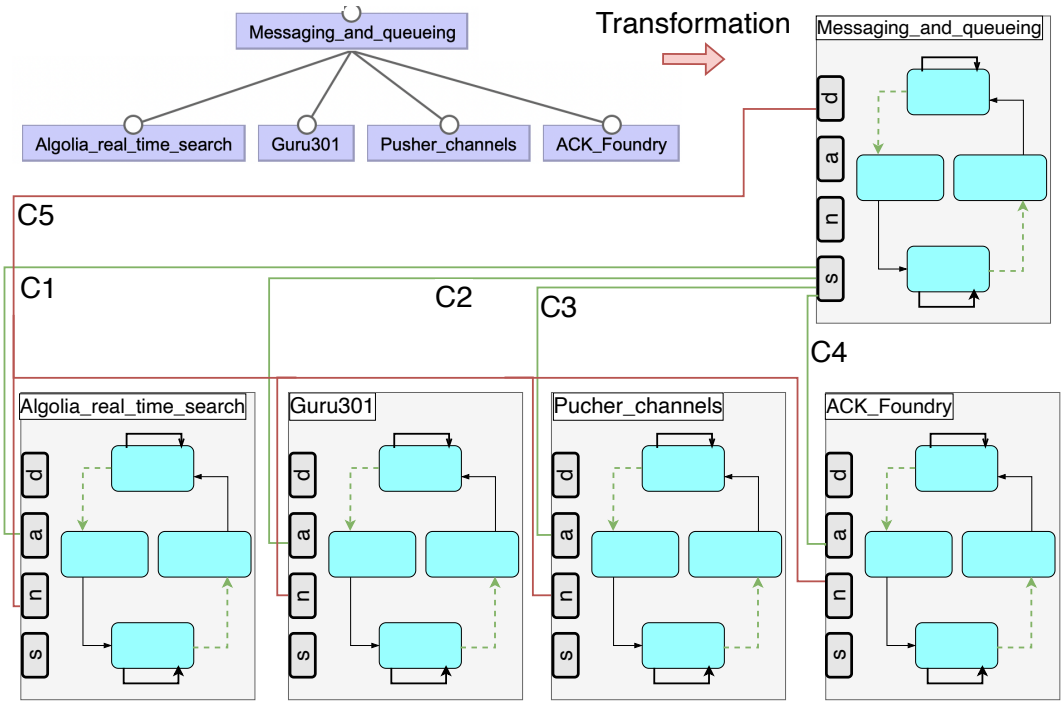

Figure 4.8: Part of the generated CBVM for the Heroku cloud FM: The behaviour of all the components is the same as shown in Fig. 4.4. For the sake of clarity, we shorten the names of the ports to the first letter.

this ensures that the configuration with `Algolia_real_time_search` *can be reached only when its dependencies are satisfied.*

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. 4.3.2.2 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 inactive states.*

Proposition 4.3.1. *Given a CBRTVM, for each interaction* e *allowed by Eqs. (4.3–4.8), 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 CBRTVM, each interaction is either a feature activation or deactivation, which involves a strongly connected component in the dependency graph.

Proof. Follows trivially from Eqs. (4.3–4.8). □

4.3.3 Stage 3: CBRTVM Integration

Stage 3, encompassing the integration of the CBRTVM into real-world case scenarios and its practical evaluation, will be covered in Chapter 6. In chapter 6 we will show how the generated CBRTVMs are integrated into practical applications and will discuss the results obtained from evaluating its performance in real-case scenarios.

Until now, we have demonstrated the FeCo4Reco transformation process for generating the CBRTVM from a feature model. In the next section, we will prove the properties about the reachable states in the generated CBRTVM.

4.4 Preserving Feature Model Semantics in CBRTVM

After having performed all the steps, the encoding process presented in Fig. 4.3 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 [122] is preserved from the FM to the CBRTVM by applying the encoding process, as the dependency graph issued from the feature model is used.

Since the CBRTVM is a JavaBIP model, it inherits the operational semantics of JavaBIP [28]. Notice that all interactions among enforceable ports correspond to either the activation of features (Eqs. (4.3–4.6)) or their deactivation (Eqs. (4.7) and (4.8)). Requesting individual feature activation or deactivation is done through notifications on spontaneous ports.

By construction, the operational semantics of the CBRTVM is represented by an LTS, whose states are implicitly described configurations with selected features, and whose transitions are labeled by interactions. Performing interactions leads to a configuration change, i.e., reconfigurations, and this section describes the properties of the reached states in the CBRTVM.

The reachable states in the CBRTVM correspond to configurations in the feature model. To reason about the properties of these reachable states, we will present a definition of the mapping function that links a state to a configuration, as defined in Definition 4.4.1.

Definition 4.4.1 (Mapping Function)**.** *Let $L = (Q, \Sigma, \rightarrow)$ be the LTS of a JavaBIP model CM generated using from FM using the FeCo4Reco transformation process. Let $\psi : Q \mapsto 2^F$ be the mapping from a state $q = (s_{f_1}, \ldots, s_{f_n})$ in Q to a configuration $\Phi = \{f_1, \ldots, f_n\} \subseteq F$ of FM defined by:*

$$\psi(q) = \{M(s_{f_1}), \ldots, M(s_{f_n})\}$$

where for every $0 \leq i \leq n$, $M(s_{f_i})$ is defined by:

$$M(s_{f_i}) = \begin{cases} \bar{f}_i & \text{if } s_{f_i} = \textit{init or } s_{f_i} = \textit{S-f} \\ f_i & \textit{otherwise} \end{cases}$$

Intuitively, the M function encodes a component state s_{f_i} into either the presence or the absence of feature f_i in configuration Φ. Specifically:

- if s_{f_i} is the initial state (`init`) or the requested state (`S-f`), then feature f_i is marked as absent ($\bar{f}_i$);
- otherwise, M maps s_{f_i} to f_i being present in configuration Φ.

We abuse notation and say that a configuration Φ in the feature model is reachable in the CBRTVM if $\psi(\Phi)$ is reachable in the LTS of the CBRTVM.

Proposition 4.4.1. *Any state reachable in the CBRTVM corresponds to a saturated partial-valid configuration.*

Proof. We proceed by induction.

Base case: the empty configuration trivially respects all dependencies and, by Assumption 4.1.1, can be completed to a valid configuration.

Induction hypothesis: if all configurations of size $\leq n$ reachable in the CBRTVM 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. (4.3–4.6). Clearly, $|\Phi'| \leq n$. Hence, by the induction hypothesis, Φ' is a saturated partial-valid configuration. Let $C = \{f \in F \mid enc(n_f).activate_f \in e\}$. By Proposition 4.3.1 and the fact that $\Phi' \subsetneq \Phi$, C is an SCC of the dependency graph. By Eqs. (4.3–4.6), the dependencies of all features in C are satisfied. Hence Φ is saturated. Furthermore, also by Eqs. (4.3–4.6), the activation of C cannot violate any exclusion constraints. Hence Φ is free from internal conflict and, by Assumption 4.1.1, it is partial-valid. □

Lemma 4.4.1. *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. 4.1.4, we have $f' \in E(f) \subseteq \Phi$. □

Proposition 4.4.2. *Let $\Phi \subset \Phi'$ be two saturated partial-valid configurations. Assume $\psi(\Phi)$ is the current state of the CBRTVM. 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 4.3.2.1 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 4.4.1, 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. (4.3–4.6), the features in $\Phi' \setminus \Phi$ can only be activated as one interaction. By observing the behaviour of components defined in Section 4.3.2.1 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. (4.3–4.6), 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 4.4.1. *For any reachable state in the CBRTVM, there exists a reachable state that corresponds to a valid configuration.*

Proof. Let Φ be a reachable configuration. By Proposition 4.4.1, it is saturated partial-valid. Hence, there exists a valid configuration $\Phi' \supseteq \Phi$. Applying Proposition 4.4.2 to Φ and Φ' proves the corollary. □

Corollary 4.4.2. *Any saturated partial-valid configuration is reachable in the CBRTVM.*

Proof. Let Φ be a saturated partial-valid configuration. Applying Proposition 4.4.2 to $\emptyset$ and Φ shows that it is reachable.

□

Lemma 4.4.2. *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. 4.7–4.8 in Sect. 4.3.2.2. □

Lemma 4.4.3. *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 4.1.4. 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 4.1.1, it is partial-valid. □

Proposition 4.4.3. *Let Φ and Φ' be two saturated partial-valid configurations. Assume $\psi(\Phi)$ is the current state in the CBRTVM. 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 4.4.3, $\Phi \cap \Phi'$ is a saturated partial-valid configuration. Hence, by Prop. 4.4.2, 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 4.4.2, 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 $\Phi' \setminus (\Phi \cap \Phi') = \Phi' \setminus \Phi$. □

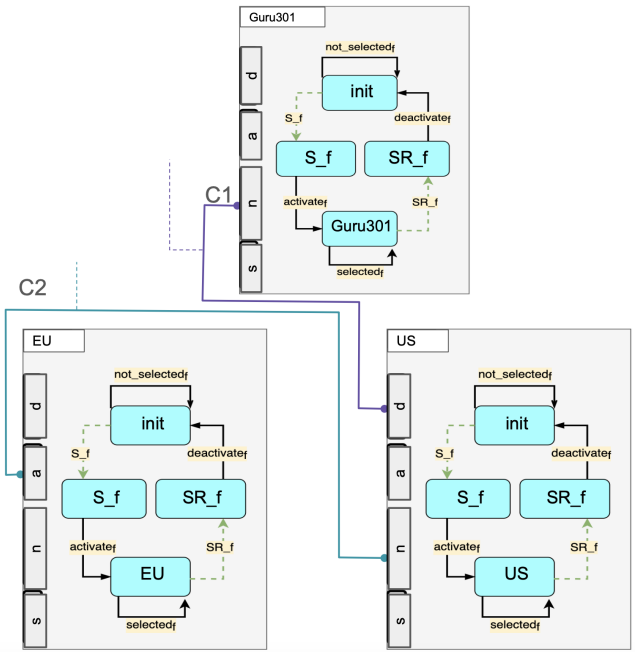

Figure 4.9: Part of the CBRTVM generated for the feature model presented in Fig. 4.5.

Based on all the theoretical results, the CBRTVM ensures safe reconfiguration management for software systems. The CBRTVM 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. 4.1.3, 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 4.4.1. *Building on Example 4.3.1, let us consider the scenario where we need to move the system from configuration*
$\alpha_1 = \{\texttt{Heroku\_Application}, \texttt{Process\_type}, \texttt{Dyno}, \texttt{Free}, \texttt{Region}, \texttt{US}, \texttt{Add\_ons}, \texttt{Messaging\_and\_queuing}, \texttt{Guru301}\}$
to
$\alpha_2 = \{\texttt{Heroku\_Application}, \texttt{Process\_type}, \texttt{Dyno}, \texttt{Free}, \texttt{Region}, \texttt{EU}\}$
by changing the region from ***US*** *to* ***EU*** *and deactivating the* ***Guru301*** *service.*

Fig. 4.9 illustrates the three components ***Guru301****,* ***EU****, and* ***US****. These components are in the* ***Guru301****,* ***init****, and* ***US*** *states, respectively. Two connectors,* $C1$ *and* $C2$*, are represented. Connector* $C1$ *indicates that the port* ***deactivate_f*** *can be executed only in synchronization with the* ***not_selected***

port of the `Guru301` *component. Connector* $C2$ *indicates that the activation of the* `activate` *port of the* EU *component requires synchronization with the* `not_selected` *port of the* `US` *component due to the mutual exclusion constraint between the* US *and the* EU *components. As we provide part of the CBRTVM, we ignore the other endpoints of the connectors, and this is represented by the dashed lines on the connectors.*

Assume the three spontaneous events are triggered by activation of the EU*, deactivation of* US *and deactivation of* $Guru301$ *from the* α_1 *state. There are six possible reconfiguration paths, as shown in Table 4.1, that can be taken to move the system from configuration* α_1 *to* α_2*. The CBRTVM can receive the reconfiguration request in any order, however, not all reconfiguration paths are valid, as certain interactions can only occur in specific states.*

Assume a deactivation request for the `US` *region is received while the system is in configuration* α_1*. In this case, the* `US` *component will receive the spontaneous request and transition to the* `SR_f` *state. However, the deactivation port (*`deactivate_f`*) of the* `US` *region cannot be executed immediately because the deactivation requires synchronization with the* `not_selected` *port of the* `Guru301` *component, which is still active and in the* `Guru301` *state. In this state, the* `not_selected` *port is not enabled. Consequently, the deactivation of the* `US` *region cannot proceed from the current configuration* α_1*, and the* `US` *component will remain in the* `SR_f` *state until the* `Guru301` *component is deactivated.*

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 feature can be deactivated since it requires synchronization with the* `not_selected` *port of* `Guru301` *component, which is already deactivated (component* `Guru301` *is in the* `init` *state where port* `not_selected` *is enabled). Therefore, the interaction for deallocating resources from the* `US` *region can be executed in synchronization with the* `not_selected` *of the component* `Guru301`*. 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 order of interactions enforced by the generated CBRTVM 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 CBRTVM is only generated once without computing the set of valid configurations. In particular, this means that we do not have to compute the reconfiguration plan. Furthermore, it drives the reconfiguration process in a “lazy” manner, by postponing feature (de)activation until it can be safely executed.

