# Design of correct-by-construction self-adaptive cloud applications using formal methods

Trinh Le Khanh


## ▶ To cite this version:

Trinh Le Khanh. Design of correct-by-construction self-adaptive cloud applications using formal methods. Symbolic Computation [cs.SC]. Université de Lille, 2023. English. NNT : 2023ULILB002 . tel-04136838v2

## HAL Id: tel-04136838
## https://theses.hal.science/tel-04136838v2

Submitted on 21 Jun 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Inria center at the University of Lille
CRIStAL laboratory
MADIS doctoral school

# Design of Correct-by-Construction Self-adaptive Cloud Applications using Formal Methods

Presented by:

**Trinh LE-KHANH**

**Thesis Committee:**

| | | |
|---|---|---|
| **Jean-Marie JACQUET** | Professor at University of Namur | Reviewer |
| **Marius BOZGA** | CNRS Research Engineer at Université Grenoble Alpes | Reviewer |
| **Olga KOUCHNARENKO** | Professor at University of Franche-Comté | Jury President / Examiner |
| **Philippe MERLE** | Inria Research Director at Centre Inria de l'Université de Lille | Supervisor |
| **Simon BLIUDZE** | Inria Researcher at Centre Inria de l'Université de Lille | Co-supervisor |

**27 January 2023**

Université de Lille

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

Inria
INVENTEURS DU MONDE NUMÉRIQUE

Centre Inria de l'Université de Lille
Laboratoire CRIStAL
Ecole Doctorale MADIS

# Conception d'applications Cloud Auto-adaptatives Correct-by-Construction à l'aide de Méthodes Formelles

Présenté par:

**Trinh LE-KHANH**

**Comite de Thèse:**

| | | |
|---|---|---|
| **Jean-Marie JACQUET** | Professeur à l'Université de Namur | Rapporteur |
| **Marius BOZGA** | Ingénieur de recherche CNRS à l'Université Grenoble Alpes | Rapporteur |
| **Olga KOUCHNARENKO** | Professeur à l'Université de Franche-Comté | Président du jury/ Examinateur |
| **Philippe MERLE** | Directeur de recherche Inria au Centre Inria de l'Université de Lille | Directeur de thèse |
| **Simon BLIUDZE** | Chercheur Inria au Centre Inria de l'Université de Lille | Co-encadrante |

**27 Janvier 2023**

# Acknowledgements

On my adventures in research, there have been moments of everything, favorable and challenging, but at all times, I have received great help and advice from surrounding people. First, I would like to express my deepest gratitude to my supervisors, Prof. Philippe MERLE and Dr. Simon BLIUDZE. Their patience, timely comments, and continuous guidance and support have made this thesis possible.

In particular, Prof. Philippe helped me point out my mistakes and gave me comments to deal with many problems during my thesis. Working with Simon, I have learned the value of research and how to become a good researcher. I can achieve good results from this study with his enthusiastic advice, precise comments, and encouragement. In addition, discussing with him in weekly meetings helped significantly improve my writing and presenting skills. Without their help, I would have had many difficulties finishing this thesis.

Besides my supervisors, I would like to thank all my thesis committee members for agreeing to review my work. A special thank you to M. Jean-Marie JACQUET and M. Marius BOZGA for being the reviewers ("rapporteurs") of my PhD thesis and Mme. Olga KOUCHNARENKO for being the examiner ("examinateur") of my dissertation. I would also like to extend my warmest thanks to my colleagues and friends in the INRIA-SPIRALS team, Lionel, Pierre, Larisa, Rémy, Antonin, Salman, and Alexandre, for maintaining a very enjoyable atmosphere.

In retrospect, I am indebted to all the people who looked after my first steps in France. Special thanks to Simon, who was my guarantee in processing paper works, and madam Trâm in the doctoral school for supporting me in the beginning and bridging me to people who can help. My life in France has been pleasant, thanks to my Vietnamese friends: Ms. Linh Giang, Kita Bùi, Minh Ngọc, Gia Nghị, and SvLille members, for helping me set up my daily life when I arrived and for other help during the last three years.

Great thanks to Prof. Phạm Ngọc Hùng for supervising my bachelor's thesis, encouraging me during my PhD, and passing on to me the lesson: "Try your best, and you will gain some things in the end". I also thank my UET colleagues: Hảo Nguyễn, Mạnh Hùng, Sơn Nguyễn, and Đình Dương, for your discussions and help.

Last but not least, I express my heartfelt gratitude to my family members, who have always encouraged me to pursue my passion, trusted in me, and shown me the value of knowledge and diligence. Sweetest thanks and a huge hug to my girlfriend, Phương, for her love and patience.

Thank you! / Merci! / Cảm ơn!
Lê Khánh Trình

# Abstract

Correctly coordinating access to cloud resources across concurrent cloud software components is essential to ensure that they satisfy user and system requirements and avoid operational faults and deadlocks. Cloud systems must be able to self-adapt to changes at run time without interruption. Traditional approaches do not separate the code of component computations from their coordination, making it difficult to debug and maintain. Changes in coordination policies require reprogramming the components and affecting other components interacting with them. This Ph.D. thesis aims to ensure that concurrent cloud application entities have the correct access to cloud resources through three main contributions:

- NaturalBIP—a pseudo-natural language for specifying functional requirements. This language is defined through an ontology-driven specification approach. This ontology precisely defines concepts and their relationships in a specific domain. Then, the specifications are written in pseudo-natural templates with placeholders restricted by ontology concepts.

- NaturalBIP Compiler—a compiler to analyze and translate the specifications written in NaturalBIP language into JavaBIP artifacts (i.e., JavaBIP GlueBuilder, data transfers, and safety properties) and BIP connectors.

- An extension of OCCIware with coordination capabilities using JavaBIP. The Finite State Machine (FSM) specification in OCCIware design is used to specify the component's behavior. Then, the coordination between them is established by JavaBIP generated using the NaturalBIP Compiler. The BIP model is computed from the BIP connectors and the configuration model to verify the deadlock-free property using iFinder, a tool for the compositional detection of deadlocks at design time.

With these contributions, I provide a toolchain to develop correct-by-construction self-adaptive cloud applications and conclude this thesis by presenting future perspectives to improve this work.

# Résumé


Il est essentiel de coordonner correctement l'accès aux ressources cloud entre les composants logiciels cloud simultanés pour s'assurer qu'ils satisfont aux exigences des utilisateurs et du système et éviter les pannes opérationnelles et les blocages. Les systèmes cloud doivent être capables de s'adapter aux changements au moment de l'exécution sans interruption. Les approches traditionnelles ne séparent pas le code des calculs des composants de leur coordination, ce qui rend difficile le débogage et la maintenance. Les changements dans les politiques de coordination nécessitent de reprogrammer les composants et d'affecter d'autres composants en interaction avec eux. Cette thèse de doctorat vise à s'assurer que les entités applicatives cloud concurrentes ont le bon accès aux ressources cloud à travers trois contributions principales :

- NaturalBIP—un langage pseudo-naturel pour spécifier les exigences fonctionnelles. Ce langage est défini par une approche de spécification basée sur l'ontologie. Cette ontologie définit précisément les concepts et leurs relations dans un domaine spécifique. Ensuite, les spécifications sont écrites dans des modèles pseudo-naturels avec des espaces réservés limités par des concepts d'ontologie.
- NaturalBIP Compiler—un compilateur pour analyser et traduire les spécifications écrites en langage NaturalBIP en artefacts JavaBIP (c'est-à-dire JavaBIP GlueBuilder, transferts de données et propriétés de sécurité) et connecteurs BIP.
- Une extension d'OCCIware avec des capacités de coordination utilisant JavaBIP. La spécification Finite State Machine (FSM) dans la conception OCCIware est utilisée pour spécifier le comportement du composant. Ensuite, la coordination entre eux est établie par JavaBIP généré à l'aide du compilateur NaturalBIP. Le modèle BIP est calculé à partir des connecteurs BIP et du modèle de configuration pour vérifier la propriété sans blocage à l'aide d'iFinder, un outil de détection compositionnelle des blocages au moment de la conception.

Avec ces contributions, je propose une chaîne d'outils pour développer des applications cloud auto-adaptatives correctes par construction et conclus cette thèse en présentant des perspectives futures pour améliorer ce travail.

# Contents


**List of Figures** ix

**List of Tables** xi

**1 Introduction** 1

1.1 Context . . . 1
1.2 Objectives . . . 2
1.3 Contributions . . . 3
1.4 List of Tools and Language . . . 4
1.5 Structure of the Thesis . . . 5

**2 State of the Art** 7

2.1 Self-adaptive Cloud Applications . . . 7
2.1.1 Standards for Managing Cloud Resources . . . 9
2.1.2 Frameworks for Managing Cloud Resources . . . 10
2.1.3 OCCIware . . . 12
2.2 Model-Based Approach . . . 15
2.2.1 Requirement Engineering . . . 15
2.2.2 Component-Based Approach . . . 16
2.2.3 Correct-by-Construction Software Development . . . 17
2.3 Used Component-based Frameworks and Tools . . . 18
2.3.1 PBL . . . 18
2.3.2 The BI(P) Framework . . . 19
2.3.3 The JavaBIP Framework . . . 22
2.3.4 iFinder . . . 24
2.4 Discussion . . . 26

**3 Domain-Specific Language for Developing Self-adaptive Applications** 29

3.1 Introduction . . . 29
3.2 NaturalBIP Language . . . 31
3.2.1 Ontology Architecture . . . 31

3.2.2 NaturalBIP Syntax and Semantics . . . 34
3.3 NaturalBIP Compiler . . . 42
3.3.1 Pre-processing . . . 43
3.3.2 Boolean Encoding . . . 46
3.3.3 Dual-Horn clauses generation . . . 47
3.3.4 JavaBIP artifacts generation . . . 53
3.3.5 BIP Connectors generation . . . 56
3.4 Summary . . . 58

**4 Towards Exogenous Coordination of Concurrent Cloud Applications 59**
4.1 Introduction . . . 59
4.2 Motivation & Running Example . . . 60
4.3 Methodology of extending coordination capability of the OCCIware design 63
4.3.1 Concepts for extending coordination capability in the OCCIware design . . . 64
4.3.2 Generate artifacts for verification . . . 68
4.3.3 Integration of JavaBIP into OCCIware implementation . . . 71
4.4 Evaluation . . . 71
4.4.1 OCCIware design and configuration model . . . 72
4.4.2 Generated artifacts from the OCCIware design . . . 72
4.4.3 Verification using iFinder . . . 76
4.4.4 Implementing and Adapting to changes . . . 77
4.5 Summary . . . 77

**5 Experimental Validation 81**
5.1 The overview of Heroku Deployer . . . 81
5.2 The HerokuDeployer microservice design . . . 83
5.2.1 The structure of HerokuDeployer . . . 83
5.2.2 Writing functional requirements in NaturalBIP language . . . 86
5.3 Generating Java artifacts and implementing the Heroku Deployer . . . 87
5.3.1 Artifacts for the verification . . . 87
5.3.2 Artifacts for the implementation . . . 88
5.4 Verifying the deadlock-freedom using iFinder . . . 90
5.5 Running the experiment . . . 93
5.6 Summary . . . 96

**6 Conclusion 99**

**Bibliography 101**

**Appendix Heroku Requirements 111**

# List of Figures


1.1 The process for developing correct-by-construction systems . . . . . . . . 2
1.2 The overview of our contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1 The overview of OCCIware—a model-driven vision to manage Everything as a Service . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.2 Ecore diagram of OCCIware Metamodel [132] . . . . . . . . . . . . . . . . . 14
2.3 BIP connectors with the corresponding sets of interactions . . . . . . . . 21

3.1 Ontology Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.2 Behavioral Ontology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.3 Domain Specific Ontology of the Tracker-Peer communication system . . 34
3.4 NaturalBIP Requirement Ontology . . . . . . . . . . . . . . . . . . . . . . 35
3.5 The decomposition of requirement **TP__01__register** . . . . . . . . . . . 40
3.6 The decomposition of requirement **TP__02__speak** . . . . . . . . . . . . 41
3.7 The process for the generation of JavaBIP artifacts from NaturalBIP requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.8 The finite state machines of Tracker and Peer . . . . . . . . . . . . . . . . 43
3.9 The pre-processing of requirement **HTP** . . . . . . . . . . . . . . . . . . 44
3.10 Collecting quantifiers, actions, and conditions from the pre-processed requirement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

4.1 The overview of *Monitor-Switch* Web application . . . . . . . . . . . . . 62
4.2 Model-Driven Managing Everything as a Service with OCCIwareBIP . . 63
4.3 The OCCIwareBIP design of the *Monitor-Switch* Web application . . . . 65
4.4 Domain-specific ontology for the cloud domain . . . . . . . . . . . . . . . 67
4.5 Update new property in the OCCI metamodel . . . . . . . . . . . . . . . . 67
4.6 The type of action `chooseServer` is `spontaneous` . . . . . . . . . . . . . 68
4.7 Connector $c$ in the tree structure . . . . . . . . . . . . . . . . . . . . . . 70
4.8 The result of the verification using the iFinder tool . . . . . . . . . . . . 77
4.9 The user interface of the *Monitor-Switch* Web application . . . . . . . . . 78

4.10 After the current server reaches the threshold, the subsequent request will be directed to another server . . . . . . 78

5.1 The overview of HerokuDeployer microservice . . . . . . 82
5.2 The overview of the Monitor-Switch Web application . . . . . . 83
5.3 The OCCIwareBIP design of *Heroku Deployer* . . . . . . 84
5.4 Class *DeployerConnector* with generated template code . . . . . . 88
5.5 The result of the verification using the iFinder tool . . . . . . 94
5.6 The result after deploying the application . . . . . . 94
5.7 The deployed Heroku Dyno . . . . . . 95
5.8 Running the microservice *compute* . . . . . . 95
5.9 Running the Web application *Monitor-Switch* . . . . . . 96

# List of Tables


2.1 Element Quantifier . . . 19
2.2 $\mathcal{AI}(P)$ and $\mathcal{AC}(P)$ representations of some basic interaction schemes . . . 22
2.3 List of analysis methods for computing invariants . . . 26

3.1 NaturalBIP grammar . . . 36
3.2 Notations used in our grammar . . . 37
3.3 The derivable `Clauses` contain `Constraints` . . . 37
3.4 Grammar rules of Compound Expression . . . 37
3.5 Element Quantifier . . . 39

4.1 Mapping between the Behavioral Ontology classes and the OCCIware metamodel concepts . . . 66

5.1 The number of lines for writing the constraints to describe the functionality choose Java language and the whole functionalities in deploying a Web application onto a free Heroku Dyno . . . 97

# Chapter 1

# Introduction

## 1.1 Context

As modern software systems, cloud applications are inherently concurrent. Their components run simultaneously and share access to resources such as virtual machines, application servers, or database managers. The cloud applications need to be monitored correctly at run time, and adaptations may be performed following the changes in available resources. When cloud applications run, there is little control over their resource use. The applications must be able to dynamically adapt their behaviors to the changes in cloud resource availability. *Correct coordination of resource access* between concurrent entities of cloud applications is critical for meeting user and system requirements while avoiding operational faults and deadlocks. Cloud applications are self-adaptive by nature, especially when users use resources in a pay-as-you-go manner. Self-adaptive systems [105] are capable of adjusting their behavior to satisfy the expected objectives. A self-adaptive system continuously monitors itself to deal with unforeseen circumstances, such as changes in the system environment, system failures, new requirements, or requirements priority changes [116]. *Continuously monitoring and assessing the correctness* of cloud systems is not trivial in the context of the application's evolution.

In traditional development, the code coordinating entities' access to available resources is interleaved with software components' business functionality. This complicates application maintenance when facing policy changes. Although maintenance can be supported by change impact analysis [82, 111, 112], *this process takes time and effort.* Exogenous models and languages [32] were introduced to deal with this problem. The exogenous approach distinctly separates computation and coordination code. This separation enhances the reusability of components. The advantages of exogenous coordination are supporting and permitting verification techniques to compute components' code and dependencies between them [25].

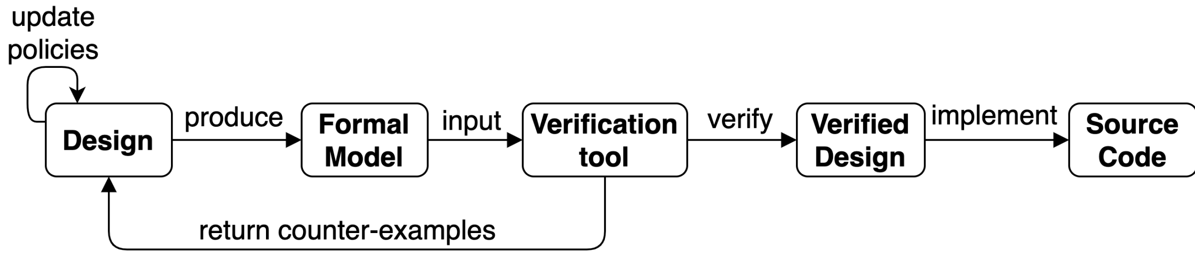


Figure 1.1: The process for developing correct-by-construction systems

System errors found at the late phase of the development process are widespread [77] and might cause enormous consequences in deployment. Fixing these errors frequently or rebuilding the system can be costly and time-consuming. Furthermore, during an application's life cycle, several versions with new features are officially released, or the coordination between cloud components may change according to changing requirements from users and the system [107]. Therefore, *early error detection of system requirements* and fixing them is essential in cloud development.

A method for dealing with this issue is the correct-by-construction (CbyC) approach, a combination of formal methods and incremental developments [49]. In CbyC development, developers create a formal model from the requirements. Based on this model, developers can prove the requirements' consistency and correctness. This approach's advantage is detecting system errors and correcting them early. Hence, minimizing the potential system errors in the implementation phase. Figure 1.1 presents the process for developing correct-by-construction self-adaptive cloud applications. In particular, when designing cloud applications, designers provide a high-level abstract model (i.e., the formal model) of the system. The formal model is analyzed to verify its functionality's correctness through a verification tool, which applies various techniques such as theorem proving or model checking. Then, the verified design is used as a stable foundation for implementing cloud applications. Unfortunately, cloud developers *lack an easy-to-use framework* supporting the correct-by-construction design of self-adaptive cloud applications.

## 1.2 Objectives

The main objective of this thesis is **to provide developers with methodologies and tools to ensure that concurrent cloud application entities access cloud resources correctly**.

Learning to develop cloud applications using correct-by-construction development methodologies is challenging for cloud designers unfamiliar with formal methods. The **first objective** is to *provide cloud designers with means to write unambiguous specifications*

*of the system functionalities.* These specifications can be used to validate the system requirements and design using correct-by-construction techniques.
To this end, I address the following research questions:

- **RQ1**: How do we provide designers with means to write functional specifications in a language that is easy to learn and use?
- **RQ2**: How can these specifications be used to validate the safety properties of the target system?

The **second objective** is to *provide tool support for an end-to-end development flow from specifications to run time enforcement of safety properties.* To reach this objective, we address the following research questions:

- **RQ3**: How to support cloud designers with means to write a high-level abstract design of a cloud application graphically?
  More concretely, cloud designers should be able to describe types of behaviors and define behavioral constraints through specifications written in an easy-to-use language.
- **RQ4**: How to effectively adapt to the change in both the design and implementation phases?

## 1.3 Contributions

This dissertation presents a methodology to design and implement correct-by-construction self-adaptive cloud applications. Figure 1.2 provides an overview of our contributions.

In Chapter 3, we address **RQ1** by proposing NaturalBIP, a pseudo-natural language for specifying functional requirements. This language supports cloud designers in writing unambiguous specifications of system functionalities. To address the **RQ2**, we provide NaturalBIP Compiler, a tool taking the specification written in NaturalBIP language as the input to generate:

- a formal model, which is used as the input for an appropriate tool to ensure that the target system operates safely, and
- Java classes with empty functions and JavaBIP annotations. In particular, each generated Java class represents a component in the OCCIware design. Then, developers write those functions to complete the component's functionalities.

In Chapter 4, we address **RQ3** by extending the OCCIware implementation with the exogenous coordination capability. OCCIware Studio is an OCCI model-driven tool built

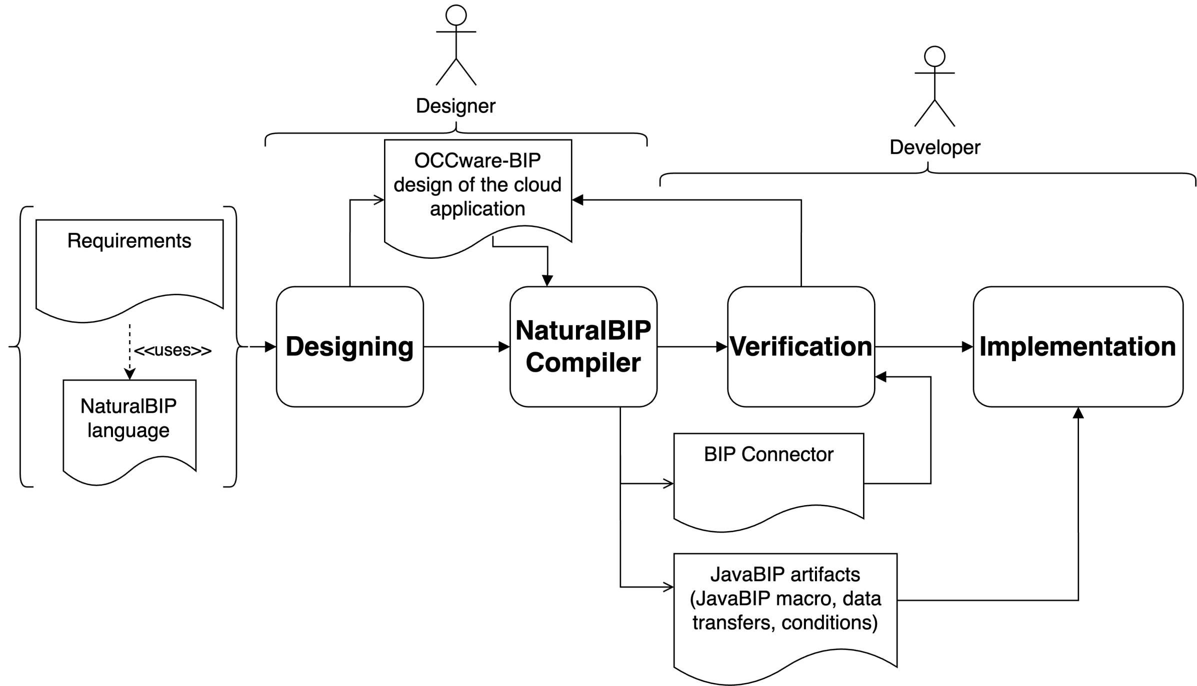


Figure 1.2: The overview of our contributions

around the OCCIware metamodel for designing, verifying, and implementing everything as a service. We start by defining some concepts in the OCCIware framework to write the specification in NaturalBIP language and specify the possible events occurring at run time. We also provide a mechanism to select some specifications within the list of available ones to achieve the customers' current demand. Following the exogenous approach, the computation code of components is independent of the coordination code generated from the selected specifications. By providing a mechanism for selecting policies and generating the corresponding implementation through NaturalBIP Compiler, we help developers quickly update the design and implementation, hence contributing toward answering **RQ4**.

## 1.4 List of Tools and Language

During this thesis, I developed tools and a language to reach the above objectives:

- NaturalBIP is a language to write the specification of functional requirements in a pseudo-natural manner.

- OCCIwareBIP Studio [8] is a platform that supports developers in creating high-level abstract designs of self-adaptive cloud applications.

- NaturalBIP Compiler [7] is a compiler for analyzing and parsing the requirements written in the NaturalBIP language to generate JavaBIP artifacts and BIP connectors.
- GeneratingBIPFile [2] is an implementation of the algorithm (see Section 4.3.2) that computes the BIP model from the BIP connectors and the configuration model.

## 1.5 Structure of the Thesis

This thesis is organized as follows.

- We begin by surveying the literature on methodologies, tools, and frameworks for developing correct-by-construction self-adaptive cloud applications in Chapter 2. Based on the survey, we suggest directions to achieve our main objective.
- In Chapter 3, we propose the NaturalBIP language for writing specifications of functional requirements in a pseudo-natural manner. We introduce an ontology-driven specification approach to restrict the specification to terms an ontology defines. The ontology provides precise concepts and the relationships between them in the domain of the system under design. This ontology is a part of an architecture to employ textual templates for the specifications. The specifications written in these templates can be formalized and verified. We also present the process used by our tool—NaturalBIP Compiler—to generate JavaBIP artifacts and BIP connectors from the design with the selected specifications.
- In Chapter 4, we present the OCCIwareBIP framework. This framework supports cloud developers in creating a high-level abstraction design of a cloud application graphically. To write cloud applications specification using the NaturalBIP language, we define a domain-specific ontology for cloud applications to express specifications from the cloud domain by mapping concepts in the OCCIware metamodel to classes in our proposed behavior ontology. After that, a configuration model is created based on the OCCIwareBIP design. From the configuration model and the OCCIwareBIP design, we propose an algorithm to generate the corresponding BIP model and the instruction for computing the system's invariant using an appropriate tool. The user then defines linear safety properties to be proven. Together with the generated instruction and BIP model, they form the input of iFinder to verify whether these properties hold and whether the design is free from deadlocks.
- In Chapter 5, we present the experimental validation of our toolchain on the example of Heroku Deployer—a microservice to deploy Web applications/microservices on Heroku automatically.

- We conclude the thesis in Chapter 6 with an overview of the contributions and a discussion of the perspectives for future work.

# Chapter 2

# State of the Art

In this chapter, we review the context of the research question stated in the chapter Introduction and related issues. The chapter is organized as follows. The main concept of self-adaptive cloud applications and cloud resource management are introduced in Section 2.1. We discuss the principles and techniques to build correct cloud applications in Section 2.2. We introduce the existing component-based frameworks and tools supporting the thesis in Section 2.3. Finally, we finish the chapter with a discussion of the approaches presented in Section 2.4.

## 2.1 Self-adaptive Cloud Applications

In the software domain, a self-adaptive system adapts itself to the changes in the environment or system policies. This system can self-configure, self-optimize, self-protect, or self-recover [113].

- **Self-configuration.** Components automatically adjust to environmental changes caused by policy changes, such as adding a new component or removing some existing components. This adaptation helps to ensure the performance of the target system.

- **Self-optimization.** Components can self-optimize resource access in response to changing workloads. This improves the performance of resource usage.

- **Self-protection.** This is the mechanism to detect undesired behaviors (e.g., unauthorized access, denial-of-service attacks) and performs appropriate actions to reduce vulnerability.

- **Self-recovery.** When system malfunctions are detected, the related components perform actions to correct the system without disrupting other components.

In our work, we consider two attributes: self-configuration and self-optimization, to develop a self-adaptive cloud application. Once the application is deployed, resource usage and requirements control are limited. It is necessary to safely manage the resources at run time by adapting entities' behaviors in reaction to resource availability changes. Components of cloud applications interact through cloud resources such as virtual machines, networks, storage, application servers, database managers, middleware services, etc. Thereby, cloud resource sharing can lead to cloud resource contention, such as deadlocks, starvation, and race conditions.

- A deadlock is a situation in which two or more components are waiting for each other to perform an action, but none can proceed because they are waiting on the other. This can result in a standstill, where the system becomes stuck and unable to progress.
- Starvation is a situation in which a component or process is prevented from making progress because other components or processes constantly block it. This can result in the component or process being unable to access the resources it needs to complete its work.
- A race condition is a situation in which the outcome of a computation depends on the order in which multiple components or processes access shared resources. This can result in unpredictable and inconsistent behavior in the system.

Several approaches can be used to address coordination issues, including using synchronization mechanisms such as mutex and semaphores to control access to shared resources.

- Mutex (mutual exclusion) is a synchronization mechanism that allows only one component or process to access a shared resource at a time. This mechanism can prevent data/state corruption by ensuring that only one component or process can access the resource at any time.
- Semaphore is a synchronization mechanism that allows multiple components or processes to access a shared resource but limits the number of components or processes that can access the resource simultaneously. This mechanism can ensure that resources are used correctly by limiting the number of components or processes that can access the resource simultaneously.

While these synchronization mechanisms can effectively address coordination issues, they also have limitations. For example, using mutex and semaphore can introduce overhead and complexity into the system and may not be appropriate for all types of systems or situations. Additionally, these mechanisms may not always be sufficient to prevent all coordination issues from occurring.

### 2.1.1 Standards for Managing Cloud Resources

In recent years, cloud computing has become the preferred delivery model for computing resources [23]. Cloud developers can manage the resource using a cloud resource management application programming interface (CRM-API) [91].

Several APIs (Application Programming Interfaces) are available for cloud resource management, depending on each cloud platform. Some common cloud platforms and the APIs they offer for resource management include:

- Amazon Web Services (AWS)[1]: AWS offers a variety of APIs for managing resources on its cloud platform, including the AWS Management Console, the AWS Command Line Interface (CLI), and the AWS SDKs (Software Development Kits).
- Microsoft Azure[2]: Azure offers some APIs for managing resources on its cloud platform, including the Azure Portal, the Azure CLI (Command Line Interface), and the Azure SDKs.
- Google Cloud Platform[3]: Google Cloud offers several APIs for managing resources on its platform, including the Google Cloud Console, the Cloud SDK (Software Development Kit), and the Cloud Client Libraries.
- IBM Cloud[4]: IBM Cloud offers some APIs for managing resources on its platform, including the IBM Cloud Console, the IBM Cloud CLI (Command Line Interface), and the IBM Cloud SDKs.
- Heroku[5] is a cloud platform that enables developers to build, run, and scale applications. It offers several APIs that can be used to manage resources on the platform, including Heroku Platform API, Heroku Connect API, Heroku Builds API, etc.

These APIs allow users to programmatically manage resources such as compute instances, virtual machines, storage, networking, and databases. However, each cloud provider proposes a CRM-API with different concepts and architectures. Therefore, the provision and management of cloud resources face problems such as compatibility while constructing multi-cloud systems with CRM-API and the flexibility of cloud management applications [132]. Several cloud computing standards have been developed to address these issues.

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com/
[3] https://cloud.google.com/
[4] https://www.ibm.com/cloud
[5] https://www.heroku.com/

**CIMI.** The Distributed Management Task Force (DMTF)[6] provides Cloud Infrastructure Management Interface (CIMI) standard [54], which focuses on managing Infrastructure-as-a-Service (IaaS) resource life-cycle by defining a RESTful (REpresentational State Transfer) API.

**OVF.** Open Virtualization Format (OVF) [9] is another standard provided by the DMTF for packaging and describing software appliances running in virtual machines. The OVF package contains one XML file describing hardware and network configuration, disk images, etc.

**CAMP.** Cloud Application Management for Platforms (CAMP) is a standard for managing Platform-as-a-Service (PaaS) resources, provided by Organization for the Advancement of Structured Information Standards (OASIS)[7]. CAMP specifies operations such as administration, monitoring tasks, and life-cycle management [84].

**OCCI.** Open Cloud Computing Interface (OCCI) is an open standard proposed for managing cloud resources [18] from the Open Grid Forum (OGF)[8]. While CIMI, OVF, and CAMP address a specific resource model (i.e., IaaS or PaaS), OCCI manages all the above cloud resources by providing a RESTful API. Thus, cloud developers can specify cloud providers using OCCI Core Model [93]—a resource-oriented model based on OCCI. The OCCI Core Model can be extended for specific purposes and accessed via a REST API [64].

**TOSCA.** Besides OCCI, the OASIS's Topology and Orchestration Specification for Cloud Applications (TOSCA) [34, 48] is a standard for describing the topology or architecture of cloud applications. TOSCA provides a language for expressing cloud applications/services and their relations (i.e., topology). The language can also describe the operations of such applications/services independently of the cloud providers (i.e., the orchestration). The advantage of TOSCA is allowing cloud designers to manage cloud resources from different providers interoperably. While TOSCA focuses on providing a description language, OCCI provides an API for managing cloud resources.

### 2.1.2 Frameworks for Managing Cloud Resources

Various frameworks have been developed based on cloud standards and language to provide a unified API for managing cloud resources across multi-cloud services.

[6] https://www.dmtf.org/
[7] https://www.oasis-open.org/
[8] https://www.ogf.org/ogf/doku.php

**Frameworks relying on the OCCI standard.** Some frameworks that support managing OCCI resources have been provided. For example, rOCCI [12], pySSF [13] and pyOCNI [10], OCCI4Java [11], and erocci [5] are implementations relying on Ruby, Python, Java, and Erlang, respectively. Although the OCCI standard supports all cloud resources, these frameworks support a specific resource model, mainly IaaS. By contrast, *OCCIware is a model-driven vision for managing Everything as a Service (XaaS)* [106, 132]. It enables modeling any type of resource by providing capabilities to design, validate, implement, deploy, and manage XaaS using OCCI.

Similar to the OCCIware metamodel, CompatibleOne Resource Description System (CORDS) [130] describes cloud applications on top of the OCCI standard. This model can describe IaaS and PaaS resources or be extended as domain-specific models for managing the resources [128]. The Advanced Capabilities for CORDS (ACCORDS) platform executes CORDS models for describing and validating user requirements, managing provision plans, and delivering cloud services.

**Frameworks relying on the TOSCA standard.** Several research and frameworks have been proposed to exploit the TOSCA standard, such as Cloudify [3], Alien4Cloud [1], TosKer [42], TOSCAMP [17], OpenTOSCA [33], TOSCA Studio [48] TORCH [123], etc.