4.5 Conclusion

In this chapter, we presented an automated approach for enforcing by construction the safe reconfiguration behaviour of software products through the automatic derivation of executable, component-based run-time variability models (CBRTVMs) from feature models. The CBRTVMs, 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 runtime. Our approach ensures the preservation of feature model semantics and constraint consistency in the generated models as established in Sect. 4.4. Thus, we answer the research question RQ1: "How to enforce domain constraints during dynamic reconfiguration at low cost?". Our approach generates CBRTVMs that enforce domain constraints encoded in the feature model.

The term *Component-Based Run-Time Variability Model* (CBRTVM) highlights the following facts: 1) the model in question is executable and can be *used at run time* to enforce the domain constraints, and 2) the set of valid configurations is never computed explicitly but is derived from components representing individual features.

The key threat to the validity of our work lies in Assumption 4.1.1. We expect this assumption to hold for a large proportion of realistic feature models. Efficiently verifying or enforcing this assumption in the general case is challenging [84]. However, we can implement additional heuristics based on the propagation of exclusion constraints to increase the proportion of feature models that satisfy the assumption.

Chapter 5

Composing Run-time Variability Models

Software evolution [42] is the continual development of system software to extend its own functionality over time by integrating new functionalities not originally modeled.

To enable modeling a system as it evolves, there must be a means to integrate sub-models encapsulating new functionality into 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.

In this chapter, we will delve into addressing research questions 2 and 3, which we introduced in Chapter 1:

RQ2 How can we enable compositionality in our approach?

RQ3 How can we ensure that the compositional approach consistently enforces domain constraints based on the semantics of the composition?

In Chapter 4, we presented an automated model transformation approach called FeCo4Reco for enforcing safe reconfiguration of software products by construction. The generated CBRTVMs are represented in the JavaBIP framework [28]. We aim to tackle RQ2 and RQ3 by enabling compositionality in the FeCo4Reco approach, allowing the composition of multiple CBRTVMs derived from different feature models (FMs). The ability to compose CBRTVMs is important, as it enables the construction of complex variability models from smaller, reusable parts. However, to preserve the semantics and properties of the original FMs during composition, it is important to use composition operators with well-defined semantics. To this end, building upon the work presented in Chapter 4, this chapter introduces a study of novel JavaBIP

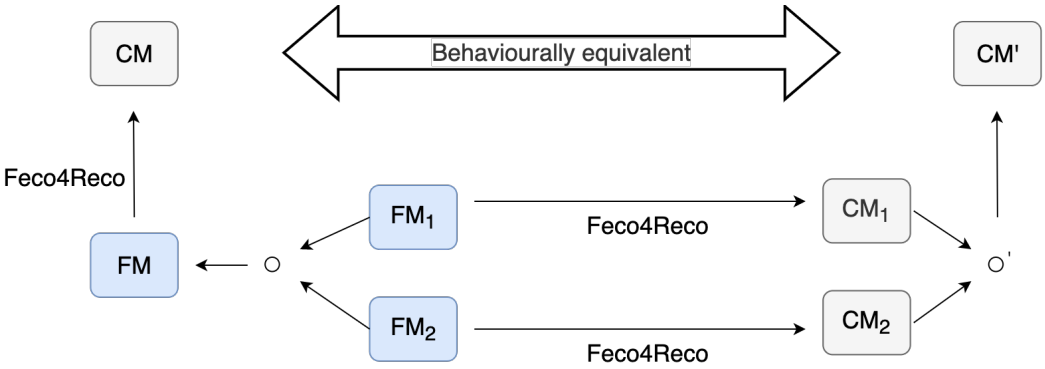

Figure 5.1: Overview of Feature and JavaBIP models composition

composition operators that enable our approach to be compositional while preserving the safety of dynamic reconfiguration. These operators correspond to the standard FM composition operators: union ($\cup$), intersection ($\cap$), and strict intersection ($\dot{\cap}$).

By defining JavaBIP composition operators that mirror these FM operators, we can compose the generated CBRTVMs based on the desired composition semantics, be it union, intersection, or strict intersection. Figure 5.1 provides an overview of our compositional approach. Our objective is to define, for each FM composition operator $\circ \in \{\cup, \cap, \dot{\cap}\}$, a corresponding JavaBIP composition operator $\circ'$, such that applying $\circ'$ to compose CM_1 and CM_2 yields a composed model CM' whereof the behaviour is equivalent to that of CM.

5.1 Composition of Feature Models

In this chapter, we adopt a denotational logic-based methodology for the composition of feature models, as outlined in [7]. 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 the three composition operators inspired by the ones in [7] and defined by the following Boolean formulae:

Intersection ($\cap$):	$\phi = \phi_{FM_1} \wedge \phi_{FM_2}$
Strict Intersection ($\dot{\cap}$):	$\phi = \left(\phi_{FM_1} \wedge not(F_2 \setminus F_1)\right) \wedge \left(\phi_{FM_2} \wedge not(F_1 \setminus F_2)\right)$
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. Notice that $[[FM_1]] \cup [[FM_2]] \subseteq [[FM]]$ holds, but $[[FM_1]] \cup [[FM_2]] = [[FM]]$ does not necessarily hold.

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 feature models, as long as the diagram is equivalent to the original formula.

5.2 Composition of CBRTVMs

This section provides a detailed presentation of CBRTVMs and of their composition. Our approach is structural. Let CM_1 and CM_2 be two 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 first describes the modification of components and macros to enhance flexibility in the activation and deactivation of features w.r.t. FeCo4Reco [72]. Then, it explains how these macros are composed for each of the composition operators presented in Section 5.1.

5.2.1 Macros for Composition

In the context of the FeCo4Reco model transformation, for each feature f, a corresponding component is generated, denoted as enc(n_f) (see Fig. 4.4). In this Chapter, we slightly alter the transition names for conciseness: we use a_f to denote $activate_f$, d_f for $deactivate_f$, s_f for $selected_f$, and ns_f for $not\_selected_f$. Each state of the generated components represents either the presence or the absence of the corresponding feature in the configuration.

In the CBRTVM generated, the coordination macros strictly encode dependencies among features. Indeed, the transformation computes the strongly connected components (SCCs) of the dependency graph and, for each feature f, defines the corresponding require and accept macros by putting

$$a_f \quad \textbf{Requires} \quad a_{f_1}, \ldots, a_{f_m}, s_{f'_1}, \ldots, s_{f'_n}, ns_{f''_1}, \ldots, ns_{f''_p}$$
$$a_f \quad \textbf{Accepts} \quad a_{f_1}, \ldots, a_{f_m}, s_{f'_1}, \ldots, s_{f'_n}, ns_{f''_1}, \ldots, ns_{f''_p}$$

where, $f_1, \ldots, f_m \in SCC_f \setminus \{f\}$, $f'_1, \ldots, f'_n \in E(f) \setminus SCC_f$, and $f''_1, \ldots, f''_p \in \chi(f)$.

To introduce greater flexibility in the (de)activation of features within the model, we modify the original macros while preserving essential properties proven in Chapter 4. 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. The statements and proofs of these results follow the proofs of the results presented in Chapter 4.

Definition 5.2.1. *(Macros for Composition) The macros for composition are defined by:*

$$a_f \ \textbf{\textit{Requires}} \ (a_{f_1}; s_{f_1}), \ldots, (a_{f_m}; s_{f_m}), (s_{f'_1}; a_{f'_1}), \ldots, (s_{f'_n}; a_{f'_n}), ns_{f''_1}, \ldots, ns_{f''_p}$$
$$a_f \ \textbf{\textit{Accepts}} \ a_{f_1}, s_{f_1}, \ldots, a_{f_m}, s_{f_m}, s_{f'_1}, a_{f'_1}, \ldots, s_{f'_n}, a_{f'_n}, ns_{f''_1}, \ldots, ns_{f''_p}, \ldots$$

where the semicolon in parentheses denotes disjunction (logical OR), whereas the comma denotes conjunction (logical AND).

Note that the dots in the Accept macros represent ports added after the saturation of the Accept macros, which is important since *Accept* macros are used to specify the list of all ports that are allowed to synchronize with the given port, as presented in Section 2.4. The need for saturation arises from the fact that the *Require* macros are transitive. Without saturation of the *Accept* macros, some transitively required ports in the *Require* macros may be excluded from the Accepts macros, leading to the restriction of those interactions. More details about the saturation process of the *Accept* macros, will be presented later in Section 5.2.3.

Example 5.2.1. *Consider the dependency graph in Fig. 5.2. In the context of SCC_2 depending on SCC_1, the modified macros for the activation ports of the components corresponding to features in SCC_2 are of the form: a_C **Requires** $(a_A; s_A), (a_B; s_B)$ and a_C **Accepts** a_A, s_A, a_B, s_B. It states that*

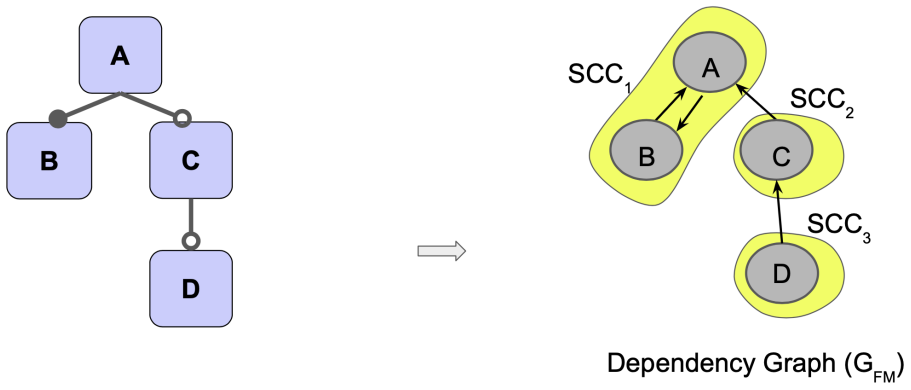

Figure 5.2: Example of a dependency graph

the a_c port can be synchronized with both a_A and a_B ports (allowing SCC_2 to be activated together with SCC_1), or with either s_A or s_B (allowing SCC_2 to be activated after SCC_1).

Let $G_{FM} = (F, E)$ be the feature model dependency graph and let $G' = (V', E')$ be the directed acyclic graph (DAG) obtained by factoring G_{FM} by its strongly connected components. The dependency relation $\prec$ on V' is defined as $s_1 \prec s_2$ if $(s_2, s_1) \in E'$. We say that s_2 *depends* on s_1.

The above modifications of the macros allow (de)activation of SCCs within the same transition as those they depend on. Definition 5.2.1 ensures that the SCC activation still respects the dependency graph. Furthermore, the results in Chapter 4 still hold.

5.2.2 Composing Requires Macros

As mentioned in the opening of Sect. 5.2, the composition of CBRTVMs is structural: composition operators are defined on the sets of 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 5.2.2. *(Composition Operators) Let ρ_1 and ρ_2 be two sets of Requires macros. We define the following composition operators:*

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

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

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

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

(A port that requires false will never be executed, as explained in Section 2.4.)

- ***Union*** *(∪):*

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

*(A port that has a "**Requires** true" can be executed as a singleton, as explained in Section 2.4.)*

5.2.3 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:

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

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. Sect 2.4). 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.

5.2.4 Composition Operators on JavaBIP models

Definition 5.2.3. *(Composition) Let* $CM_1 = (\mathcal{C}_1, \rho_1, \alpha_1)$ *and* $CM_2 = (\mathcal{C}_2, \rho_2, \alpha_2)$ *be JavaBIP models as defined in Sect. 2.4. Their composition by* $\circ' \in \{\cup, \cap, \dot{\cap}\}$ *is the JavaBIP model* $CM' = (\mathcal{C}', \rho', \alpha')$ *where:*

- $\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2$ *is the union of their components,*
- ρ' *is the composed require macros, i.e.* $\rho' = \rho_1 \circ' \rho_2$ *as defined in Sect 5.2.2,*
- α' *is the saturated accept macros as defined in Sect 5.2.3.*

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 [7], 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 chapter, 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 Sect. 5.2.4). 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.

5.3 A Bisimulation for Correctness and Compositionality Results

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.

Bisimulation is a binary relation commonly used in Concurrency Theory (e.g., [120]) 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) on CBRTVMs.

Definition 5.3.1 ($\mathcal{P}$-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 $u = \bigcup_{i=1}^{k} l_i$.*

Notice that Def. 5.3.1 does not exclude the possibility of $\mathcal{P}$ holding on the intermediate states of a $\mathcal{P}$-path.

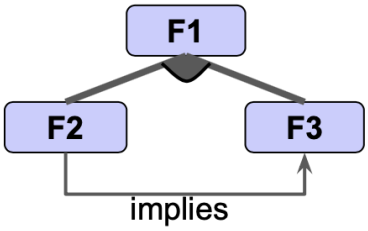

Figure 5.3

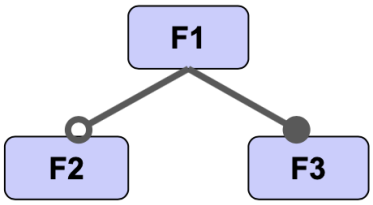

Figure 5.4

Figure 5.5: Two feature models that have the same set of valid configurations.