- Cloudify [3] and Alien4Cloud [1] provide their own proprietary Domain-Specific Language (DSL) based on TOSCA for modeling and deploying cloud applications. However, providing new DSLs with constraints makes templates not portable to various frameworks.

- By separating the definition of cloud applications and containers, TosKer [42] allows flexibly managing cloud systems consisting of containers and running applications. On the other hand, the separation also increases the complexity of the usage.

- TOSCAMP [17] is the combination of TOSCA and CAMP for orchestrating multi-cloud applications using policies. Cloud designers must declare both deploying applications and policies to ensure those policies are applied.

- OpenTOSCA [33] is a platform for deploying and managing TOSCA-modeled cloud applications. It converts TOSCA descriptions into executable actions and sends these actions to the cloud through the corresponding API.

- TOSCA Studio [48] is a model-driven cloud orchestration framework based on the combination of TOSCA and OCCI. This framework maps the TOSCA-based description into the OCCI meta-model through Ecore meta-modeling. However, when TOSCA updates a new resource type, mapping it into OCCI is a challenge.

- TORCH [123] converts the TOSCA model of the application into the BPMN workflow and dataflow models. These models specify the operations that can be executed using the BPMN engine. By separating the provisioning workflow from the invocation of the cloud services enforcing the provisioning, it takes the benefits of maintainability and scalability.

**Other frameworks.** Deltacloud [4] is an implementation of the CIMI standard. It contains API servers and drivers for several cloud providers. Deltacloud supports constructing an interoperable solution by wrapping several clouds using the corresponding driver. However, it has been retired since 2015.

Claudia [114] is a service management system that allows deploying and scaling services automatically based on the infrastructure and service status. Claudia defines these services using the Service Description File (SDF), whose syntax is based on the OVF standard. ASCETiC [62] is an open architecture proposed in the homonymous EU project to optimize energy cost-effectiveness in multi-cloud. It uses the OVF specification to express information about deploying Virtual Machines on IaaS providers.

CloudML [41, 63] allows describing cloud services and application components in provisioning cloud resources. However, CloudML is just a language to specify the deployment and management concerns by textual syntax. ClOud sOlution design tooL (COOL) [98] supports cloud developers in the design of cloud architectures from high-level requirements. The COOL architectures can be specified using CloudML and deployed via OCCIware.

In this thesis, we choose OCCIware because it supports managing Everything as a Service (XaaS). The OCCIware provides a means for specifying the behaviors associated with OCCIware entities as a Finite State Machine (FSM). These FSM specifications are already available for cloud infrastructure elements such as virtual machines, networks, and storage and the cloud applications elements such as databases. The specifications make it possible to guide the deployment of cloud applications based on available cloud resources. While such FSMs in the OCCIware models can be leveraged to monitor and coordinate the activities of the corresponding entities, *there are currently no such mechanisms available.*

### 2.1.3 OCCIware

Our OCCIwareBIP framework (see Chapter 4) relies on OCCIware [132]—an OCCI-based model-driven cloud resource management framework. This framework allows cloud architects to construct cloud computing modeling frameworks that target specific cloud domains, such as infrastructure management, elasticity management, etc. Using a simple resource-oriented metamodel can deal with any kind of resource-based software. It also

drastically reduces development time by supporting the *Models@run.time* approach [35] and code generation [106]. In the development phase, the OCCIware approach relies on model-driven engineering (MDE), which allows for raising the abstraction level of a system as a model adhering to a metamodel defines the modeling language. The OCCIware approach consists of (i) *OCCIware Studio* (yellow boxes) and (ii) *OCCIware Runtime* (purple boxes), as shown in Figure 2.1.

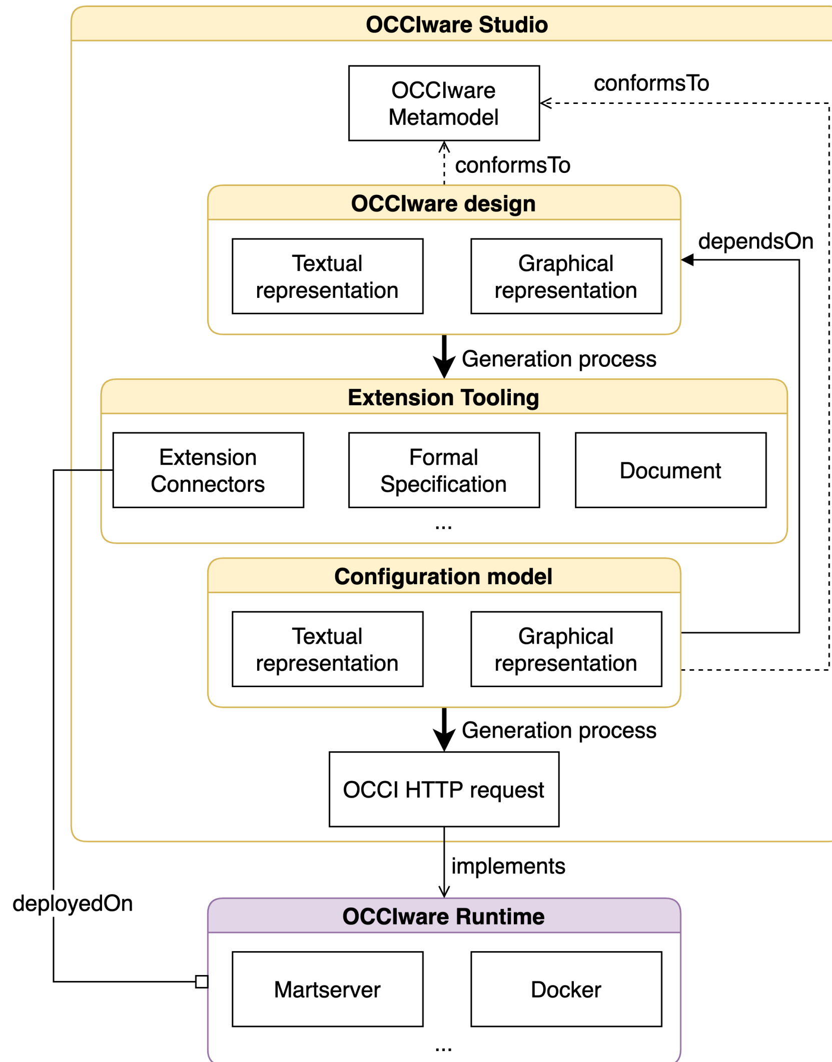


Figure 2.1: The overview of OCCIware—a model-driven vision to manage Everything as a Service

**OCCIware Studio.** OCCIware Studio is built on top of Eclipse to provide a means to design the resource of cloud applications (i.e., OCCIware design). The OCCIware design can be represented both textually and graphically to describe components conforming to *OCCIware Metamodel* concepts (Figure 2.2) and the system's constraints. `Resource`,

`Link`, `Entity`, `Kind`, `Mixin`, `Action`, `Category`, and `Attribute` are eight OCCI core concepts colored in gray. `Resource` is the center of the OCCI Core Model, and it owns a set of `links`, which describe the relationship between two resources. `Entity` is an abstract concept and is typed by a `Kind` and could be annotated by several `Mixin` instances. While `Kind` indicates the immutable type of the OCCI entity, `Mixin` represents cross-cutting attributes and actions that can be added to an OCCI entity. A `Kind` or Mixin contains a set of actions describing specific behaviors. `Kind`, `Mixin`, and `Action` inherit from `Category` to identify themselves uniquely by using a scheme and a term and owning a set of `attributes`. The remaining classes can be found in [132].

Figure 2.2: Ecore diagram of OCCIware Metamodel [132]

After creating a OCCIware design, developers can trigger the *generation process* to generate a set of *Extension Tooling* artifacts, including *Documents*, *Formal Specifications*, *Extension Connectors*, etc. *Extension Connectors* are Java classes containing templates for the necessary business code of the extension's components. Developers then complete these templates to maintain the synchronization between the designed configurations and running ones on the *OCCIware Runtime*.

On the cloud developers' side, they can create a configuration model conforming to the *OCCIware Metamodel* and depending on OCCIware design. To deploy and manage this configuration model in the cloud, applications send OCCI HTTP requests to the OC-CIware Runtime extended with the deployed *Extension Connector*. OCCIware toolchain supports generating a cURL-based script, which interacts with OCCIware Runtime via both OCCI HTTP Protocol [100] and OCCI Text Rendering [59] to instantiate OCCI

entities into an offline OCCI-compliant runtime. The OCCIware Runtime then invokes the appropriate *Extension Connector* to create corresponding resources.

**OCCIware Runtime.** *OCCIware Runtime* is a generic OCCI-compliant *models@run.time* support and includes a cloud resource container and tools for deployment, execution, and supervision of Everything as a Service (XaaS).

The models in OCCIware Studio provide abstractions for managing cloud resources but *lack the ones related to coordination*, which are synchronization constraints that enforce restrictions on the interaction patterns between components [53].

## 2.2 Model-Based Approach

The system design aims to define the system's architecture, components, or data to satisfy requirements [45]. Validating the design is needed to ensure that the specification is valid for the implementation. Natural languages are ambiguous [115]; hence, they are not suitable for the specification of requirement to create a quality design. Formal model-based development is the process of implementing the system correctly. In this process, requirements are transformed into formal specifications [26], which describe the system's structure and behaviors together with external stimuli [133]. Then, the formal specification will be used to verify some properties, such as consistency or completeness.

### 2.2.1 Requirement Engineering

An ontology-driven specification is an approach for formulating system requirements in natural language. Numerous research on requirement engineering based on ontology [52], including representing domain knowledge [55, 71, 127, 121, 119], formalizing requirements [85, 134], verifying some properties of the requirements (e.g., consistency [129, 68, 97], completeness [47, 57, 99]), or validating requirements [129, 85, 90]. In particular, starting from an *upper ontology* describing the general concepts for using in multiple contexts, developers define the *domain-specific ontology* to define domain-specific concepts and their relationship.

Pattern-based specifications are textual templates filled by defined ontology concepts. Restricting the specification to terms from an ontology removes the ambiguity of the natural language. To ensure a unique interpretation for each specification, some approaches employ requirement boilerplates [90, 96, 88]. However, the more boilerplates, the more effort, and time developers need to learn and apply them correctly. In our work, we provide templates, which are structures with placeholders that ontology concepts can replace. By defining the semantics of each template, we ensure the unique interpretation

of each specification and flexibility in writing them. Therefore, reducing the effort and time to learning our language.

*Ontology-Driven Conceptual Modeling* (ODCM) [125] is the application of ontological theories to improve the theory and practice of *conceptual modeling*. Conceptual modeling represents aspects of the physical world using models for communicating, learning, and problem-solving between users. The ability to detect and correct errors depends on the *conceptual models*' quality. However, many conceptual models lacked an adequate specification of terminologies and semantics of the underlying models, which led to the inconsistency of knowledge's interpretation and use [73]. Therefore, ontologies were introduced to provide a foundation for conceptual modeling. *Ontology* is "the set of things whose existence is acknowledged by a particular theory or system of thought" [79]. Ontology can be used to evaluate conceptual modeling languages of frameworks (e.g., Web Ontology Language (OWL), Unified Modeling Language (UML)) [74], become the theoretical foundations of a conceptual model [103, 75], or improve semantic interoperability [70].

### 2.2.2 Component-Based Approach

Most programs are implemented using *endogenous* [32] coordination, where coordination codes are incorporated within computational codes of components. For complex distributed systems, it is hard to debug and maintain because developers must recheck all related components codes and other components interacting with them. Meanwhile, in the *exogenous coordination* [32] approach, each component implements its functions by only computation codes and interacts with the others through external ports. This high level of abstraction makes it easy to reuse components to build large-scale systems like cloud applications and tackle the difficulty in debugging and maintaining of the development following the endogenous coordination approach.

The computation and coordination are separated in the *component-based design*, where the computational entities are the system's components. Connectors coordinate the interaction among these components. Some formal approaches for modeling, compositing, and analyzing connectors include the algebra of stateless connectors [43], the Tile model [66], nets with boundaries [120], process calculi, REO [21], and BIP [28].

- **The algebra of stateless connectors.** Bruni et al. [43] present an algebra consisting of five classes of basis stateless connectors. Then they used graphs to describe concrete structures, tiles to represent operational and observational semantics, and tick-tables to provide denotational semantics.

- **The Tile model.** The Tile model [66] provides a flexible and appropriate semantic configuration for concurrent systems [44, 61]. This configuration defines operational

and abstract semantics to suitable connectors classes in the algebra of stateless connectors.

- **Process calculi.** Calculi such as CSP [78], CCS [95], the $\pi$-calculus [117], process algebras [31, 30, 65], and the actor model [15] are computation models to handle the complexities in constructing concurrent systems. Capizzi et al. [46] introduced the aspect-oriented technique to redesign a distributed program by altering only the coordinating aspects while keeping the computational code untouched. Scholten et al. [118] proposed a variant of the $\pi$-calculus that allows different processes to communicate and impose exogenous coordination through user-defined channel types.

- **The Reo coordination model.** Reo Scripting Language (RSL) [21] is a declarative channel-based language supporting exogenous coordination. This language is used with Constraint Automata Reactive Module Language (CARML) in Vereofy toolkit [25] to model the system. Vereofy also supports model checking of specific components and composite systems with safety properties written in Linear Time Logic (LTL) [108] and Branching-time Stream Logic (BTSL) [83].

- **The BIP framework.** Behavior-Interaction-Priority (BIP) [27, 36] is a framework providing a mechanism for coordinating concurrent components. *Correctness-by-construction* is a significant feature of BIP. Its composability can be used to preserve the deadlock-freedom of the underlying behaviors using appropriate tools such as iFinder [6]. JavaBIP [37, 92] is an open-source Java version of the BIP framework that relies on Java annotations, component APIs, and specification files to coordinate concurrent components.

Our work is based on BIP and JavaBIP frameworks to support designing, verifying, and implementing correct-by-construction self-adaptive cloud applications.

### 2.2.3 Correct-by-Construction Software Development

System errors found at the late phase of the development process are popular [77]. Undetected system errors in the requirement analysis cause enormous consequences in the deployment. Fixing these errors or rebuilding the system is costly and time-consuming. The Correctness by Construction (CbyC) approach is a solution to address this problem. The CbyC combines formal methods and incremental developments [49]. It starts by constructing a formal model from the system requirements. This model can be used as the input for model checkers or theorem provers to validate and verify whether the design meets its specification and whether the specification meets the requirements. This process has been applied in industry and demonstrated its effectiveness for decreasing

errors and increasing productivity in developing safety-critical applications [76, 14] or distributed systems (e.g., cloud applications) [58].

In the implementation phase, developers choose the appropriate language to implement the design depending on the target system. The selected language must have supporting frameworks/tools to analyze and verify the correctness effectively. For example, BIP and JavaBIP frameworks are suitable for developing cloud applications due to their rigorous and unambiguous semantics. In particular, BIP allows the creation of an executable model to verify whether the system under design satisfies a specific property (e.g., the deadlock-freedom). To ensure the high-level abstraction of the model, only relevant actions are represented as FSM's transitions. All the allowed synchronizations (i.e., the interactions) between the concurrent components are specified in BIP *glue*.

As modern software systems, cloud applications are inherently concurrent. Thus, CbyC is an appropriate approach to reach our objective. In summary, the advantages of CbyC are proven by the following:

- It addresses defects early when the changes are cheap. The testing process is to confirm that the target system works instead of debugging it.

- It ensures the safety properties of the system through verification.

## 2.3 Used Component-based Frameworks and Tools

This section introduces frameworks and tools that are important to this work, including PBL - an open-source library to handle Boolean formulas, BIP and JavaBIP frameworks, and the iFinder - the model checker used to verify the deadlock-freedom of the target system.

### 2.3.1 PBL

PBL[9] is a general-purpose Python library for working with Boolean formulas. It can parse and manipulate standard DIMACS (i.e., Discrete Mathematics and Theoretical Computer Science) files and a proprietary language for the formulas. This library allows programmers to quickly complete binary decision diagrams (BDD) and SAT projects by focusing on core algorithms rather than Boolean expressions. Furthermore, the code is written with readability in mind, so algorithms (introduced in [40]) and rendering methods can be easily read, inspected, or modified. The Boolean formulas can be represented as a recursive dictionary or as a list of lists with the supporting types described in Table 2.1:

[9] https://github.com/tyler-utah/PBL

Table 2.1: Element Quantifier

| Type | Explanation | Corresponding symbol |
|---|---|---|
| const | constant type | `True` or `False` |
| var | variable type | variables with the name including text and numbers |
| neg | negative type | ∼ |
| and | operator "and" | & |
| or | operator "or" | \| |
| xor | operator "exclusive or" | `XOR` |
| impl | operator "imply" | `=>` |
| eqv | operator "iff" | `<=>` |

"var" can be an expression or a variable. If it is a variable, it has a string name and a number for converting to conjunctive normal form (CNF). In this form, the negative values denote the negation of corresponding variables.

PBL provides functions to convert the Boolean formulas to the Negation Normal Form (NNE) or CNF. Listing 2.1 shows an example with a Boolean formula (line 2) and its corresponding CNF clauses (line 4).

Listing 2.1: A Boolean formula and its corresponding CNF clauses

```
1  // Boolean formula
2  ((x1 => y1 & y2) & (x2 => y2 & y3)) & ~(x3 => x4)
3  // CNF clauses
4  (~x1 | y1) & (~x1 | y2) & (~x2 | y2) & (~x2 | y3) & x3 & ~x4
```

In our work, we extend PBL to compute dual-Horn clauses, which are used to generate coordination codes in BIP and JavaBIP (see Section 3.3 of Chapter 3).

### 2.3.2 The BI(P) Framework

Behavior-Interaction-Priority (BIP) [28] 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).

The first layer (*Behavior*) presents atomic components with fixed activities considered ports, which are pairwise distinct. The components are modeled as automata, in which their transitions are labeled by sets of ports.

**Definition 2.3.1 (Component)**

A component $C = (S, P, \rightarrow)$ is a transition system where $S$ is the set of states, $P$ is the set of ports, and $\rightarrow \subseteq S \times 2^P \times S$ is the set of transitions.

For all $s$, $s' \in S$ and $p \in P$, we write $s \xrightarrow{p} s'$ to denote the labeled transition $(s, p, s')$ $\in \rightarrow$. It means $a$ is enabled in $s$ (i.e., $s \xrightarrow{p}$) iff $s'$ exists.

The second layer (*Interaction*) defines a mechanism to coordinate components. Interactions are sets of *ports* that determine the allowable synchronizations between components.

**Definition 2.3.2 (Interaction)**

Let $P$ be the set of ports, an interaction over $P$ is a non-empty set $a$, such that $a \subseteq P$.

In [38], the authors defined $\mathcal{AI}(P)$ to provide a clear and convenient notation for manipulating sets of interactions.

Priorities in the third layer of BIP (*Priority*) establish scheduling limits and resolve conflicts simultaneously when many interactions are enabled. We named *Glue* refers to the interaction and priority layers together. In brief, the BIP engine drives the execution of a BIP system by cyclically applying the following protocol:

1. At the current state, each component sends to the BIP engine all the possible transitions;

2. The BIP engine selects an interaction that meets the glue requirements, conducts the data transmission, and notifies all components involved;

3. The notified components perform the functions related to the corresponding transitions.

Connectors are used to define an interaction model in a structured manner. Simon et al. proposed a *causal semantic* [39] for the *Algebra of Connectors* $\mathcal{AC}(P)$ [38], which has the syntax is defined as follows:

$$
\begin{aligned}
&s ::= [0] \mid [1] \mid [p] \mid [x] \text{ (synchrons)} \\
&t ::= [0]' \mid [1]' \mid [p]' \mid [x]' \text{ (triggers)} \\
&x ::= s \mid t \mid x \cdot x
\end{aligned}
\tag{2.1}
$$

where $p \in P$, '$\cdot$' is a binary operator (called *fusion*), and brackets '$[\cdot]'$' and '$[\cdot]$' are unary *typing* operators.

In $\mathcal{AI}(P)$, the fusion operator is synchronization, and a connector is the set of allowed interactions based on two kinds of connected ports: *synchron* or *trigger* (Figure 2.3a). The semantic of $\mathcal{AC}(P)$ is the function $|\cdot| : \mathcal{AC}(P) \rightarrow \mathcal{AI}(P)$ defined by the following

rules:

$$
\begin{aligned}
&|p| = p, \\
&\left|\prod_{i=1}^{n}[x_i]\right| = \prod_{i=1}^{n}|x_i|, \\
&\left|\prod_{i=1}^{n}[x_i]' \cdot \prod_{j=1}^{m}[y_j]\right| = \sum_{i=1}^{n}|x_i| \prod_{k\neq i}(1+|x_k|)\prod_{j=1}^{m}(1+|y_j|),
\end{aligned}
\tag{2.2}
$$

where $\prod$ is the synchronization operator and $\sum$ is the synchronization operator of $\mathcal{AI}(P)$ for $p \in P \cup \{0,1\}$ and $x_1, \ldots, x_n, y_1, \ldots, y_m \in \mathcal{AC}(P)$. If all connected ports are synchrons, this is a *rendezvous* synchronization, i.e., a connector defines exactly one interaction comprising all its ports (Figure 2.3b). If there are one or more trigger ports, this is a *broadcast* synchronization, i.e., The connector describes the interactions consisting of all non-empty subsets of the connected ports containing one or more trigger ports(Figure 2.3c).

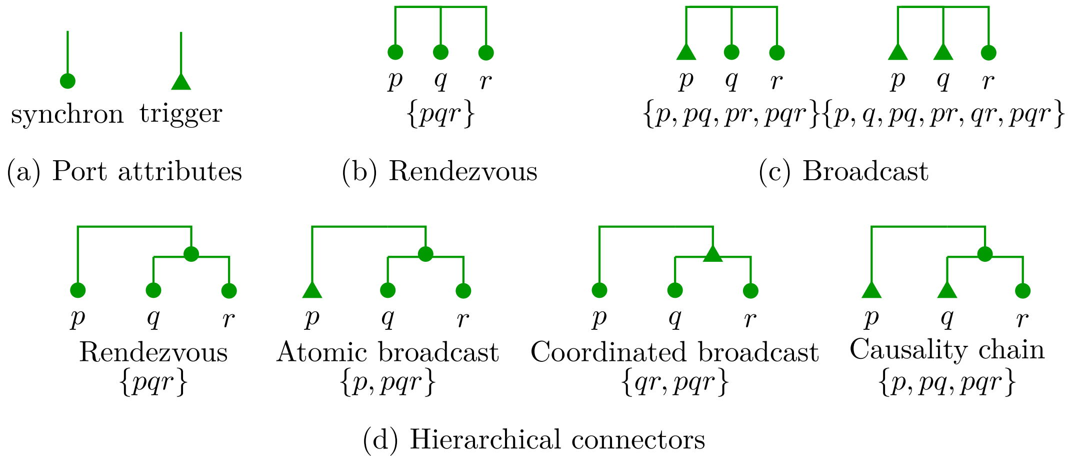


Figure 2.3: BIP connectors with the corresponding sets of interactions

In the hierarchical connector, the interaction from each subconnector forms an allowed interaction according to the port labeling of the connector nodes. For instance, the causal chain connector in Figure 2.3d has a trigger port $p$ and a synchron, which is a binary broadcast subconnector $q$▸——•$r$. Thus the causal chain connector allows interactions are the single port $p$ and any combination of $p$ with some interaction of $q$▸——•$r$. The subconnector $q$▸——•$r$ allows interactions $q$ and $qr$. As a result, the allowed interactions of the causal chain are $\{p,\ pq,\ pqr\}$. Table 2.2 shows the representation in $\mathcal{AI}(P)$ (column 2) and $\mathcal{AC}(P)$ (column 3) for some interaction schemes of Figure 2.3.

Table 2.2: $\mathcal{AI}(P)$ and $\mathcal{AC}(P)$ representations of some basic interaction schemes

| | $\mathcal{AI}(P)$ | $\mathcal{AC}(P)$ |
|---|---|---|
| Rendezvous | $pqr$ | $pqr$ |
| Broadcast | $p(1+q)(1+r)$ | $p'qr$ |
| Atomic broadcast | $p(1+qr)$ | $p'[qr]$ |
| Coordinated broadcast | $qr(1+p)$ | $[qr]'p$ |
| Causality chain | $p(1+q(1+r))$ | $p'[q'r]]$ |

### 2.3.3 The JavaBIP Framework

JavaBIP [37] is a Java implementation of the BIP framework. Component *behaviors* in JavaBIP can be represented by a Finite State Machine (FSM), which has a finite number of states and a finite number of transitions between them. JavaBIP framework uses Java annotations associated with class, method, and parameter declaration to represent the FSMs. In particular, there are six kinds of annotations, including:

- *@ComponentType*: the annotation declares a component type with the FSM's name and initial state. For example, the below code illustrates a class `Peer` with the annotation that defines the component's name as `Peer`, and the initial state of its FSM is *PeerInit*.

```
@ComponentType(initial="PeerInit", name="demo.Peer")
public class Peer{ ... }
```

- *@Ports* and *@Port*: these annotations declare the name and the type of transition and usually be set before defining *ComponentType*. There are three types of FSM transformations: executable, spontaneous, and internal. The enforceable transitions are controlled by the engine through the notification of executors at each execution cycle. Spontaneous transitions are considered environmental events of a component. Finally, internal transitions allow a component to execute a behavior immediately. Only enforceable transitions are used for synchronization between components.

```
@Ports({
    @Port(name="speak", type=PortType.enforceable)
  , @Port(name="listen", type=PortType.enforceable)
  , @Port(name="register", type=PortType.enforceable)
  , @Port(name="unregister", type=PortType.enforceable)
})
@ComponentType ...
```

- *@Guard*: this annotation is used for boolean methods to define the guard of transitions.

```
@Guard(name="is_registered_to")
public boolean is_registered_to(@Data(name="Tracker2Peer_data") String trackerID)
{ ... }
```

- *@Transition*: declares a transition of an FSM specified by event name, `source` and `target` states, and a guard expression (if necessary). Guard expression of a transition can be a single guard or comprise a set of guards using three logical operators: negation '!', the conjunction '&', and disjunction '|'.

```
@Transitions({
    @Transition(name="speak", source="PeerInit", target="Registered", guard="
        is_registered_to")
})
public void speak(){ ... }
```

- *@Data*: this annotation can be used in two aspects as follows:
  - To define data provided by components, which is declared by the non-void method (lines 1 and 2 in the below block), or
  - To define data required by components, which is used as method parameters (line 5).

```
1 @Data(name="Tracker2Peer_data", accessTypePort=AccessType.any)
2 public String trackerID() { ... }
3
4 @Guard(name="is_registered_to")
5 public boolean is_registered_to(@Data(name="Tracker2Peer_data") String trackerID)
6 { ... }
```

In [39], the authors demonstrate that the interactions set defined by a BIP connector can be characterized by a Boolean formula that is a conjunction of implications of the form:

$$p \Rightarrow a_1 \vee \ldots \vee a_n \quad (2.3)$$

with port $p$ is considered as *effect* and each $a_i$ ($i \in [1, n]$), being a conjunction of several ports, is considered *causes*. Obviously, for $p$ to engage in an interaction, at least one $a_i$ for $i \in [1, n]$, in which all the belonging ports participate. Therefore, in an interaction, $p$ can participate if there is some participation of $a_i$. To specify the constraint (2.3) in JavaBIP, one uses the macro notation:

$$p \quad \textbf{Require} \quad a_1; \ldots; a_n$$

For example, the encodings of Broadcast 2 (Figure 2.3c) is

$$r \quad \textbf{Require} \quad p; q$$

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

$$p \quad \textbf{Accept} \quad a, \qquad \text{which formally means} \qquad p \Rightarrow \bigwedge_{\substack{q \in P \setminus a \\ q \neq p}} \overline{q} \tag{2.4}$$

where $P$ is the set of all ports of all the BIP components in the system.

In JavaBIP, the *data transfer* is performed before the execution of the transition. At present, JavaBIP does not implement any *priority* except the so-called *maximal progress* priority applied by default to all JavaBIP systems, where interactions larger in terms of set inclusion are prioritized.

### 2.3.4 iFinder

iFinder [102] is the improvement of D-Finder [29], a tool for verifying component-based systems described in the BIP language. iFinder proposed a compositional verification technique to avoid the combinatorial explosion of component-based systems. It uses interaction invariants for components instead of searching for adequate assumptions. The interaction invariants are considered "cooperation tests" [20] because they allow subtracting states infeasible by the semantics.

iFinder consists of two highly connected tools[10]: *ifinder* and *ichecker*. The first tool computes invariants for each component using a specific "analysis" method as described in the ".inv" file. The latter gathers all the invariants and uses them to prove properties. It uses computed invariants in *the ".inv"* file and properties described in the "*.pro*" file to collect information such as the properties to be proven, the components on which the properties shall apply, the BIP model, etc. Then, *ichecker* generates an SMT file containing the conjunction of the invariants and the negation of the safety properties as input to the Z3 SMT solver [50]. If the result is unsatisfied, the properties are invalid, and the Z3 solver returns a counter-example. Developers must refine the design and repeat the process until satisfactory results. The input of the iFinder tool consists of three components as follows:

1. **BIP model** (*.bip* file): contains the information of the components extracted from the design and the configuration file.

[10] https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/bip/IFinder/-/tree/real-time-marius

Listing 2.2: The structure of the BIP model

```
1  package <package_name>
2    port type <Port_type>()
3    atom type <Component_name> ([parameters])
4      export port <Port_type> <Port_name>()
5      place <state_name>+
6       initial  to <state_name>
7      on <Port_name> from <state_name_1> to <state_name_2> do {...}
8    end
9    connector type <Connector_name>(<Port_type> p1, <Port_type> p2, ..., <Port_type> pn)
10     [export port ep()]
11     define p1' p2 ... pn
12   end
13   compound type <package_name> Compound()
14     component <Component_name> <Component_instances>+
15     connector <Connector_name> <Connector_instance>()
16   end
17 end
```

Listing 2.2 shows the template of a BIP model with the elements in "<..>" are mandatory placeholders and " [..] " are optional. The BIP model starts by declaring the package name (i.e., the name of the target system). For example, "`package monitorswitch`". After that, users define the port type used in the whole system. By default, we set it as "`Port`" (line 2). Each component has an FSM presented by an "`atom type`" (line 3), which uses "*port*" and "*place*" to declare the FSM's transitions and states, respectively (lines 4 and 5). It determines the initial state by the keyword "`initial to`" (line 6). Then, all the transitions of the FSM are defined as the pattern in line 7. Each connector is defined by a function "`connector type`" with a list of child node ports (lines 9-12). If a node is a trigger, we encode it following a symbol "prime" (i.e., '). If the considering connector is a subconnector of a bigger connector, it has an "*export port*" to interact with other ports at the same level of the connector tree. Finally, in function "`compound type`" (lines 13-16), we declare concrete instances of each atom type (i.e., component) and the concrete connectors generated from the component instances and "`connector type`".

2. **Invariants instructions** (*.inv* file): it defines instructions for computing invariants using the analysis method. In the verification step, the invariants will be computed based on the defined analysis methods specified for components. Table 2.3 shows some analysis methods for computing invariants.

   Taking the command "—at Switch —a the atom—control" as an example, it illustrates an instruction that the component (`-at Switch`) will be analyzed by `atom-control` analysis method.

Table 2.3: List of analysis methods for computing invariants

| Analysis methods | Explanation |
|---|---|
| `trap` | trap invariants in Petri nets |
| `liner` | linear invariants in Petri nets |
| `control-reachability` | control reachability (ignoring data / clocks) |
| `zone-reachability` | full zone reachability, if timed automata |
| `atom-control` | (only for atomic components) direct state/place enumeration |
| `interaction-time` | (only for compound components) the method of history clocks |
| `guest` | user provided invariant in external file |

3. **Safety properties** (*.pro* file): properties that need to be proven. Those properties are written in the prefix way, and their variables are viewed in terms of Petri nets [51]. Looking at the property (>=(+ m0_Init s0_Init) 2) as an example, the variable `s0_Active` indicates that `s0` is started in state `Init` (following the FSM described in the design). Since FSM is a special case of Petri nets, the number of tokens in any given place cannot exceed 1. The sum of the token's number in the corresponding two places in the clause is at least 2. It means that `s0` and `m0` are both started.

## 2.4 Discussion

There are some requirements for building correct self-adaptive cloud applications:

1. We need a framework to create a correct-by-construction cloud design. This framework provides a means to specify concepts of cloud components and related compositional operators for correct-by-construction development.

2. Instead of applying coordination techniques such as monitors, message forwarding, or semaphores, the framework should assist cloud designers in specifying coordination by using well-founded and organized concepts.

3. The component's behavior and coordination should be described in a high-level abstraction to help cloud architects efficiently and correctly design complex applications.

4. The framework should be able to derive a correct and efficient implementation.

This chapter presents the current state of the art of principles, concepts, and techniques supporting our objective, including the correct-by-construction approach, cloud resource management, and model-based approach. This thesis combines all these approaches to develop correct-by-construction self-adaptive cloud applications. OCCIware is an open-source model-driven framework for designing cloud applications. We address the first requirement by introducing some new usage of OCCIware entities' properties (Section 4.3.1) and leveraging FSM specifications (Section 4.4.2).

We deal with the second requirement by using the BIP framework. BIP framework provides a means to design correct-by-construction systems following the exogenous coordination, which specifies the coordination separately to computation. The coordination code is represented by BIP connectors constructed by the trigger, synchron ports, or sub-connectors.