Example 5.3.1. *Consider the feature models shown in Figures 5.3 and 5.4. These two feature models have the same set of valid configurations:*

1. *F1, F3*
2. *F1, F2, F3*

However, they have different sets of saturated partial-valid configurations. This is because the dependency graphs and the sets of strongly connected components (SCCs) are not the same, as depicted in Figures 5.6 and 5.7. Trivially, in Figure 5.7, F1 and F3 form an SCC, while in Figure 5.3, each feature forms an SCC by itself. Even though the set of valid configurations is the same for the two feature models, their different structures lead to different sets of saturated partial-valid configurations.

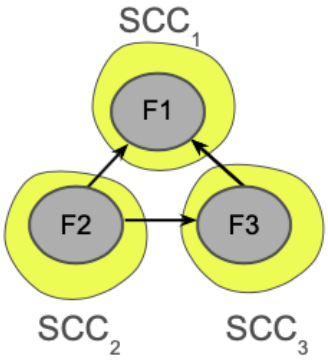

Figure 5.6

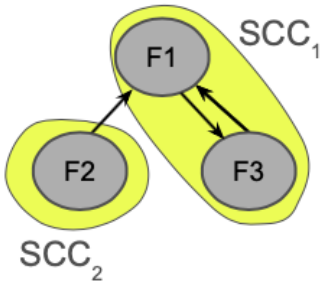

Figure 5.7

We now introduce the notion of the multi-step $\mathcal{UP}$-bisimulation, which allows us to compare behaviors of two LTSs (Labeled Transition Systems) with respect to states that satisfy a given predicate. It tolerates violations of interaction atomicity, meaning that a single transition in one LTS can be matched by multiple transitions in the other LTS, as long as the resulting sets of observable actions (those not part of the unobservable ports $\mathcal{U}$) coincide.

Definition 5.3.2 (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. $\rightarrow \subseteq Q \times \left\{\{p\} \,|\, p \in P\right\} \times Q$ with $P = P_1 = P_2$, multi-step $\mathcal{UP}$-bisimulation reduces to the classical bisimulation by taking $\mathcal{P} = \mathit{true}$ and $\mathcal{U} = \emptyset$.

Definition 5.3.3 (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.*

Let FM_1 and FM_2 be two feature models, $\circ \in \{\cup, \cap, \dot{\cap}\}$, CM and CM' be the CBRTVMs derived as in Fig. 5.1 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 activated or deactivated following given activation or deactivation 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 to be (see Fig. 4.4)

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

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 $\psi(q)$ is a valid configuration of the composed feature model $FM_1 \circ FM_2$ (c.f. Fig. 5.1).

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

The following lemmas and corollaries presented in this section are used to prove the main results in this chapter, namely Proposition 5.3.1, for the intersection, union, and strict intersection operators.

Lemma 5.3.1. *Let $\psi(q)$ be a configuration in FM and $q \xrightarrow{u} q'$ be a transition in L_{CM} corresponding to a spontaneous event, i.e. $u = \{S\text{-}f\}$ or $u = \{SR\text{-}f\}$ for some feature f. Then $\psi(q) = \psi(q')$.*

Proof. As detailed in the operational semantics of CM in Section 2.4, state q in L_{CM} contains a state from the $enc(n_f)$ component, denoted as s_f, where s_f is the component's current active state which can be init, S-f, SR-f, or f as shown in Fig.4.4. By Def. 4.4.1, a spontaneous transition of label $S\text{-}f$ in component $enc(n_f)$ is from state init to state $S\text{-}f$. Both states init and S-f are mapped to the absence of feature f, denoted as $\bar{f}$, in configuration $\psi(q)$. The same applies for transition $SR\text{-}f$. Therefore, a spontaneous transition does not modify the feature configuration. Then $\psi(q') = \psi(q)$. □

Lemma 5.3.2. *Let FM be the composed feature model from FM_1 and FM_2 using intersection operator such that $\phi_{fm} = \phi_{FM_1} \wedge \phi_{FM_2}$. A valid configuration $\Phi \in [[FM]]$ iff $\Phi \cap F_i \in [[FM_i]]$ for i = 1,2.*

Proof.

$$\begin{aligned} \Phi \models \phi_{FM} &\Rightarrow \Phi \models (\phi_{FM_1} \wedge \phi_{FM_2}) && \text{(by definition of } \phi_{FM}) \\ &\Rightarrow \Phi \models \phi_{FM_1} \wedge \Phi \models \phi_{FM_2} && \text{(semantics of } \wedge) \end{aligned}$$

Applying the configuration to only the relevant features:

$$\Phi \cap F_1 \models \phi_{FM_1}$$
$$\Phi \cap F_2 \models \phi_{FM_2}$$

Similarly, the reverse direction can be shown using the semantics of the conjunction operator. □

Corollary 5.3.1. *Let F_1, F_2 be the feature sets of FM_1 and FM_2 respectively. Let FM be a feature model such that $\phi_{FM} = \phi_{FM1} \wedge \phi_{FM2}$. If $\Phi \in [[FM]]$ then $\Phi_1 = F_1 \cap \Phi$ and $\Phi_2 = F_2 \cap \Phi$ are valid in respectively FM_1 and FM_2, and $\Phi = \Phi_1 \cup \Phi_2$.*

Proof. By Lemma 5.3.2, given that Φ is a valid configuration in FM, it follows that $\Phi \cap F_i$ is a valid configuration in FM_i for $i = 1, 2$. Building on this, the corollary comes from:

$$\Phi_1 \cup \Phi_2 = (F_1 \cap \Phi) \cup (F_2 \cap \Phi) = (F_1 \cup F_2) \cap \Phi = \Phi$$

□

Lemma 5.3.3. *Let FM be the composed feature model from FM_1 and FM_2 using intersection operator ($\circ = \cap$). For any valid configuration Φ' in FM and any feature $f \in \Phi'$, it holds that $SCC_{G_{FM_1}}(f) \cup SCC_{G_{FM_2}}(f) \subseteq \Phi'$.*

Proof. Let $f \in \Phi'$. By Lemma 5.3.2, since $\Phi' \in [[FM]]$, it follows that $\Phi' \cap F_1 \in [[FM_1]]$ and $\Phi' \cap F_2 \in [[FM_2]]$ where F_1, F_2 are the feature sets of FM_1, FM_2 respectively. In particular, $SCC_{G_{FM_1}}(f) \subseteq \Phi' \cap F_1$ and $SCC_{G_{FM_2}}(f) \subseteq \Phi' \cap F_2$. It follows that $SCC_{G_{FM_1}}(f) \subseteq \Phi'$ and $SCC_{G_{FM_2}}(f) \subseteq \Phi'$. Taking the union on both sides proves the claim that $SCC_{G_{FM_1}}(f) \cup SCC_{G_{FM_2}}(f) \subseteq \Phi'$. □

Lemma 5.3.4. *Let FM be the composed feature model from FM_1 and FM_2 using $\circ = \cap$ operator. For any valid configuration $\Phi' \in [[FM]]$ and a feature $f \in \Phi'$, it holds that $E(f) \subseteq \Phi'$ where $E(f) = E_{G_{FM_1}}(f) \cup E_{G_{FM_2}}(f)$.*

Proof. Given $\phi_{FM} = \phi_{FM_1} \wedge \phi_{FM_2}$. Since $\Phi' \models \phi_{FM}$, Φ' satisfies the constraints in both FM_1 and FM_2. This means for any $f \in \Phi'$, all dependency relations for f hold in both models. As $E(f)$ consolidates all such dependencies across the models, any feature $f' \in E(f)$ must satisfy the constraints as well. Therefore, if $f \in \Phi'$, then $E(f) \subseteq \Phi'$. □

Lemma 5.3.5. *Let FM be the composed feature model from FM_1 and FM_2 using intersection ($\circ = \cap$) operator. For any valid configuration Φ' in FM and any feature $f \in \Phi'$, it holds that $\chi_{G_{FM_1}}(f) \cup \chi_{G_{FM_2}}(f) \not\subset \Phi'$.*

Proof. By Lemma 5.3.2, Φ' is valid in FM, then $\Phi' \cap F_i$ is valid in FM_i for $i = 1, 2$. Since $\Phi' \cap F_i$ is valid in FM_i for $i = 1, 2$, then $\chi_{FM_i}(f) \cap \Phi' = \emptyset$ for $i = 1, 2$. Thus, $\chi_{G_{FM_1}}(f) \cup \chi_{G_{FM_2}}(f) \not\subset \Phi'$ concludes the proof. □

Lemma 5.3.6. *Let FM be the composed feature model from FM_1 and FM_2 using union operator such that $\phi_{fm} = \phi_{FM_1} \vee \phi_{FM_2}$. A valid configuration $\Phi \in [[FM]]$ if $\Phi \cap F_1 \in [[FM_1]]$ or $\Phi \cap F_2 \in [[FM_2]]$ or both.*

Proof. Φ is valid configuration in $[[FM]]$, then $\Phi \models \phi_{FM}$. By semantics of disjunction, this implies $\Phi \models \phi_{FM_1}$ or $\Phi \models \phi_{FM_2}$. Applying Φ only to the relevant feature sets F_1 and F_2:

$$\Phi \cap F_1 \models \phi_{FM_1} \text{ or } \Phi \cap F_2 \models \phi_{FM_2}$$

Therefore, $\Phi \cap F_i$ is valid in FM_i for $i = 1$ or $i = 2$ or both. □

Lemma 5.3.7. *Let FM be the composed feature model from FM_1 and FM_2 using the union ($\circ = \cup$) operator. For any valid configuration Φ' in FM and any feature $f \in \Phi'$, it holds that $SCC_{G_{FM_1}}(f) \cup E_{G_{FM_1}}(f) \subseteq \Phi'$ or $SCC_{G_{FM_2}}(f) \cup E_{G_{FM_1}}(f) \subseteq \Phi'$ or both.*

Proof. Let $f \in \Phi'$. By Lemma 5.3.6, since $\Phi' \in [[FM]]$, it follows that $\Phi' \cap F_1 \in [[FM_1]]$ or $\Phi' \cap F_2 \in [[FM_2]]$ where F_1, F_2 are the feature sets of FM_1, FM_2 respectively. Without loss of generality, assume $\Phi' \cap F_1 \in [[FM_1]]$. In particular, this means $SCC_{G_{FM_1}}(f) \subseteq \Phi' \cap F_1$ and $E_{G_{FM_1}}(f) \subseteq \Phi' \cap F_1$. Therefore, $SCC_{G_{FM_1}}(f) \subseteq \Phi'$ and $E_{G_{FM_1}}(f) \subseteq \Phi'$. By similar reasoning, if $\Phi' \cap F_2 \in [[FM_2]]$, then $SCC_{G_{FM_2}}(f) \subseteq \Phi'$ and $E_{G_{FM_2}}(f) \subseteq \Phi'$. Thus, this proves the claim that $SCC_{G_{FM_1}}(f) \subseteq \Phi'$ or $SCC_{G_{FM_2}}(f) \subseteq \Phi'$ or both. □

Lemma 5.3.8. *Let FM be the composed feature model from FM_1 and FM_2 using the union operator. For any valid configuration Φ' in FM and any feature $f \in \Phi'$, it holds that $\chi_{G_{FM_1}}(f) \not\subset \Phi'$ or $\chi_{G_{FM_2}}(f) \not\subset \Phi'$, or $\chi_{G_{FM_1}}(f) \cup \chi_{G_{FM_2}}(f) \not\subset \Phi'$.*

Proof. Let Φ' be a valid configuration in FM and $f \in \Phi'$. By Lemma 5.3.6, either $\Phi' \cap F_1 \in [[FM_1]]$, in which case $\chi_{G_{FM_1}}(f) \not\subset \Phi'$ since $\Phi' \cap F_1$ is valid in FM_1; or $\Phi' \cap F_2 \in [[FM_2]]$, in which case $\chi_{G_{FM_2}}(f) \not\subset \Phi'$ since $\Phi' \cap F_2$ is valid in FM_2; or both, in which case $\chi_{G_{FM_1}}(f) \cup \chi_{G_{FM_2}}(f) \not\subset \Phi'$ since $\Phi' \cap F_1$ is valid in FM_1 and $\Phi' \cap F_2$ is valid in FM_2. Therefore, either $\chi_{G_{FM_1}}(f) \not\subset \Phi'$, $\chi_{G_{FM_2}}(f) \not\subset \Phi'$, or $\chi_{G_{FM_1}}(f) \cup \chi_{G_{FM_2}}(f) \not\subset \Phi'$ holds. □

Proposition 5.3.1. *Let FM_1 and FM_2 be two feature models. Let $L = (Q, P, \rightarrow)$ and $L' = (Q', P', \rightarrow)$ be, respectively, the semantic LTSs of the CBRTVMs CM and CM' as shown in Figure 5.1. Then $CM \simeq_{\mathcal{UP}} CM'$ with $\mathcal{P}$ and $\mathcal{U}$ be defined as above.*

5.3.1 Proof of Intersection Case

Proof. (Intersection Case) **From L_{CM} to $L'_{CM'}$:** Let $q_1 \overset{u}{\Rightarrow} q_{n+1}$ be any multi-step transition in L_{CM}, with $u = l_1 \dots l_n$. Let q'_1 be the state in $L'_{CM'}$ such that $(q_1, q'_1) \in R$. Let $\Phi = \psi(q_1)$ and $\Phi' = \psi(q_{n+1})$. By Definition 5.3.1, Φ and $\Phi' \in [[FM]]$. Without loss of generality, intermediate states are assumed to correspond to non-valid configurations; indeed, otherwise, $q_1 \overset{u}{\Rightarrow} q_{n+1}$ can be divided into shorter parts respecting this assumption.

Consider the following cases:

Case 1: Spontaneous interaction. Let u be an interaction corresponding to the execution of a spontaneous event received in L_{CM}: u can be either {`S-f`} or {`SR-f`} (cf. Fig. 4.4). State q_1 is a tuple of component states including one of $enc(n_f)$ component. Starting from q_1 and according to u, $enc(n_f)$ performs the transition from `init` to `S-f` when $u =$ {`S-f`}, or from `f` to `SR-f` when $u =$ {`SR-f`}.

As $(q_1, q'_1) \in R$, the component states in q'_1 in $L'_{CM'}$ mirror those in q_1. Thus, $enc(n_f)$ in q'_1 can execute the same interaction u, leading to q'_2. By Lemma 5.3.1, both $\mathcal{P}(q_2)$ and $\mathcal{P}(q'_2)$ hold, ensuring $(q_2, q'_2) \in R$ satisfying conditions of a multi-step $\mathcal{UP}$-bisimulation.

Case 2: Interactions Involving Enforceable Ports. Consider $u = l_1 \dots l_n$ in the system L_{CM}, where both l_1 and l_n correspond to an interaction involving enforceable ports.

Let $L'_{CM'}$ be over B that is the set of components that correspond to the features in $\Phi' \setminus \Phi$. Based on the composed macros for the intersection operator in Def. 5.2.2, firing the port a_f for any component $b \in B$ requires firing:

$$a_f \textbf{ Requires } (a_{f'_1}; s_{f'_1}), \dots, (a_{f'_m}; s_{f'_m}), \quad | \ f'_i \in SCC_f = SCC_{G_{FM_1}}(f) \cup SCC_{G_{FM_2}}(f) \setminus \{f\} \tag{5.1a}$$

$$(a_{f''_1}; s_{f''_1}), \dots, (a_{f''_n}; s_{f''_n}), \quad | \ f''_i \in E(f) = E_{G_{FM_1}}(f) \cup E_{G_{FM_2}}(f) \setminus SCC_f \tag{5.1b}$$

$$ns_{f'''_1}, \dots, ns_{f'''_k} \quad | \ f'''_i \in \chi(f) = \chi_{FM_1}(f) \cup \chi_{FM_2}(f) \tag{5.1c}$$

Since state q_n is reached in L_{CM} and u involves interactions with enforceable ports, for any component $b \in B$ the state becomes f. This implies that in L_{CM}, port a_f is fired for every component $b \in B$. Consequently, a_f must first be enabled, and to be enabled the component should either have already received and executed a spontaneous activation event before the multi-step transition u, or can be received and executed starting from q'_1, i.e., there is an interaction

l_i among those of u, $1 < i < n$, and $l_i = \{\texttt{S-f}\}$ or $l_i = \{\texttt{SR-f}\}$. Thus, given that the same components B are concerned in CM', it follows that CM' will receive the same set of spontaneous events as in u.

W.l.g, let us assume that in L'_{CM} all spontaneous activation events are received starting from q'_1. By Case 1, these spontaneous events can be executed and:

$$q'_1 \xrightarrow{l_1} q'_2 \dots q'_{m-1} \xrightarrow{l_{m-1}} q'_m$$

Upon reaching state q'_m in $L'_{CM'}$, it is established that port a_f is enabled for each component b in B. Now, we need to prove the existence of a transition $q'_m \xrightarrow{l_m} q'_{m+1}$ in L'_{CM}, with $l_m = \{a_f \mid f \in \Phi' \backslash \Phi\}$. To substantiate this transition, we must demonstrate two key aspects for every port a_f of a component b in B: firstly, the right-hand side of the require macro for a_f, as specified in the composed model CM', is enabled; and secondly, that the accept macro for a_f accepts all the required ports.

Given that $SCC_{G_{FM_1}}(f) \cup SCC_{G_{FM_2}}(f) \subseteq \Phi'$ and $E(f) = E_{G_{FM_1}}(f) \cup E_{G_{FM_2}}(f) \subseteq \Phi'$, as supported by Lemma 5.3.3 and Lemma 5.3.4, we can conclude that the ports a'_f for all components corresponding to features in both SCC_f and $E(f)$ are enabled, thus satisfying conditions in Eq. 5.1a and in Eq. 5.1b.

Furthermore, by Lemma 5.3.5, $\chi_{G_{FM_1}}(f) \cup \chi_{G_{FM_2}}(f) \not\subset \Phi'$, thus for all components corresponding to features in $\chi_{G_{FM_1}}(f) \cup \chi_{G_{FM_2}}(f)$, port $ns_{f'}$ is enabled, satisfying conditions in Eq. 5.1c.

Additionally, the saturation of accept macros guarantees that all ports required by a_f are accepted for the interaction (cf. Sect. 5.2.3). Therefore, this concludes the existence of a transition of label $l' = \{a_f\} \cup \{a'_f \mid f' \in SCC_f \cup E(f)\} \cup \{ns_{f'''} \mid f''' \in \chi(f)\}$, where $l'_m = l' \backslash \{s_{f_1}, \dots, s_{f_n}, ns_{f_1} \dots, ns_{f_n}\}$, keeping only the observable ports. Consequently, we can assert that

$$q'_m \xrightarrow{l'_m} q'_{m+1}$$

Consequently, the existence of $u' = l'_1 \dots l'_m$ in L'_{CM} is established, where $\bigcup_{i=1}^{n} l_i$ in L_{CM} is equal to $\bigcup_{i=1}^{m} l'_i$ in L'_{CM}. Furthermore, the attainment of all requested features in Φ' within L'_{CM} and the equality $\bigcup_{i=1}^{n} l_i = \bigcup_{i=1}^{m} l'_i$ ensure that $\mathcal{P}(q_{m+1})$ holds. This implies that the component states in q_{n+1} and q_{m+1} are the same, thereby confirming that $(q_{n+1}, q'_{m+1}) \in R$.

The same reasoning can be applied to the deactivation. Indeed, any synchronized activation of a set of features can be reversed by the corresponding synchronized deactivation of the same features.

From L'_{CM} to L_{CM}: Let q_1 be the state in L_{CM} such that $(q'_1, q_1) \in R$ and $q'_1 \overset{u'}{\Longrightarrow} q'_n$ be a multi-step transition in $L'_{CM'}$, where $u' = l'_1 \dots l'_n$. Let $\Phi' = \psi(q'_n)$. Since $\mathcal{P}(q'_n)$ holds, then $\Phi' \in [[FM]]$. Therefore, given that L_{CM} is generated from FM through the FeCo4Reco transformation process, as outlined in [72], by Lemma 4.4.3, we deduce that state q_{m+1}, where $\Phi' = \psi(q_{m+1})$, can be reached within L_{CM}. Hence, there exists a multi-step transition u from q_1, where $u = u'$, and $(q'_n, q_{m+1}) \in R$, and we are done. □

5.3.2 Proof of Strict Intersection Case

Proof sketch. (Strict Intersection Case) The reasoning for strict intersection follows that of the intersection case, with the primary distinction being the exclusion of unique features. However, this is tackled through the composition's encoding, wherein the macros of both models incorporate (p ***Requires*** *false*) for every port p found exclusively in one of the models ($p \in P_1 \setminus P_2 \cup P_2 \setminus P_1$). Intuitively, this means that ports that are exclusive to a model will never participate in any interaction. □

5.3.3 Proof of Union Case

Proof. (Union Case) The proof is similar to the intersection case up until the composition of the required macros. For the union composition operator, the require macros are composed as disjunctions instead of conjunctions (cf. Def. 5.2.2) and extended with "p **Requires** *true*" for $p \in P_1 \setminus P_2 \cup P_2 \setminus P_1$. Specifically, firing the port a_f for any component $b \in B$ requires firing:

$$a_f \textbf{ Requires } (a_{f'_1}; s_{f'_1}), \dots, (a_{f'_m}; s_{f'_m}), \quad | \ f'_i \in SCC_f = SCC_{G_{FM_1}}(f) \setminus \{f\} \tag{5.2a}$$

$$(a_{f''_1}; s_{f''_1}), \dots, (a_{f''_n}; s_{f''_n}), \quad | \ f''_i \in E_{G_{FM_1}}(f) \setminus SCC_f \tag{5.2b}$$

$$ns_{f'''_1}, \dots, ns_{f'''_k} \quad | \ f'''_i \in \chi_{FM_1}(f) \tag{5.2c}$$

or requires firing:

$$a_f \textbf{ Requires } (a_{f'_1}; s_{f'_1}), \ldots, (a_{f'_m}; s_{f'_m}), \quad | \ f'_i \in SCC_f = SCC_{G_{FM_2}}(f) \setminus \{f\} \tag{5.3a}$$

$$(a_{f''_1}; s_{f''_1}), \ldots, (a_{f''_n}; s_{f''_n}), \quad | \ f''_i \in E_{G_{FM_2}}(f) \setminus SCC_f \tag{5.3b}$$

$$ns_{f'''_1}, \ldots, ns_{f'''_k} \quad | \ f'''_i \in \chi_{FM_2}(f) \tag{5.3c}$$

The same reasoning for the spontaneous events can also be applied, thus, w.l.g, let us assume that in L'_{CM} all spontaneous activation events are received starting from q'_1, then these spontaneous events can be executed and:

$$q'_1 \xrightarrow{l_1} q'_2 \ldots q'_{m-1} \xrightarrow{l_{m-1}} q'_m$$

Upon reaching state q'_m in $L'_{CM'}$, it is established that port a_f is enabled for each component b in B. Now, we need to prove the existence of a transition $q'_m \xrightarrow{l_m} q'_{m+1}$ in L'_{CM}, with $l_m = \{a_f \mid f \in \Phi' \setminus \Phi\}$. To substantiate this transition, we must prove two key aspects for every port a_f of a component b in B: firstly, the right-hand side of the require macro for a_f, as specified in the composed model CM', is enabled; and secondly, the accept macro for a_f accepts all the required ports.

By Lemma 5.3.7, one has $SCC_{G_{FM_1}}(f) \subseteq \Phi'$ and $E_{G_{FM_1}}(f) \subseteq \Phi'$, or $SCC_{G_{FM_2}}(f) \subseteq \Phi'$ and $E_{G_{FM_2}}(f) \subseteq \Phi'$, or both, it follows that the ports a'_f for all components corresponding to features in $SCC_{G_{FM_1}}(f) \cup E_{G_{FM_1}}(f)$, $SCC_{G_{FM_2}}(f) \cup E_{G_{FM_2}}(f)$, or both, are enabled. Consequently, this satisfies the conditions in either Eq. 5.2a and 5.2b, Eq. 5.3a and 5.3b, or both. Then, by Lemma 5.3.8, either $\chi_{FM_1}(f) \not\subset \Phi'$, $\chi_{FM_2}(f) \not\subset \Phi'$, or $\chi_{FM_1}(f) \cup \chi_{FM_2}(f) \not\subset \Phi'$. Consequently, for all components corresponding to features in $\chi(f)$, the port $ns_{f'}$ is enabled. This ensures compliance with the conditions specified in either Eq. 5.2c, Eq. 5.3c, or both. Therefore, at least one of Eq. 5.2 or Eq. 5.3 is satisfied.

Additionally, the saturation of accept macros guarantees that all ports required by a_f are accepted for the interaction (cf. Sect. 5.2.3). Therefore, this concludes the existence of a transition of label $l' = \{a_f\} \cup \{a'_f \mid f' \in SCC_f \cup E(f)\} \cup \{ns_{f'''} \mid f''' \in \chi(f)\}$, where $l'_m = l' \setminus \{s_{f_1}, \ldots, s_{f_n}, ns_{f_1} \ldots, ns_{f_n}\}$, keeping only the observable ports from state q'_m, such that:

$$q'_m \xrightarrow{l'_m} q'_{m+1}$$

Consequently, the existence of $u' = l'_1 \dots l'_m$ in L'_{CM} is established, where $\bigcup_{i=1}^{n} l_i$ in L_{CM} is equal to $\bigcup_{i=1}^{m} l'_i$ in L'_{CM}. Furthermore, the attainment of all requested features in Φ' within L'_{CM} and the equality $\bigcup_{i=1}^{n} l_i = \bigcup_{i=1}^{m} l'_i$ ensure that $\mathcal{P}(q_{m+1})$ holds. This implies that the component states in q_{n+1} and q_{m+1} are the same, thereby confirming $(q_{n+1}, q'_{m+1}) \in R$.

The same reasoning can be applied to the deactivation. Indeed, any synchronized activation of a set of features can be reversed by the corresponding synchronized deactivation of the same features.

From L'_{CM} to L_{CM}: The proof follows the same reasoning as in the intersection case. □

5.3.4 Congruence of the $\mathcal{UP}$-bisimulation

Multi-step $\mathcal{UP}$-bisimulation is not a congruence on JavaBIP models in general 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 5.2.