However, learning to use BIP is not easy for cloud developers unfamiliar with formal methods. Thus, we proposed a language that allows cloud developers to specify requirements to develop correct-by-construction cloud applications using BIP (Section 3.2). Some related research provides boilerplates for writing specifications. However, developers must learn to use numerous boilerplates depending on the target domain. To reduce the effort and time of the learning activity, our ontology architect provides patterns, which are the general textual templates for specifying requirements in a pseudo-natural manner. By introducing the NaturalBIP language, we addressed the third requirement.

We answer the fourth requirement by developing NaturalBIP Compiler (Section 3.3), which analyzes the specification written in the NaturalBIP language to compute BIP model and Java BIP artifacts. The BIP model are part of the input for iFinder to verify whether the design is deadlock-free. JavaBIP artifacts, including JavaBIP GlueBuilder, data transfers, and safety properties (represented as conditions/constraints), are used to support developers in implementing the target cloud applications correctly.

# Chapter 3

# Domain-Specific Language for Developing Self-adaptive Applications

This chapter introduces our language called NaturalBIP for specifying behavioral constraints of self-adaptive applications. We begin by explaining why we decided to create the NaturalBIP language in Section 3.1. Section 3.2 defines the ontology architecture to construct the NaturalBIP language. The architecture includes an ontology specifying system concepts and relationships between them, a component for describing conditions, and a domain-specific language for naturally writing functional requirements. The NaturalBIP Compiler to translate the specification to JavaBIP artifacts and BIP connectors is presented in Section 3.3. Finally, Section 3.4 is a summary of this chapter.

## 3.1 Introduction

**Context.** Over the last decade, exogenous coordination has become a promising approach for managing the complexity of the design and implementation of distributed systems by controlling components from the outside. BIP is a framework supporting exogenous coordination. It allows building complex designs from smaller designs to enforce given properties. The challenge is applying BIP in designing cloud applications since cloud designers have no background knowledge of BIP. To develop an application in an exogenous approach, developers have to conduct a lot of manipulations that are difficult to learn. In contrast, there is no baseline for specifying clear and consistent requirements in natural language. They are written based on personal knowledge with some implicit expectations.

To reduce the specifying and coding efforts, we propose NaturalBIP language for developing correct-by-construction self-adaptive applications. This language applies

an ontology-driven specification approach for formulating functional requirements in a pseudo-natural language. We adopt the approach proposed by Stachtiari et al. [122] using boilerplates [80] in combination with a conceptual model. More concretely, boilerplates are semi-completed specifications containing placeholders that can be filled by concepts specified in the conceptual model or the ontology of the target system.
Our proposed language should be able to:

- specify functional requirements and properties in a language similar to English sentences,
- transform the specifications to BIP connectors for verification, and
- transform the specifications to JavaBIP macro code for the implementation.

**Methodology.** An ontology explicitly declares concepts in a well-defined knowledge domain. It consists of classes, their individuals, properties, relations, and axioms, forming the overall theory that describes the ontology. Ontologies encode relationships between concepts used in placeholders. Using the proper tool support, engineers can avoid ambiguous references and maintain the relationships between concepts described in requirements. We defined grammar to derive patterns that are pseudo-natural language templates with placeholders to capture requirements. Each derived pattern is associated with a formal representation in a logical language. By using these patterns, it addresses the ambiguity of natural language requirements. According to [110], a boilerplate comprises attributes and fixed syntax elements. For example, in the boilerplate (3.1), “executes” is a fixed syntax element, while $\langle instance \rangle$ and $\langle action \rangle$ are placeholder attributes for user input.

$$\langle instance \rangle \quad \text{executes} \quad \langle action \rangle \tag{3.1}$$

Figure 1.2 provides an overview of the methodology to support the development of concurrent applications following the exogenous approach:

- We proposed an architecture to formulate well-defined semantic requirements by applying an ontology-driven specification approach. In particular, we introduce an ontology that precisely defines concepts and semantic relationships [56, 60, 89] of components in a concurrent system and provides a grammar to write functional requirements.
- Software designers construct the application design and write requirements using the defined ontology and grammar in the NaturalBIP language.

- The written requirements and design model are sent to NaturalBIP Compiler to check the syntax and semantics before generating corresponding BIP connectors and JavaBIP artifacts for verifying and implementing the application following the exogenous approach.

**Running example.** We use an example of *Tracker-Peer communication* to illustrate the use of our proposed language.

**Example 3.1.1 (Tracker-Peer communication [37])**

Consider the Tracker-Peer example inspired by a wireless audio protocol for peer-to-peer communication. There are two component types: Tracker and Peer. The protocol allows an arbitrary number of peers to communicate along a random number of wireless communication channels, and a unique tracker manages each channel. There is the list of requirements written in natural language as follows:

- **Req_log**: "Every tracker shall log whenever a peer registering or unregistering to it."
- **Req_action**: "Registered peers shall either speak to the channel or listen to other registered peers in the channel."
- **Req_constraint**: "At most one registered peer shall speak at the moment."

## 3.2 NaturalBIP Language

We propose an ontology architecture for specifying the functional requirements of self-adaptive applications. Based on the proposed ontology architecture, we define the syntax for our language in the context-free grammar form to interpret functional requirements and parse them to generate BIP connectors.

### 3.2.1 Ontology Architecture

Figure 3.1 shows the architecture overview, which includes ontologies, components, and the relationships between them to preserve the semantics and resolve conflicts and ambiguities. The requirements are written following the semantic definitions of NaturalBIP Requirement Ontology (NRO). The NRO imports the Domain Specific Ontology (DSO), transitively imports Behavioral Ontology (BO), and uses DSO instances.

Behavioral Ontology (BO) defines core concepts (e.g., `Subject`, `Action`, `State`, `Finite State Machine` (FSM)) that are used as requirements specification elements. It plays the role of top-level ontology, supporting broad semantic compatibility among domain-specific ontologies.

A Domain-Specific Ontology (DSO) contains the domain-specific classes of the system to be designed. DSO imports all classes in the top-level ontology (i.e., the BO) and further specializes in them. The DSO instances define corresponding instances or variables for the corresponding `Subject` in DSO.

The NaturalBIP Requirement Ontology (NRO) provides sentence structures using boilerplate forms with placeholders to construct a well-formed requirement, thus, decreasing specification errors.

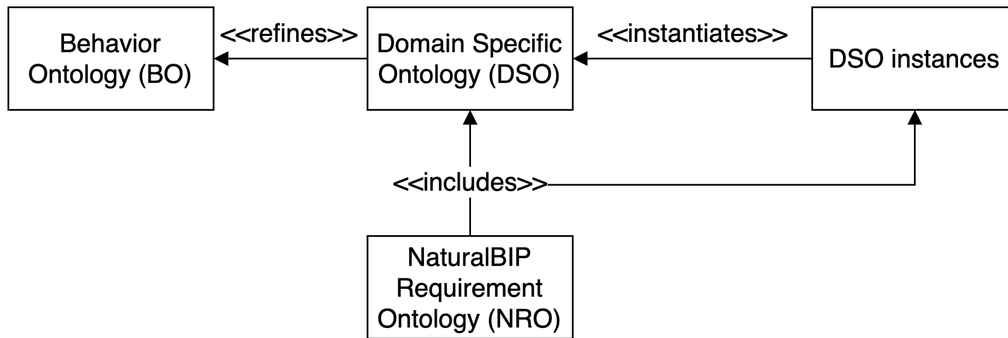


Figure 3.1: Ontology Architecture

#### 3.2.1.1 Behavioral Ontology (BO)

This ontology contains concepts for specifying functional requirements and their relationships, such as components, actions, states, etc. These concepts are semantically interrelated, as shown in the conceptual overview in Figure 3.2. The BO concepts and their relationships are introduced and explained as follows:

- A `Subject` represents an interactive element in a concurrent system, and it `has` a finite state machine. Its status is determined by a State (via `isIn` property). A `Subject` can inherit from another `Subject` via the `extends` property, and `performs` actions.

- Each finite state machine (`FSM`) contains a set of transitions (i.e., `TransitionSet`) and a set of states (i.e., `StateSet`).

- An `Action` denotes a behavior of a related `Subject`. One `Action` belongs to a `TransitionSet` in a `FSM` of the `Subject`. The execution of `Action` might change the `State` of the `Subject` (via `sets` property).

- Each `State` belongs to the `StateSet` of the `Subject`'s `FSM` and its value can be set by executing an `Action`.

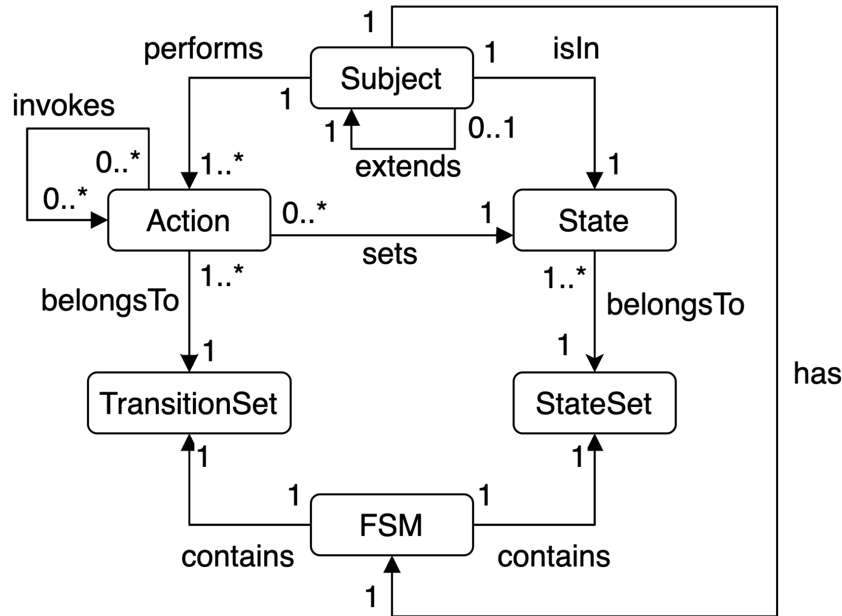


Figure 3.2: Behavioral Ontology

#### 3.2.1.2 Domain Specific Ontology (DSO)

DSO describes the vocabulary to express a system domain. The DSO concepts and their interactions provide a semantic model of the system's domain. Figure 3.3 shows the DSO for the Tracker-Peer communication system conforms to Example 3.1.1.

There are two `Subjects` are `Tracker` and `Peer`. Each has a `FSM` which contains a set of transition and state. The `TransitionSet` of `Tracker` includes `broadcast` and `log`, while `Peer` consists of `register`, `unregister`, `speak`, and `listen`. State `TrackerInit` belongs to the `StateSet` of `Tracker`, and states `PeerInit`, `Registered` belong to the `StateSet` of `Peer`.

#### 3.2.1.3 NaturalBIP Requirement Ontology (NRO)

Figure 3.4 shows the ontology for encoding elements of requirement specifications. The detail elements and the relations between them are defined as follows:

- The `Requirement` in our language is a `Clause` scoped by one or more `Quantifiers` (via `hasClause` and `hasQuantifier` properties).
- Each `Clause` always has a `Main` clause (`hasMain` property) and may have a Constraint (`hasConstraint` property).
- The `Main` clause describes the occurrence of actions and may specify `State Values` scoped by corresponding `Quantifiers`.

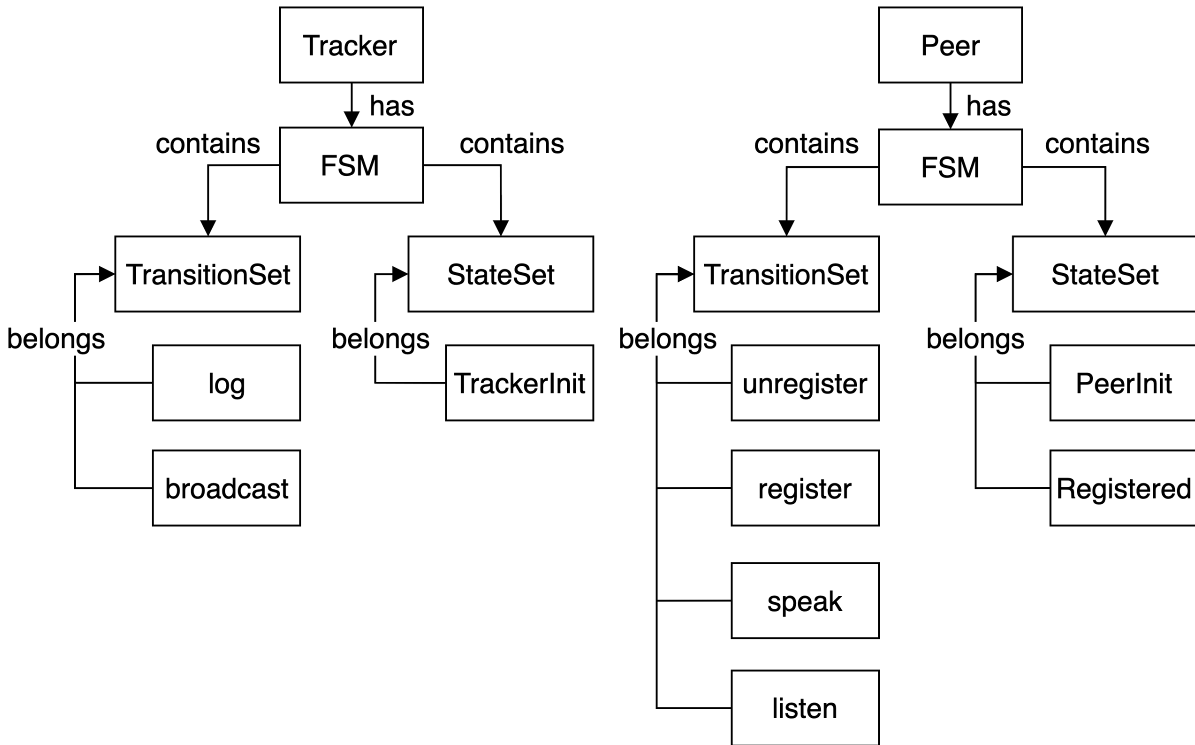


Figure 3.3: Domain Specific Ontology of the Tracker-Peer communication system

- The `Constraint` clause specifies the observation of events and may describe `State Values` scoped by corresponding `Quantifiers`.
- `Quantifier` is a component determining the scope of application for specific classes in functional requirements. A `Quantifier` might contain `Conditions`.
- Component `Condition` defines conditions or constraints that affect classes, instances, or the relations between them.
- `Observing Event`, `Occurring Action`, and `State Value`: These components specify the observation of an event, the occurrence of an action, or the state's value, respectively.

`Quantifier`, `Condition`, `Occurring Action`, `State Value`, and `Observing Event` are boilerplate syntax containing fixed elements and placeholders.

### 3.2.2 NaturalBIP Syntax and Semantics

The grammar rules of our language are presented in Table 3.1 with some notations defined in Table 3.2. The grammar rules are left-associative, and non-terminal symbols will be parsed until reaching terminal symbols. A `Requirement` is always scoped by at least one

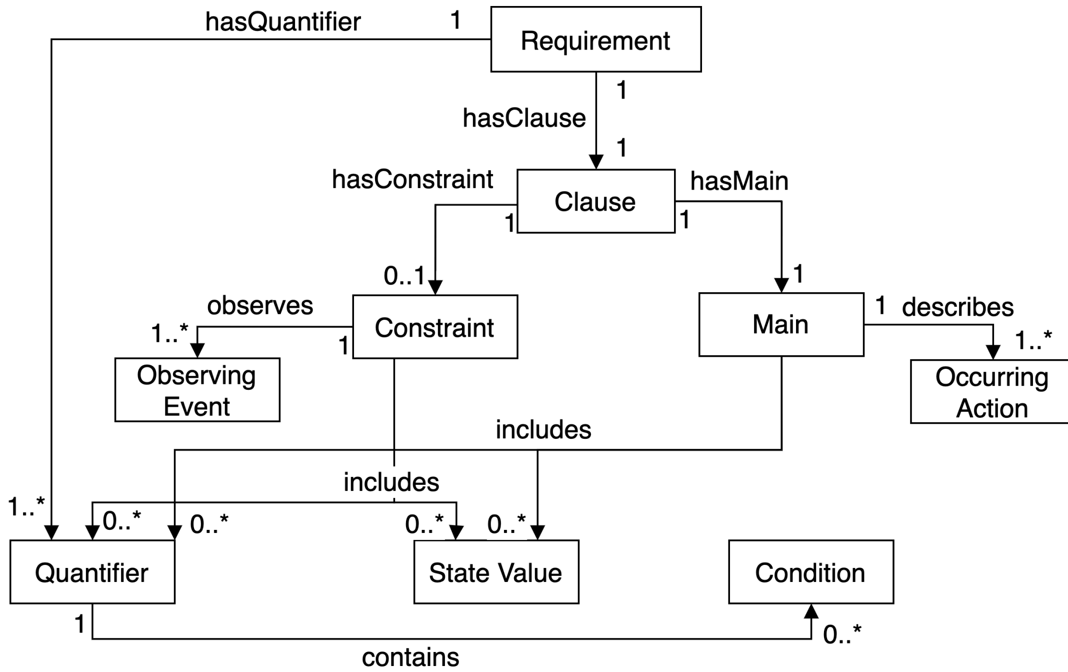


Figure 3.4: NaturalBIP Requirement Ontology

`Quantifier` and contains a `Main` clause. `Constraint` clause is optional in a `Requirement`, and if it exists, this clause will be separated from the `Main` clause by a comma symbol (i.e., ","). As described in Table 3.1, `Main` is constructed by `compound expressions` including `Occurring Actions`, `State Values`, or `Conditions` with `connective` operators. If the `Main` clause and `Constraint` clause use a new variable, the new variable must be scoped by a `Quantifier`. Table 3.4 illustrates operators to associate elements in a `compound expressions` including: conjunction operator (i.e., "**and**") and disjunction operator (i.e., "**or**").

Table 3.3 presents all the derivable phrases of a requirement if it contains constraints. The `Constraint` is defined after conjunction words (i.e., "*if*", "*while*", "*after*", "*before*"), and the `constraints expression` consists of `Observing Events`, `State Values`, or `Condition` with `connective` operators. The `Quantifier` is optional in `Constraint`.

`Observing Event`, `Occurring Action`, `State Value`, `Condition`, and `Quantifier` are atomic components that contain placeholders to be filled with information adhering to the design of the target application (i.e., DSO).

**State Value.** This element describes that an instance of a `Subject` reaches a specific state. For example, "*p is in Registered state*" expresses that instance `p` is in the state `Registered`.

Table 3.1: NaturalBIP grammar

<table>
<tr><td>⟨Requirement⟩</td><td>::=</td><td>(⟨Quantifier⟩)+ ⟨Clause⟩</td></tr>
<tr><td>⟨Clause⟩</td><td>::=</td><td>⟨Main⟩<br>| ⟨Constraint⟩, ⟨Main⟩</td></tr>
<tr><td>⟨Quantifier⟩</td><td>::=</td><td>⟨quantification of amount⟩ **subject␣instance**,<br>[⟨separator⟩ ⟨Condition⟩, ]</td></tr>
<tr><td>⟨Constraint⟩</td><td>::=</td><td>⟨conjunction⟩ ⟨constraint expression⟩</td></tr>
<tr><td>⟨Main⟩</td><td>::=</td><td>⟨compound expression⟩</td></tr>
<tr><td>⟨compound expression⟩</td><td>::=</td><td>⟨compound expression⟩ *synchronized with* ⟨compound expression⟩<br>| ⟨compound expression⟩ ⟨connective⟩ ⟨State Value⟩<br>| ⟨compound expression⟩ ⟨connective⟩ ⟨Occurring Action⟩<br>| [⟨Quantifier⟩] ⟨State Value⟩<br>| [⟨Quantifier⟩] ⟨Occurring Action⟩</td></tr>
<tr><td>⟨constraint expression⟩</td><td>::=</td><td>⟨constraint expression⟩ ⟨connective⟩ ⟨state value⟩<br>| ⟨constraint expression⟩ ⟨connective⟩ ⟨Observing Event⟩<br>| [⟨Quantifier⟩] ⟨State Value⟩<br>| [⟨Quantifier⟩] ⟨Observing Event⟩</td></tr>
<tr><td>⟨State Value⟩</td><td>::=</td><td>**instance** *is in* **state_value** *state*</td></tr>
<tr><td>⟨Observing Event⟩</td><td>::=</td><td>**instance** *executes* **action**<br>| **instance** *does not execute* **action**</td></tr>
<tr><td>⟨Occurring Action⟩</td><td>::=</td><td>**instance** *shall* **action**<br>| **instance** *shall not* **action**<br>| (**instance** *shall either* ***action***$_1$ *or* ***action***$_2$ *or* ... *or* ***action***$_n$)</td></tr>
<tr><td>⟨Condition⟩</td><td>::=</td><td>⟨Condition⟩ ⟨connective⟩ ⟨Condition⟩<br>| **instance** *is* phenomenon [**instances**]<br>| **instance** *has* phenomenon [**instances**]<br>| **instance** *can* do phenomenon [**instances**]</td></tr>
<tr><td>⟨conjunction⟩</td><td>::=</td><td>*if* | *while* | *after* | *before*</td></tr>
<tr><td>⟨connective⟩</td><td>::=</td><td>*and* | *or*</td></tr>
<tr><td>⟨quantification of amount⟩</td><td>::=</td><td>*for all* | *for any* | *for every*<br>| *there is a* | *there is one* | *there are some*</td></tr>
<tr><td>⟨separator⟩</td><td>::=</td><td>*where* | *such that*</td></tr>
</table>

**Observing Event.** This element is used in `Constraint` to specify the observation of an event. For example, "*t executes broadcast*" denotes the observation of an event that

Table 3.2: Notations used in our grammar

| Symbol | Meaning |
|---|---|
| $\langle \ldots \rangle$ | Non-terminal symbol |
| $[\ldots]$ | Optional symbol |
| $\ldots \mid \ldots$ | Disjunction logic operator |
| $(\ldots)+$ | The number of elements inside the parentheses is at least one |
| **bold string** | The placeholder with semantics defined or deduced from the DSO |
| *emphasis string* | Keyword |
| normal string | Normal meaning in the natural language |

Table 3.3: The derivable `Clause`s contain `Constraint`s

| Derivable Clause | Explanation |
|---|---|
| **If** ⟨Constraint⟩, ⟨Main⟩ | Globally, ⟨Main⟩ occurs next to the occurrence of ⟨Constraint⟩ |
| **While** ⟨Constraint⟩, ⟨Main⟩ | Globally, ⟨Main⟩ occurs during the observation of ⟨Constraint⟩ |
| **After** ⟨Constraint⟩**,** ⟨Main⟩ \| ⟨Main⟩ **after** ⟨Constraint⟩ | Eventually, ⟨Main⟩ occurs after the occurrence of ⟨Constraint⟩ |
| **Before** ⟨Constraint⟩**,** ⟨Main⟩ \| ⟨Main⟩ **before** ⟨Constraint⟩ | ⟨Main⟩ can be observed until the occurrence of ⟨Constraint⟩ |

Table 3.4: Grammar rules of Compound Expression

| Grammar | Explanation |
|---|---|
| `elem_1` **and** ... **and** `elem_n` | Globally, elements `elem_1` to `elem_n` occur at the same time. |
| `elem_A` **synchronized with** `elem_B` | Globally, the occurrence of `elem_A` is synchronized with the occurrence of `elem_B`. |
| `elem_A` **or** `elem_B` | At the current state, `elem_A` or `elem_B` or both can be observed. |

`Tracker t` executes the `broadcast` action, where `t` is an instance variable of `Tracker`

(cf. Example 3.1.1).

**Occurring Action.** This component specifies the occurrence of actions in `Main` including the execution (i.e., **instance** *shall* **action**, **instance** *shall not* **action**), and exclusive disjunction operator (i.e., "(*shall either* ... *or* ...)").

- "*p shall listen*" expresses that action `listen` of `Peer p` must be executed.
- "*p shall not speak*" expresses that action `speak` of `Peer p` must not be executed.
- "*(p shall either speak or listen)*" expresses that two actions `speak` and `listen` of instance `p` cannot occur at the same time.

**Quantifier.** This element is mandatory to determine the application scope of the requirement. The quantifier is defined in the `Quantifier` rule in Table 3.1 and explained in Table 3.5. Looking at the requirement in Example 3.1.1, the quantifier for Peer can be presented as follows:

- "*For all Peer p,*" expresses that for all instance `p` of component `Peer`, or
- "*There are some Peer p, such that p is registered to t,*" expresses a condition applied for some instance `p` of component `Peer`, or
- "*There is a Tracker t, where t is not null,*" expresses another derivation of quantifiers with a condition but uses a different separator.

**Conditions.** This element describes the constraints or conditions to execute actions. The template of this block is defined in Formula (3.2):

**instance** *is/has/can* phenomenon [**instances**] (3.2)

Following the template, the first placeholder **instance** is the affected object of other **instances** if they exist. The phenomenon is defined by the collocation of lexical units of English from the Gellish English Dictionary [124] and the WordNet Lexical Database [94]. For examples, "`p` *has* no capacity", "`p` *is* registered to `t`".

Let us consider the requirements in Example 3.1.1, the functional requirements for **Req_log** can be written in our language as two following requirements:

- **TP_01_register**: "For any Peer p, there is a Tracker t, if p executes register, t shall log."

Table 3.5: Element Quantifier

| **Blocks** | **Explanation** | **Values** |
|---|---|---|
| ⟨quantification of amount⟩ | Mandatory. To specify the scope for applying the requirement. There are two types of ⟨quantification of amount⟩: universal quantifier and existential quantifier. | Universal quantifier: “all”, “any”, “every” Existential quantifier: “a”, “one”, “some”. |
| **subject** | Mandatory. Subjects affected by the quantifier. | The value of this block is mapped from the corresponding `Subject` in the BO. |
| **instance** | Mandatory. The specific variable of the subject. | This value can be inferred from the value of the `Subject` in BO and vocabulary in LO. |
| ⟨*separator*⟩ | Optional. Indicator to start describing conditions related to the quantifier's instance. | “where”, “such that”. |
| ⟨*Condition*⟩ | Optional. It appears following separators to describe the condition(s) of variables. | |

- **TP_01_unregister**: “For any Peer p, there is a Tracker t, such that p is registered to t, if p executes unregister, t shall log.”

- **TP_01_mutex**: “For any Peer p, there is a Tracker t, if t executes log, p shall register or p shall unregister.”

Figure 3.5 illustrates the decomposition of the requirement **TP_01_register**. The requirement is decomposed following the defined grammar until the terminal symbol (i.e., `Quantifier`, `Observing Event`, `Occurring Action`, or `Condition`) is reached. The specific values (i.e., the bold strings in the grammar) are mapped or referred from the DSO and DSO instance. In particular, requirement **TP_01_register** has `Main` and `Constraint` clauses and scopes them by `Quantifier-01` and `Quantifier-02`. The `Constraint` clause specifies the observation of event “p executes register”, and the `Main`

clause specifies the occurrence of action "t shall log". **DSO instances** contains the values deduced from the **DSO**'s concepts. Then, the placeholders use those values to complete the requirements.

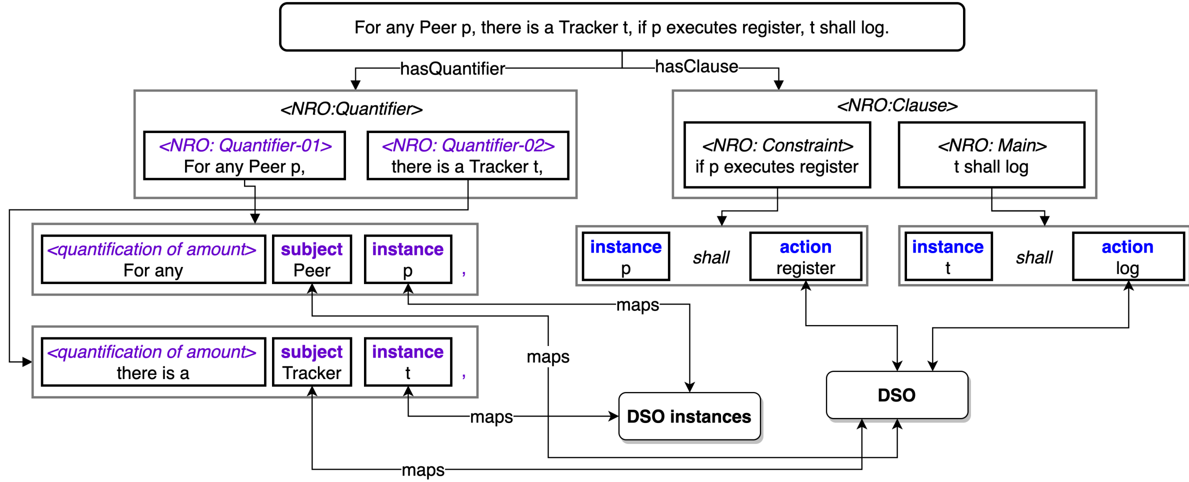


Figure 3.5: The decomposition of requirement **TP_01_register**

The original requirements **Req_action** and **Req_constraints** are related to the action *speak* of subject *Peer* and action `broadcast` of subject `Tracker`. Therefore, we write those requirements in NaturalBIP as follows:

- **TP_02_mutex**: "For any Peer p, (p shall either speak or listen)."
- **TP_02_speak**: "For any Peer p, there is a Tracker t, such that p is registered to t, if t executes broadcast, p shall speak and for all Peer p1, where p1 is registered to t and p1 is different with p, p1 shall not speak."
- **TP_02_listen**: "For any Peer p, there is a Tracker t where p is registered to t, if t executes broadcast, p shall listen or for all Peer p1, such that p1 is registered to t and p1 is different with p, p1 shall listen."

**TP_02_mutex** describes a constraint that a specific instance of `Peer` cannot `speak` and `listen` at a time. In the case that there are more than one instance of `Peer` registered to the same `Tracker`'s instance, **TP_02_speak** states that at most one instance of `Peer` can `speak`, while **TP_02_listen** claims that more than one instance of `Peer` can `listen` in a specific moment. The three requirements have the same **TP_02** part in their name because they associate each other following the description in **Req_action** and **Req_constraint** in Example 3.1.1.

Figure 3.6 presents the decomposition of requirement **TP_02_speak**. This requirement includes `Main` and `Constraint` clauses. Both of them are affected by `quantifier-01`

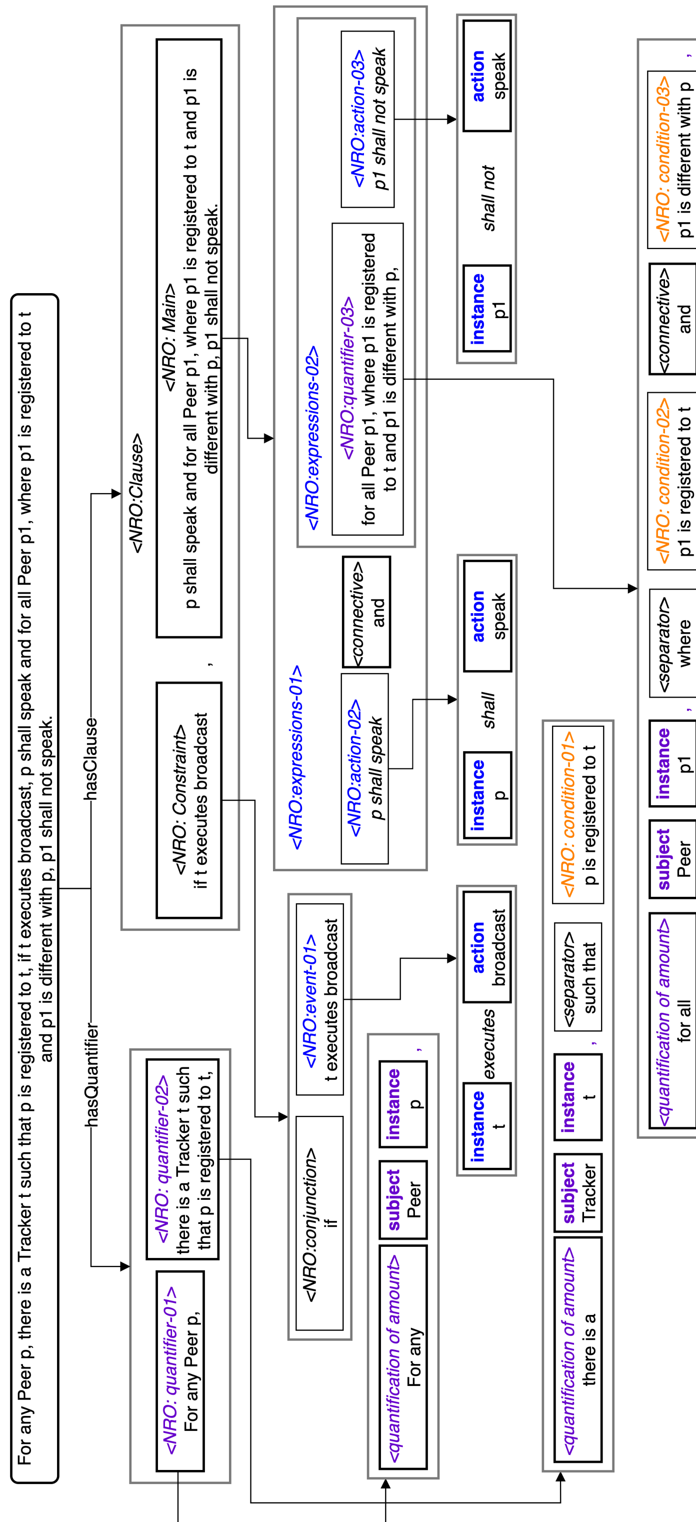


Figure 3.6: The decomposition of requirement **TP__02__speak**

and `quantifier-02`. `quantifier-02` contains `condition-01`, which describes the constraint between variables `p` and `t`. The `Constraint` describes the observation of event `event-01`, which expresses that instance `t` of `Tracker` executes action `broadcast`. After observing that event, the `Main` clause contains a compound expression `expressions-01`, which specifies the occurrence of action `action-02` and a sub-compound expression (i.e., `expressions-02`). `expressions-02` includes `quantifier-03` and `action-03`, where `quantifier-03` expresses variable `p1` and a conjunction of two conditions, `condition-02` and `condition-03`; and `action-03` expresses that `p1` shall not `speak`.

## 3.3 NaturalBIP Compiler

In Section 3.2, we have proposed the NaturalBIP language for writing functional requirements for cloud applications. In this section, we propose an automated generation of the artifacts, including the JavaBIP macro, data transfers, and constraints from the functional requirements written in the NaturalBIP language. Figure 3.7 shows the overall process of the NaturalBIP compiler step by step, together with the corresponding input and output data:
Input: (i) the application design, and (ii) the NaturalBIP requirements
Output: JavaBIP artifacts for the implementation

- Step 1 *Pre-processing*: Analyzing the NaturalBIP requirements to get the information of Quantifier, Interaction, and Condition; and skolemizing the requirements (cf. Section 3.3.1).

- Step 2 *Boolean Encoding*: Generating Boolean formulas from requirements by handling patterns.

- Step 3 *Conjunctive Normal Form (CNF) Transformation*: Using PBL—an open-source tool for handling Boolean formulas following algorithms proposed in [40]—to generate CNF clauses (see Section 2.3.1).

- Step 4 *Dual-Horn Generator*: Calculating dual-Horn clauses from the CNF obtained in the previous step. The dual-Horn clauses present the interactions between the system's components [39].

- Step 5 *BIP Generator*: Generating artifacts, including JavaBIP macros, data transfers, conditions, and BIP connector in the form of the algebra of connectors (AC) [39].

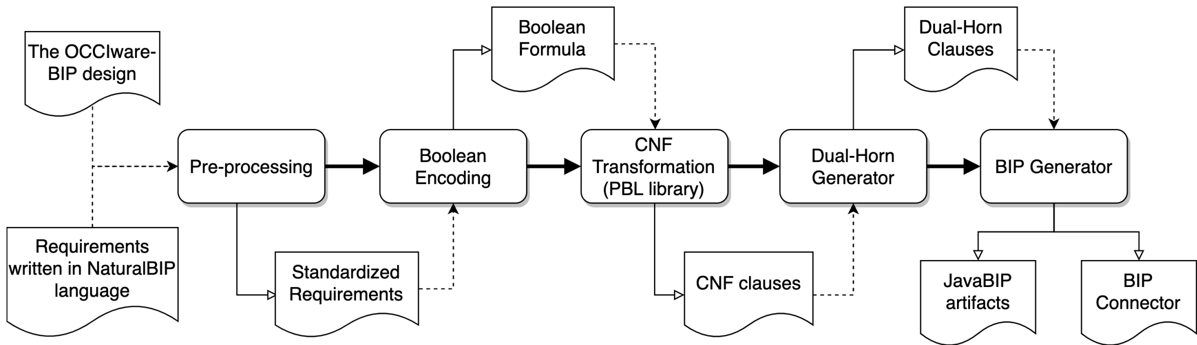


Figure 3.7: The process for the generation of JavaBIP artifacts from NaturalBIP requirements

### 3.3.1 Pre-processing

Pre-processing converts the requirements written in NaturalBIP into input for PBL (i.e., Boolean formulas). This process is performed in the following steps:

1. **Pre-processing of NaturalBIP requirements.**

   The output of the NaturalBIP compiler is the JavaBIP macro, which describes the interaction between system components through actions. Therefore, requirements written in the NaturalBIP contain descriptive information about some components' states that must be transformed into the corresponding actions. Based on the FSM specification in the design, the transformation converts state description to the disjunction of actions leading to that state. For example,

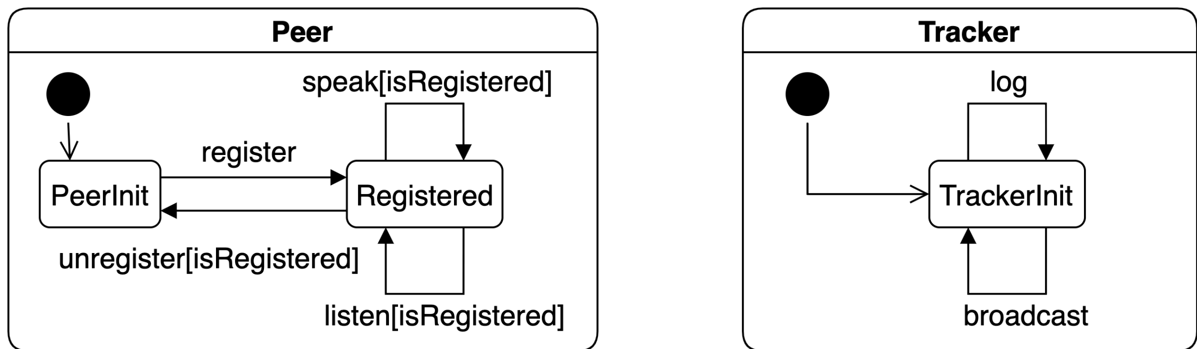


Figure 3.8: The finite state machines of Tracker and Peer

   To depict the use of some special expressions, consider a hypothetical requirement named **HTP** (i.e., hypothetical Tracker-Peer example) that specifies "*For any Tracker t, while t is in TrackerInit, there is a Peer p, where p is registered to t, (p shall either speak or listen) and p shall not unregister.*", where `Peer` and `Tracker` are components having the corresponding FSMs described in Figure 3.8.

State `TrackerInit` has two incoming transitions: `log` and `broadcast`. Thus, state `TrackerInit` is replaced with two actions `log` and `broadcast` and "*while*" connective is replaced by "*if*" because "*while*" specifies the state values. The original requirement will be rewritten as "*For any Tracker t, if (t shall log or t shall broadcast), there is a Peer p, where p is registered to t, (p shall either speak or listen) and p shall not unregister.*" (cf. Step 1 in Figure 3.9).

The expression "`either ...or ...`" in our language will be presented by "`A XOR B`" in Boolean formulas. The requirement **HTP** becomes "*For any Tracker t, if (t shall log or t shall broadcast), there is a Peer p, where p is registered to t, (p shall speak XOR p shall listen) and p shall ~unregister.*" (cf. steps 2 and 3 in Figure 3.9).

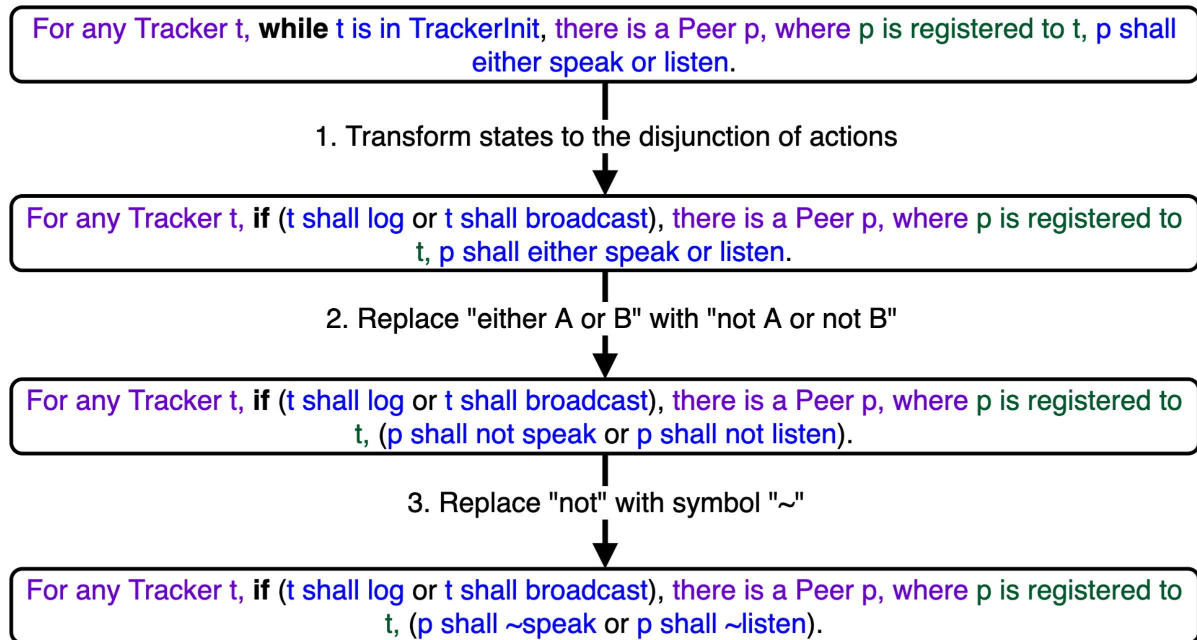


Figure 3.9: The pre-processing of requirement **HTP**

2. **Collecting components**
   After being pre-processed, the requirement is analyzed to extract quantifiers and actions following the corresponding grammar rules in Table 3.1 (cf. steps (1) and (2) in Figure 3.10). To collect the conditions, first, remove the quantifiers and actions obtained in steps (1) and (2) from the request (cf. step (3) in Figure 3.10). Then split the remaining string with the "*and*"/"*or*" operators as keywords; remove the keywords (i.e., *if*, *while*) and parentheses to get the conditions (cf. step (4) in Figure 3.10). Finally, actions and conditions are converted to variables for Boolean formulas and updated into the requirement (cf. step (5) in Figure 3.10).

3. **Skolemization**
   Skolemization is a process of removing quantifiers from a predicate logic formula.

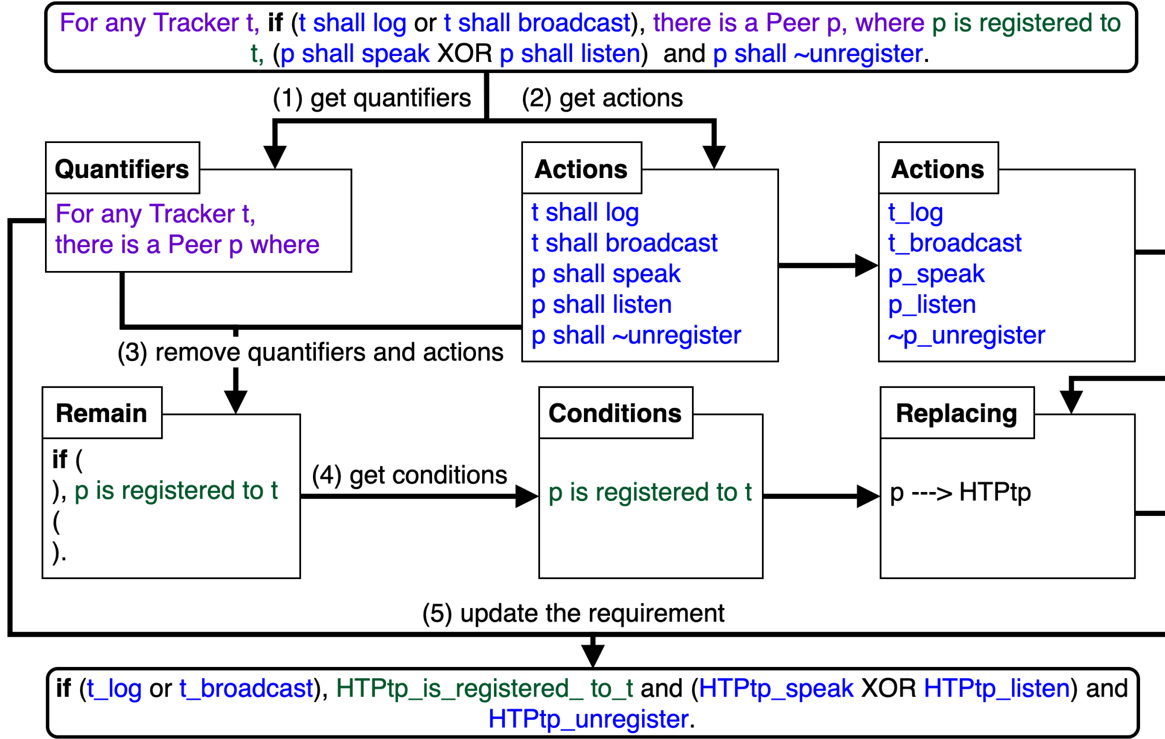


Figure 3.10: Collecting quantifiers, actions, and conditions from the pre-processed requirement

Assume that a requirement $REQ$ has an existential quantifier describing the instance $ei$ of a class $C$ and a universal quantifier describing instance $ui$ of class $C'$. If the existential quantifier is inside the universal quantifier, $ei$ will be replaced with a new value composed by the concatenation of the requirement name, instance name in the universal quantifier, and its name (cf. equation(1.4)). In particular, $ei$ will be replaced with $REQuiei$.

$$ei = \begin{cases} REQuiei, & \text{if } ei \text{ is bounded in } ui. \\ ei, & \text{otherwise.} \end{cases} \tag{3.3}$$

In this way, the instance $p$ in the hypothetical example **HTP** is replaced with $HTPtp$. Thus, the **HTP** becomes: "*if (t_log or t_broadcast), HTPtp_is_registered_to_t and (HTPtp_speak XOR HTPtp_listen) and ~HTPtp_unregister.*" (cf. step (5) in Figure 3.10).

Regarding the requirement **TP_02**, which includes three related requirements **TP_02_mutex**, **TP_02_speak** and **TP_02_listen**, its Boolean formulas after the pre-processing are shown in Listing 3.1:

Listing 3.1: `req_file` contains the Boolean formulas of requirement **TP_02**

```
1 // File: TP_02.txt
2 TP_02_speak = ((p_speak => (p1_is_registered_to_TP02speakptt & p1_is_different_with_p & ~p1_speak &
      p_is_registered_to_TP02speakptt & TP02speakptt_broadcast)))
3 TP_02_mutex = (p_speak XOR p_listen)
4 TP_02_listen = ((p_listen => (p_is_registered_to_TP02listenptt & TP02listenptt_broadcast))) & ((p1_listen => (
      p1_is_registered_to_TP02listenptt & p1_is_different_with_p & p_is_registered_to_TP02listenptt &
      TP02listenptt_broadcast)))
5 Main_Exp: TP_02_speak & TP_02_mutex & TP_02_listen
```

### 3.3.2 Boolean Encoding

Boolean Encoding is the process of converting requirements containing `Constraint` clauses (i.e., *"if"*, *"while"*, *"after"*, and *"before"*) into Boolean formulas. Among grammar rules in Table 3.3, the pre-processing step converted "while" clause to "if" clause. Assume that the `compound expression` in Main contains a disjunction of $n$ sub-expression (i.e., $\bigvee_{i=1}^{n} expr_i$) and $expr_i$ are conjunctions of elements `state value`, `observing event`, `occurring action`, or `Condition` (as shown in Table 3.1). We denote the conjunctions as $\bigwedge_{j=1}^{|expr_i|} elm_j^i$

We write $Elm$ to denote an element in the `Main` clause. The grammar rules for *"if"*, *"while"*, and *"after"* are presented in formulas (3.4):

$$\begin{aligned} &\textit{if/after}\ \texttt{Constraint},\ \texttt{Main} \\ &\equiv \textit{if/after}\ \texttt{Constraint},\ \bigvee_{i=1}^{n} \bigwedge_{j=1}^{|expr_i|} elm_j^i \end{aligned} \tag{3.4}$$

This pattern will be converted to Boolean formulas as equation (3.5):

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{|expr_i|} (elm_j^i \Longrightarrow \texttt{Constraint} \wedge \bigwedge_{j' \neq j} elm_{j'}^i) \tag{3.5}$$

As mentioned in Table 3.3, the grammar "Before *Constraint*, *Main*" states that "globally, *Constraint* is absent in the occurrence of *Main*", so the corresponding Boolean formula for this grammar is shown in formula (3.6):

$$\bigwedge_{i}^{n} \bigwedge_{j}^{|expr_i|} (elm_j^i \Longrightarrow \sim\texttt{Constraint} \wedge \bigwedge_{j' \neq j} elm_{j'}^i) \tag{3.6}$$

### 3.3.3 Dual-Horn clauses generation

Given a formula $F$ in conjunctive normal form (CNF), where $F = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ and $C_i = \neg x_1 \vee \ldots \neg x_m \vee y_1 \vee \cdots \vee y_k$, where $\neg x_j (j \in [1, m])$ are negative literals and $y_l (l \in [1, k])$ are positive literals, the dual-Horn clauses are computed as follows:

1. For each clause $C_i$ in $F$:
   - If $C_i$ contains at most one negative literal (i.e., $m <= 1$), keep $C_i$
   - Otherwise, split $C_i$ into the disjunction of multiple clauses. $C_i = C_i^1 \vee \cdots \vee C_i^m = (\neg x_1 \vee y_1 \cdots \vee y_k) \vee \cdots \vee (\neg x_m \vee y_1 \cdots \vee y_k)$
2. Conjoin the resulting clauses with the logical "and" (i.e., $\wedge$) operator to obtain the final formula. Assume that clause $C_1$ contains more than one negative literal, the final formula $F = (C_1^1 \wedge C_2 \wedge \cdots \wedge C_n) \vee \cdots \vee (C_1^m \wedge C_2 \wedge \cdots \wedge C_n)$ is the disjunction of dual-horn clauses.

The process to generate dual-Horn clauses [39] consists of the following steps:

- **Step 1: generate CNF clauses using PBL library.**

  Listing 3.2 shows CNF clauses generated from Boolean expressions (described in Listing 3.1) using the PBL library.

Listing 3.2: CNF clauses generated from the Boolean formulas in `req_file` in Listing 3.1

```
1  // 1. generated CNF clauses
2  (~(p_speak) | p1_is_registered_to_TP02speakptt)
3  & (~(p_speak) | p1_is_different_with_p)
4  & (~(p1_speak) | ~(p_speak))
5  & (~(p_speak) | p_is_registered_to_TP02speakptt)
6  & (~(p_speak) | TP02speakptt_broadcast)
7  & (p_speak | p_listen)
8  & (~(p_speak) | p_speak)
9  & (~(p_listen) | p_listen)
10 & (~(p_speak) | ~(p_listen))
11 & (~(p_listen) | p_is_registered_to_TP02listenptt)
12 & (~(p_listen) | TP02listenptt_broadcast)
13 & (~(p1_listen) | p1_is_registered_to_TP02listenptt)
14 & (~(p1_listen) | p1_is_different_with_p)
15 & (~(p1_listen) | p_is_registered_to_TP02listenptt)
16 & (~(p1_listen) | TP02listenptt_broadcast)
```

- **Step 2: synthesize clauses that have the same negative elements**

  Considering any two different clauses `cnf_clauses`[i] and `cnf_clauses`[j] in the given CNF formula, we denote by `cnf_clauses`[i].neg and `cnf_clauses`[j].neg the sets of negative variables and by `cnf_clauses`[i].pos and `cnf_clauses`[j].pos the

sets of positive variables in `cnf_clauses`[i] and `cnf_clauses`[j], respectively. If two negative sets of `cnf_clauses`[i] and `cnf_clauses`[j] are equivalent (line 10 in Algorithm 1), assign `cnf_clauses`[i].neg to `temp_clauses`.neg. The `temp_clauses`.pos is the Cartesian product of `cnf_clauses`[i].pos and `cnf_clauses`[j].pos (lines 12 and 13). If the `temp_clauses`.pos is not empty, add it into the `synthesized_cnf_clauses` (lines 17-19). For example, the two CNF clauses "(~x1 | y1 | y2) & (~x1 | z1 | z2)" will be replaced by "~x1 | y1&z1 | y1&z2 | y2&z1 | y2&z2".

**Algorithm 1** Combine clauses having the same negative variables

```
Require: cnf_clauses                                   ▷ given set of CNF clauses
Ensure: synthesized_cnf_clauses
 1: synthesized_cnf_clauses := {}
 2: combined_clauses := {}
 3: i := 0                                             ▷ counter variable for iteration
 4: while i < cnf_clauses.size() do
 5:     if cnf_clauses[i] ∈ combined_clauses then     ▷ ignore combined clauses
 6:         continue
 7:     end if
 8:     initialize temp_clauses                        ▷ create a temporary clause
 9:     for j ∈ [i + 1, cnf_clauses.size()] do
10:         if cnf_clauses[i].neg = cnf_clauses[j].neg then
11:             temp_clause.neg := cnf_clauses[i].neg
12:             cnf_clauses[i].pos := cnf_clauses[i].pos × cnf_clauses[j].pos
13:             temp_clause.pos := cnf_clauses[i].pos
14:             combined_clauses := combined_clauses ∪ {cnf_clauses[j]}
15:         end if
16:     end for
17:     if temp_clause.pos.size() != 0 then
18:         synthesized_cnf_clauses := synthesized_cnf_clauses ∪ temp_clause
19:     end if
20:     i := i+1
21: end while
```

Listing 3.3: `cnf_clauses` after being synthesized

```
1 (~(p_speak) | p1_is_different_with_p & TP02speakptt_broadcast & p_is_registered_to_TP02speakptt &
      p1_is_registered_to_TP02speakptt)
2 & (~(p_listen) | p_is_registered_to_TP02listenptt & TP02listenptt_broadcast)
3 & (~(p1_listen) | p1_is_different_with_p & p1_is_registered_to_TP02listenptt & TP02listenptt_broadcast &
      p_is_registered_to_TP02listenptt)
4 & (~(p1_speak) | ~(p_speak))
5 & (p_speak | p_listen)
6 & (~(p_speak) | ~(p_listen))
```

- **Step 3: collect conditions to relevant negative actions**
  We introduce *a causal rule* (Definition 3.3.1) that provides mechanism for computing BIP connectors [39].

  **Definition 3.3.1 (Causal rule [39])**
  A *causal rule* is a $\mathbb{B}[P]$ formula $E \Longrightarrow C$, where $E$ (the *effect*) is a constant *true* or an action variable $p \in P$; and $C$ (the *cause*) is a constant (*true* or *false*) or a disjunction of interactions, i.e., $\bigvee_{i=1}^{n} a_i$ where, for $i \in [1, n]$, $a_i$ are conjunctions of action variables (i.e., $a \in 2^P$).

  There are three types of *causal rules* with constants:

  1. $true \Longrightarrow a$ expresses that the considered interaction contains $a$,
  2. The form $p \Longrightarrow true$ are satisfied by all interactions, and
  3. $p \Longrightarrow false$ expresses that the considered interaction does not contain $p$.

  **Definition 3.3.2 (Interaction)**
  An interaction $a \in 2^P$ satisfies a formula $R \in \mathbb{B}[P]$ (denoted $a \models R$) iff the corresponding boolean valuation satisfies $R$. A term $x \in \mathcal{AI}(P)$ (i.e., *algebra of interactions* [38]) satisfies $R$ (denoted $x \models R$) iff all interactions belonging to $x$ satisfy $R$, that is

  $$x \models R \stackrel{def}{\Longleftrightarrow} \forall a \in x, a \models R$$

  A non-constant *causal rule* can be represented in the form $\overline{p} \vee \bigvee_{i=1}^{n} a_i$ where, $p$ is a variable belonging to the action set $P$ of the whole system (i.e., $p \in P$), and $a_i$ with $i \in [1, n]$ are conjunctions of action variables (i.e., $a \in 2^P$). Therefore, if one of the negative variables in a clause is a condition, it must be combined with the related negative action by De Morgan's laws [104]. For example, consider the clause "~p_speak | ~p1_listen | ~p_is_registed_to_t", where "p_is_registed_to_t" is a condition and the action "p_speak" is related to the condition "p_is_registed_to_t" because they have the same instance value "p". Thus, they must be combined to have a new clause "~(p_speak &p_is_registed_to_t) | ~p1_listen", which presents two actions in the corresponding causal rule. Listing 3.3 has no change because the clauses do not contain any negative conditions.

- **Step 4: generate the sets of dual-Horn clauses**
  Dual-Horn clauses are the disjunctions of variables whereof at most one is negative and, consequently, consists of non-constant *causal rules*. The synthesized CNF clauses in Listing 3.3 contain two clauses having two negative variables (lines 4 and 6). Thus, four sets of dual-Horn clauses are computed, as shown in Listing 3.4.

Listing 3.4: The generated dual-Horn clauses

```
1  (
2      (~(p_speak) | p1_is_different_with_p & TP02speakptt_broadcast & p_is_registered_to_TP02speakptt &
           p1_is_registered_to_TP02speakptt)
3      & (~(p_listen) | p_is_registered_to_TP02listenptt & TP02listenptt_broadcast)
4      & (~(p1_listen) | p1_is_different_with_p & p1_is_registered_to_TP02listenptt & TP02listenptt_broadcast &
           p_is_registered_to_TP02listenptt)
5      & (~(p1_speak))
6      & (p_speak | p_listen)
7      & (~(p_speak))
8  )
9  | (
10     (~(p_speak) | p1_is_different_with_p & TP02speakptt_broadcast & p_is_registered_to_TP02speakptt &
           p1_is_registered_to_TP02speakptt)
11     & (~(p_listen) | p_is_registered_to_TP02listenptt & TP02listenptt_broadcast)
12     & (~(p1_listen) | p1_is_different_with_p & p1_is_registered_to_TP02listenptt & TP02listenptt_broadcast &
           p_is_registered_to_TP02listenptt)
13     & (~(p_speak))
14     & (p_speak | p_listen)
15     & (~(p_speak))
16 )
17 | (
18     (~(p_speak) | p1_is_different_with_p & TP02speakptt_broadcast & p_is_registered_to_TP02speakptt &
           p1_is_registered_to_TP02speakptt)
19     & (~(p_listen) | p_is_registered_to_TP02listenptt & TP02listenptt_broadcast)
20     & (~(p1_listen) | p1_is_different_with_p & p1_is_registered_to_TP02listenptt & TP02listenptt_broadcast &
           p_is_registered_to_TP02listenptt)
21     & (~(p1_speak))
22     & (p_speak | p_listen)
23     & (~(p_listen))
24 )
25 | (
26     (~(p_speak) | p1_is_different_with_p & TP02speakptt_broadcast & p_is_registered_to_TP02speakptt &
           p1_is_registered_to_TP02speakptt)
27     & (~(p_listen) | p_is_registered_to_TP02listenptt & TP02listenptt_broadcast)
28     & (~(p1_listen) | p1_is_different_with_p & p1_is_registered_to_TP02listenptt & TP02listenptt_broadcast &
           p_is_registered_to_TP02listenptt)
29     & (~(p_speak))
30     & (p_speak | p_listen)
31     & (~(p_listen))
32 )
```

Since each dual-Horn clause is a *causal rule*, a dual-Horn set is a *system of causal rules* defined in Definition 3.3.3 [39].

**Definition 3.3.3 (System of Causal rules)**

A *system of causal rules* is a set $R = \{p \Longrightarrow x_p\}_{p \in P^t}$, where $P^t \stackrel{def}{=} P \cup \{true\}$. An interaction $a \in 2^P$ satisfies the system $R$ (denoted $a \models R$), iff $a \models \bigwedge_{p \in P^t}(p \Longrightarrow x_p)$. We denote by $|R|$ the union of interactions satisfying $R$:

$$|R| \stackrel{def}{=} \sum_{a \models R} a$$

Listing 3.5 shows four systems of causal rules for the corresponding dual-Horn clauses in Listing 3.4 by removing conditions.

Listing 3.5: The systems of causal rules corresponding to the sets of dual-Horn clauses

```
1  {
2      p_speak => TP02speakptt_broadcast,
3      p_listen => TP02listenptt_broadcast,
4      p1_listen => TP02listenptt_broadcast,
5      p1_speak => false,                    // ~(p1_speak)
6      true => p_speak | p_listen,
7      p_speak => false
8  },
9  {
10     p_speak => TP02speakptt_broadcast,
11     p_listen => TP02listenptt_broadcast,
12     p1_listen => TP02listenptt_broadcast,
13     p_speak => false,
14     true => p_speak | p_listen,
15     p_speak => false
16 },
17 {
18     p_speak => TP02speakptt_broadcast,
19     p_listen => TP02listenptt_broadcast,
20     p1_listen => TP02listenptt_broadcast,
21     p1_speak => false,
22     true => p_speak | p_listen,
23     p_listen => false
24 },
25 {
26     p_speak => TP02speakptt_broadcast,
27     p_listen => TP02listenptt_broadcast,
28     p1_listen => TP02listenptt_broadcast,
29     p_speak => false,
30     true => p_speak | p_listen,
31     p_listen => false
32 }
```

The causal rules can be simplified as follows:

$$\{p \implies a_1, p \implies a_2\} \rightsquigarrow \{p \implies a_1 \wedge a_2\} \tag{3.7}$$

Furthermore, notice that $a_1 \vee a_1 a_2 = a_1$, thus causal rules can be also simplified accordingly:

$$(p \implies (a_1 \vee a_1 a_2)) \rightsquigarrow (p \implies a_1) \tag{3.8}$$

All the causal rules are simplified and absorbed using (3.7) and (3.8). Listing 3.6 illustrates the system of causal rule following simplification and absorption. In this listing, the causal rule $p \implies false$ and its port $p$ are also removed.

Listing 3.6: The systems of causal rules after simplifying and absorbing

```
1  {
2      p_listen => TP02listenptt_broadcast,
3      p1_listen => TP02listenptt_broadcast,
4      true => p_listen
5  },
```

```
6  {
7      p_listen  => TP02listenptt_broadcast,
8      p1_listen  => TP02listenptt_broadcast,
9      true  => p_listen
10 },
11 {
12     p_speak => TP02speakptt_broadcast,
13     p1_listen  => TP02listenptt_broadcast,
14     true  => p_speak
15 },
16 {
17     p1_listen  => TP02listenptt_broadcast,
18 }
```

Each system of causal rules contains causal rules with constant (cf. Definition 3.3.1), including $true \implies \bigvee_i^n p_i$ and $p_i \implies true$, where $p_i$ are actions in the considered system of causal rules. Listing 3.7 presents the systems of causal rules after adding the causal rules with constant.

Listing 3.7: The systems of causal rules after adding causal rules with constant

```
1  {
2      p_listen  => TP02listenptt_broadcast,
3      p1_listen  => TP02listenptt_broadcast,
4      p_listen  => true, p1_listen  => true, TP02listenptt_broadcast => true,
5      true  => p_listen | p1_listen  | TP02listenptt_broadcast
6  },
7  {
8      p_listen  => TP02listenptt_broadcast,
9      p1_listen  => TP02listenptt_broadcast,
10     p_listen  => true, p1_listen  => true, TP02listenptt_broadcast => true,
11     true  => p_listen | p1_listen  | TP02listenptt_broadcast
12 },
13 {
14     p_speak => TP02speakptt_broadcast,
15     p1_listen  => TP02listenptt_broadcast,
16     p_speak => true, p1_listen => true, TP02listenptt_broadcast => true, TP02speakptt_broadcast => true
17     true  => p_speak | p1_listen | TP02listenptt_broadcast | TP02speakptt_broadcast
18 },
19 {
20     p1_listen  => TP02listenptt_broadcast,
21     p1_listen  => true, TP02listenptt_broadcast => true,
22     true  => p1_listen | TP02listenptt_broadcast
23 }
```

- **Step 5: saturate the systems of causal rules**

  The systems of causal rules need to be saturated to construct BIP connectors.

**Definition 3.3.4 (Saturated System of Causal Rules [39])**

A system of causal rules $\{p_i \implies x_i\}_{i=1}^n$ is *saturated* iff, for all $i \in [1, n]$, $x_i = x_i[(x_j p_j)/p_j]$, where $x_i[(x_j p_j)/p_j]$ is obtained by substituting $(x_j p_j)$ for $p_j$ in $x_i$, for all $j \neq i$.

For example, consider a system of causal rules $\mathcal{CR}(P) = \{p \implies a_p, q \implies pa_q\}$, where $p, q \in P$ are ports, and $a_p, a_q \in 2^P$ are interactions. We obtain the saturated system of causal semantic $\mathcal{CR}_{sat}(P) = \{p \implies a_p, q \implies pa_pa_q\}$ by substituting $pa_p$ for $p$ in the second rule. Obviously, $\mathcal{CR}(P)$ and $\mathcal{CR}_{sat}(P)$ are equivalent:

$$
\begin{aligned}
&(p \implies a_p) \wedge (q \implies pa_q) = (\overline{p} \vee a_p) \wedge (\overline{q} \vee pa_q) \\
&= \overline{p}\,\overline{q} \vee a_p\overline{q} \vee pa_pa_q = \overline{pq} \vee a_p\overline{q} \vee pa_pa_qa_p \\
&= (\overline{p} \vee a_p) \wedge (\overline{q} \vee pa_pa_q) = (p \implies a_p) \wedge (q \implies pa_pa_q)
\end{aligned}
$$

Listing 3.8 shows the saturated systems of causal rules for the corresponding one in Listing 3.7.