Proposition 5.3.2. *Let CM_1, CM_2, and CM_3 be three models 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 operator $\circ' \in \{\cup, \cap, \dot{\cap}\}$, 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. First of all, notice that any $\mathcal{P}$-path can be extended to a longer $\mathcal{P}$-path in the semantic LTS of the same CBRTVM by firing unobservable `not_selected` ports after the firing of the corresponding (`de`)`activate` ports.

Consider a $\mathcal{P}'$-path π in $CM_1 \circ' CM_3$. Let π_1 and π_3 be its projections onto CM_1 and CM_3, respectively. The path π_1 is a $\mathcal{P}$-path and, by multi-step $\mathcal{UP}$-bisimilarity has an equivalent $\mathcal{P}$-path π_2 in CM_2. Assume that π_2 cannot be synchronised with an extension of π_3 and consider the first state where this happens. That means that the firing of some port p in π_2 or π_3 is blocked by a require macro modified by the composition operator (see Defn. 5.2.2), i.e. either p **Requires** *false* (strict intersection), or p **Requires** L_1, L_2.

The first case (p **Requires** *false*) can only occur when $p \in (P_1 \setminus P_2) \cup (P_2 \setminus P_1)$, meaning that p is an enforceable port corresponding to a dead feature. However, such a port cannot be fired in the first place.

Thus, all possible cases lead to contradictions, proving the proposition. □

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.

5.4 Conclusion

In this chapter, we introduced and studied composition operators for CBRTVMs, enabling automated construction of component-based systems in a modular fashion. It provides re-usability by allowing CBRTVM models to be reused across systems and instances through composition, flexibility by enabling composing models 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—well-established variability models—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 run-time reconfiguration.

With the proposed encoded composition operators, we have answered RQ2 by enabling compositionality through the introduction of novel CBRTVM composition operators that correspond to the well-studied FM composition operators as presented in Section 5.2. Furthermore, in Section 5.3, we have addressed RQ3 by ensuring that the composed CBRTVMs preserve the semantics of the composed feature models, ensuring the correctness of the encoding.

Finally, to prove the correctness and compositionality of our approach, we have introduced a novel multi-step $\mathcal{UP}$-bisimulation relation. While it is not a congruence on JavaBIP models in general, we have shown that it is a congruence on the sub-algebra of JavaBIP models generated by CBRTVMs.

Chapter 6

Practical and Experimental Validation

To demonstrate the practical applicability and effectiveness of our approach, we conducted several experiments. First, we implemented the FeCo4Reco transformation process using the ATLAS Transformation Language (ATL) and integrated the generated JavaBIP runtime CBRTVM with a cloud computing system. We measured the runtime overhead of the CBRTVM and showed that it is negligible.

Next, we validated that the CBRTVM enforces safe reconfigurations by deploying a simple web application on the Heroku cloud. We showed through two scenarios that the CBRTVM prevents unsafe reconfigurations. Finally, we evaluated the composition operators - union, intersection, and strict intersection - implemented in the macro composer. Using the Heroku and CloudWatch feature models, we composed their CBRTVMs and demonstrated through reachability analysis that the composed CBRTVM behaves as expected theoretically for the three operators.

6.1 Implementation of FeCo4Reco Transformation Process

We have implemented the model transformation process of FeCo4Reco using the ATLAS Transformation Language (ATL) [81]. ATL allows specifying rules for transforming a source model conforming to a source metamodel into a target model conforming to a target metamodel.

In our implementation, the source models are feature models in XML format conforming to the feature model metamodel [116] (Figure 6.1). The target models are component-based models conforming to the JavaBIP metamodel [94] (Figure 6.2).

The ATL transformation rules specify how entities in the feature model (e.g., features, constraints) are mapped to components, ports, and coordination macros in the JavaBIP model. The output of the ATL transformation is an XML file representing the JavaBIP architectural model.

This XML file is then parsed using the DOM API in Java to generate the JavaBIP runtime executable including:

- Java classes representing the BIP components and their ports
- Glue code implementing the coordination between components

The source code of our implementation is available on Zenodo [2] for reproducibility.

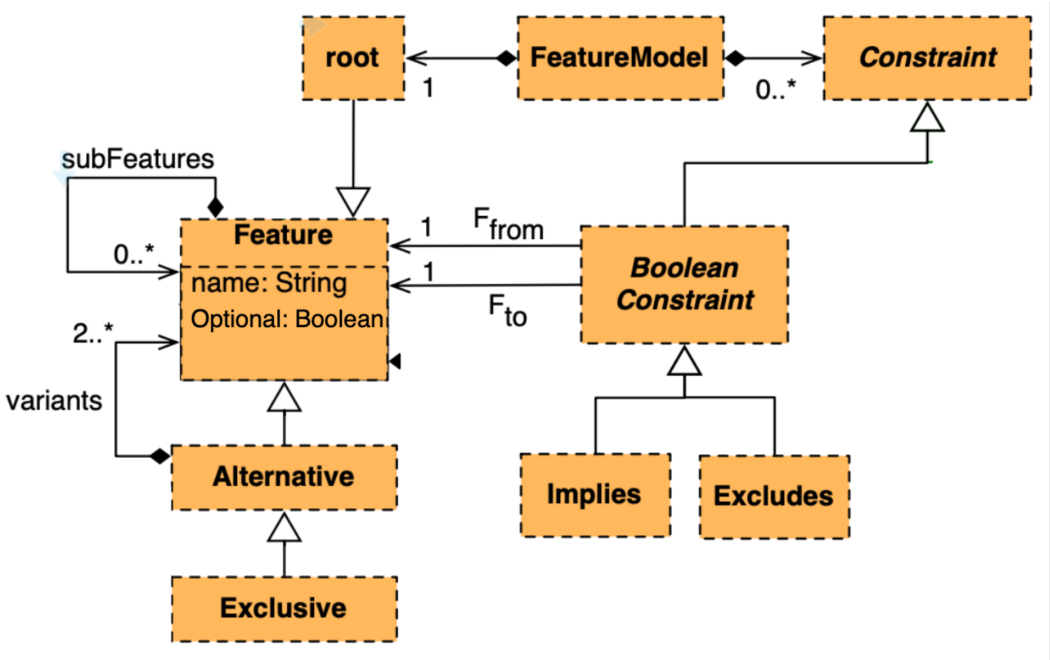

Figure 6.1: Feature model metamodel.

Figure 6.3 shows the FeCo4Reco architecture that enables stages 3 (Sect.4.3). The CBRTVM 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

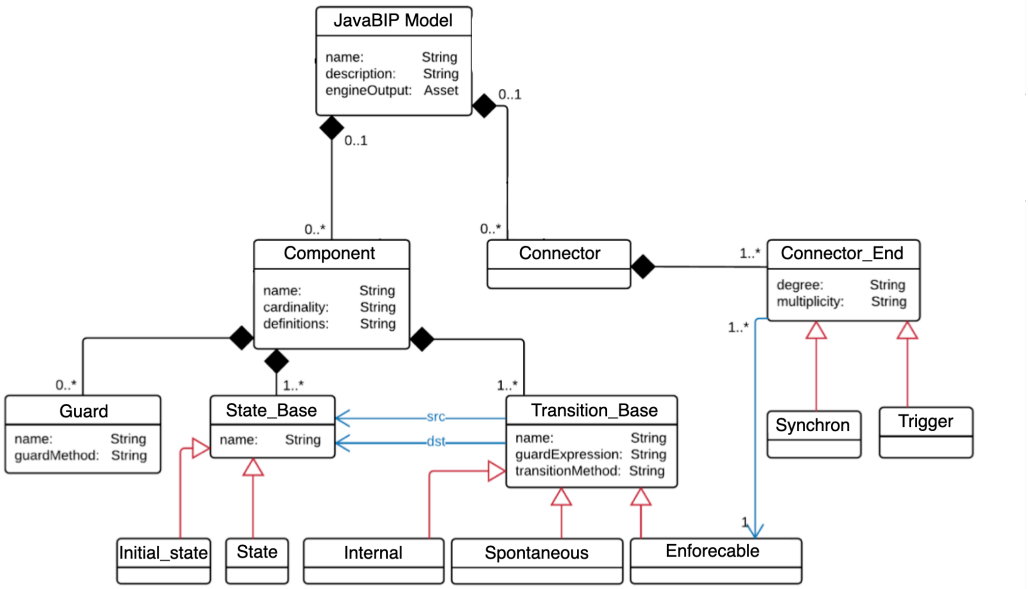

Figure 6.2: JavaBIP metamodel.

spontaneous event notifications (cf. Chapter 2, Sect. 2.4). This can be done by different means depending on the requirements of the Cloud computing system. For this work, we have designed a dashboard application with two buttons (Activate and Deactivate) for each feature as shown in Fig. 6.4.

6.2 Heruko Deployer Overview

The Heroku cloud deployer project was Simon's project that enables deployment of Java servlet applications on the Heroku cloud platform. It utilizes the Heroku CLI to provision resources and manage the deployment process. Heroku CLI is a tool for interacting with the Heroku platform programmatically. It provides a set of commands that enable developers to configure and manage Heroku resources and deploy applications directly from the command line.

We have extended this project by integrating the HerokuDeployer component, as illustrated in Figure 6.6, into the CBRTVM. This integration allows for the automatic deployment of applications to the Heroku cloud, specifically targeting the Java Virtual Machine (JVM) region in the US or EU with the free dyno resource configuration. Furthermore, we have implemented a user interface within the CBRTVM that facilitates the activation or deactivation of Heroku add-ons to the deployed application. Through this interface, the Guru301 add-on can be requested to be provisioned to the application.

The `HerokuDeployer` component as shown in Figure 6.5 initiates its operation in the *init* state. Subsequent to this initial phase, the configuration

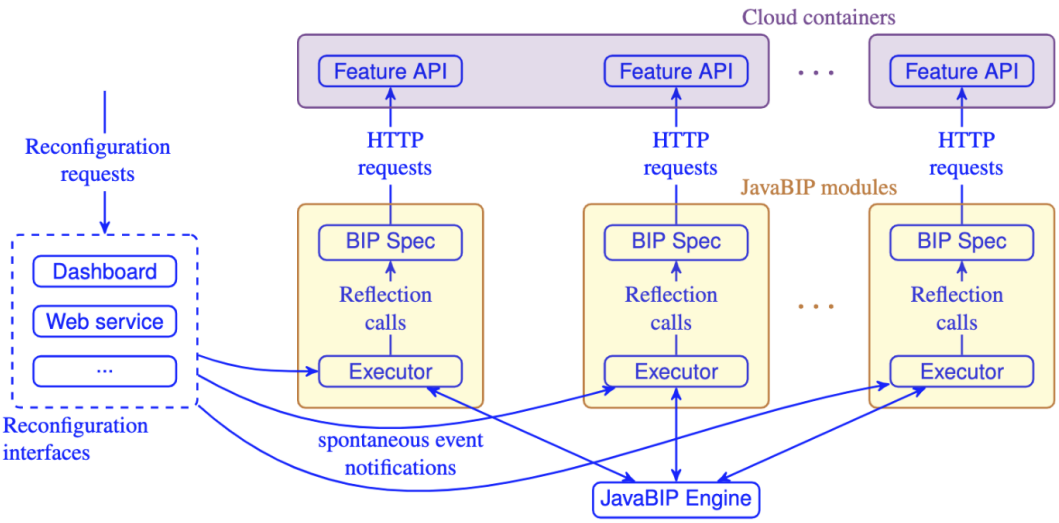

Figure 6.3: Integration of the JavaBIP CBRTVM with a Cloud Computing System.

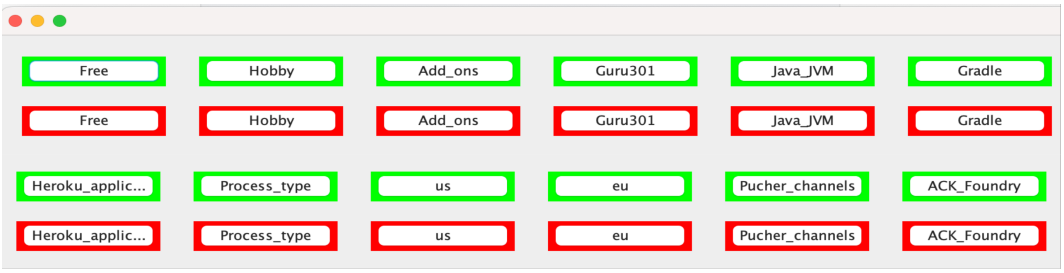

Figure 6.4: An interface for users to trigger requests.

is set to utilize a free dyno through the execution of the command `heroku dyno:type=free`. Following this configuration, the container is designated to the EU region by the command `heroku config:set HEROKU_REGION=eu`. The final step in the configuration process involves the selection of the JVM buildpack, accomplished with `heroku buildpacks:set heroku/java`. Each of these transitions is facilitated by API calls specifically designed to allocate the necessary resources.

Once the configuration of the resources is set up, the deployment of the locally built WAR file to the Heroku container is achieved through the use of the command:

```
heroku war:deploy /path/to/your/app.war --app your-app-name
```

This sequence of operations allows the deployment process within the Heroku environment.