Listing 3.8: Saturated systems of causal rules

```
1  {
2      p_listen  => p_listen & TP02listenptt_broadcast,
3      p1_listen  => p1_listen & TP02listenptt_broadcast,
4      TP02listenptt_broadcast => true,
5      true  => TP02listenptt_broadcast
6  },
7  {
8      p_listen  => p_listen & TP02listenptt_broadcast,
9      p1_listen  => p1_listen & TP02listenptt_broadcast,
10     TP02listenptt_broadcast => true,
11     true  => TP02listenptt_broadcast
12 },
13 {
14     p_speak => TP02speakptt_broadcast & p_speak,
15     p1_listen  => p1_listen & TP02listenptt_broadcast,
16     TP02listenptt_broadcast => true,
17     TP02speakptt_broadcast => true,
18     true  => TP02listenptt_broadcast | TP02speakptt_broadcast
19 },
20 {
21     p1_listen  => p1_listen & TP02listenptt_broadcast,
22     TP02listenptt_broadcast => true,
23     true  => TP02listenptt_broadcast
24 }
```

### 3.3.4 JavaBIP artifacts generation

JavaBIP macros represent BIP connectors in Java implementation to coordinate the components' activities at runtime. Algorithm 2 presents how to generate the macro codes from the dual-Horn clauses representing the systems of causal rules in Listing 3.7.

For each clause in the considered dual-Horn set (lines 5 and 6 in Algorithm 2), if it is a non-constant clause (lines 8), save it into the `requires` set (line 9) to generate corresponding JavaBIP macro for require part (lines 17-19). For example, the clause "~( p_speak) | TP02speakptt_broadcast" is used to synthesize JavaBIP macro "port(Peer. class

, "speak").requires(Tracker.class, "broadcast")". In this transformation, the instances' names are replaced with the corresponding class names in the design (line 16).

Each sub-set in the `dualHorn_clauses_sets` contains a clause presenting the causal rule "$true \implies \vee_{i=1}^{n} p_i$" (line 10). The set of `accepts` is computed by the union operation of such clauses (line 11) in all the sub-set. Based on the `accepts`, we generate "`accepts_macro`" by each element in `accepts` accepts all the others (lines 20-22).

Listing 3.9 explains the process of generating "`accepts_macro`". From the set of actions (line 2), we replace instances names with classes name (line 5) then generate "accepts" part for JavaBIP macro (lines 8-10). Finally, the JavaBIP macro is generated as shown in Listing 3.10.

**Algorithm 2** JavaBIP macro generation

```
 1: Input: dualHorn_clauses                         ▷ sets of dual-Horn clauses
 2: Output: javaBIP_GlueBuilder                     ▷ JavaBIP macros
 3: init accepts
 4: init requires
 5: for sub_set ∈ dualHorn_clauses_sets do
 6:     for clause ∈ sub_set do
 7:         if clause.pos ≠ ∅ then
 8:             if clause.neg ≠ ∅ then              ▷ This is a non-constant clause
 9:                 requires := requires ∪ {clause}
10:             else                                ▷ This clause contains set of interactions
11:                 accepts := accepts ∪ clause.pos
12:             end if
13:         end if
14:     end for
15: end for
16: //replace instance names with corresponding class names in requires and accepts
17: for clause ∈ requires do
18:     require_macro += "port("+clause.neg+").requires("+",".join(clause.pos)+");"
19: end for
20: for elm_i ∈ accepts do
21:     accepts_macro += "port(" + elm_i + ").accepts(" + ",".join (accepts \ {elm_j})
    + ");"
22: end for
23: return require_macro + accept_macro
```

Listing 3.9: Accept parts generated from the set of actions

```
1 //accepts — set of actions
2 accepts = {'TP02listenptt_broadcast', 'p_speak', 'TP02speakptt_broadcast', 'p_listen', 'p1_listen'}
3
4 // replace instances names with classes name
5 accepts = {'Tracker.broadcast', 'Peer.speak', 'Tracker.broadcast', 'Peer. listen ', 'Peer. listen '}
```

```
6
7  // generate corresponding accepts_macro
8  port(Peer.class, "listen").accepts(Peer.class, "speak", Peer.class, "listen", Tracker.class, "broadcast");
9  port(Peer.class, "speak").accepts(Peer.class, "listen", Tracker.class, "broadcast");
10 port(Tracker.class, "broadcast").accepts(Tracker.class, "broadcast", Peer.class, "speak", Peer.class, "listen");
```

Listing 3.10: Generated JavaBIP macros

```
1 port(Peer.class, "listen").requires(Tracker.class, "broadcast");
2 port(Peer.class, "speak").requires(Tracker.class, "broadcast");
3 port(Peer.class, "listen").accepts(Peer.class, "listen", Tracker.class, "broadcast", Peer.class, "speak");
4 port(Peer.class, "speak").accepts(Peer.class, "listen", Tracker.class, "broadcast");
5 port(Tracker.class, "broadcast").accepts(Peer.class, "listen", Tracker.class, "broadcast", Peer.class, "speak");
```

The `GlueBuilder` contains JavaBIP macros and data transfers between components at runtime. To generate such information, we consider the conditions. If the condition template is "$\langle instance_1 \rangle$ **is/has/can something** $\langle instance_i \rangle$" (where $i \geq 0$) and the subject of $\langle instance_1 \rangle$ is different from the subject of $\langle instance_i \rangle$, there is a data transfer from $\langle subject_i \rangle$ to $\langle subject_1 \rangle$. Thus, the generated data transfer for requirement **TP_02** is "data(Tracker.class,"Tracker2Peer_data").to(Peer.class,"Tracker2Peer_data");". This data transfer is generated from condition "`is_registered_to`" which is for "TP02speaktp_is_registered_to_t, TP02listentp_is_registered_to_t, p1_is_registered_to_t".

As mentioned in Section 2.3.3, JavaBIP classes contain `@Guard` functions specifying the condition to fire FSM's transitions. Each condition is considered as a `@Guard` (i.e., a boolean function). The boolean functions are generated as a template with the default returning value set as "*True*". Depending on the context of the requirements, developers write the actual content for those functions. The parameters of those functions are generated depending on the kind of $\langle subject_i \rangle$:

- If $\langle subject_i \rangle$ is empty, there is no parameter,
- If $\langle subject_i \rangle$ is the same as $\langle subject_1 \rangle$, generate a normal parameter,

```
In Peer.java:
    @Guard(name="is_different_from")
    public boolean is_different_from(Peer Peer_ins) {
        return true;
    }
```

- If $\langle subject_i \rangle$ is different from $\langle subject_1 \rangle$, the generated parameter is a `@Data` received through the specified data transfers.

```
In Peer.java:
    @Guard(name="is_registered_to")
    public boolean is_registered_to(@Data(name="Tracker2Peer_data") Tracker
        Tracker_ins) {
        return true;
```

}

### 3.3.5 BIP Connectors generation

Algorithm 3 describes the process of generating BIP connectors. This algorithm is a variation of the algorithm constructing a causal interaction tree from a saturated system of causal rules [39], and the correctness argument for our algorithm is similar to that in [39]. The input of Algorithm 3 is a set of atomic interactions computed from the system of causal rules. Let $X = \{p \Longrightarrow x_p\}_{p \in P^t}$ be a saturated system of causal rules, with $x_p = \bigvee_{i=1}^{n_p} a_i^p$, $atomic_interaction_set = \{pa_p^p | p \in P \cup \{true\}, i \in [1, n_p]\}$. The output is BIP connectors, which are the textual representation of $\mathcal{AC}(P)$.

**Algorithm 3** BIP connectors generation

```
    Input: ais                ▷ set of atomic interactions computed from dual-Horn clauses
    Output: bip_connector                          ▷ BIP connectors formalized in AC(P)
 1: procedure genConnector(ais)
 2:     initalize triggers, remains, result_str
 3:     triggers = {ais_i | card(ais_i) = 1}
 4:     remains = ais \ triggers
 5:     triggers = triggers ∪ get_intersection(remains)
 6:     if triggers = ∅ then
 7:         result_str += gen_rendezvous_str (remains)
 8:     else
 9:         result_str += gen_triggers_str (triggers)
10:     end if
11:     for each trigger in triggers do
12:         dependent_elements = get_dependent_elements(remains, trigger)
13:         if is_connector(dependent_elements) then
14:             result_str += genConnector(dependent_elements)
15:         else
16:             result_str += gen_rendezvous_str(dependent_elements)
17:         end if
18:     end for
19:     return result_str
20: end procedure
```

Consider a set of interactions $\{p, pq, pqrt\}$, and each element is a set of ports. Thus, the input for the algorithm is "ais = $\{\{p\}, \{p, q\}, \{p, q, r, t\}\}$". In the first steps, the algorithm detects and separates the triggers (triggers) (i.e., "$\{\{p\}\}$") and the remaining elements (remains) (i.e., "$\{\{p, q\}, \{p, q, r, t\}\}$"), where the triggers are sets with a single element (lines 3 and 4).

Function *get_intersection*(`remains`) takes input as the `remains` to compute an interaction that is a sub-interaction of all `remains`' elements. If the computed one is not empty and does not contain elements in `triggers`, it will be added into `triggers` (line 5). In this example, although the intersection of "$\{p, q\}$" and "$\{p, q, r, t\}$" is "$\{p, q\}$", `triggers` contains "$\{p\}$". Thus, `triggers` does not add "$\{p, q\}$" into it.

Considering `triggers`, if `triggers` is empty, the input presents a Rendezvous connector (lines 6 and 7). Function *gen_rendezvous_str*(`remains`) generates the textual representation of the Rendezvous connector. For instance, if `triggers` $= \{\}$ and `remains` $= \{\{p, q\}\}$, `result_str` is "`(p)-(q)`" (i.e., $pq$ in $\mathcal{AC}(\mathrm{P})$). Otherwise, generating the textual representation of triggers (line 9). At this step, because `triggers` $= \{\{p\}\}$ the `result_str` appends "`(p)'`" using function *gen_triggers_str*(`triggers`).

In the next step, function *get_dependent_elements*(`remains`, `trigger`) calculates elements in `remains`which are dependent on each element in `triggers` (line 12). For example, for `triggers` $= \{\{p\}\}$ and `remains` $= \{\{p, q\}, \{p, q, r, t\}\}$, the function returns $\{\{q\}, \{q, r, t\}\}$. If `dependent_elements` is a connector (line 13), invoke the algorithm recursively with the input is `dependent_elements` (line 14). In particular, function *is_connector*(`dependent_elements`) checks whether the intersection of any two elements in `dependent_elements` is not empty. If it is not empty, `dependent_elements` is considered a set that constructs a connector. In this example, `dependent_elements` is a connector because the intersection of $\{q\}$ and $\{q, r, t\}$ is $\{q\}$ (i.e., not empty). Thus, we run the algorithm recursively on the set $\{\{q\}, \{q, r, t\}\}$. Following the above steps, there is another trigger (i.e., "`(q)'`") and the `dependent_elements = ['r', 't']`. If `dependent_elements` is not a connector (line 15), generate a textual representation for *synchron* elements as a Rendezvous connector (line 16). As a result, the BIP connector of $\{\{q\}, \{q, r, t\}\}$ is $q'[rt]$ (the corresponding textual representation is "`(q)'-[(r)-(t)]`") and the final BIP connector is $p'[q'[rt]]$ (the corresponding textual representation is "`(p)'-[(q)'-[(r)-(t)]]`").

Listing 3.11 illustrates the generated BIP connectors (lines 7-9) from the sets of atomic interactions (lines 2-4) conforms to the saturated systems of causal rules in Listing 3.8.

Listing 3.11: Generated BIP connectors

```
1  // sets of atomic interactions
2  {{{'TP02listenptt_broadcast'}, {'p_listen', 'TP02listenptt_broadcast'}, {'p1_listen', 'TP02listenptt_broadcast'
       }},
3  {{'p1_listen', 'TP02listenptt_broadcast'}, {'p_speak', 'TP02speakptt_broadcast'}, {'TP02listenptt_broadcast'}, {'
       TP02speakptt_broadcast'}},
4  {{'p1_listen', 'TP02listenptt_broadcast'}, {'TP02listenptt_broadcast'}}}
5
6  // corresponding generated BIP connectors
7  (Tracker.broadcast)'-(Peer.listen)-(Peer.listen)
8  (Tracker.broadcast)'-(Tracker.broadcast)'-(Peer.speak)-(Peer.listen)
```

```
9 (Tracker.broadcast)'—(Peer. listen )
```

## 3.4 Summary

Based on the ontology-driven requirement engineering and semantic analysis approach, we proposed NaturalBIP—a domain-specific language to specify the functional requirements of cloud applications. In this language, we defined grammar rules and templates with ontology-based semantics to tackle the ambiguity of the natural language syntax and express the semantic relationships and implicit assumptions through the information in the NRO.

We also proposed the NaturalBIP compiler for producing BIP connectors and JavaBIP GlueBuilder to coordinate component interactions at runtime following an exogenous approach. The compilation includes:

1. analyzing the requirements to collect quantifiers and actions defined in the design and explore the system's conditions/constraints, then generate corresponding Boolean formulas for each requirement,
2. computing dual-Horn clauses representing all possible allowed interactions of the system,
3. generating BIP connectors specify the interactions between the system's components,
4. generating JavaBIP artifacts, including JavaBIP macros, data transfers, and boolean functions (i.e., guards) for classes,
5. handles some basic errors, such as undefined actions or states of classes from the requirement written in NaturalBIP language.

In general, we provide a tool for system designers to write functional requirements explicitly (cf. Section 3.2) and developers to implement the application following an exogenous approach (cf. Section 3.3). We demonstrated the whole process through the example of Tracker-Peer communication.

# Chapter 4

# Towards Exogenous Coordination of Concurrent Cloud Applications

This chapter proposes a methodology to develop and maintain cloud applications following the exogenous approach. To illustrate the idea, we propose a new framework named OCCIwareBIP, which integrates JavaBIP—a framework for the exogenous coordination of concurrent Java components into OCCIware—a framework for designing cloud applications. From the OCCIwareBIP design, we generate an executable model to verify the deadlock-free property of the cloud application. Finally, we present an example to show the ability of our approach to guarantee the safety and benefits of modularization in developing cloud applications.

The rest of this chapter is structured as follows: We introduce the context and motivation leading to our contributions in Section 4.1. Section 4.2 specifies an example used in the rest of the chapter. Section 4.3 provides an overview of our OCCIwareBIP toolchain and presents algorithms to generate artifacts supporting the exogenous coordination and deadlock-freedom verification. Section 4.4 demonstrates how the example is implemented to evaluate the proposed approach. Finally, we conclude the chapter in Section 4.5.

## 4.1 Introduction

Cloud computing has become the dominant delivery model for computing resources [24]. It plays an essential role in several business models and academic research by providing services such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) [16]. When cloud applications run, there is little control over their resource use and requirements. The applications must be able to dynamically adapt their behaviors to the changes in cloud resource availability. Essentially, any application entity, beyond simply computing a specific function, must communicate and share resources with other entities. Correct coordination of resource access between concurrent

entities of cloud applications is critical for meeting user and system requirements while avoiding operational faults and deadlocks.

In traditional development, the codes coordinating entities' access to available resources are often interleaved with software components' business functionality. This complicates application maintenance when facing policy changes. Although maintenance can be supported by change impact analysis [82, 111, 112], this process takes time and effort.

*Exogenous* models and languages [32] were introduced to deal with this problem. The *exogenous* approach distinctly separates computation and coordination code. This separation enhances the reusability of components. Furthermore, other applications with similar interaction patterns may reuse the coordination specification. The advantages of exogenous coordination are supporting and permitting verification techniques to compute components' code and dependencies between them [25]. Behavior-Interaction-Priority (BIP) [27, 36] is a framework for the component-based design of correct-by-construction applications. It provides a mechanism for coordinating concurrent components. JavaBIP [37, 92] is an open-source Java implementation of the BIP framework for coordinating concurrent components, relying on annotations, component APIs, and external specification files.

The main contribution of this chapter is to propose a methodology to develop and maintain cloud applications following the exogenous approach. To illustrate the idea, we propose a new framework named OCCIwareBIP, which integrates the OCCIware design framework with the exogenous coordination capability using JavaBIP. In the OCCIwareBIP framework, we introduce new concepts for describing action types and specifying behavioral constraints in our NaturalBIP language. Our framework allows developers to generate an executable model to safely manage cloud entities' access to the resources at run time. We leverage the inherent modularity of exogenous coordination to enable cloud architects easily substitute coordination policies according to the user and system requirements. In addition, the OCCIwareBIP supports users in generating input for iFinder [6] from the cloud application design and the configuration models. iFinder is a compositional deadlock detection tool to ensure the design is deadlock-free.

## 4.2 Motivation & Running Example

This section describes an example that we use to illustrate our approach to ensuring the safety of concurrent cloud application development and the modularity it provides for the specification of coordination policies. Intuitively, *safety properties* state that *a specified bad condition never occurs*. In concurrent cloud applications, safety properties can formalize access rules to resources. For example, “The number of sequential requests to the same server will never exceed a given threshold”. However, the current design

platforms have no means to specify such information. Thus, it leads to the research question (**RQ3**): *How to support cloud designers with a means to write a high-level abstraction design of a cloud application graphically?*

In the life cycle of cloud applications, the functionalities can be updated depending on the change in system/user requirements, which are unpredictable. Modularity provides "option values" for creating new or improving existing functions in software evolution, primarily when a system must satisfy unforeseen future demands [87]. For instance, in the (fictional) example below, we consider four different situations triggering the deployment of a database to be used by the application. Modular applications, where the coordination and computation codes are separated, can be easily adapted to the change of requirements by substituting the appropriate components and adjusting the coordination specification without modifying the functional components. Thus, we need to answer the research question (**RQ4**): *How to effectively adapt to the change in both the design and implementation phases?*

To demonstrate our proposed approach, we use the example of Monitor-Switch, a Web application deployed on a Heroku Dyno to direct the requests to a list of servers. In particular, this Web application ensures that when a server reaches the specified threshold for the number of requests, the subsequent activities are switched to run on another server. The application is initialized by choosing a server from a list of servers. The user can then click the button "*compute random number*" to request a number from 0 to 100 computed by the microservice *ComputeRN*.

The Web application consists of two components, including *Switch* and *Monitor*. *Switch* provides actions to pick a list of servers (i.e., action *addServer*), choose an initial server from the list (i.e., action *chooseServer*), or remove all the picked servers (i.e., action removeAllServers). *Monitor* communicates to the *Switch* through action *receiveSwitchConfirm*. Then, the user can send requests to the server using *receiveRandomNumberRequest*. The server returns a random number if the number of requests is within the allowed threshold. Otherwise, the *Monitor* will redirect the request to another server. The switching process is synchronized between the *Switch* and *Monitor*.

**Structural Constraints.** In the case where the *Heroku Dyno* was created without any database add-on, it is necessary to be able to deploy one for saving data when running the *Monitor-Switch* application.

**Behavioral Constraints.** 1) The application can request random numbers from a specific server at most five times; 2) We consider four different (fictional) scenarios for adding a database add-on to the current *Heroku Dyno* from the running Web application:

1. User clicks on the "*Create Database add-on*" button (i.e., Monitor).

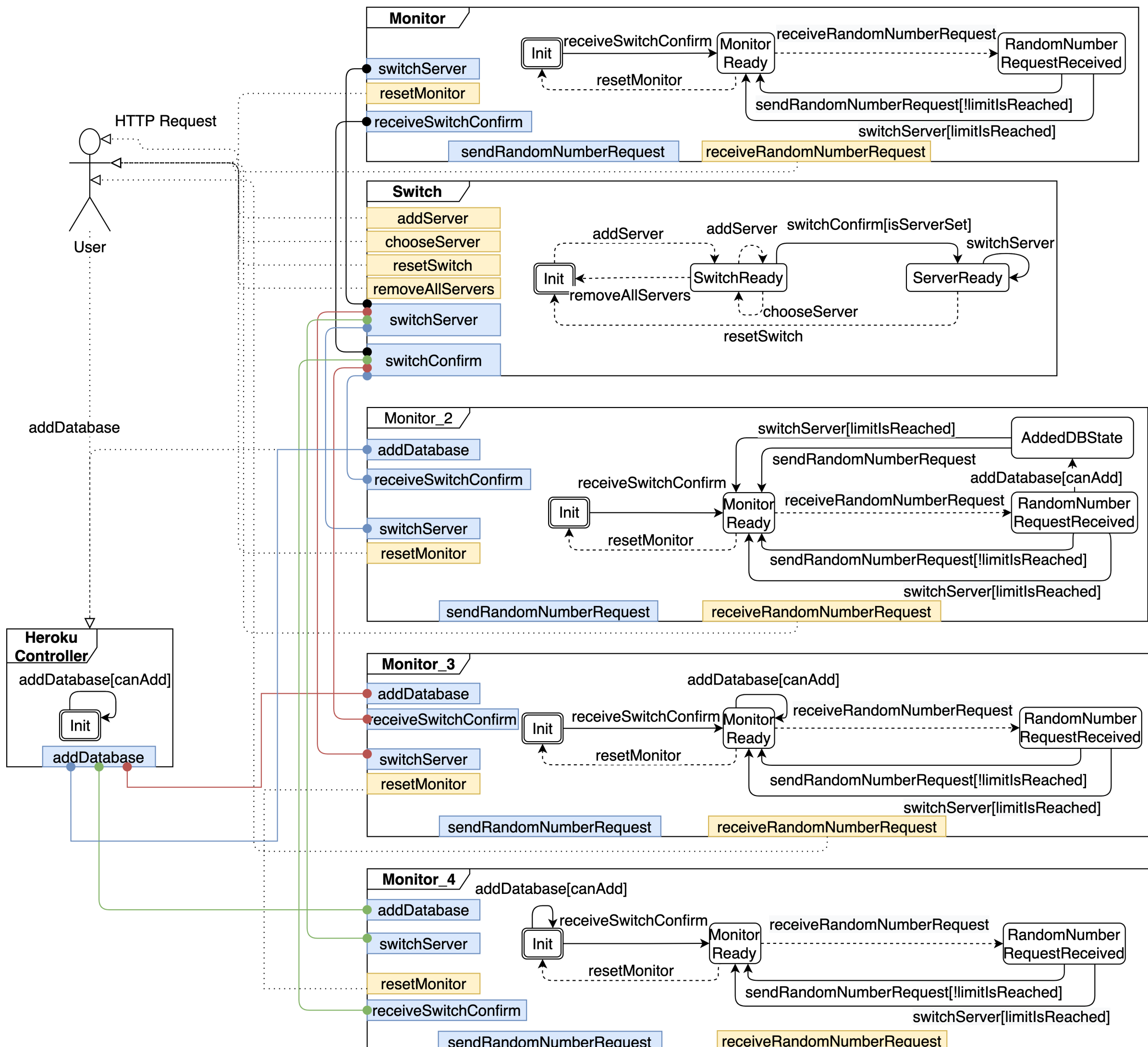


Figure 4.1: The overview of *Monitor-Switch* Web application

2. The returned number is greater than 70 (i.e., *Monitor_2*): When the user sends a request to the server, and the server calculates a random number value greater than 70, trigger the *addDatabase* action. We assume the random number returned represents the server resource consumption from the corresponding request. Under the above assumption, when the consumption exceeds a certain threshold (i.e., 70), then trigger *addDatabase*.

3. the limitation is nearly reached (i.e., *Monitor_3*): Before reaching the limit of requests, the *addDatabase* will be fired.

4. the system switches to another server (i.e., *Monitor_4*): After switching servers, the *addDatabase* will be fired.

Action *addDatabase* in components *Monitors* is synchronized with the corresponding action in the *HerokuController* component.

Figure 4.1 presents the overview of the Web application *Monitor-Switch*, which consists of a `Switch` and four different scenarios of `Monitor`. Each component has an FSM, where the dash arrows denote actions executed by users, and the connectors present the interactions between these components.

## 4.3 Methodology of extending coordination capability of the OCCIware design

This section describes the workflow of the OCCIwareBIP toolchain, which extends the OCCIware framework to allow the safe management of resources at run time, as shown in Figure 4.2. The light blue boxes show pre-existing artifacts from the OCCIware framework we extended; the dark blue ones are developed in this work.

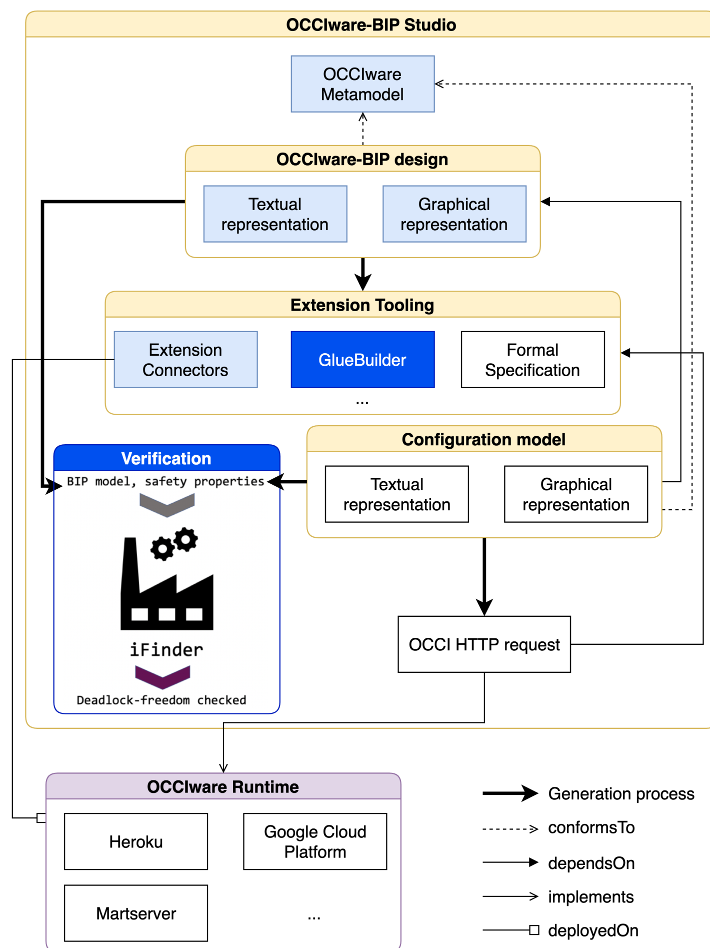


Figure 4.2: Model-Driven Managing Everything as a Service with OCCIwareBIP

To assist OCCIware architects in designing cloud applications following the exogenous approach, we extend the OCCIware Metamodel described in [131] by defining different concepts required to model components and the coordination between them (Section 4.3.1). Section 4.3.2 describes algorithms to prepare the BIP model and the ".inv" file (i.e., instructions to compute the system's invariants) for the verification using the iFinder model checker. Since the algorithm to generate JavaBIP macro has been introduced in Chapter 3, we describe how to generate the implementation of the component and prove the equivalence between JavaBIP macro and generated BIP connector in Section 4.3.3.

### 4.3.1 Concepts for extending coordination capability in the OCCIware design

As introduced in Section 4.1, the OCCIware design supports users to specify cloud components and their structural constraints. In contrast, behavior constraints are implemented by developers in the implementation phase. With this approach, the developers must track and analyze impacted codes whenever behavior constraints change, and this process is time-consuming. To quickly adapt to those changes, we defined the concept "`Specification`" specifying those behavior constraints in design.

**Concepts for specifying behavioral constraints.** Figure 4.3 illustrates the OCCIwareBIP design of a Web application named *Monitor-Switch*, where `Monitor`, `Monitor_2`, `Monitor_3`, `Monitor_4`, `Switch`, and `HerokuController` are components (the `Monitor_i` components encode the corresponding scenario discussed above); and `Specification` is the new concept specifying the coordination between those components. Components' behaviors can be represented by an FSM, consisting of a finite set of states and a finite set of transitions between these states. The transitions are associated with the corresponding actions of each OCCIware component.

Listing 4.1: Requirements are specified using "`Specification`"

```
1 <annotations name="Specification">
2     <annotation id="switchServer_3">For any Monitor_3 m, there is a Switch s, such that m is
          reached the threshold , m shall switchServer synchronized with s shall switchServer.</
          annotation>
3     <annotation id="switchConfirm_3">For any Monitor_3 m, there is a Switch s, m shall
          receiveSwitchConfirm synchronized with s shall switchConfirm.</annotation>
4     <annotation id="addDB_3">For any Monitor_3 m, there is a HerokuController hc, such that hc can
          add database, m shall addDatabase synchronized with hc shall addDatabase.</annotation>
5     <annotation id="switchServer_4">For any Monitor_4 m, there is a Switch s, m shall switchServer
          synchronized with s shall switchServer.</annotation>
6     <annotation id="switchConfirm_4">For any Monitor_4 m, there is a Switch s, m shall
          receiveSwitchConfirm synchronized with s shall switchConfirm.</annotation>
```

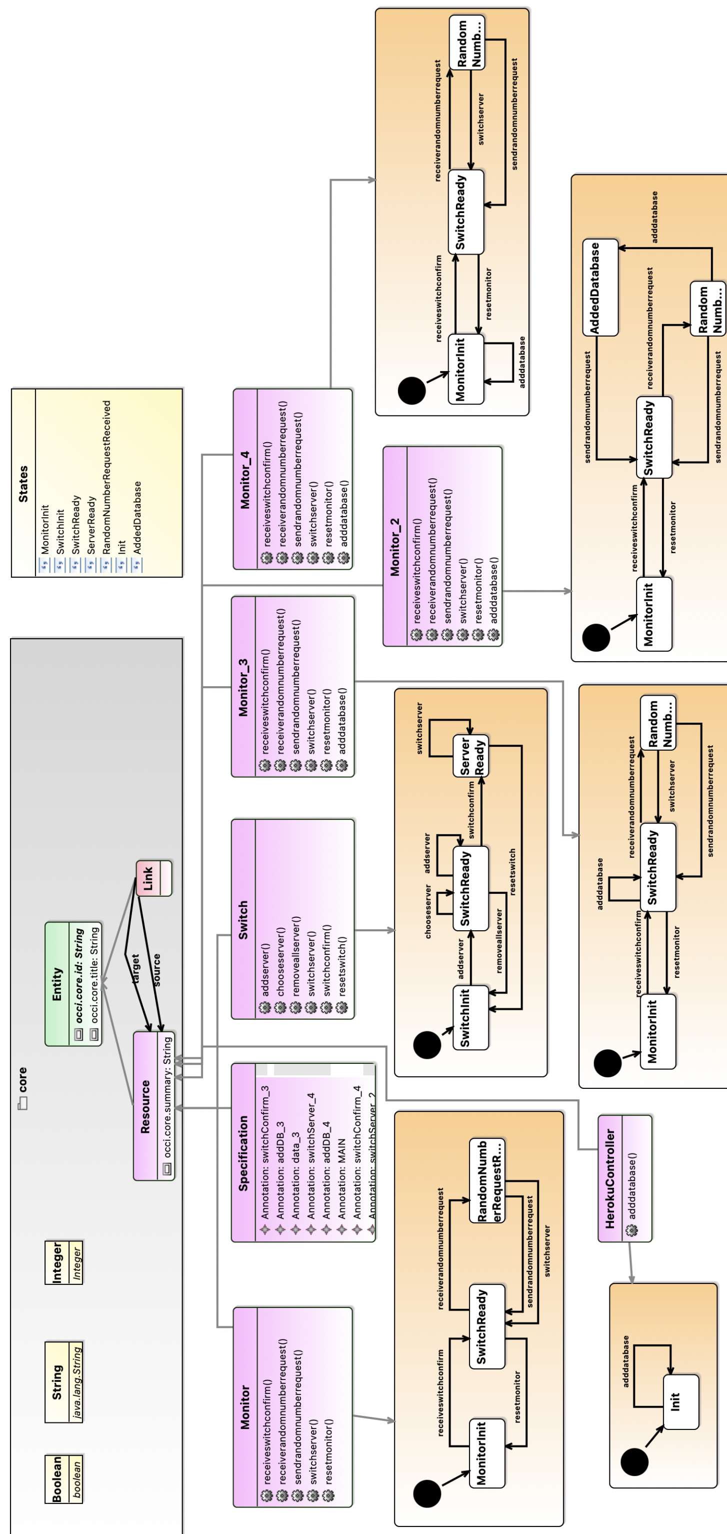


Figure 4.3: The OCCIwareBIP design of the *Monitor-Switch* Web application

```
7    <annotation id="addDB_4">For any Monitor_4 m, there is a HerokuController hc, such that hc can
         add database, m shall addDatabase synchronized with hc shall addDatabase.</annotation>
8    ...
9    <annotation id="MAIN">switchServer_3, switchConfirm_3, addDB_3</annotation>
10 </annotations>
```

Listing 4.1 shows the `Specification` in details. Each *annotation* specifies:

- A functional requirements written in our NaturalBIP language (lines 2-9); or
- Annotation "`MAIN`" indicates the selected requirements from the list of requirements (line 10). To select all the requirements, the value of "`MAIN`" can be set as "*all*". To select all except a small set of requirements, "`MAIN`" is written in the template "except *requirement_i*", where *requirement_i* are deselected requirements.

**Domain-Specific Ontology for the Cloud Domain.** To express specification from the cloud domain, we construct a DSO encompassing the cloud domain by mapping the OCCIware metamodel concepts to the corresponding classes in the Behavioral Ontology (see Section 3.2.1.1) as shown in Table 4.1.

Table 4.1: Mapping between the Behavioral Ontology classes and the OCCIware metamodel concepts

| **OCCIware metamodel concepts** | **Behavioral Ontology classes** |
|---|---|
| Kind | Subject |
| Mixin | Subject |
| FSM | FSM |
| Transition | Transition |
| State | State |
| Action | Action |

Figure 4.4 illustrates our DSO for the cloud domain, which is the semantic model for writing specifications in the NaturalBIP language, following the mapping in Table 4.1.

**Concepts for specifying action types.** When extending the coordination capability of the OCCIware design, actions in the OCCIware design should be able to describe observed events (i.e., *spontaneous*), reactions when observing the event (i.e., *enforceable*), or internally updates the component's states (i.e., *internal*). Therefore, we update the OCCI Metamodel by defining a new property named "actiontype" to specify each action type (Figure 4.5).