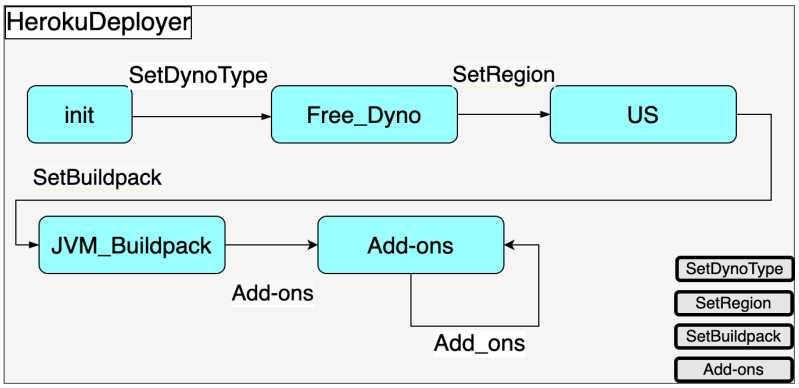

Figure 6.5: Heroku deployer component.

6.2.1 Performance Evaluation

On the overhead measures To evaluate the performance overhead introduced by our approach, we conducted experiments by executing the generated JavaBIP runtime and comparing it to direct reconfiguration for feature models of varying sizes. The feature models were randomly generated with different numbers of features: 100, 200, and 300 features.

For each feature model size, we generated 20 random feature models and applied our FeCo4Reco transformation to produce the corresponding JavaBIP runtime. We then measured the time and memory overhead incurred during the reconfiguration process, which involves transitioning the system from one valid configuration α_1 to another valid configuration α_2. These configurations were selected randomly.

As depicted in Figure 6.7, the coordination logic introduced by our approach incurs a small constant overhead in terms of both time (measured in milliseconds) and memory (measured in megabytes). Notably, this overhead was observed to be negligible compared to the typical resource requirements of cloud applications, demonstrating the practical applicability of our approach in real-world scenarios.

6.2.2 Safe Reconfiguraiton

On safe reconfiguration To illustrate that the reconfigurations are performed safely we deployed a simple web application onto the Heroku cloud with the CBRTVM generated from the Heroku cloud FM (Fig. 4.1). The CBRTVM is running on a local server using Tomcat 9. It intercepts (re)-configuration requests via APIs using HTTP request methods. The CBRTVM acts on the received requests to control migrating the system along a safe path to the desired configuration. A simple configuration for the web application is dyno

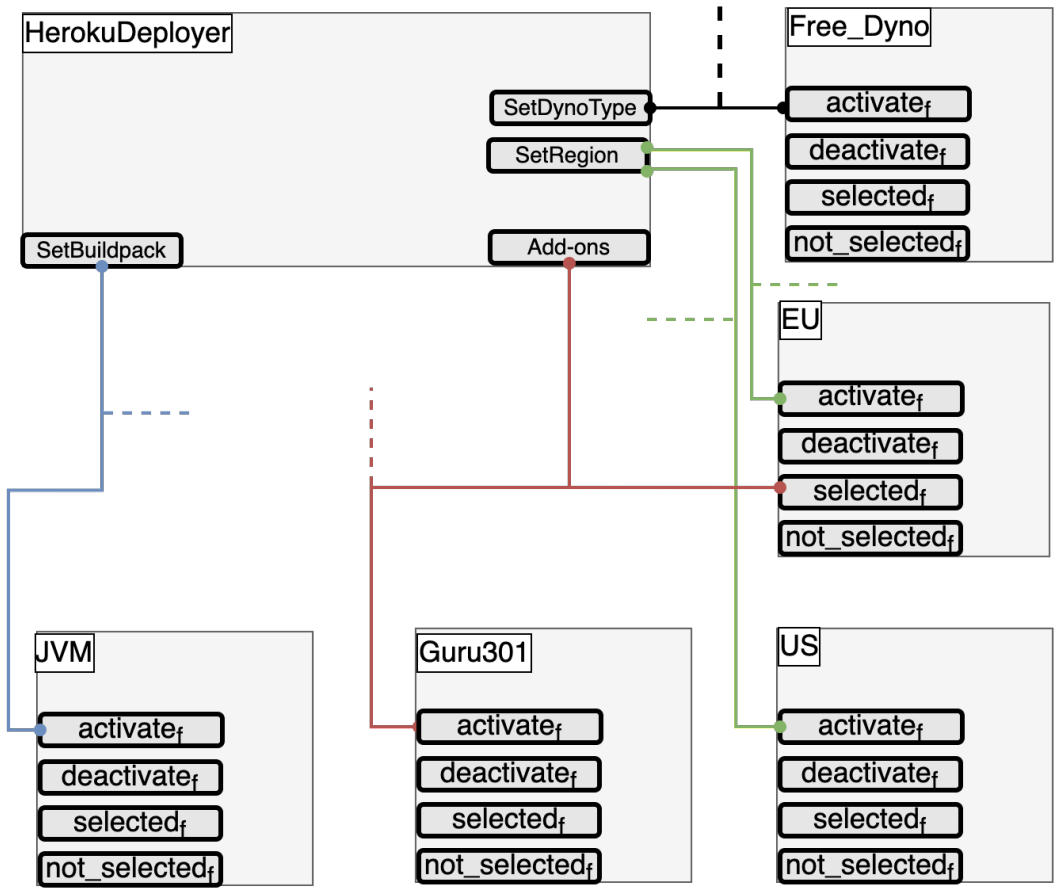

Figure 6.6: The overview of Heroku Deployer integrated into CBRTVM.

free (free container), region (EU), and build-pack (Java jvm). Consider two reconfiguration scenarios in relation to Example 4.3.1:

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 CBRTVM transition for activating `Guru301` is triggered synchronously with the selection of the `US` region. The final configuration is shown in Fig. 6.8.
Scenario 2: In this case, the `Guru301` service is not added to the web application. Upon receiving the request, the CBRTVM postponed the activation of `Guru301` service until the application region becomes `us`. The final configuration is shown in Fig. 6.9.

Furthermore, Figures 6.8 and 6.9 show the final configurations for two scenarios described in Sect. 6 obtained with the generated CBRTVM.

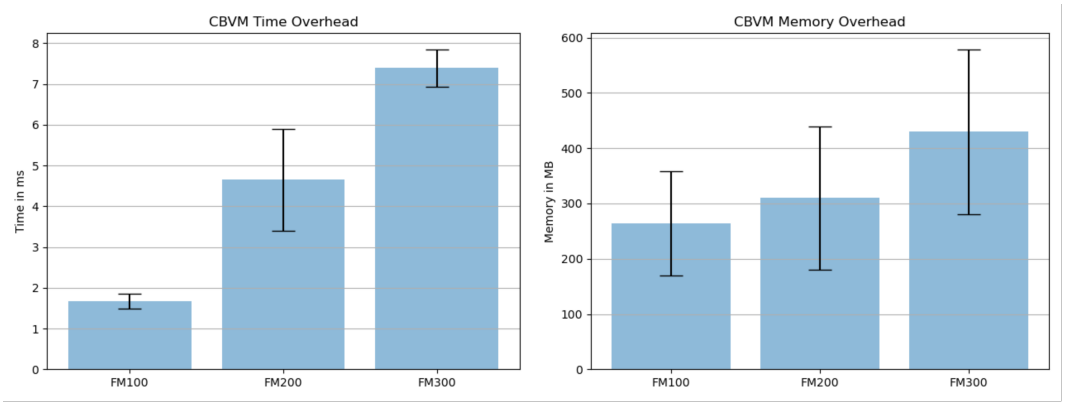

Figure 6.7: Model overhead (average values for the generated FMs).

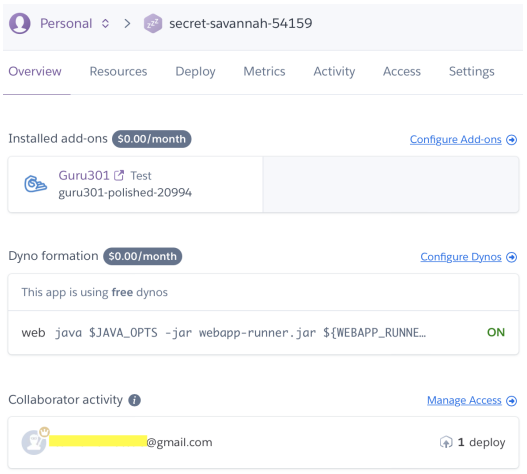

Figure 6.8: Using the generated CBRTVM to request the `Guru301` add-on to the `us` region knowing that `Guru301` requires `us`: success.

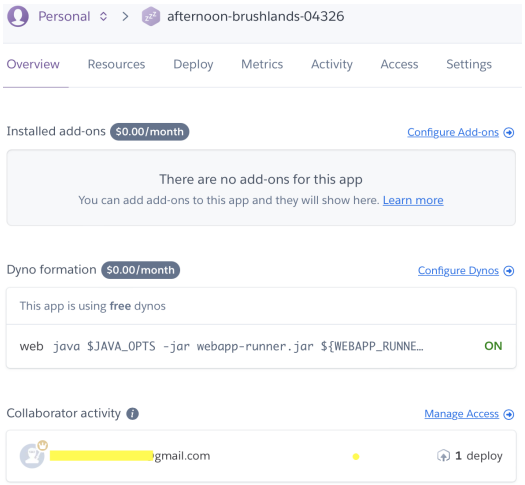

Figure 6.9: Using the generated CBRTVM to request the `Guru301` add-on to the `eu` region knowing that `Guru301` requires `us`: no add-ons.

This experiment demonstrates the effectiveness of the CBRTVM in ensuring safe reconfigurations of cloud-based applications. By intercepting and coordinating reconfiguration requests, the CBRTVM guarantees that the system transitions along a safe path to the desired configuration, adhering to the constraints specified in the feature model. The two scenarios illustrate the CBRTVM's ability to handle dependencies between components, such as the requirement for the *Guru*301 add-on to be added to the application only when the resources of the application are allocated in the *US* region.

Table 6.1: 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

6.3 Composition Operators Experimental Validation

In this section, we present an experimental evaluation of the composition operators developed for enabling compositionality in the FeCo4Reco approach. We use the Heroku Cloud platform and a new CloudWatch monitoring service to illustrate and validate the three composition operators: union, intersection, and strict intersection. The evaluation is carried out using the FeCo4Reco model transformation to derive executable Component-Based Run-Time Variability Models (CBRTVMs) from feature models.

6.3.1 Running 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 [1]. 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 Fig. 6.10.

Suppose a new monitoring service, CloudWatch, 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 Fig. 6.11.

To incorporate this service, the Heroku FM and the CloudWatch FM must be composed into one model that includes these new functionalities.

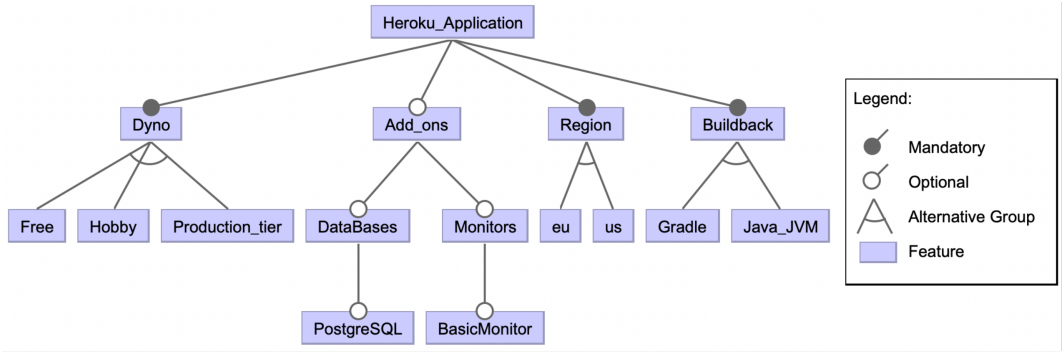

Figure 6.10: Simplified Heroku Cloud feature model.

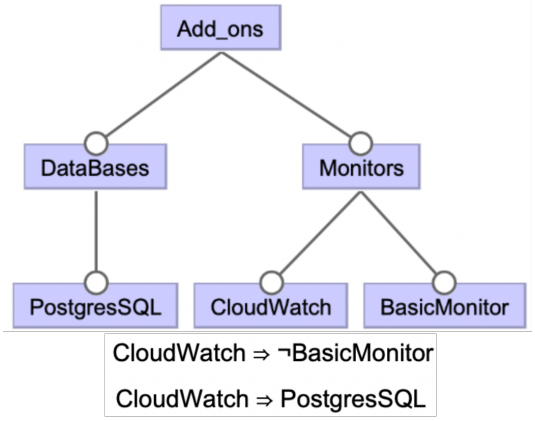

Figure 6.11: CloudWatch FM.

6.3.2 Experimental Setup and Evaluation

For evaluation purposes, we use the CloudWatch feature model as in Fig. 6.11, 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 this Chapter 5. Using the FeCo4Reco model transformation [72], 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 Chapter 5, Section 5.2, 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 specifi-

cations sets of CBRTVMs to compose and a resulting macro file for a chosen composition operator are then packaged to create a composed CBRTVM.

Table 6.1 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 6.12 sums up the reachability outcomes across the operators used for the composition.