When declaring actions in the OCCIware design, if the value of "*actiontype*" is `spontaneous` or `internal`, the transition is `spontaneous` or `internal`, respectively.

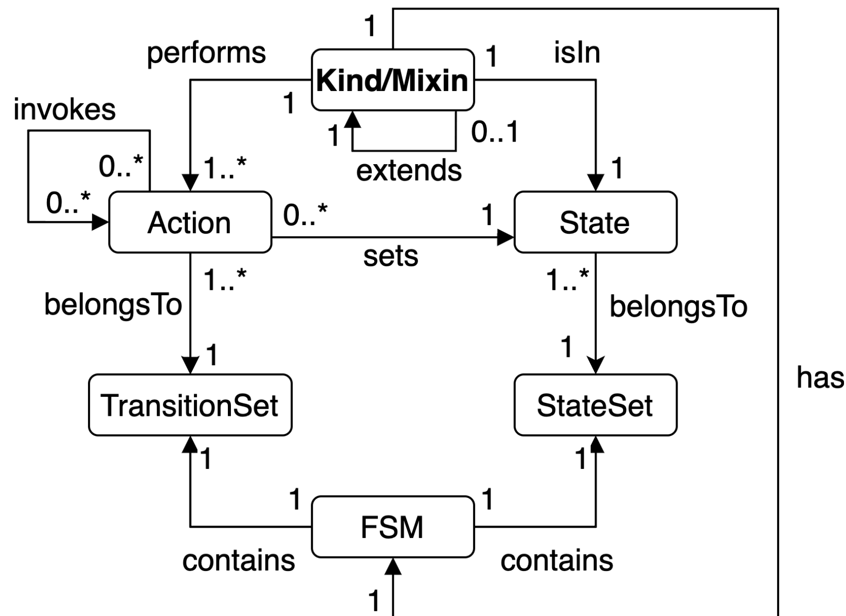


Figure 4.4: Domain-specific ontology for the cloud domain

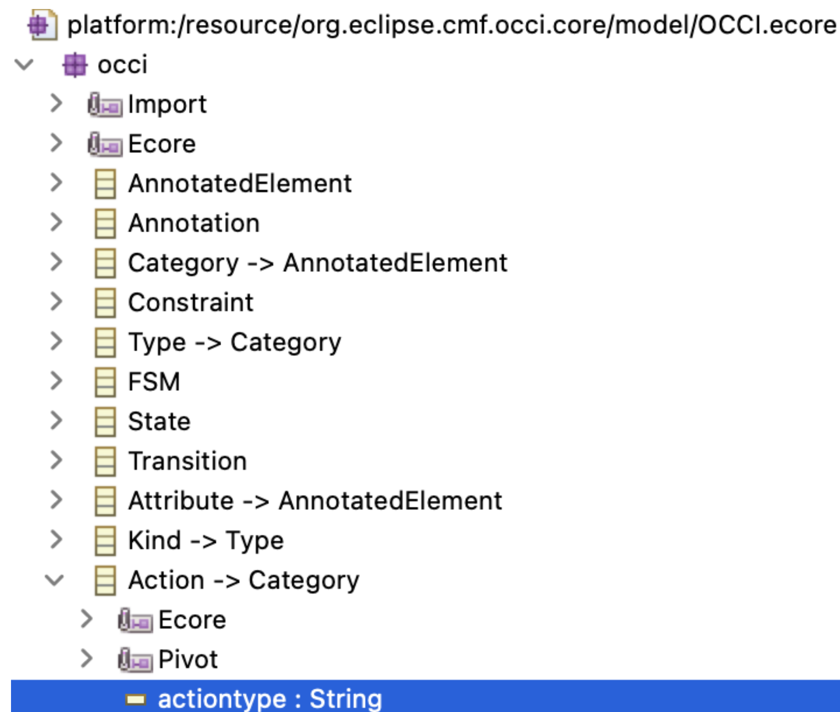


Figure 4.5: Update new property in the OCCI metamodel

The transition is `enforceable` if the value of "*actiontype*" is `enforceable` or empty (by

default). Figure 4.6 depicts that the action `chooseServer` in component `Switch` is a spontaneous action.

Properties
Name: chooseServer
Scheme: http://occiwarebip.org/monitorswitch/switch/action#
Title:
Description:
Actiontype: spontaneous

Figure 4.6: The type of action `chooseServer` is `spontaneous`

### 4.3.2 Generate artifacts for verification

The design can be verified from the specified behavioral constraints to check whether it satisfies some safety properties and is free from deadlocks. In this work, we use iFinder—a compositional deadlock detection and verification tool for BIP models. iFinder expects a BIP model, a *.inv* file, and a set of linear safety properties on input. The *.inv* file specifies the instructions for computing the system's invariants. It is generated as follows:

- Each component in the OCCIwareBIP design is generated following the template "−at <OCCIwareBIP Component> −a atom−control"
- Specifying a compound element to control the whole system "−ct <Application_Name>Compound −a control−reachability"

Options `-at` and `-ct` represent atom type and compound type, respectively. Option `-a` specifies the applied analysis method (see Section 2.3.4).

The BIP model is generated from the OCCIwareBIP design and the OCCI configuration model. Since the BIP model encodes behavior constraints for specific instances, as opposed to component types (Kinds and Mixins) specified in OCCIware, a configuration model is needed to generate BIP connectors. The pseudo-code in Algorithm 4 presents the steps for generating BIP connector definitions. The input is the textual representation of the BIP connector, and the output is the set of connector type definitions in the BIP language. Each definition represents a flat connector, while the input connector can be hierarchical. Thus, a BIP connector might be represented by more than one connector type definition.

**Algorithm 4** Generate connectors for BIP models

```
    Input: connector                        ▷ BIP connector in textual representation
    Output: list_connector_definition ▷ corresponding connector type definitions
 1: TreeNode root = createTree("root", connector)
 2: list_connector_definition = ""
 3: genConnectorTemplate(root)
 4: procedure genConnectorTemplate(TreeNode node)
 5:     for child in children do
 6:         genConnectorTemplate(child)
 7:     end for
 8:     if children.size() > 0 then
 9:         list_connector_definition += "connector type " + nodeName +
    "_define("
10:         for i in children.size() do
11:             list_connector_definition += addParameter("Port p" + i)
12:         end for
13:         if nodeName != "root" then
14:             list_connector_definition += "\n export port Port ep()"
15:         end if
16:         list_connector_definition += "\n define "
17:         for i in children.size() do
18:             if children[i].isTrigger() then
19:                 list_connector_definition += "p" + i + "’"
20:             else
21:                 list_connector_definition += "p" + i
22:             end if
23:         end for
24:         list_connector_definition += "\n end \n"
25:     end if
26: end procedure
```

In the initial state, we encode the textual representation of the BIP connector to a tree structure (line 1). Each node has *nodeName* property to specify its name and *children* property to describe its child nodes. Consider the BIP connector from input as a tree with a root node (i.e., *nodeName* is “`root`”), leaf nodes (i.e., *nodeName* is port type), and compound nodes (i.e., *nodeName* is “c$i$”, where $i$ is the index number). For instance, connector $c$ in Figure 4.7 can be presented as a tree with a `root` node having a leaf node named `s.act` and a compound node named `c1`. `c1` has two leaf nodes `t.act` and `v.act`. The next step generates the connector type definitions by executing function genConnectorTemplate() for the root node (line 3).

The algorithm is first recursively applied to each child node (lines 5-7). If the node under consideration is not a leaf, it is a compound node corresponding to a sub-connector

and is represented by a connector type (lines 8-12). If the *nodeName* is not "root" (i.e., this node is a sub-connector), an export port should be declared to allow that sub-connector to interact with siblings (lines 13-15). An apostrophe after the variable name denotes a trigger in the BIP language (line 19). Finally, from the generated template functions of connectors and the given configuration model defining the port type and corresponding port instances, we generate concrete connectors, which are the combinations of the port's instances.

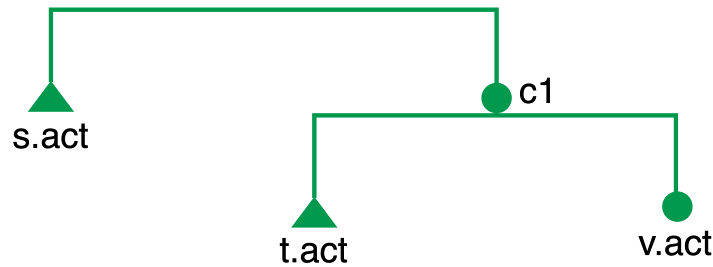


Figure 4.7: Connector $c$ in the tree structure

Consider connector c = (s.act)'−[(t.act)'−(v.act)] in Figure 4.7, let the configuration model config_model = {s:{s0,s1}, t:{t0,t1}, v:{v0}} specify instances of each class s, t, v. The template of the connector is generated by function genConnectorTemplate() (line 5 - Algorithm 4) as lines 1-7 in Listing 4.2. Firstly, function genConnectorTemplate() generates the template for c1 (lines 1-4). Then, the connector type root is generated to connect port s.act and the export port (ep) of c1 (lines 5-8).

Finally, the concrete connectors will be generated as a list of all possible combinations of component instances in the configuration model (lines 13-19). Since connector c1 is the sub-connector of connector c, it interacts with port instances of "s" through the export port "ep" (lines 16-19).

Listing 4.2: generated BIP connectors in the BIP language for connector c

```
1  connector type connector_c1_define(Port p1, Port p2)
2      export port Port ep()
3      define p1' p2
4  end
5  connector type connector_root_define(Port p1, Port p2)
6      define p1' p2
7  end
8  compound type connectorCompound()
9      component t t0, t1
10     component s s0, s1
11     component v v0
12
13     connector connector_c1_define connector_c1_0(t0.act, v0.act)
```

```
14      connector connector_c1_define connector_c1_1(t1.act, v0.act)
15
16      connector connector_root_define connector_root_0(s0.act, connector_c1_0.ep)
17      connector connector_root_define connector_root_1(s0.act, connector_c1_1.ep)
18      connector connector_root_define connector_root_2(s1.act, connector_c1_0.ep)
19      connector connector_root_define connector_root_3(s1.act, connector_c1_1.ep)
20  end
```

### 4.3.3 Integration of JavaBIP into OCCIware implementation

Following the exogenous approach, the application is implemented with the separation of the computation code and coordination code. The computation codes are Java classes generated by mapping information from the OCCIware design, where:

- Kind and Mixin components in the OCCIware design are the Java classes with the annotation `@ComponentType`.
- Each action in the OCCIware component is a function with annotation `@Transition` labeled by the action's name from one state to another state or itself.

The coordination code that describes the interaction between the components and the data exchange is the *GlueBuilder* class, which is generated following Section 3.3.4.

In [39], it is shown that the connectors and the set of dual-Horn clauses that they are generated from define the same set of interactions. The textual representations we generate for JavaBIP (GlueBuilder macros) and BIP (connectors) correspond precisely to the dual-Horn clauses and the generated connectors. They are, therefore, also equivalent among themselves. Thus, verification results obtained on the BIP model remain valid for the JavaBIP one.

## 4.4 Evaluation

This section demonstrates how the example specified in Section 4.2 is developed following the process in Figure 4.2. In particular, the process steps are as follows:

- Step 1. Create the OCCIware design for the target application.
- Step 2. Generate artifacts for verification and implementation.
- Step 3. Specify safety properties. Then, use those properties, the BIP model, and the .inv file computed from Step 2 as the input for iFinder to verify whether the OCCIware design is deadlock-free.
- Step 4. Write concrete code of the generated template Java classes to complete the implementation.

### 4.4.1 OCCIware design and configuration model

Figure 4.3 presents the OCCIware design of the *Monitor-Switch*. This Web application illustrates the advantages of the exogenous approach (i.e., ensuring modularity). It consists of different policies for reacting to various events. Our approach allows users to specify those policies as components, and then users can pick ones following the corresponding synchronization constraints.

Four *Monitor* components (i.e., *Monitor*, *Monitor_2*, *Monitor_3*, and *Monitor_4*) for adding a database add-on into the current *Heroku Dyno*, corresponding to scenarios specified in Section 4.2. Component *HerokuController* is the API invoking a function from *HerokuDeployer* to create a database add-on.

OCCIware Studio also provides a means to create the configuration model, which consists of concrete instances of components defined in the OCCIware design [131]. For example, Listing 4.3 illustrates the configuration model, where each component has an instance.

Listing 4.3: The configuration model of Monitor-Switch application

```
1 configuration use "http://occiwarebip.org/monitorswitch#/" resource "urn:uuid:ecbc6aee−2269−4
    fb8−a413−c9a3fd81f303" : monitorswitch.Monitor title "monitor"
2 { } resource "urn:uuid:256089b1−5d9e−4abd−ba07−a392a929af4e" : monitorswitch.Monitor_2 title
    "monitor_2"
3 { } resource "urn:uuid:92924e87−3871−49e2−96c6−1692d6177796" : monitorswitch.Monitor_3 title
    "monitor_3"
4 { } resource "urn:uuid:dcf8234a−38c9−482f−bd2e−1d81de6140c4" : monitorswitch.Monitor_4 title
    "monitor_4"
5 { } resource "urn:uuid:c317de8b−cb9a−429e−8e87−7fdec132bb3f" : monitorswitch.Switch title "
    switch"
6 { } resource "urn:uuid:c317123b−4444−5555−6666−77c132bb3f77" : monitorswitch.
    HerokuController title "herokucontroller"
7 { }
```

### 4.4.2 Generated artifacts from the OCCIware design

From the OCCIware design, we generate artifacts that include (1) Java classes with JavaBIP annotations, (2) JavaBIP GlueBuilder specifying the coordinations and data transfers, and (3) the textual representation of BIP connectors. Then, taking the BIP connectors and the configuration model, we generate the (4) BIP model and compute (5) instructions for computing invariants of the target application.

**Java classes with JavaBIP annotations.** Once the OCCIware model is completed, developers can trigger the generation process to generate artifacts in `Extension Tooling` (Figure 4.2). `Extension Connectors` contains computation codes with JavaBIP annota-

tions, while coordination codes are separated and defined in `Glue Builder`. Listing 4.4 shows a part of the generated `Monitor_3` class from the corresponding element in the OCCIware design in Figure 4.3.

- The annotation *ComponentType* defines the component's name is `Monitor_3` and the initial state of its FSM is `MonitorInit` (line 7).
- Annotations *@Ports* and *@Port* declare the name and type of kind-specific actions (lines 1-5). For example, `receiveRandomNumberRequest` and `resetMonitor` are *spontaneous* actions, and `sendRandomNumberRequest`, `receiveSwitchConfirm`, `switchServer`, and `addDatabase` are *enforceable* actions.
- Annotation `@Transition` specifies the event name, source state, target state, and a guard expression. Guard expression of a transition can be a single guard or a set of guards using logical operators: negation '!', the conjunction '&', and disjunction '|'. For instance, lines 10-12 depicts a transition labeled `sendRandomNumberRequest` from `RandomNumberRequestReceived` to `SwitchReady`. This transition can be fired if and only if it satisfies the guard "`!is_reached_the_threshold`", which means the threshold is not reached.
- `@Guard`: this annotation is used for specifying boolean methods to define conditions to trigger a transition (lines 16 and 19).
- Annotation `@Data` is declared before a non-void method to define data provided by a component to other components. `@Data` can also be used in method parameters (line 20) to represent received data from another one.

Listing 4.4: `Monitor_3` class with JavaBIP annotations (cf. the automaton in Figure 4.3)

```
1  @Ports({
2  @Port(name = "receiveRandomNumberRequest", type = PortType.spontaneous)
3  , @Port(name = "sendRandomNumberRequest", type = PortType.enforceable)
4  , @Port(name = "addDatabase", type = PortType.enforceable), @Port(name = "switchServer", type =
       PortType.enforceable)
5  , @Port(name = "receiveSwitchConfirm", type = PortType.enforceable) , @Port(name = "
       resetMonitor", type = PortType.spontaneous)
6  })
7  @ComponentType(initial = "MonitorInit", name = "monitorswitch.connector.Monitor_3")
8  public class Monitor_3 extends HttpServlet{
9
10   @Transitions({
11         @Transition(name = "sendRandomNumberRequest", source = "
               RandomNumberRequestReceived", target = "SwitchReady", guard = "!
               is_reached_the_threshold"),
12     })
```

```
13      public void sendRandomNumberRequest() throws IOException {...}
14      // similarly for the other kind-specific actions
15
16      @Guard(name = "is_reached_the_threshold")
17      public boolean is_reached_the_threshold() {...}
18
19      @Guard(name="can_add_database_through")
20      public boolean can_add_database_through(@Data(name="HerokuController2Monitor_3_data")
            HerokuController HerokuController_ins) {}
21  }
```

**JavaBIP GlueBuilder and BIP connectors.** As shown in Listing 4.1, there are three selected requirements are "*switchServer_3*", "*switchConfirm_3*", and "*addDB_3*". Thanks for the NaturalBIP Compiler (Section 3.3), the selected requirements are parsed and analyzed to compute the JavaBIP macro codes and the data transfers between components. Listing 4.5 shows the generated textual representation of BIP connectors (lines 4, 10, and 16) and JavaBIP GlueBuilder from the selected requirements.

Listing 4.5: The generated JavaBIP GlueBuilder

```
1   public class GlueBuilder_Specification extends GlueBuilder {
2       @Override
3       public void configure(){
4           //switchServer_3: (Switch.switchServer)-(Monitor_3.switchServer)
5           port(Switch.class, "switchServer").requires(Monitor_3.class, "switchServer");
6           port(Monitor_3.class, "switchServer").requires(Switch.class, "switchServer");
7           port(Switch.class, "switchServer").accepts(Monitor_3.class, "switchServer");
8           port(Monitor_3.class, "switchServer").accepts(Switch.class, "switchServer");
9
10          //switchConfirm_3: (Monitor_3.receiveSwitchConfirm)-(Switch.switchConfirm)
11          port(Switch.class, "switchConfirm").requires(Monitor_3.class, "receiveSwitchConfirm");
12          port(Monitor_3.class, "receiveSwitchConfirm").requires(Switch.class, "switchConfirm");
13          port(Monitor_3.class, "receiveSwitchConfirm").accepts(Switch.class, "switchConfirm");
14          port(Switch.class, "switchConfirm").accepts(Monitor_3.class, "receiveSwitchConfirm");
15
16          //addDB_3: (Monitor_3.addDatabase)-(HerokuController.addDatabase)
17          port(HerokuController.class, "addDatabase").requires(Monitor_3.class, "addDatabase");
18          port(Monitor_3.class, "addDatabase").requires(HerokuController.class, "addDatabase");
19          port(Monitor_3.class, "addDatabase").accepts(HerokuController.class, "addDatabase");
20          port(HerokuController.class, "addDatabase").accepts(Monitor_3.class, "addDatabase");
21
22          data(HerokuController.class,"HerokuController2Monitor_3_data").to(Monitor_3.class,"
                HerokuController2Monitor_3_data");
23      }
24  }
```

**BIP model and the instruction for computing invariants.** From the OCCIware design (Figure 4.3) and the configuration model (Listing 4.3), we apply the process in Section 4.3.2 to generate the BIP model and compute the instructions. Listing 4.6 illustrates a part of the generated BIP model. Since three requirements specify three *synchon* connectors, the BIP model contains three definitions of connector (lines 6-14). Based on the configuration model, the compound type declares instances (lines 17-22) and generates concrete connectors (lines 23-25).

Listing 4.6: The generated BIP model

```
1  package monitorswitch
2      port type Port()
3      atom type Switch()
4      end
5      ...
6      connector type switchServer_3_root_define(Port p1, Port p2)
7          define p1 p2
8      end
9      connector type switchConfirm_3_root_define(Port p1, Port p2)
10         define p1 p2
11     end
12     connector type addDB_3_root_define(Port p1, Port p2)
13         define p1 p2
14     end
15
16     compound type monitorswitchCompound()
17         component HerokuController herokucontroller()
18         component Monitor monitor()
19         component Switch switch()
20         component Monitor_2 monitor_2()
21         component Monitor_3 monitor_3()
22         component Monitor_4 monitor_4()
23         connector switchServer_3_root_define switchServer_3_root_0(monitor_3.switchServer, switch.
               switchServer)
24         connector switchConfirm_3_root_define switchConfirm_3_root_0(monitor_3.
               receiveSwitchConfirm, switch.switchConfirm)
25         connector addDB_3_root_define addDB_3_root_0(monitor_3.addDatabase, herokucontroller.
               addDatabase)
26     end
```

Listing 4.7 shows the content of the .inv file, which provides instructions to iFinder to compute invariants of the system, where the components are analyzed using the `atom-control` method (lines 2-7) and the compound element is analyzed using `control-reachability` (line 10). Those analysis methods are introduced in Table 2.3).

Listing 4.7: Instructions for computing system's invariants

```
1  # atom control
```

```
2  -at Switch -a atom-control
3  -at Monitor -a atom-control
4  -at Monitor_2 -a atom-control
5  -at Monitor_3 -a atom-control
6  -at Monitor_4 -a atom-control
7  -at HerokuController -a atom-control
8
9  # compound control reachability
10 -ct monitorswitchCompound -a control-reachability
```

### 4.4.3 Verification using iFinder

Listing 4.8 illustrates the properties that must be preserved to keep the application deadlock-free. The property is written in prefix syntax and presents that instance switch is in state `SwitchInit`, at least one of the instances `monitor`, `monitor_2`, `monitor_3`, and `monitor_4` are in state `MonitorInit`, and instance `herokucontroller` is in state `Init`. It means, in a moment, there is one instance of `Switch`, one instance of `HerokuController`, and at least one instance of `Monitor*` in the initial state (line 2). Line 3 describes there is a moment, if instance `switch` is in state `ServerReady`, monitor `monitor_3` shall add a database (i.e., `monitor_3` is in state `SwitchReady` and instance `herokucontroller` is in `Init`).

Listing 4.8: The properties in the .prop file

```
1  (or
2      (and (= switch_SwitchInit 1) (>= (+ monitor_MonitorInit monitor_2_MonitorInit
           monitor_3_MonitorInit monitor_4_MonitorInit) 1) (= herokucontroller_Init 1))
3      (=> (= switch_ServerReady 1) (and (= monitor_3_SwitchReady 1)) (= herokucontroller_Init 1))
4  )
```

Then, the deadlock-free property of the Monitor-Switch design can be verified by executing the following command:

```
ichecker.sh -p monitorswitch -r monitorswitchCompound -i inv_1.inv -s prop_1.pro
```

The verification process loads 1) package model by the argument `-p monitorswitch`; 2) instructions for computing invariants of the system by the argument `-i inv_1.inv`; and 3) the safety properties by `-s prop_1.pro`. After loading those inputs, *iFinder* generates an SMT model to be checked by Z3. A counter-example is returned if the result is `invalid`.

Fig. 4.8 shows the verification result. The BIP model is satisfied the property described in the ".prop" file. The verification time is acceptable (i.e., 0.733s).

```
trinhlk@ubuntu: ~/Downloads/IFinder/ujf.verimag.bip.ifinder/examples/monitorswitch
[ichecker] generate invariant by 'atom-control' ...
[ichecker] generate invariant by 'atom-control' done
[ichecker] invariant recorded for 'herokucontroller.'
[ichecker] process line '-ct monitorswitchCompound -a control-reachability' ...
[ichecker] locate compound type 'monitorswitchCompound' ... ok
[ichecker] instantiate component type 'monitorswitchCompound' ... ok
[ichecker] check if 'control-reachability' is applicable ... yes
[ichecker] generate invariant by 'control-reachability' ...
[ichecker] construct the Petri net ... |P|=17, |T|=3 done
[ichecker] reached 1 markings
[ichecker] generate invariant by 'control-reachability' done
[ichecker] invariant recorded for ''
[ichecker] process invariant specification 'monitorswitch-scheme.inv' done
[ichecker] load property 'monitorswitch-deadlock.pro' ... done
[ichecker] generate SMT script 'monitorswitch-scheme-monitorswitch-deadlock' ...
 done
[ichecker] run SMT solver ...
valid

real    0m0.733s
user    0m1.101s
sys     0m0.064s
trinhlk@ubuntu:~/Downloads/IFinder/ujf.verimag.bip.ifinder/examples/monitorswitc
h$
```

Figure 4.8: The result of the verification using the iFinder tool

### 4.4.4 Implementing and Adapting to changes

The application is ready after implementing the component's functions as shown in Figure 4.9, where the button "*Compute random number*" is defined to invoke the *ComputeRN* microservice to request a random number, and the "*Create Database add-on*" button is to request *HerokuDeployer* to invoke *addDatabase* function. As described in Section 4.2 and Listing 4.1, the Web application is implemented following the policy `Monitor_3`, which will add database after reaching the fourth request "*Compute random number*".

Additionally, when the request number reaches the threshold (i.e., five requests in sequence), the next request will be redirected to another server. Figure 4.10 illustrates this functionality that the "*current server*" is directed from server "`../compute`" to server "`../compute2`".

Finally, when the developers want to change the policies due to a change in user or system requirements, this can be achieved by simply choosing related annotations in the "`MAIN`" annotation of "*Specification*" from the OCCIware model (see Listing 4.1).

## 4.5 Summary

In the design process, we defined some concepts to specify/update system synchronizations and behavioral constraints in the design. It helps cloud architects adapt quickly to any future system or user requirements change.

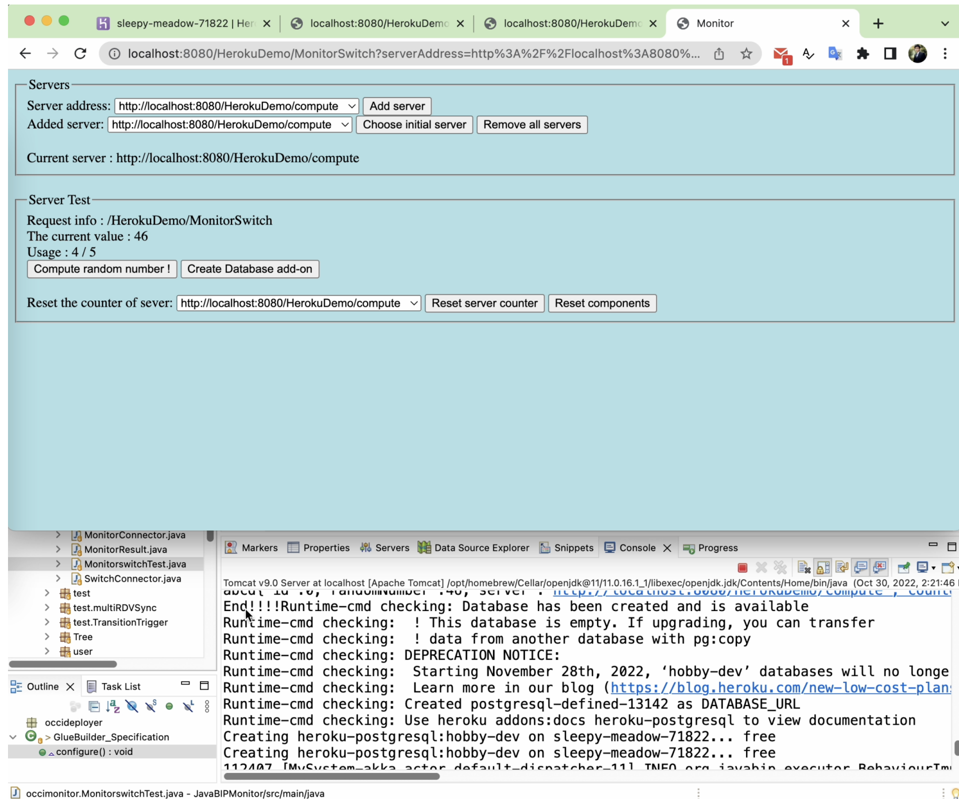


Figure 4.9: The user interface of the *Monitor-Switch* Web application

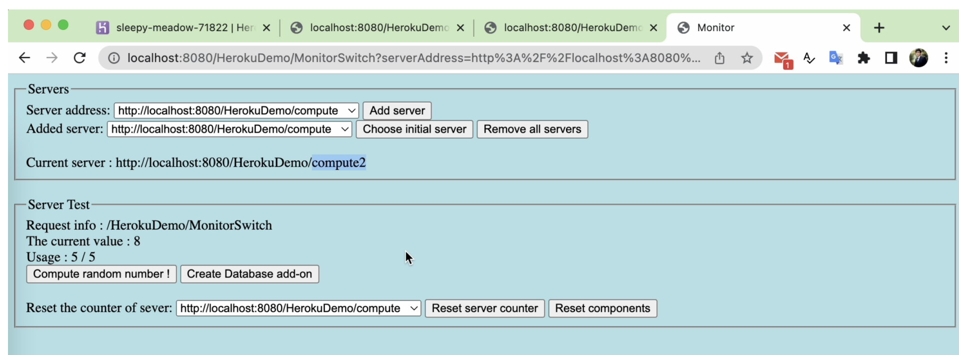


Figure 4.10: After the current server reaches the threshold, the subsequent request will be directed to another server

We introduced the concept of the “actiontype” property to specify the type of actions in the OCCIware design. We leveraged the FSM specifications of components and the OCCIware annotation to write behavioral constraints in NaturalBIP language to generate automatically:

- JavaBIP annotations and macro code for the exogenous coordination of the cloud resource accesses by components at run time.

- The input for the iFinder tool consists of the corresponding BIP model and instructions for computing system invariants.

With the concepts of `Specification` and OCCIware annotations defined in this section, designers can specify the interactions and data transfers between components and the safety properties that need to be held at run time, thus addressing the **RQ3**.

By integrating JavaBIP into OCCIware, we provided a framework to develop cloud applications following the exogenous approach. This approach reduces the dependency between elements and helps developers quickly adapt to the change in requirements by substituting the appropriate components or adjusting the coordination specification independently. Therefore, addressing the **RQ4**. Furthermore, our approach not only supports OCCIware but can also be applied to other open-source cloud design platforms supporting FSM specifications.

# Chapter 5

# Experimental Validation

In this chapter, we illustrate the usage of our toolchain for generating artifacts to support cloud developers in designing, validating, verifying, implementing, and deploying cloud applications in the exogenous approach. We introduce the experiment in Section 5.1. Then, Section 5.2 presents the OCCIwareBIP design of the Heroku Deployer microservice consisting of its structure and the specifications using our NaturalBIP language. In Section 5.3, we present the artifacts generated using our NaturalBIP for verifying and implementing the target application. After completing the generated Java template classes, we deploy the application and run it on the Heroku platform. The detail of this process is described in Section 5.5. Finally, we summary this chapter in Section 5.6.

## 5.1 The overview of Heroku Deployer

*HerokuDeployer* is a microservice for deploying a Web application/microservice automatically on the Heroku platform. *HerokuDeployer* was Simon's project to demonstrate the application of legacy JavaBIP on the cloud. I extend this project to illustrate the usage of our toolchain in the development of cloud applications following the exogenous approach. The experiment includes:

- *HerokuDeployer*: This microservice is developed following the exogenous approach for automatically deploying the Monitor-Switch Web application on Heroku. It can run on IaaS such as Google Compute or the local machine by calling Heroku APIs and executing Heroku CLI commands through the terminal.

  Figure 5.1 shows the overview of the *HerokuDeployer* microservice. The `Dyno type` of the deploying `Heroku Dyno` is set from the two connectors between `HerokuDeployer` and `DynoType` components. Similarly, the region, add-ons, and language of the `Dyno` are specified by connectors from `HerokuDeployer` to `Region` and `Buildpacks` components, respectively. Each language supports a limited set

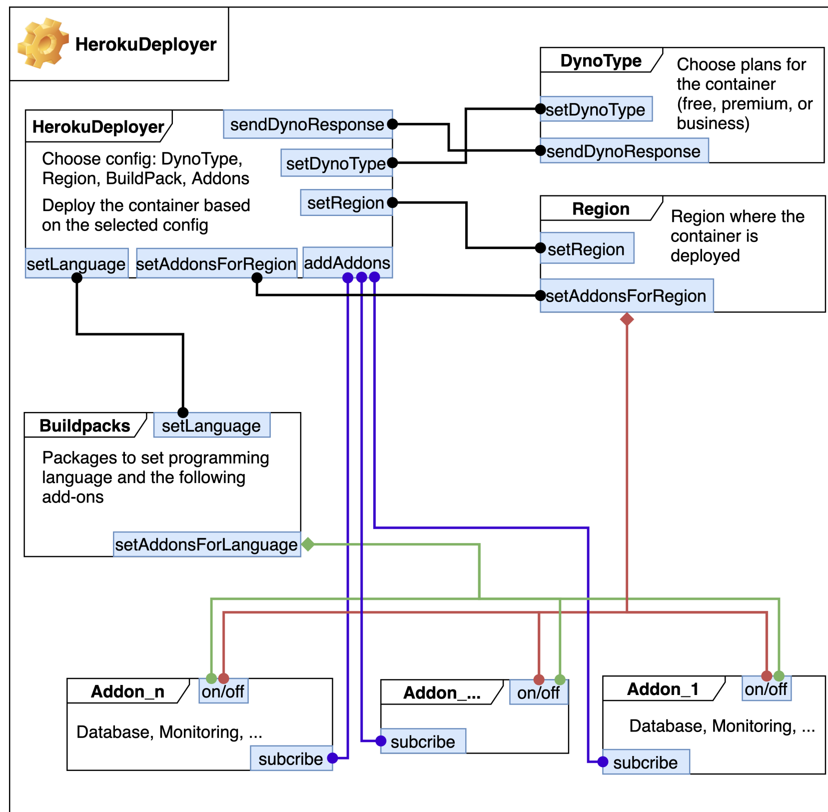


Figure 5.1: The overview of HerokuDeployer microservice

of add-ons. Thus, there is a diamond arrow (i.e, a trigger connector in BIP) from `setAddonsForLanguage` of `Buildpacks` to `Addons` components. In the same way, a diamond arrow from `setAddonsForRegion` of `Region` to `Addons` components.

- *Monitor-Switch*: As introduced in Chapter 4, this Web application illustrates the advantages of the exogenous approach (i.e., ensuring modularity). It consists of different policies for reacting to various events. Our approach allows users to specify those policies as components, and then users can pick ones following the corresponding synchronization constraints.

  Figure 5.2 presents the overview of the Web application *Monitor-Switch*. There are four `Monitor` components (i.e. `Monitor`, `Monitor_2`, `Monitor_3`, and `Monitor_4`) for adding a database add-on into the current `Heroku Dyno`, corresponding to scenarios specified in Section 4.2. Component `HerokuController` is the API invoking a function from `HerokuDeployer` to create a database add-on.

- *Compute*: This is a simple microservice generating a random integer number.

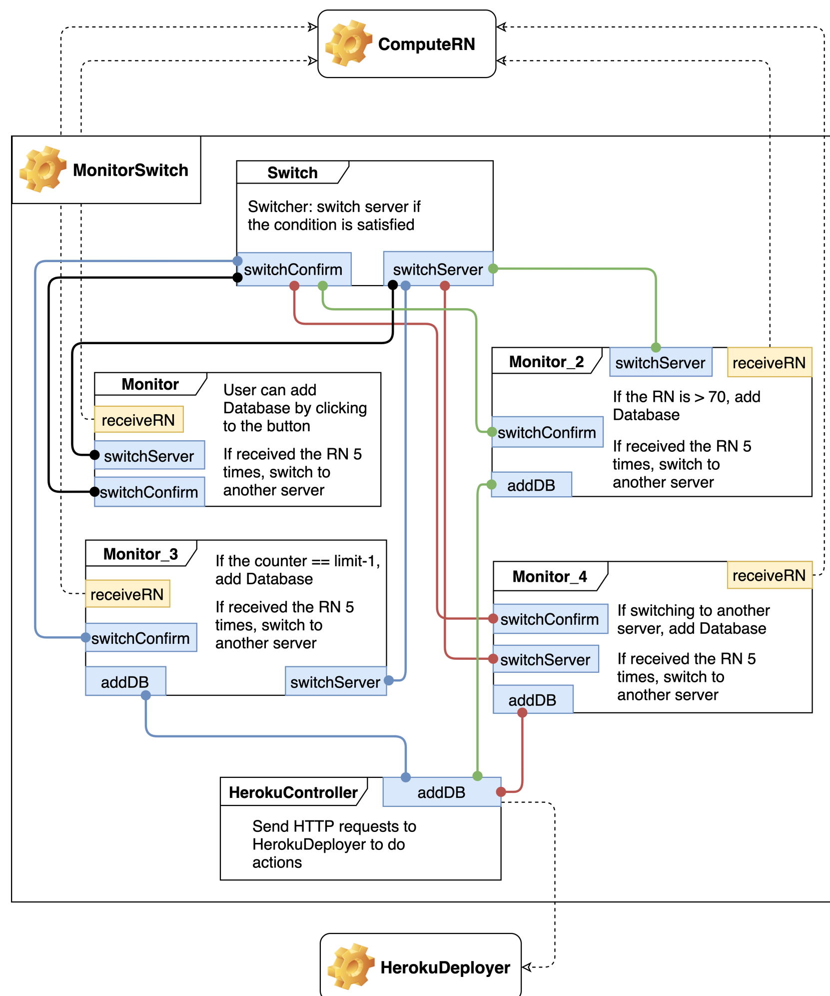


Figure 5.2: The overview of the Monitor-Switch Web application

## 5.2 The HerokuDeployer microservice design

To deploy a Web application/microservice on the Heroku platform, end-users directly set the plans, region, and programming language of the Heroku Dyno—a Linux container running programs in response to a user-specified command[1]; and choose some add-ons *depending on the selected configuration.*

### 5.2.1 The structure of HerokuDeployer

Figure 5.3 provides the OCCIware design of Heroku-Deployer, including:

[1]https://www.heroku.com/dynos

Figure 5.3: The OCCIwareBIP design of *Heroku Deployer*

**HerokuDynoType.** This component is the *Dyno type*[2], which specifies the appropriate plan of the Heroku container (e.g., free, hobby, standard) depending on the project's size, ranging from demo projects to high-traffic production services.

**HerokuRegion.** Users can deploy Heroku applications across many geographies[3]. The regions available to a specific app are determined by whether deployed to the Common Runtime or a Private Space.

**Addons.** Heroku provides add-ons[4], which are fully maintained components, services, or infrastructure provided by a third-party provider or Heroku. In this example, we use four add-ons are:

- *HerokuPostgres*[5] and *HerokuClearDBMySQL*[6]: two add-ons for build applications using SQL databases.
- *HerokuScoutAPM*[7] and *HerokuNewRelicAPM*[8]: two add-ons for performance monitoring and troubleshooting Web applications.

**HerokuBuildpack.** This component contains a collection of scripts for retrieving dependencies, the generated assets, compiled code, and other information depending on the programming language[9]. (The slug compiler[10] assembles this output into a slug. Heroku support for Ruby, Python, Java, Clojure, Node.js, Scala, Go, and PHP is implemented via a set of open-source buildpacks.)

**Deployer.** This component specifies the process of deploying a microservice/Web application on Heroku. Users start choosing the configuration (i.e., DynoType, region, buildpack, add-ons). Then, a Heroku application will be created automatically following the constraints described in the **Specification**.

After choosing the DynoType and region, the Deployer will select the corresponding values through the connections to HerokuDynoType and HerokuRegion. There are some **structural constraints** when selecting add-ons because the available add-ons depend on the region (i.e., HerokuRegion) or the programming language (i.e., HerokuBuildpacks).

[2]https://devcenter.heroku.com/articles/dyno-types
[3]https://devcenter.heroku.com/articles/regions
[4]https://elements.heroku.com/addons
[5]https://www.heroku.com/postgres
[6]https://devcenter.heroku.com/articles/cleardb
[7]https://devcenter.heroku.com/articles/scout
[8]https://devcenter.heroku.com/articles/newrelic
[9]https://devcenter.heroku.com/articles/buildpacks
[10]https://devcenter.heroku.com/articles/slug-compiler

If users select add-ons not supported in a region or by a language, those add-ons cannot be active. For example:

- HerokuPostgres and HerokuClearDBMySQL do not support the language Gradle. Hence, if the users select the language as Gradle, the two add-ons cannot activate;
- HerokuScoutAPM supports Python, Ruby, and Php;
- HerokuNewRelicAPM does not support Scala, Clojure, Gradle, Go, etc

The **behavioral constraint** of the deployment is "User can deploy a free Heroku application in US or EU without the definition of language or add-ons."

### 5.2.2 Writing functional requirements in NaturalBIP language

After determining the components and constraints, we analyze the constraints to have concrete functional requirements. Consider a functionality for choosing a language. We have the following requirements:

- If the user selects Java language, the language in HerokuBuildpack is set as Java through the Deployer.
- When HerokuBuildpack sets the add-ons for Java, the Deployer shall set the add-ons for Java, HerokuPostgres, HerokuClearDBMySQL, and HerokuNewRelicAPM can activate, and HerokuScoutAPM shall not activate.
- When the Deployer sets the add-ons for Java and HerokuPostgres, HerokuClearDB-MySQL and HerokuNewRelicAPM can activate, and HerokuScoutAPM shall not activate, HerokuBuildpack shall set the add-ons for Java.
- If the Deployer decides to add HerokuPostgres, HerokuPostgres shall be added with the free plan.

These requirements will be written in our NaturalBIP language as shown in Listing 5.1. There are 105 functional requirements in Appendix 6 for the microservice to automatically deploy the Heroku application. Listing 5.1 shows a part of the whole requirements. In particular, these requirements describe constraints for setting Java as a language when deploying a Heroku Dyno.

Listing 5.1: Requirements for setting Java as language for a Heroku application

```
1 <annotation id="deployersetJava">There is a HerokuBuildpack buildpack, a Deployer deployer,
    buildpack shall setJava synchronized with deployer shall setJava.</annotation>
2 <annotation id="buildpackpostgres1">There is a HerokuBuildpack buildpack, a HerokuPostgres postgres
    , buildpack shall setAddonsForJava synchronized with postgres shall on.</annotation>
```

```
3 <annotation id="buildpackClearDB1">There is a HerokuBuildpack buildpack, a HerokuClearDBMySQL
     cleardb, buildpack shall setAddonsForJava synchronized with cleardb shall on.</annotation>
4 <annotation id="buildpackScout1">There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout,
     buildpack shall setAddonsForJava synchronized with scout shall off.</annotation>
5 <annotation id="buildpackNewRelic1">There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM
     newrelic, buildpack shall setAddonsForJava synchronized with newrelic shall on.</annotation>
6 <annotation id="buildpackdeployer1">There is a HerokuBuildpack buildpack, a Deployer deployer, if
     buildpack executes setAddonsForJava, deployer shall setAddonsForJava.</annotation>
7 <annotation id="synthesisBuildpack1">There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM
     newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a
     Deployer deployer, if deployer executes setAddonsForJava and cleardb executes on and postgres
     executes on and scout executes off and newrelic executes on, buildpack shall setAddonsForJava.
     </annotation>
```

# 5.3 Generating Java artifacts and implementing the Heroku Deployer

Following the process in Section 4.3, our OCCIwareBIP toolchain generates Java classes with JavaBIP annotations and JavaBIP macros from the specified requirements to coordinate the interactions between components.

## 5.3.1 Artifacts for the verification

As mentioned in Figure 1.2 (Section 3.1), the NaturalBIP Compiler generates BIP connectors from the given functional requirements written in the NaturalBIP language. Listing 5.2 presents some BIP connectors generated from the corresponding requirements in Listing 5.1:

Listing 5.2: The generated BIP connectors for setting Java language

```
1 <annotation id="deployersetJava_1">(Deployer.setJava)—(HerokuBuildpack.setJava)</annotation>
2 <annotation id="buildpackpostgres1_1">(HerokuBuildpack.setAddonsForJava)—(HerokuPostgres.on)</
     annotation>
3 <annotation id="buildpackClearDB1_1">(HerokuBuildpack.setAddonsForJava)—(HerokuClearDBMySQL.
     on)</annotation>
4 <annotation id="buildpackScout1_1">(HerokuBuildpack.setAddonsForJava)—(HerokuScoutAPM.off)</
     annotation>
5 <annotation id="buildpackNewRelic1_1">(HerokuBuildpack.setAddonsForJava)—(HerokuNewRelicAPM.
     on)</annotation>
6 <annotation id="buildpackdeployer1_1">(HerokuBuildpack.setAddonsForJava)'—(Deployer.
     setAddonsForJava)</annotation>
7 <annotation id="synthesisBuildpack1_1">[(HerokuPostgres.on)—(HerokuClearDBMySQL.on)—(
     HerokuScoutAPM.off)—(Deployer.setAddonsForJava)—(HerokuNewRelicAPM.on)]'—(
     HerokuBuildpack.setAddonsForJava)</annotation>
```

### 5.3.2 Artifacts for the implementation

In Figure 5.4, the left column shows the package *occideployer* containing Java classes of the Heroku Deployer application. Each class with the ending "*Connector*" is generated from the corresponding component in the OCCIwareBIP design of Heroku Deployer. The canvas shows a part of generated code for the component *Deployer*.

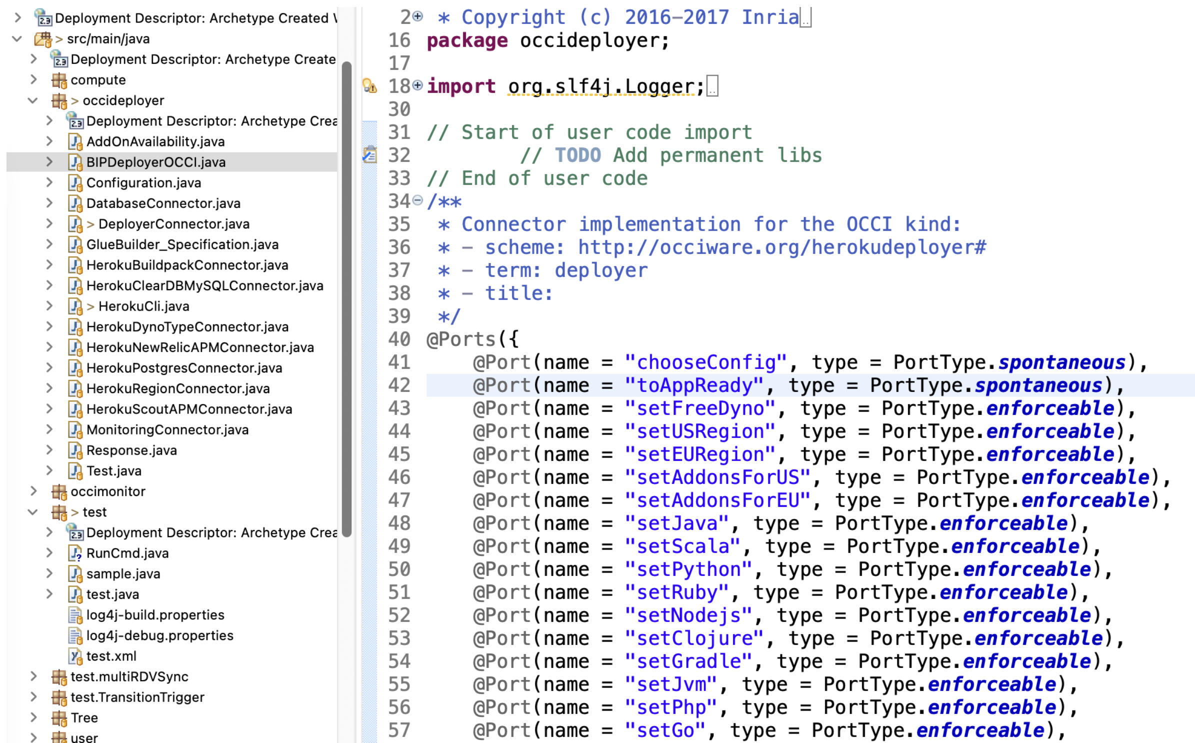


Figure 5.4: Class *DeployerConnector* with generated template code

Class *GlueBuilder_Specification* encodes the coordination between components using the JavaBIP macro. For example, Listing 5.3 illustrates the generated JavaBIP macro for corresponding requirements in Listing 5.1.

Listing 5.3: The generated JavaBIP macro for setting Java language

```
1 //————deploysetJava
2 port(HerokuBuildpackConnector.class, "setJava").requires(DeployerConnector.java, "setJava");
3 port(DeployerConnector.class, "setJava").requires(HerokuBuildpackConnector.java, "setJava");
4 port(HerokuBuildpackConnector.class, "setJava").accepts(DeployerConnector.java, "setJava");
5 port(DeployerConnector.class, "setJava").accepts(HerokuBuildpackConnector.java, "setJava");
6
7 //————buildpackpostgres1
8 port(HerokuPostgresConnector.class, "on").requires(HerokuBuildpackConnector.java, "
      setAddonsForJava");
9 port(HerokuBuildpackConnector.class, "setAddonsForJava").requires(HerokuPostgresConnector.java, "on
      ");
```

```
10 port(HerokuBuildpackConnector.class, "setAddonsForJava").accepts(HerokuPostgresConnector.java, "on");
11 port(HerokuPostgresConnector.class, "on").accepts(HerokuBuildpackConnector.java, "setAddonsForJava");
12
13 //————buildpackClearDB1
14 port(HerokuClearDBMySQLConnector.class, "on").requires(HerokuBuildpackConnector.java, "setAddonsForJava");
15 port(HerokuBuildpackConnector.class, "setAddonsForJava").requires(HerokuClearDBMySQLConnector.java, "on");
16 port(HerokuBuildpackConnector.class, "setAddonsForJava").accepts(HerokuClearDBMySQLConnector.java, "on");
17 port(HerokuClearDBMySQLConnector.class, "on").accepts(HerokuBuildpackConnector.java, "setAddonsForJava");
18
19 //————buildpackScout1
20 port(HerokuScoutConnector.class, "off").requires(HerokuBuildpackConnector.java, "setAddonsForJava");
21 port(HerokuBuildpackConnector.class, "setAddonsForJava").requires(HerokuScoutConnector.java, "off");
22 port(HerokuBuildpackConnector.class, "setAddonsForJava").accepts(HerokuScoutConnector.java, "off");
23 port(HerokuScoutConnector.class, "off").accepts(HerokuBuildpackConnector.java, "setAddonsForJava");
24
25 //————buildpackNewRelic1
26 port(HerokuNewRelicAPMConnector.class, "on").requires(HerokuBuildpackConnector.java, "setAddonsForJava");
27 port(HerokuBuildpackConnector.class, "setAddonsForJava").requires(HerokuNewRelicAPMConnector.java, "on");
28 port(HerokuBuildpackConnector.class, "setAddonsForJava").accepts(HerokuNewRelicAPMConnector.java, "on");
29 port(HerokuNewRelicAPMConnector.class, "on").accepts(HerokuBuildpackConnector.java, "setAddonsForJava");
30
31 //————buildpackdeployer1
32 port(DeployerConnector.class, "setAddonsForJava").requires(HerokuBuildpackConnector.java, "setAddonsForJava");
33 port(HerokuBuildpackConnector.class, "setAddonsForJava").accepts(HerokuBuildpackConnector.java, "setAddonsForJava");
34 port(HerokuBuildpackConnector.class, "setAddonsForJava").accepts(HerokuBuildpackConnector.java, "setAddonsForJava");
35
36 //————synthesisBuildpack1_1
37 port(HerokuBuildpackConnector.class, "setAddonsForJava").requires(DeployerConnector.class, "setAddonsForJava", HerokuClearDBMySQLConnector.class, "on", HerokuPostgresConnector.class, "on", HerokuScoutAPMConnector.class, "off", HerokuNewRelicAPMConnector.class, "on");
38
```

```
39 port(DeployerConnector.class, "setAddonsForJava").requires(HerokuClearDBMySQLConnector.class, "on
   ", HerokuPostgresConnector.class, "on", HerokuScoutAPMConnector.class, "off",
   HerokuNewRelicAPMConnector.class, "on");
40 port(HerokuClearDBMySQLConnector.class, "on").requires(DeployerConnector.class, "setAddonsForJava",
    HerokuPostgresConnector.class, "on", HerokuScoutAPMConnector.class, "off",
   HerokuNewRelicAPMConnector.class, "on");
41 port(HerokuPostgresConnector.class, "on").requires(DeployerConnector.class, "setAddonsForJava",
   HerokuClearDBMySQLConnector.class, "on", HerokuScoutAPMConnector.class, "off",
   HerokuNewRelicAPMConnector.class, "on");
42 port(HerokuScoutAPMConnector.class, "off").requires(DeployerConnector.class, "setAddonsForJava",
   HerokuClearDBMySQLConnector.class, "on", HerokuPostgresConnector.class, "on",
   HerokuNewRelicAPMConnector.class, "on");
43 port(HerokuNewRelicAPMConnector.class, "on").requires(DeployerConnector.class, "setAddonsForJava",
   HerokuClearDBMySQLConnector.class, "on", HerokuPostgresConnector.class, "on",
   HerokuScoutAPMConnector.class, "off");
44
45 port(DeployerConnector.class, "setAddonsForJava").accepts(HerokuClearDBMySQLConnector.class, "on"
   , HerokuPostgresConnector.class, "on", HerokuScoutAPMConnector.class, "off",
   HerokuNewRelicAPMConnector.class, "on", HerokuBuildpackConnector.class, "setAddonsForJava");
46 port(HerokuClearDBMySQLConnector.class, "on").accepts(DeployerConnector.class, "setAddonsForJava",
   HerokuPostgresConnector.class, "on", HerokuScoutAPMConnector.class, "off",
   HerokuNewRelicAPMConnector.class, "on", HerokuBuildpackConnector.class, "setAddonsForJava");
47 port(HerokuPostgresConnector.class, "on").accepts(DeployerConnector.class, "setAddonsForJava",
   HerokuClearDBMySQLConnector.class, "on", HerokuScoutAPMConnector.class, "off",
   HerokuNewRelicAPMConnector.class, "on", HerokuBuildpackConnector.class, "setAddonsForJava");
48 port(HerokuScoutAPMConnector.class, "off").accepts(DeployerConnector.class, "setAddonsForJava",
   HerokuClearDBMySQLConnector.class, "on", HerokuPostgresConnector.class, "on",
   HerokuNewRelicAPMConnector.class, "on", HerokuBuildpackConnector.class, "setAddonsForJava");
49 port(HerokuNewRelicAPMConnector.class, "on").accepts(DeployerConnector.class, "setAddonsForJava",
   HerokuClearDBMySQLConnector.class, "on", HerokuPostgresConnector.class, "on",
   HerokuScoutAPMConnector.class, "off", HerokuBuildpackConnector.class, "setAddonsForJava");
50 port(HerokuBuildpackConnector.class, "setAddonsForJava").accepts(DeployerConnector.class, "
   setAddonsForJava", HerokuClearDBMySQLConnector.class, "on", HerokuPostgresConnector.class, "on
   ", HerokuScoutAPMConnector.class, "off", HerokuNewRelicAPMConnector.class, "on");
```

## 5.4 Verifying the deadlock-freedom using iFinder

iFinder is a compositional deadlock detection and verification tool for BIP models. Regarding the experiment of Heroku Deployer, the input for running iFinder include:

- *herokudeployer.bip*: This is the BIP model of the HerokuDeployer. This file presents each component as Petri nets (lines 1-13 in Listing 5.4) and defines the connector types between them (lines 15-39) and concrete connectors depending on the component's instances (lines 43-50).

Listing 5.4: A part of the generated Heroku Deployer's BIP model

```
1  atom type HerokuDynoType()
2      data string occi_core_title
3      data string occi_core_id
4      data string occi_core_summary
5      export port Port reset1()
6      export port Port sub1()
7      export port Port sendDynoResponse()
8      place Init , Free
9       initial to Init do {}
10     on sub1 from Init to Free do {}
11     on sendDynoResponse from Free to Free do {}
12     on reset1 from Free to Init do {}
13 end
14 ...
15 connector type deployersetJava_1_root_define(Port p1, Port p2)
16     define p1 p2
17 end
18 connector type buildpackpostgres1_1_root_define(Port p1, Port p2)
19     define p1 p2
20 end
21 connector type buildpackClearDB1_1_root_define(Port p1, Port p2)
22     define p1 p2
23 end
24 connector type buildpackScout1_1_root_define(Port p1, Port p2)
25     define p1 p2
26 end
27 connector type buildpackNewRelic1_1_root_define(Port p1, Port p2)
28     define p1 p2
29 end
30 connector type buildpackdeployer1_1_root_define(Port p1, Port p2)
31     define p1' p2
32 end
33 connector type synthesisBuildpack1_1_c00_define(Port p1, Port p2, Port p3, Port p4, Port p5)
34     export port Port ep()
35     define p1 p2 p3 p4 p5
36 end
37 connector type synthesisBuildpack1_1_root_define(Port p1, Port p2)
38     define p1' p2
39 end
40 ...
41 compound type herokudeployerCompound()
42      ...
43     connector deployersetJava_1_root_define deployersetJava_1_root_0(buildpack.setJava, deployer
           .setJava)
44     connector buildpackpostgres1_1_root_define buildpackpostgres1_1_root_0(postgres.fon,
           buildpack.setAddonsForJava)
```

```
45     connector buildpackClearDB1_1_root_define buildpackClearDB1_1_root_0(postgres.fon,
           buildpack.setAddonsForJava)
46     connector buildpackScout1_1_root_define buildpackScout1_1_root_0(postgres.off, buildpack.
           setAddonsForJava)
47     connector buildpackNewRelic1_1_root_define buildpackNewRelic1_1_root_0(postgres.fon,
           buildpack.setAddonsForJava)
48     connector buildpackdeployer1_1_root_define buildpackdeployer1_1_root_0(buildpack.
           setAddonsForJava, deployer.setAddonsForJava)
49     connector synthesisBuildpack1_1_c00_define synthesisBuildpack1_1_c00_0(deployer.
           setAddonsForJava, scout.off, postgres.fon, relic.fon, cleardb.fon)
50     connector synthesisBuildpack1_1_root_define synthesisBuildpack1_1_root_0(
           synthesisBuildpack1_1_c00_0.ep, buildpack.setAddonsForJava)
51 end
```

The BIP connector "`synthesisBuildpack1_1`" is a hierarchical connector with a sub-connector, which is the synchronization of five ports. Thus, this connector is encoded by two connectors in the BIP model, the first one is a rendezvous (i.e., `c00`), and the latter is the broadcast between the exported port from the sub-connector and a normal port.

- *herokudeployer-scheme.inv*: This file provides instruction to iFinder to compute invariants.

Listing 5.5: The .inv file describes the instruction for computing system's invariants

```
1  # atom control
2  -at HerokuBuildpack -a atom-control
3  -at Database -a atom-control
4  -at HerokuClearDBMySQL -a atom-control
5  -at HerokuDynoType -a atom-control
6  -at HerokuNewRelicAPM -a atom-control
7  -at HerokuPostgres -a atom-control
8  -at HerokuRegion -a atom-control
9  -at HerokuScoutAPM -a atom-control
10 -at Monitoring -a atom-control
11 -at Deployer -a atom-control
12
13 # compound control reachability
14 -ct herokudeployerCompound -a control-reachability
```

As shown in Listing 5.5, there are atomic components including `HerokuBuildpack`, `Database`, `HerokuClearDBMySQL`, `HerokuDynoType`, `HerokuNewRelicAPM`, `HerokuPostgres`, `HerokuRegion`, `HerokuScoutAPM`, `Monitoring`, and `Deployer`. The compound element is analyzed using `control-reachability`.

- `herokudeployer-deadlock.pro`: This file describes the property that needs to be held to avoid deadlock situations. As described in Section 5.2.1, we have the

behavioral constraint that "*User can deploy a free Heroku application in US or EU without the definition of language or add-ons.*"

Listing 5.6: The system's properties is specified in prefix format

```
1 (=> (and (= deployer_AppPushed 1) (= dynotype_Free 1) (or (= region_USAddonsSet 1) (=
      region_EUAddonsSet 1)))
2     (and
3         (>= (+ postgres_HobbyDev cleardb_Ignite) 0)
4         (>= (+ scout_Chair relic_Wayne) 0)
5         (>= (+ buildpack_Java buildpack_Scala buildpack_Python buildpack_Ruby
              buildpack_Nodejs buildpack_Clojure buildpack_Gradle buildpack_Jvm buildpack_Php
              buildpack_Go) 0)
6     )
7 )
```

Listing 5.6 illustrates the constraint. The parenthesis in the first line specifies an application is pushed to a free Dyno Type in the US or EU. Lines 2-6 describe that specifying add-ons and languages is not required.

Then, the design is verified whether it satisfies safety properties by executing the following command:

```
ichecker.sh -p herokudeployer -r herokudeployerCompound -i inv_1.inv -s prop_1.pro
```

The verification process loads 1) package model by the argument `-p herokudeployer`; 2) instructions for computing invariants of the system by the argument `-i inv_1.inv`; and 3) the safety properties by `-s prop_1.pro`. After loading those inputs, *iFinder* generates an SMT model to be checked by Z3. A counter-example is returned if the result is `invalid`.

Fig. 5.5 shows the verification result. The BIP model is satisfied the property described in the ".prop" file. The verification time is acceptable (i.e., 0.843s) for checking over 105 BIP connectors between 10 components.

## 5.5 Running the experiment

From the generated JavaBIP artifacts, including Java classes with template codes and JavaBIP macro (Section 5.3.2), developers complete codes for template Java classes and then run them with the coordination described in JavaBIP macro. In this running, we use APIs supporting HTTP Get methods to deploy the *Monitor-Switch* application onto Heroku free container (Dyno) in the US with Heroku Postgres database addon and Java language.

**Deploying.** The deployment starts by calling APIs which use HTTP Get methods. For example, the request `http://localhost:8080/HerokuDemo/BIPDeployerOCCI?req=deploy&`

```
trinhlk@ubuntu: ~/Downloads/IFinder/ujf.verimag.bip.ifinder/examples/herokudeployer
[ichecker] generate invariant by 'atom-control' done
[ichecker] process line '-at Deployer -a atom-control' ...
[ichecker] locate atom type 'Deployer' ... ok
[ichecker] instantiate component type 'Deployer' ... ok
[ichecker] check if 'atom-control' is applicable ... yes
[ichecker] generate invariant by 'atom-control' ...
[ichecker] generate invariant by 'atom-control' done
[ichecker] invariant recorded for 'deployer.'
[ichecker] process line '-ct herokudeployerCompound -a control-reachability' ...
[ichecker] locate compound type 'herokudeployerCompound' ... ok
[ichecker] instantiate component type 'herokudeployerCompound' ... ok
[ichecker] check if 'control-reachability' is applicable ... no
[ichecker] process invariant specification 'herokudeployer-scheme.inv' done
[ichecker] load property 'herokudeployer-deadlock.pro' ... done
[ichecker] generate SMT script 'herokudeployer-scheme-herokudeployer-deadlock' .
.. done
[ichecker] run SMT solver ...
valid

real    0m0.843s
user    0m1.338s
sys     0m0.111s
trinhlk@ubuntu:~/Downloads/IFinder/ujf.verimag.bip.ifinder/examples/herokudeploy
er$
```

Figure 5.5: The result of the verification using the iFinder tool

region=us&buildpack=jvm&addon=heroku−postgresql&pushApp=true specifies a *deploy* action to create a free Heroku Dyno in the US with the build pack *JVM* (Java virtual machine), and the add-on Heroku Postgres. The last parameter (i.e., *pushApp*) determines whether the Heroku Dyno will be created in the logged-in Heroku account. When the application is deployed, it returns a JSON file describing the information of container name, region, and application URL, as shown in Figure 5.6. In particular, we run it in localhost with

localhost:8080/Heroku...

{"OwnerEmail":"trinhlk.vnu@gmail.com","AppName":"whispering-retreat-40514","RegionName":"us","WebUrl":"https://whispering-retreat-40514.herokuapp.com/"}

Figure 5.6: The result after deploying the application

Tomcat 9 and JRE 1.8 for the JavaBIP library. To deploy the Monitor-Switch application on Heroku, we package the project in Figure 5.4 to "*HerokuDemo.war*" and then use it to deploy the Heroku application.

**Using the deployed Web application.** After deploying the Web application on Heroku, we have a Heroku Dyno named "`whispering-retreat-40514`" with a free plan with the Heroku Postgres add-on, as shown in Figure 5.7.

Figure 5.7: The deployed Heroku Dyno

Users enter the servlet URL of the corresponding application/microservice to use this application. For instance, when users enter the URL: "whispering−retreat−40514.herokuapp .com/compute", they send a request to the deployed microservice named *compute*, the server will return the content in JSON format as Figure 5.8.

Figure 5.8: Running the microservice *compute*

Similarly, users can run the Web application Monitor-Switch on this Heroku container by entering the following URL: "whispering−retreat−40514.herokuapp.com/MonitorSwitch" (Figure 5.9). Figure 4.9 also describes the *Monitor-Switch* application but running on the local machine, while Figure 5.9 shows the running of the Web application on a Heroku container.

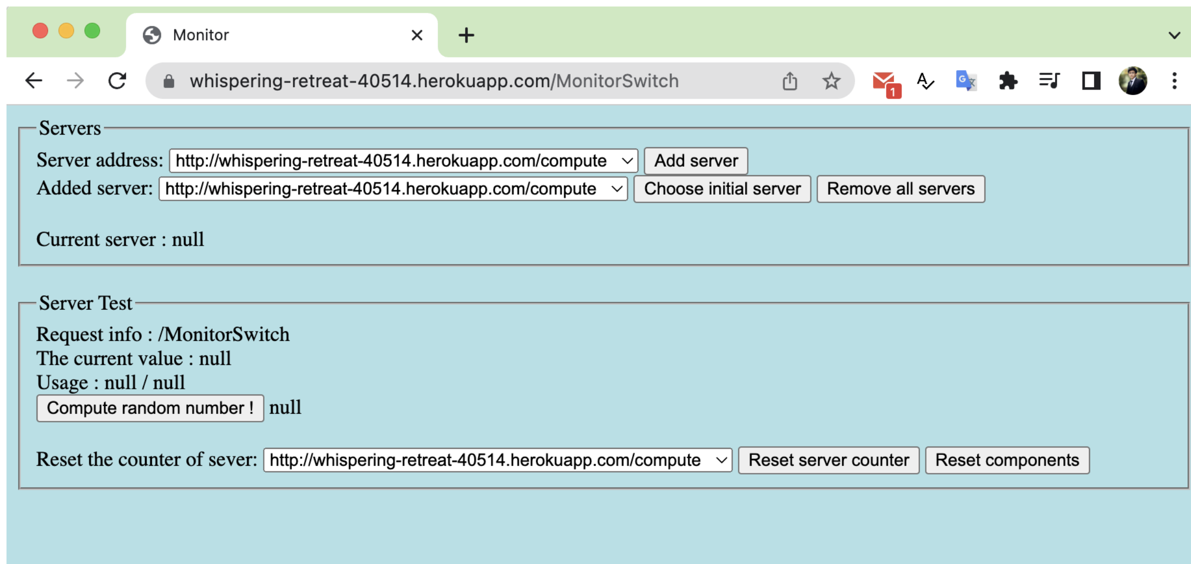


Figure 5.9: Running the Web application *Monitor-Switch*

## 5.6 Summary

This chapter presents the results of our proposal through the development of Heroku Deployer—a microservice to automatically deploy a Web application/microservice on the Heroku platform. Recall that we extend the OCCIware design to be able to specify the coordination between cloud application entities using our NaturalBIP language. We have implemented the NaturalBIP Compiler to generate (1) BIP connectors corresponding to the specification in the NaturalBIP language and (2) JavaBIP macro representing the coordination between entities in Java. The first artifact is then used along the configuration model to generate iFinder's input for verifying the deadlock freedom. The latter is a part of the implementation.