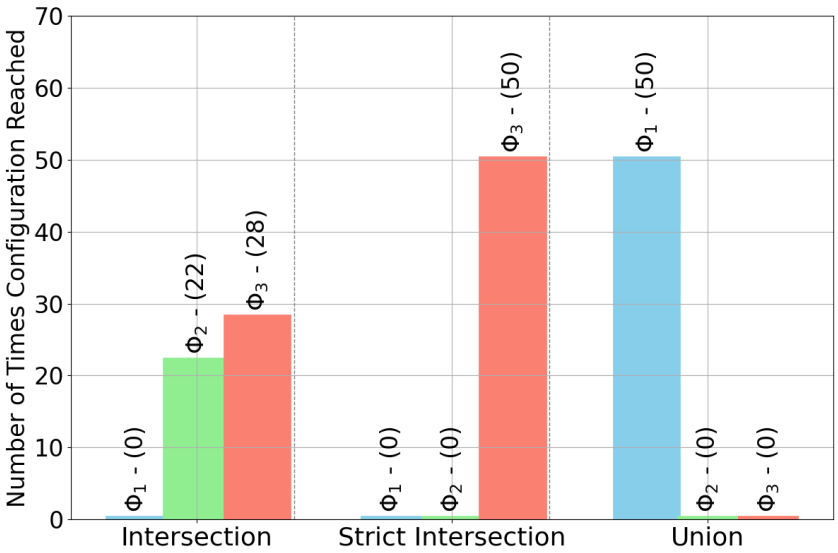

Figure 6.12: Observed configurations.

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 Fig. 6.12, 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 Fig. 6.12. 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 6.1). 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 6.1. 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.

This outcome serves as a practical validation of the behavior of the encoded operators for the CBRTVMs.

6.4 Conclusion

The FeCo4Reco transformation process is implemented using the ATLAS Transformation Language (ATL), enabling the generation of executable CBRTVM from the input feature model. The performance evaluation results demonstrate that the overhead introduced by the CBRTVM is negligible in terms of both time and memory, making our approach viable for practical application in real-world systems.

Regarding the enforcement of domain constraints during the reconfiguration process, our theoretical and practical validation confirms that the generated CBRTVMs effectively preserve the feature model constraints during system reconfigurations. The CBRTVM ensures that the system transitions along safe paths, adhering to the specified constraints and handling dependencies between features correctly.

Furthermore, our approach exhibits compositional properties, as shown by the practical evaluation of the composition operators implemented in the macros composer. Reachability analysis on the composed CBRTVMs demonstrates behavior that aligns with the theoretical results for each operator, allowing valid configurations to be reached while preventing invalid ones.

Chapter 7

Conclusion and Perspective

In this chapter, we summarize our thesis dissertation by discussing the challenges and goals addressed, and we outline our contributions. Then, we discuss our short-term and long-term perspectives related to the work presented in this dissertation.

7.1 Summary of the Dissertation

Modern software systems are increasingly distributed and controllable via APIs. They leverage a variety of software services, particularly with the widespread adoption of cloud computing. Cloud computing represents a promising paradigm for realizing the vision of utility computing, where developers can access services on-demand based on their requirements. These resources encompass computational and memory resources, virtual machines, application servers, databases, etc. System administrators thus take advantage of the flexibility of this provisioning model and rely on cloud environments to host their applications. However, managing reconfiguration is not a straightforward task, highlighting the need to address our first research question (RQ1): *How to enforce domain constraints during dynamic reconfiguration at low cost?* Approaches presented in the state-of-the-art approaches focus either on generating a target configuration without detailing the reconfiguration plan needed to reach it from the current configuration or using formal models which are complex to build. Consequently, system administrators require an approach to allow reconfiguration of the application while only enforcing reconfiguration in safe transitions.

Furthermore, software systems often evolve over time with the addition of new components or services. These new elements can expand the configuration space of the system. Thus, there is a need for our approach to be compositional to support system evolution and accommodate the integration of new services and models into the existing CBRTVM. This aligns with the research questions

we aimed to address, specifically RQ2: *How can we enable compositionality in our approach?* and RQ3: *How can we ensure that the compositional approach consistently enforces domain constraints based on the semantics of the composition?*

In this dissertation, we presented an automated approach for enforcing by construction the safe reconfiguration behaviour of software products through the automatic derivation of executable, component-based run-time variability models from feature models. The component-based run-time variability models, control the application behaviour by handling reconfiguration requests and executing them in such a way 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 Chapter 4, Sect. 4.4. Additionally, the feasibility and effectiveness of our approach are demonstrated through an evaluation of the overhead induced by the CBRTVM 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. Therefore, our approach provides a solution to RQ1.

In addition, we introduce 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. 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. By introducing composition operators for CBRTVMs and establishing the multi-step $\mathcal{UP}$-bisimulation equivalence, we address the research questions RQ2 and RQ3.

7.2 Perspectives

Although the work presented in this dissertation covers the need for reconfiguration and evolution of software systems deployed in complex environments, there is still work that can be done to improve our research. In this section, we thus discuss some short-term and long-term perspectives that should be considered in the continuation of this work.

7.2.1 Short-term Perspectives

A key threat to the validity of our approach lies in Assumption 4.1.1, which states that all considered feature models are such that any configuration free from internal conflict is partial-valid. Although we expect this assumption to hold for a large proportion of realistic feature models, efficiently verifying or enforcing it in the general case is challenging [84]. To extend our approach's applicability to a broader range of feature models, we plan to develop stronger heuristics for enforcing this assumption. One potential strategy involves upfront propagation of exclude constraints in the feature model hierarchy before component generation. This would prevent reaching intermediate configurations that, although conflict-free, cannot be extended to valid configurations due to exclude constraints at lower levels in the hierarchy. The propagation could be achieved by traversing the feature model upwards, collecting exclude constraints at each level, and adding these propagated constraints to the parent nodes. H

Moreover, we intend to generalize our approach to handle constraints among features formulated as arbitrary Boolean formulas in addition to the feature model, rather than being limited to require and exclude constraints. Handling such additional constraints is challenging as it requires efficient encoding and reasoning about their semantics alongside the feature model within the CBRTVM formalism. One potential solution is to first represent the additional constraints as a dual Horn formula. The dual Horn representation of these additional constraints can then be encoded into the glue macros of the CBRTVM formalism. This allows for efficient reasoning over the combined feature model and additional Boolean constraints within our approach. Furthermore, we aim to incorporate temporal constraints over features [130]. For instance, a temporal constraint could specify that once a high-performance database plan is selected, the system cannot transition to a lower-performance plan during its lifetime. However, extending our approach to support temporal constraints poses significant challenges. It requires capturing the temporal semantics within the formalism used for feature modeling and encoding them into the CBRTVM.

Finally, we look forward to incorporating AI capabilities that enable dynamic, context-aware reconfigurations of the system based on detected changes in the operating environment. By leveraging AI tools, the system could continuously monitor and interpret contextual factors, triggering autonomous reconfigurations.

7.2.2 Long-term Perspectives

To further enhance the automation and applicability of our approach, we need to investigate cloud ontologies and API integration. Currently, in the generated

CBRTVMs, the API calls required for feature activation and deactivation are manually added for each transition. Thus, one long-term perspective is to explore ways to automatically associate API calls with features and integrate them into the transformation process from feature models to CBRTVMs. This would involve analyzing cloud ontologies and service descriptions to map feature activations and deactivations to the corresponding API calls required for provisioning or de-provisioning the associated cloud resources and services.

In addition, to streamline the user experience, we aim to provide intelligent default value assignments for feature group selection by extending the feature model with requirements specifications mapped to an ontology such as cloud ontology. Leveraging this ontological knowledge representation, we can employ engines to automatically derive suitable default feature values adhering to specified constraints and preferences extracted from the requirements. While constructing accurate ontologies from natural language and developing efficient reasoning algorithms for potentially large ontologies introduces challenges, this approach simplifies configuration through automated decision-making aligned with requirements. This reduces manual user effort.

Furthermore, since the JavaBIP engine depends on binary decision diagrams (BDDs) to inform components of its intended states each cycle, we also want to conduct a rigorous analysis of the space complexity of the BDDs used in our methodology. BDDs provide a compact representation of Boolean functions, enabling efficient evaluation and manipulation of the feature constraints. However, while BDDs are more compact than truth tables, their size can still grow exponentially in the number of Boolean variables for certain function representations. Given that feature models can express constraints over hundreds or thousands of features, the BDD representations used in our approach may become a source of space complexity issues. A rigorous analysis of the space complexity is important for understanding the scalability limits and identifying potential optimizations or alternative representations.

Finally, we will explore a more generalized composition approach that goes beyond the predefined operators currently presented. By allowing user-defined composition, we intend to provide greater flexibility in constructing CBRTVMs.

Bibliography

[1] Heroku: Cloud platform. https://www.heroku.com.

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

[3] Mathieu Acher, Philippe Collet, Franck Fleurey, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. Modeling context and dynamic adaptations with feature models. In *4th International Workshop Models@ run. time at Models 2009 (MRT'09)*, page 10, 2009.

[4] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Composing feature models. In *International Conference on Software Language Engineering*, pages 62–81. Springer, 2009.

[5] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. 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*, pages 3–19. Springer, 2010.

[6] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. 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*, pages 17–33. Springer, 2010.

[7] Mathieu Acher, Benoit Combemale, Philippe Collet, Olivier Barais, Philippe Lahire, and Robert B France. 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*, pages 352–369. Springer, 2013.

[8] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on services computing*, 11(2):430–447, 2017.

[9] Nick Antonopoulos and Lee Gillam. *Cloud computing*, volume 51. Springer, 2010.

[10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[11] Mohsen Asadi, Ebrahim Bagheri, Dragan Gašević, Marek Hatala, and Bardia Mohabbati. Goal-driven software product line engineering. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 691–698, 2011.

[12] Mohsen Asadi, Gerd Gröner, Bardia Mohabbati, and Dragan Gašević. Goal-oriented modeling and verification of feature-oriented product lines. *Software & Systems Modeling*, 15:257–279, 2016.

[13] Paul Attie, Eduard Baranov, Simon Bliudze, Mohamad Jaber, and Joseph Sifakis. A general framework for architecture composability. *Formal Aspects of Computing*, 28(2):207–231, 2016.

[14] Muhammad Ali Babar, Lianping Chen, and Forrest Shull. Managing variability in software product lines. *IEEE software*, 27(3):89–91, 2010.

[15] Felix Bachmann and Paul Clements. *Variability in software product lines*. Carnegie Mellon University, Software Engineering Institute, 2005.

[16] Sundramoorthy Balaji and M Sundararajan Murugaiyan. Waterfall vs. v-model vs. agile: A comparative study on sdlc. *International Journal of Information Technology and Business Management*, 2(1):26–30, 2012.

[17] Luciano Baresi and Clément Quinton. Dynamically evolving the structural variability of dynamic software product lines. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 57–63. IEEE, 2015.

[18] 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.

[19] Don Batory. Feature models, grammars, and propositional formulas. In *Int. Conf. on Software Product Lines*, pages 7–20. Springer, 2005.

[20] Julian Bellendorf and Zoltán Ádám Mann. Specification of cloud topologies and orchestration using tosca: a survey. *Computing*, 102(8):1793–1815, 2020.

[21] David Benavides, Alexander Felfernig, José A Galindo, and Florian Reinfrank. Automated analysis in feature modelling and product configuration. In *Safe and Secure Software Reuse: 13th International Conference on Software Reuse, ICSR 2013, Pisa, June 18-20. Proceedings 13*, pages 160–175. Springer, 2013.

[22] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information systems*, 35(6):615–636, 2010.

[23] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.

[24] Nelly Bencomo, Svein Hallsteinsen, and Eduardo Santana De Almeida. A view of the dynamic software product line landscape. *Computer*, 45(10):36–41, 2012.

[25] 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.

[26] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. Opentosca–a runtime for tosca-based cloud applications. In *Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings 11*, pages 692–695. Springer, 2013.

[27] Ekaba Bisong and Ekaba Bisong. An overview of google cloud platform services. *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*, pages 7–10, 2019.

[28] 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.

[29] Mohammad Ubaidullah Bokhari, Qahtan Makki, and Yahya Kord Tamandani. A survey on cloud computing. In *Big data analytics: proceedings of CSI 2015*, pages 149–164. Springer, 2018.

[30] Clarissa Borba and Carla Silva. A comparison of goal-oriented approaches to model software product lines variability. In *Advances in Conceptual Modeling-Challenging Perspectives: ER 2009 Workshops CoMoL, EThe-CoM, FP-UML, MOST-ONISW, QoIS, RIGiM, SeCoGIS, Gramado, Brazil, November 9-12, 2009. Proceedings 28*, pages 244–253. Springer, 2009.

[31] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. A theory of software product line refinement. *Theoretical Computer Science*, 455:2–30, 2012.

[32] Jan Bosch and Rafael Capilla. Dynamic variability in software-intensive embedded system families. *Computer*, 45(10):28–35, 2012.

[33] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, September 2006.

[34] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In *International Symposium on Component-based Software Engineering*, pages 7–22. Springer, 2004.

[35] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[36] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06)*, pages 40–48. IEEE, 2006.

[37] Brendan Burns. *Designing distributed systems: patterns and paradigms for scalable, reliable services.* " O'Reilly Media, Inc.", 2018.

[38] Jessie Carbonnel, Marianne Huchard, André Miralles, and Clémentine Nebut. Feature model composition assisted by formal concept analysis. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, volume 2, pages 27–37. SciTePress, 2017.

[39] Everton Cavalcante, André Almeida, Thais Batista, Nélio Cacho, Frederico Lopes, Flavia C Delicato, Thiago Sena, and Paulo F Pires. Exploiting software product lines to develop cloud computing applications. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, pages 179–187, 2012.

[40] 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.

[41] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42(10):37–43, 2009.

[42] Ned Chapin, Joanne E Hale, Khaled Md Khan, Juan F Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[43] Maverick Chardet, Hélène Coullon, and Christian Pérez. Predictable efficiency for reconfiguration of service-oriented systems with concerto. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 340–349. IEEE, 2020.

[44] Maverick Chardet, Hélène Coullon, Dimitri Pertin, and Christian Pérez. Madeus: A formal deployment model. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 724–731. IEEE, 2018.

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

[46] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability management in software product lines: a systematic review. In *Proceedings of the 13th International Software Product Line Conference*, pages 81–90. Citeseer, 2009.

[47] Lianping Chen and Muhammad Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology*, 53(4):344–362, 2011.

[48] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002.

[49] H Cloud. The nist definition of cloud computing. *National Institute of Science and Technology, Special Publication*, 800(2011):145, 2011.

[50] Maxime Cordy, Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking adaptive software with featured transition systems. *Assurances for Self-Adaptive Systems: Principles, Models, and Techniques*, pages 1–29, 2013.

[51] Maxime Cordy, Xavier Devroey, Axel Legay, Gilles Perrouin, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Jean-François Raskin. A decade of featured transition systems. In Maurice H. ter Beek, Alessandro Fantechi, and Laura Semini, editors, *From Software Engineering to Formal Methods and Tools, and Back*, volume 11865 of *LNCS*, pages 285–312. Springer, 2019.

[52] Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L Wolf. {NaaS}:{Network-as-a-Service} in the cloud. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, 2012.

[53] Hélène Coullon, Ludovic Henrio, Frédéric Loulergue, and Simon Robillard. Component-based distributed software reconfiguration: a verification-oriented survey. *ACM Computing Surveys*, 2023.

[54] Hélène Coullon, Ludovic Henrio, Frédéric Loulergue, and Simon Robillard. Component-based distributed software reconfiguration: A verification-oriented survey. *ACM Comput. Surv.*, 56(1):2:1–2:37, 2024.

[55] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries. In *Generic Programming: International Seminar on Generic Programming Dagstuhl Castle, Germany, April 27–May 1, 1998 Selected Papers*, pages 25–39. Springer, 2000.

[56] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings 3*, pages 266–283. Springer, 2004.

[57] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, 10(1):7–29, 2005.

[58] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 211–220, 2006.

[59] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *11th International Software Product Line Conference (SPLC 2007)*, pages 23–34. IEEE, 2007.

[60] Mohit Dhingra, J Lakshmi, and SK Nandy. Resource usage monitoring in clouds. In *2012 ACM/IEEE 13th International Conference on Grid Computing*, pages 184–191. IEEE, 2012.

[61] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, and Jakub Zwolakowski. Optimal provisioning in the cloud. *Technical Report*, 2013.

[62] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Aeolus: A component model for the cloud. *Information and Computation*, 239:100–121, 2014.

[63] Roberto Di Cosmo, Stefano Zacchiroli, and Gianluigi Zavattaro. Towards a formal component model for the cloud. In *International Conference on Software Engineering and Formal Methods*, pages 156–171. Springer, 2012.

[64] Yucong Duan, Guohua Fu, Nianjun Zhou, Xiaobing Sun, Nanjangud C Narendra, and Bo Hu. Everything as a service (xaas) on the cloud: origins, current and future trends. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 621–628. IEEE, 2015.

[65] Clemens Dubslaff. Compositional feature-oriented systems. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods - 17th Int. Conf. SEFM 2019, Proc.*, volume 11724 of *LNCS*, pages 162–180. Springer, 2019.

[66] Rim El Ballouli. *Modeling self-configuration in Architecture-based self-adaptive systems*. PhD thesis, Université Grenoble Alpes, 2019.

[67] Rim El Ballouli, Saddek Bensalem, Marius Bozga, and Joseph Sifakis. Dr-bip-programming dynamic reconfigurable systems. Technical report, Technical report TR-2018-3, Verimag Research Report, 2018.

[68] Rim El Ballouli, Saddek Bensalem, Marius Bozga, and Joseph Sifakis. Programming dynamic reconfigurable systems. *International Journal on Software Tools for Technology Transfer*, 23(5):701–719, 2021.

[69] 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.

[70] Salman Farhat, Simon Bliudze, and Laurence Duchien. Safe dynamic reconfiguration of concurrent component-based applications. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 108–111, 2022.

[71] Salman Farhat, Simon Bliudze, Laurence Duchien, and Olga Kouchnarenko. Run-time coordination of reconfiguration requests in cloud computing systems. Research Report 9504, Inria, April 2023.

[72] Salman Farhat, Simon Bliudze, Laurence Duchien, and Olga Kouchnarenko. Toward run-time coordination of reconfiguration requests in cloud computing systems. In *International Conference on Coordination Languages and Models*, pages 271–291. Springer, 2023.

[73] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. Modeling and validating dynamic adaptation. In *Models in Software Engineering: Workshops and Symposia at MODELS 2008, Toulouse, France, September 28-October 3, 2008. Reports and Revised Selected Papers 11*, pages 97–108. Springer, 2009.

[74] Gregor Gössler and Joseph Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005.

[75] Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. Providing database as a service. In *Proceedings 18th International Conference on Data Engineering*, pages 29–38. IEEE, 2002.

[76] Anubhav Hanjura. *Heroku cloud application development*. Packt Publishing Ltd, 2014.

[77] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *10th international conference on autonomic computing (ICAC 13)*, pages 23–27, 2013.

[78] James A Hess, E William, and A Novak. Feature-oriented domain analysis (foda) feasibility study kyo c. kang, sholom g. cohen. 1990.

[79] Mike Hinchey, Sooyong Park, and Klaus Schmid. Building dynamic software product lines. *Computer*, 45(10):22–26, 2012.

[80] Sven Jörges, Anna-Lena Lamprecht, Tiziana Margaria, Ina Schaefer, and Bernhard Steffen. A constraint-based variability modeling framework. *International Journal on Software Tools for Technology Transfer*, 14:511–530, 2012.

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

[82] 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.

[83] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, pages 311–320, 2008.

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

[85] Chris Kemp and Brad Gyger. *Professional Heroku Programming*. John Wiley & Sons, 2013.

[86] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[87] A Khalid. Cloud computing technology: services and opportunities. *Pakistan Journal of Science*, 65(3), 2013.

[88] Nane Kratzke. A brief history of cloud application architectures. *Applied Sciences*, 8(8):1368, 2018.

[89] 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.

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

[91] Leonidas Lymberopoulos, Emil Lupu, and Morris Sloman. An adaptive policy-based framework for network services management. *Journal of Network and systems Management*, 11:277–303, 2003.

[92] Frank D Macías-Escrivá, Rodolfo Haber, Raul Del Toro, and Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, 40(18):7267–7279, 2013.

[93] Sunilkumar S Manvi and Gopal Krishna Shyam. Resource management for infrastructure as a service (iaas) in cloud computing: A survey. *Journal of network and computer applications*, 41:424–440, 2014.

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

[95] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[96] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240, 2009.

[97] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 13–22. IEEE, 2005.

[98] Andreas Metzger and Klaus Pohl. Software product line engineering and variability management: achievements and challenges. *Future of software engineering proceedings*, pages 70–84, 2014.

[99] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models@ run. time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.

[100] Christopher Olive. Cloud computing characteristics are key. *General Physics Corporation*, 2011.

[101] Carlos Parra, Anthony Cleve, Xavier Blanc, and Laurence Duchien. Feature-based composition of software architectures. In *Software Architecture: 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings 4*, pages 230–245. Springer, 2010.

[102] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 33–42. IEEE, 2012.

[103] Kai Petersen and Claes Wohlin. A comparison of issues and advantages in agile and incremental development between state of the art and an industrial case. *Journal of systems and software*, 82(9):1479–1490, 2009.

[104] Kai Petersen, Claes Wohlin, and Dejan Baca. The waterfall model in large-scale development. In *Product-Focused Software Process Improvement: 10th International Conference, PROFES 2009, Oulu, Finland, June 15-17, 2009. Proceedings 10*, pages 386–400. Springer, 2009.

[105] Martin Pfannemuller, Christian Krupitzer, Markus Weckesser, and Christian Becker. A dynamic software product line approach for adaptation planning in autonomic computing systems. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 247–254. IEEE, 2017.

[106] Frantisek Plásil, Dusan Balek, and Radovan Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. In *Proceedings. Fourth International Conference on Configurable Distributed Systems (Cat. No. 98EX159)*, pages 43–51. IEEE, 1998.

[107] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer, 2005.

[108] Klaus Pohl, Günter Böckle, Frank van Der Linden, Günter Böckle, Klaus Pohl, and Frank van der Linden. A framework for software product line engineering. *Software Product Line Engineering: Foundations, Principles, and Techniques*, pages 19–38, 2005.

[109] Klaus Pohl and Andreas Metzger. Variability management in software product line engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 1049–1050, 2006.

[110] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

[111] Clément Quinton. *Cloud Environment Selection and Configuration: A Software Product Lines-Based Approach*. PhD thesis, Université Lille 1, 2014.

[112] Clément Quinton, Nicolas Haderer, Romain Rouvoy, and Laurence Duchien. Towards multi-cloud configurations using feature models and ontologies. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 21–26, 2013.

[113] Clément Quinton, Daniel Romero, and Laurence Duchien. Cardinality-based feature models with constraints: a pragmatic approach. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, page 162–166, New York, NY, USA, 2013. Association for Computing Machinery.

[114] Clément Quinton, Daniel Romero, and Laurence Duchien. Cardinality-based feature models with constraints: a pragmatic approach. In *Proceedings of the 17th International Software Product Line Conference*, pages 162–166, 2013.

[115] Clément Quinton, Daniel Romero, and Laurence Duchien. Automated selection and configuration of cloud environments using software product lines principles. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 144–151. IEEE, 2014.

[116] 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.

[117] Aaqib Rashid and Amit Chaturvedi. A study on resource pooling, allocation and virtualization tools used for cloud computing. *International Journal of Computer Applications*, 975(8887):43, 2017.

[118] Eric Rutten, Nicolas Marchand, and Daniel Simon. Feedback control as mape-k loop in autonomic computing. In *Software Engineering for Self-Adaptive Systems III. Assurances: International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*, pages 349–373. Springer, 2017.

[119] Farhat Salman, Simon Bliudze, Laurence Duchien, and Olga Kouchnarenko. Composing run-time variability models. In *Proceedings of The European Conference on Software Architecture (ECSA 2024)*, 2024. under review.

[120] Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[121] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th International Workshop on Variability Modeling of Software-Intensive Systems*, pages 119–126, 2011.

[122] 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.

[123] Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Automated merging of feature models using graph transformations. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 489–505. Springer, 2007.

[124] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 1063–1075, 2019.

[125] Joseph Sifakis. A framework for component-based construction. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 293–299. IEEE, 2005.

[126] Marianne Simonot and Virginia Aponte. A declarative formal approach to dynamic reconfiguration. In *Proceedings of the 1st international workshop on Open component ecosystems*, pages 1–10, 2009.

[127] Akanksha Singh, Smita Sharma, Shipra Ravi Kumar, and Suman Avdesh Yadav. Overview of paas and saas and its application in cloud computing. In *2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH)*, pages 172–176. IEEE, 2016.

[128] Ian Sommerville, Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Edition: Software engineering. *Instructor*, 2019.

[129] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Extending feature models with relative cardinalities. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 79–88, 2016.

[130] 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.

[131] OASIS Standard. Topology and orchestration specification for cloud applications version 1.0, 2013.

[132] Vijayan Sugumaran, Sooyong Park, and Kyo C Kang. Software product line engineering. *Communications of the ACM*, 49(12):28–32, 2006.

[133] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, 47(1):1–45, 2014.

[134] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[135] Frank J Van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.

[136] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition, 2008.

[137] Vijay Varadharajan and Udaya Tupakula. Security as a service model for cloud environment. *IEEE Transactions on network and Service management*, 11(1):60–75, 2014.

[138] Anthony T Velte Toby J Velte and Ph D Robert Elsenpeter. *Cloud computing*. 2010.

[139] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *11th Int. Conf. SPLC 2007*, pages 233–242. IEEE Computer Society, 2007.

[140] David M Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[141] 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.

[142] Andreas Wittig and Michael Wittig. *Amazon Web Services in Action: An in-depth guide to AWS*. Simon and Schuster, 2023.

[143] Jiyi Wu, Lingdi Ping, Xiaoping Ge, Ya Wang, and Jianqing Fu. Cloud storage as the infrastructure of cloud computing. In *2010 International conference on intelligent computing and cognitive informatics*, pages 380–383. IEEE, 2010.

[144] 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.

[145] Tewfik Ziadi and Jean-Marc Jézéquel. Software product line engineering with the uml: Deriving products. In *Software Product Lines*, pages 557–588. Springer, 2006.