We have demonstrated the usage of our NaturalBIP language to specify the Heroku Deployer microservice. By providing the NaturalBIP compiler, we save developers time. Instead of learning to write BIP connectors (as shown in Listing 5.2), BIP models (Listing 5.4) for the verification, and JavaBIP macros for the implementation (Listing 5.3) manually, our toolchain supports generating those artifacts automatically from the specifications written in NaturalBIP language (Listing 5.1). Row 1 of Table 5.1 compares the number of lines needs to specify the functionality, which is choosing Java as a language for deploying a Heroku Dyno. Instead of writing 100 lines for specifying the requirement using BIP connectors, writing verification code in the BIP model, and JavaBIP macros in the implementation, users need only 7 lines to specify the requirement in our NaturalBIP language. Row 2 of Table 5.1 illustrates that developers need 105 lines for specifying all the deployment functionalities using our NaturalBIP language instead of 1375 lines for

Table 5.1: The number of lines for writing the constraints to describe the functionality choose Java language and the whole functionalities in deploying a Web application onto a free Heroku Dyno

| **Functionality** | **NaturalBIP specification** (number of specifications) | **Generated artifacts** (lines of code) | | |
|---|---|---|---|---|
| | | **BIP connectors** | **JavaBIP macros** | **Connectors in the BIP model** |
| Choose Java language | 7 | 7 | 50 | 43 |
| | | 100 | | |
| All | 105 | 105 | 774 | 496 |
| | | 1375 | | |

specifying those functionalities using BIP connectors (105 lines), writing verification code in the BIP model (496 lines)[11], and JavaBIP macros (774 lines)[12] in the implementation.

[11] https://github.com/TrinhLK/GeneratingBIPFile/blob/master/output/herokudeployer.bip

[12] https://github.com/TrinhLK/JavaBIPonCloudDemo/blob/main/HerokuDemo/src/main/java/occideployer/GlueBuilder_Specification.java

# Chapter 6

# Conclusion

In this chapter, we briefly summarize our work on this thesis and present our contributions. Then, we discuss potential future work on developing correct-by-construction self-adaptive cloud applications.

**Summary** We have proposed an ontology-driven approach to express functional requirements in Chapter 3. Our main intention is that users unfamiliar with exogenous coordination language can write the specification in a pseudo-natural language. Our methodology generates the artifacts automatically to express the exogenous coordination between components, data transfers, and safety properties. In Chapter 4, we have extended the coordination capability of the OCCIware design and generated artifacts for verifying the deadlock-free property of the target application. Our methodology is evaluated through the Heroku Deployer and the Monitor-Switch Web application. The former shows the ability of our methodology to develop a complex application, and the latter illustrates the development of a correct-by-construction cloud application.

**Contributions**

- The main contribution of this thesis is proposing a methodology and providing tools for developing correct-by-construction self-adaptive cloud applications.
- Our approach is based on the BIP framework, which provides the capability for developing correct-by-construction systems. To reduce the time and cost of learning BIP, we proposed NaturalBIP to help users write functional requirements naturally.
- A tool with NaturalBIP Compiler has been developed. It supports developers in automatically generating JavaBIP macro, data transfers, and safety properties without the help of JavaBIP experts.
- We have extended the OCCIware Studio by defining new concepts for writing requirements in our NaturalBIP language and specifying the FSM specification in

more detail. The FSM specification is used to generate computation code in the implementation, and the BIP model for the verification.

**Future work** We discuss the current limitations of our framework and plan future work that could further improve our research on developing correct-by-construction cloud applications.

- The first perspective is validating our ontology for the overall coverage of the cloud domain. Some analysis tools allow evaluating an ontology in both qualitative and quantitative aspects [72, 69, 86, 67, 101, 126], such as the depth of inheritance tree for classes or their ancestor, etc., or identifying potential problems of an ontology. However, no evaluation method can guarantee an ontology is "sound" [109].

- Currently, safety properties for verification are written manually following the form provided by iFinder. This task is not easy for a cloud developer. Thus, a future direction is automatically deriving such properties from the requirements.

- According to the toolchain, we define the concept of "actiontype" in the OCCIware meta-model to describe the transition type (*enforceable*, *spontaneous*, or *internal*), but it's still not intuitive. Therefore, in the future, there will be some upgrades in the tool to make it more convenient for developers to use. In addition, we will let some cloud developers use our toolchain to evaluate and improve the user experience.

- The implementation quality can be guaranteed in the future by generating test cases from the formal model.

- Our methodology is not limited to OCCIware Studio. It can be applied to any design framework supporting FSM specification. Using our method on other design platforms is also a potential direction.

- In the example of Heroku Deployer, specifying the FSM specification takes time and effort, especially for complex components. Therefore, one of our potential directions is developing algorithms and infrastructure for generating the FSM specification from given execution traces of the system [19, 22, 81].

# Bibliography


[1] Alien4Cloud. https://alien4cloud.github.io/.

[2] BIP model generator. https://github.com/TrinhLK/GeneratingBIPFile.

[3] Cloudify. https://cloudify.co/.

[4] Deltacloud. https://deltacloud.apache.org/drivers.html#drivers.

[5] erocci. http://erocci.ow2.org/.

[6] ifinder. https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/bip/IFinder/-/tree/real-time-marius.

[7] NaturalBIP Compiler. https://github.com/TrinhLK/PBL-BIP.

[8] OCCIwareBIP Studio. https://github.com/TrinhLK/OCCIwareBIP-Studio.

[9] Open Virtualization Format (OVF). https://www.dmtf.org/standards/ovf.

[10] PyOCNI: A Python implementation of an extended OCCI with a JSON serialization. https://github.com/all4innov/pyOCNI.

[11] RESTful OCCI 4 Java. https://github.com/occi4java/occi4java.

[12] rOCCI - A Ruby OCCI Framework. http://gwdg.github.io/rOCCI/.

[13] Service Sharing Facility in Python. https://github.com/tmetsch/pyssf.

[14] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* 01 2005.

[15] Gul A Agha. A model of concurrent computation in distributed systems. *the MIT Press*, 1986.

[16] Yousra Abdul Alsahib S. Aldeen, Mazleena bt. Salleh, and Mohammad Abdur Razzaque. A survey paper on privacy issue in cloud computing. *Research Journal of Applied Sciences, Engineering and Technology*, 10:328–337, 2015.

[17] Kena Alexander, Choonhwa Lee, Eunsam Kim, and Sumi Helal. Enabling end-to-end orchestration of multi-cloud applications. *IEEE Access*, 5:18862–18875, 2017.

[18] Edmonds Andy, Metsch Thijs, Papaspyrou Alexander, and Richardson Alexis. Toward an opencloud standard, 2012.

[19] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, nov 1987.

[20] Krzysztof R. Apt, Nissim Francez, and Willem P. de Roever. A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 2(3):359–385, jul 1980.

[21] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Math. Struct. Comput. Sci.*, 14:329–366, 2004.

[22] George Argyros and Loris D'antoni. The Learnability of Symbolic Automata. In *CAV*, 2018.

[23] Michael Armbrust, Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, 04 2010.

[24] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, April 2010.

[25] Christel Baier, Tobias Blechmann, Joachim Klein, Sascha Klüppelholz, and Wolfgang Leister. Design and Verification of Systems with Exogenous Coordination Using Vereofy. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 97–111, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[26] Deborah Anne Baker. The Use of Requirements in Rigorous System Design. 1982.

[27] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, May 2011.

[28] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, page 3–12, USA, 2006. IEEE Computer Society.

[29] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-Finder: A Tool for Compositional Deadlock Detection and Verification. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009*, volume 5643 of *Lecture Notes in Computer Science*, pages 614–619, Grenoble, France, June 2009. Springer.

[30] JA Bergstra. Process Algebra: Specification and Verification in Bisimulation Semantics. *CWI monographs*, 4:61–94, 1986.

[31] Jan A Bergstra and Jan Willem Klop. Process Algebra for Synchronous Communication. *Information and control*, 60(1-3):109–137, 1984.

[32] Marco Bernardo and Francesco Franzè. Exogenous and Endogenous Extensions of Architectural Types. pages 40–55, 04 2002.

[33] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. Opentosca — a runtime for tosca-based cloud applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing - Volume 8274*, ICSOC 2013, page 692–695, Berlin, Heidelberg, 2013. Springer-Verlag.

[34] Tobias Binz, Gerd Breiter, Frank Leyman, and Thomas Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(3):80–85, 2012.

[35] G. Blair, N. Bencomo, and R. B. France. Models@run.time. *Computer*, 42(10):22–27, 2009.

[36] S. Bliudze and J. Sifakis. The Algebra of Connectors—Structuring Interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, Oct 2008.

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

[38] Simon Bliudze and Joseph Sifakis. The Algebra of Connectors - Structuring Interaction in BIP. In *International Conference On Embedded Software (EMSOFT)*, pages 11–20, Salzbourg, Austria, October 2007. ACM New York, NY, USA.

[39] Simon Bliudze and Joseph Sifakis. Causal semantics for the Algebra of Connectors (extended abstract). In Frank de Boer and Marcello Bonsangue, editors, *FMCO 2007*, number 5382 in LNCS, pages 179–199, Berlin Heidelberg, 2008. Springer-Verlag. See the journal version.

[40] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, Berlin, Heidelberg, 2007.

[41] Eirik Brandtzæg, Sébastien Mosser, and Parastoo Mohagheghi. Towards CloudML, a Model-based Approach to Provision Resources in the Clouds. 2012.

[42] Antonio Brogi, Luca Rinaldi, and Jacopo Soldani. TosKer: a synergy between TOSCA and Docker for orchestrating multicomponent applications. *Software: Practice and Experience*, 48(11):2061–2079, 2018.

[43] Roberto Bruni, Ivan Lanese, and Ugo Montanari. A Basic Algebra of Stateless Connectors. *Theoretical Computer Science*, 366(1-2):98–120, 2006.

[44] Roberto Bruni and Ugo Montanari. Dynamic Connectors for Concurrency. *Theoretical computer science*, 281(1-2):131–176, 2002.

[45] Dennis M Buede and William D Miller. The Engineering Design of Systems: Models and Methods. 2016.

[46] Sirio Capizzi, Riccardo Solmi, and Gianluigi Zavattaro. From Endogenous to Exogenous Coordination Using Aspect-Oriented Programming. In Rocco De Nicola, Gian-Luigi Ferrari, and Greg Meredith, editors, *Coordination Models and Languages*, pages 105–118, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[47] Ionut Cardei, Mihai Fonoage, and Ravi Shankar. Model Based Requirements Specification and Validation for Component Architectures. In *2008 2nd Annual IEEE Systems Conference*, pages 1–8. IEEE, 2008.

[48] Stéphanie Challita, Fabian Korte, Johannes Erbel, Faiez Zalila, Jens Grabowski, and Philippe Merle. Model-Based Cloud Resource Management with TOSCA and OCCI. *Software and Systems Modeling*, pages 1–23, February 2021.

[49] Martin Croxford. Correctness by Construction : A Manifesto for High-Integrity Software. 2006.

[50] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. volume 4963, pages 337–340, 04 2008.

[51] Jack B. Dennis. *Petri Nets*, pages 1525–1530. Springer US, Boston, MA, 2011.

[52] Diego Dermeval, Jéssyka Vilela, Ig Ibert Bittencourt, Jaelson Castro, Seiji Isotani, Patrick Brito, and Alan Silva. Applications of ontologies in requirements engineering: a systematic review of the literature. *Requirements Engineering*, 21(4):405–437, 2016.

[53] Peter Dinges and Gul Agha. Scoped Synchronization Constraints for Large Scale Actor Systems. In Marjan Sirjani, editor, *14th International Conference on Coordination Models and Languages (COORDINATION)*, volume LNCS-7274 of *Coordination Models and Languages*, pages 89–103, Stockholm, Sweden, June 2012. Springer.

[54] Inc. (DMTF) Distributed Management Task Force. Cloud infrastructure management interface (cimi) model and restful http-based protocol, 2016.

[55] Dang Viet Dzung and Atsushi Ohnishi. Improvement of Quality of Software Requirements with Requirements Ontology. In *2009 Ninth International Conference on Quality Software*, pages 284–289. IEEE, 2009.

[56] Dang Viet Dzung and Atsushi Ohnishi. Improvement of Quality of Software Requirements with Requirements Ontology. In *2009 Ninth International Conference on Quality Software*, pages 284–289, 2009.

[57] Dang Viet Dzung and Atsushi Ohnishi. A Verification Method of Elicited Software Requirements using Requirements Ontology. In *2012 19th Asia-Pacific Software Engineering Conference*, volume 1, pages 553–558. IEEE, 2012.

[58] Jonas Eckhardt, Tobias Mühlbauer, Jose Meseguer, and Martin Wirsing. Semantics, distributed implementation, and formal analysis of klaim models in maude. *Science of Computer Programming*, 99, 01 2014.

[59] Andy Edmonds and Thijs Metsch. Open Cloud Computing Interface - Text Rendering. OGF Published Document GWD-R-P.229, Global Grid Forum, Open Grid Forum, P.O. Box 1738, Muncie IN 47308, USA, September 2016. Accessed: 2017-1-17.

[60] Stefan Farfeleder, Thomas Moser, Andreas Krall, Tor Stålhane, Inah Omoronyia, and Herbert Zojer. Ontology-Driven Guidance for Requirements Elicitation. In *The Semanic Web: Research and Applications*, pages 212–226. Springer Berlin Heidelberg, 2011.

[61] GianLuigi Ferrari and Ugo Montanari. Tile formats for located and mobile systems. *Information and computation*, 156(1-2):173–235, 2000.

[62] A Juan Ferrer, D Garcia, et al. Ascetic: adapting service lifecycle towards efficient clouds. *European Project Space: cases and examples. SCITEPRESS*, pages 89–106, 2014.

[63] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 887–894, 2013.

[64] Roy Thomas Fielding and Richard N. Taylor. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, 2000. AAI9980887.

[65] Wan Fokkink. *Introduction to Process Algebra*. springer science & Business Media, 1999.

[66] Fabio Gadducci and Ugo Montanari. The Tile Model. *Proof, language, and interaction: Essays in honour of Robin Milner*, page 133, 2000.

[67] Aldo Gangemi, Carola Catenacci, Massimiliano Ciaramita, and Jos Lehmann. Modelling Ontology Evaluation and Validation. In *Lecture Notes in Computer Science*, pages 140–154. Springer Berlin Heidelberg, 2006.

[68] Smita Ghaisas and Nirav Ajmeri. Knowledge-Assisted Ontology-Based Requirements Evolution. In *Managing requirements knowledge*, pages 143–167. Springer, 2013.

[69] Asunción Gómez-Pérez. Ontology Evaluation. In *Handbook on Ontologies*, pages 251–273. Springer Berlin Heidelberg, 2004.

[70] Peter Green, Michael Rosemann, Marta Indulska, and Chris Manning. Candidate Interoperability Standards: An Ontological Overlap Analysis. *Data Knowl. Eng.*, 62(2):274–291, aug 2007.

[71] Gerhard Griessnig, Roland Mader, Thomas Peikenkamp, Bernhard Josko, Martin Törngren, and Eric Armengaud. CESAR: Cost-Efficient Methods and Processes for Safety Relevant Embedded Systems. *Embedded World*, 2010.

[72] Thomas R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *Int. J. Hum.-Comput. Stud.*, 43(5–6):907–928, dec 1995.

[73] Michael Grüninger, Katy Atefi, and Mark S Fox. Ontologies to support process integration in enterprise engineering. *Computational & Mathematical Organization Theory*, 6(4):381–394, 2000.

[74] Giancarlo Guizzardi and Terry Halpin. Ontological Foundations for Conceptual Modelling. *Appl. Ontol.*, 3(1–2):1–12, jan 2008.

[75] Giancarlo Guizzardi and Veruska Zamborlini. Using a Trope-Based Foundational Ontology for Bridging Different Areas of Concern in Ontology-Driven Conceptual Modeling. *Sci. Comput. Program.*, 96(P4):417–443, dec 2014.

[76] A. Hall and R. Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software*, 19(1):18–25, 2002.

[77] Marcus Hanikat. Towards a Correct-by-Construction design flow: A case-study from railway signaling systems, 2021.

[78] Charles Antony Richard Hoare et al. *Communicating Sequential processes*, volume 178. Prentice-hall Englewood Cliffs, 1985.

[79] Ted Honderich. *The Oxford Companion to Philosophy*. Oxford University Press, 2005.

[80] Elizabeth Hull, Ken Jackson, and Jeremy Dick, editors. *Requirements Engineering*. Springer, London, 2011.

[81] Natasha Yogananda Jeppu, Tom Melham, and Daniel Kroening. Active Learning of Abstract System Models from Traces using Model Checking [Extended], 2021.

[82] M.S. Kilpinen, Claudia Eckert, and P. Clarkson. The Emergence of Change at the Systems Engineering and Software Design Interface: An Investigation of Impact Analysis. 01 2007.

[83] Sascha Klüppelholz and Christel Baier. Symbolic Model Checking for Channel-based Component Connectors. *Electronic Notes in Theoretical Computer Science*, 74:19–37, 07 2009.

[84] Stefan Kolb and Cedric Röck. Unified Cloud Application Management. In *2016 IEEE World Congress on Services (SERVICES)*, pages 1–8. IEEE, 2016.

[85] Zong-yong Li, Zhi-xu Wang, Ai-hui Zhang, and Yong Xu. The Domain Ontology and Domain Rules Based Requirements Model Checking. *International Journal of Software Engineering and Its Applications*, 1(1):89–100, 2007.

[86] Adolfo Lozano-Tello and Asunción Gomez-Perez. ONTOMETRIC:A Method to Choose the Appropriate Ontology. *Journal of Database Management*, 15(2):1–18, apr 2004.

[87] Alan MacCormack, John Rusnak, and Carliss Baldwin. The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry. *SSRN Electronic Journal*, 12 2007.

[88] Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. Resa: An Ontology-Based Requirement Specification Language Tailored to Automotive Systems. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, 2015.

[89] Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. ReSA: An ontology-based requirement specification language tailored to automotive systems. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015.

[90] Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. Specification and Semantic Analysis of Embedded Systems Requirements: From Description Logic to Temporal Logic. In *International Conference on Software Engineering and Formal Methods*, pages 332–348. Springer, 2017.

[91] J.P. Martin-Flatin. Challenges in cloud management. *IEEE Cloud Computing*, 1:66–70, 05 2014.

[92] Anastasia Mavridou, Valentin Rutz, and Simon Bliudze. Coordination of dynamic software components with JavaBIP. In *Proceedings of the 14th International Conference Formal Aspects of Component Software (FACS)*, volume 10487 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2017.

[93] Thijs Metsch, Andy Edmonds, Ralf Nyren, and A Papaspyrou. Open cloud computing interface–core. In *Open Grid Forum, OCCI-WG, Specification Document. Available at: http://forge. gridforum. org/sf/go/doc16161*. Citeseer, 2010.

[94] George A. Miller. WordNet: A Lexical Database for English. *Commun. ACM*, 38(11):39–41, nov 1995.

[95] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[96] Konstantinos Mokos, Theodoros Nestoridis, Panagiotis Katsaros, and Nick Bassiliades. Semantic Modeling and Analysis of Natural Language System Requirements. *IEEE Access*, 10:84094–84119, 2022.

[97] Thomas Moser, Dietmar Winkler, Matthias Heindl, and Stefan Biffl. Requirements Management with Semantic Technology: An Empirical Study on Automated Requirements Categorization and Conflict Analysis. In *International Conference on Advanced Information Systems Engineering*, pages 3–17. Springer, 2011.

[98] Hamid R. Motahari Nezhad, Karen Yorov, Peifeng Yin, Taiga Nakamura, Scott Trent, Gil Shurek, Takayuki Kushida, and Uma Subramanian. COOL: A Model-Driven and Automated System for Guided and Verifiable Cloud Solution Design. pages 194–198, 10 2017.

[99] Tuong Huan Nguyen, Bao Quoc Vo, Markus Lumpe, and John Grundy. Kbre: A framework for knowledge-based requirements engineering. *Software Quality Journal*, 22(1):87–119, 2014.

[100] Ralf Nyren, Andy Edmonds, Thijs Metsch, and Boris Parak. Open Cloud Computing Interface - HTTP Protocol. OGF Published Document GWD-R-P.223, Global Grid Forum, Open Grid Forum, P.O. Box 1738, Muncie IN 47308, USA, September 2016. Accessed: 2017-1-17.

[101] Leo Obrst, Werner Ceusters, Inderjeet Mani, Steve Ray, and Barry Smith. The Evaluation of Ontologies. In *Semantic Web*, pages 139–158. Springer US, Boston, MA, 2007.

[102] Jorge Ocon, Francisco Colmenero, Karl Buckley, Saddek Bensalem, Iulia Dragomir, Spyridon Karachalios, Mark Woods, F. Pommerening, and Thomas Keller. Using the ERGO Framework for Space Robotics in A Planetary and An Orbital Scenario. 10 2018.

[103] Andreas L. Opdahl, Giuseppe Berio, Mounira Harzallah, and Raimundas Matulevičius. An Ontology for Enterprise and Information Systems Modelling. *Appl. Ontol.*, 7(1):49–92, jan 2012.

[104] Gerard O'Regan. *A Brief History of Computing.* Springer Publishing Company, Incorporated, 1 edition, 2008.

[105] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, 1999.

[106] Jean Parpaillon, Philippe Merle, Olivier Barais, Marc Dutoo, and Fawaz Paraiso. OCCIware - A Formal and Tooled Framework for Managing Everything as a Service. In CEUR, editor, *Projects Showcase @ STAF'15*, volume 1400 of *Proceedings of the Projects Showcase @ STAF'15*, pages 18 – 25, L'Aquila, Italy, July 2015.

[107] José D'Abruzzo Pereira, Rui Silva, Nuno Antunes, Jorge L. M. Silva, Breno de França, Regina Moraes, and Marco Vieira. A platform to enable self-adaptive cloud applications using trustworthiness properties. In *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '20, page 71–77, New York, NY, USA, 2020. Association for Computing Machinery.

[108] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[109] Joe Raad and Christophe Cruz. A survey on ontology evaluation methods. In *Proceedings of the 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management.* SCITEPRESS - Science and and Technology Publications, 2015.

[110] A. Rajan and T. Wahl. *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems.* 10 2014.

[111] Vaclav Rajlich. A Model and a Tool for Change Propagation in Software. *ACM SIGSOFT Software Engineering Notes*, 25:72, 01 2000.

[112] Xiaoxia Ren, B.G. Ryder, M. Stoerzer, and F. Tip. Chianti: a Change Impact Analysis Tool for Java Programs. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 664–665, 2005.

[113] João David Marques dos Santos Ribeiro. *A Dashboard for Decision Support in Self-Adaptive Cloud Applications.* PhD thesis, Universidade de Coimbra, 2022.

[114] Luis Rodero-Merino, Luis M Vaquero, Victor Gil, Fermín Galán, Javier Fontán, Rubén S Montero, and Ignacio M Llorente. From infrastructure delivery to service management in clouds. *Future Generation Computer Systems*, 26(8):1226–1240, 2010.

[115] Allan Berrocal Rojas and Gabriela Barrantes Sliesarieva. Automated Detection of Language Issues Affecting Accuracy, Ambiguity and Verifiability in Software Requirements Written in Natural Language. In *Proceedings of the NAACL HLT 2010 Young Investigators Workshop on Computational Approaches to Languages of the Americas*, YIWCALA '10, page 100–108, USA, 2010. Association for Computational Linguistics.

[116] Maria Salama, Rami Bahsoon, and N Bencomo. Managing trade-offs in self-adaptive software architectures: A systematic mapping study. *Managing trade-offs in adaptable software architectures*, pages 249–297, 2017.

[117] Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes.* Cambridge university press, 2003.

[118] Juan Scholten, Farhad Arbab, Frank Boer, and Marcello Bonsangue. Mocha-pi: an Exogenous Coordination Calculus based on Mobile Channels. volume 1, pages 436–442, 01 2005.

[119] Armands Slihte, Janis Osis, and Asnate Jansone. Using use cases for domain modeling.

[120] Paweł Sobociński. Representations of petri net interactions. In *International Conference on Concurrency Theory*, pages 554–568. Springer, 2010.

[121] Amina Souag, Camille Salinesi, Isabelle Wattiau, and Haris Mouratidis. Using security and domain ontologies for security requirements analysis. In *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, pages 101–107. IEEE, 2013.

[122] Emmanouela Stachtiari, Anastasia Mavridou, Panagiotis Katsaros, Simon Bliudze, and Joseph Sifakis. Early validation of system requirements and design through correctness-by-construction. *Journal of Systems and Software*, 145:52–78, November 2018.

[123] Orazio Tomarchio, Domenico Calcaterra, Giuseppe Di Modica, and Pietro Mazzaglia. Torch: a tosca-based orchestrator of multi-cloud containerised applications. *Journal of Grid Computing*, 19(1):1–25, 2021.

[124] Andries van Renssen. Gellish: a Generic Extensible Ontological Language - Design and Application of a Universal Data Structure. 2005.

[125] Michaël Verdonck, Frederik Gailly, Sergio de Cesare, and Geert Poels. Ontology-driven conceptual modeling: A systematic literature mapping and review. *Appl. Ontology*, 10:197–227, 2015.

[126] Denny Vrandečić. Ontology Evaluation. In *Handbook on Ontologies*, pages 293–313. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[127] Hai H Wang, Danica Damljanovic, and Jing Sun. Enhanced Semantic Access to Formal Software Models. In *International Conference on Formal Engineering Methods*, pages 237–252. Springer, 2010.

[128] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, and Jian Cao. A model-driven framework for interoperable cloud resources management. In *Service-Oriented Computing: 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings*, page 186–201, Berlin, Heidelberg, 2016. Springer-Verlag.

[129] Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. Formal Consistency Checking over Specifications in Natural Languages. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1677–1682. IEEE, 2015.

[130] Sami Yangui, Iain-James Marshall, Jean-Pierre Laisne, and Samir Tata. Compatibleone: The open source cloud broker. *J. Grid Comput.*, 12(1):93–109, mar 2014.

[131] Faiez Zalila, Stephanie Challita, and Philippe Merle. A Model-Driven Tool Chain for OCCI. In Herve Panetto, Christophe Debruyne, Walid Gaaloul, Mike Papazoglou, Adrian Paschke, Claudio Agostino Ardagna, and Robert Meersman, editors, *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, pages 389–409, Cham, 2017. Springer International Publishing.

[132] Faiez Zalila, Stéphanie Challita, and Philippe Merle. Model-Driven Cloud Resource Management with OCCIware. *Future Generation Computer Systems*, 99:260 – 277, October 2019.

[133] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM transactions on Software Engineering and Methodology (TOSEM)*, 6(1):1–30, 1997.

[134] Yingzhou Zhang and Weifeng Zhang. Description Logic Representation for Requirement Specification. In *International Conference on Computational Science*, pages 1147–1154. Springer, 2007.

# Heroku Requirements

- **deployFreeDyno**: There is a HerokuDynoType dynotype, a Deployer deployer, dynotype shall sub1 synchronized with deployer shall setFreeDyno.
- **deployResponse**: There is a HerokuDynoType dynotype, a Deployer deployer, dynotype shall sendDynoResponse synchronized with deployer shall receiveDynoResponse.
- **setUSRegion**: There is a HerokuRegion region, a Deployer deployer, region shall toUS synchronized with deployer shall setUSRegion.
- **setEURegion**: There is a HerokuRegion region, a Deployer deployer, region shall toEU synchronized with deployer shall setEURegion.
- **availableAddons**: There is a HerokuRegion region, there is a HerokuPostgres postgres, there is a HerokuClearDBMySQL cleardb, there is a HerokuScoutAPM scout, there is a HerokuNewRelicAPM newrelic, if region executes setAddonsForEU or region executes setAddonsForUS, postgres shall on or cleardb shall on or scout shall on or newrelic shall on.
- **addonsForUS**: There is a HerokuRegion region, there is a HerokuPostgres postgres, there is a HerokuClearDBMySQL cleardb, there is a HerokuScoutAPM scout, there is a HerokuNewRelicAPM newrelic, there is a Deployer deployer, if postgres executes on and cleardb executes on and scout executes on and newrelic executes on and deployer executes setAddonsForUS, region shall setAddonsForUS.
- **addonsForEU**: There is a HerokuRegion region, there is a HerokuPostgres postgres, there is a HerokuClearDBMySQL cleardb, there is a HerokuScoutAPM scout, there is a HerokuNewRelicAPM newrelic, there is a Deployer deployer, if postgres executes on and cleardb executes on and scout executes on and newrelic executes on and deployer executes setAddonsForEU, region shall setAddonsForEU.
- **setadnsForUS**: There is a HerokuRegion region, a Deployer deployer, if region executes setAddonsForUS, deployer shall setAddonsForUS.

- **setadnsForEU**: There is a HerokuRegion region, a Deployer deployer, if region executes setAddonsForEU, deployer shall setAddonsForEU.
- **deployersetJava**: There is a HerokuBuildpack buildpack, a Deployer deployer, buildpack shall setJava synchronized with deployer shall setJava.
- **deployersetScala**: There is a HerokuBuildpack buildpack, a Deployer deployer, buildpack shall setScala synchronized with deployer shall setScala.
- **deployersetPython**: There is a HerokuBuildpack buildpack, a Deployer deployer, buildpack shall setPython synchronized with deployer shall setPython.
- **deployersetRuby**: There is a HerokuBuildpack buildpack, a Deployer deployer, buildpack shall setRuby synchronized with deployer shall setRuby.
- **deployersetNodejs**: There is a HerokuBuildpack buildpack, a Deployer deployer, buildpack shall setNodejs synchronized with deployer shall setNodejs.
- **deployersetClojure**: There is a HerokuBuildpack buildpack, a Deployer deployer, buildpack shall setClojure synchronized with deployer shall setClojure.
- **deployersetGradle**: There is a HerokuBuildpack buildpack, a Deployer deployer, buildpack shall setGradle synchronized with deployer shall setGradle.
- **deployersetJvm**: There is a HerokuBuildpack buildpack, a Deployer deployer, buildpack shall setJvm synchronized with deployer shall setJvm.
- **deployersetPhP**: There is a HerokuBuildpack buildpack, a Deployer deployer, buildpack shall setPhP synchronized with deployer shall setPhp.
- **deployersetGo**: There is a HerokuBuildpack buildpack, a Deployer deployer, buildpack shall setGo synchronized with deployer shall setGo.
- **buildpackpostgres1**: There is a HerokuBuildpack buildpack, a HerokuPostgres postgres, buildpack shall setAddonsForJava synchronized with postgres shall on.
- **buildpackpostgres2**: There is a HerokuBuildpack buildpack, a HerokuPostgres postgres, buildpack shall setAddonsForScala synchronized with postgres shall on.
- **buildpackpostgres3**: There is a HerokuBuildpack buildpack, a HerokuPostgres postgres, buildpack shall setAddonsForPython synchronized with postgres shall on.
- **buildpackpostgres4**: There is a HerokuBuildpack buildpack, a HerokuPostgres postgres, buildpack shall setAddonsForRuby synchronized with postgres shall on.

- **buildpackpostgres5**: There is a HerokuBuildpack buildpack, a HerokuPostgres postgres, buildpack shall setAddonsForNodejs synchronized with postgres shall on.
- **buildpackpostgres6**: There is a HerokuBuildpack buildpack, a HerokuPostgres postgres, buildpack shall setAddonsForClojure synchronized with postgres shall on.
- **buildpackpostgres7**: There is a HerokuBuildpack buildpack, a HerokuPostgres postgres, buildpack shall setAddonsForGradle synchronized with postgres shall off.
- **buildpackpostgres8**: There is a HerokuBuildpack buildpack, a HerokuPostgres postgres, buildpack shall setAddonsForJvm synchronized with postgres shall on.
- **buildpackpostgres9**: There is a HerokuBuildpack buildpack, a HerokuPostgres postgres, buildpack shall setAddonsForPhp synchronized with postgres shall on.
- **buildpackpostgresm1**: There is a HerokuBuildpack buildpack, a HerokuPostgres postgres, buildpack shall setAddonsForGo synchronized with postgres shall on.
- **buildpackClearDB1**: There is a HerokuBuildpack buildpack, a HerokuClearDB-MySQL cleardb, buildpack shall setAddonsForJava synchronized with cleardb shall on.
- **buildpackClearDB2**: There is a HerokuBuildpack buildpack, a HerokuClearDB-MySQL cleardb, buildpack shall setAddonsForScala synchronized with cleardb shall on.
- **buildpackClearDB3**: There is a HerokuBuildpack buildpack, a HerokuClearDB-MySQL cleardb, buildpack shall setAddonsForPython synchronized with cleardb shall on.
- **buildpackClearDB4**: There is a HerokuBuildpack buildpack, a HerokuClearDB-MySQL cleardb, buildpack shall setAddonsForRuby synchronized with cleardb shall on.
- **buildpackClearDB5**: There is a HerokuBuildpack buildpack, a HerokuClearDB-MySQL cleardb, buildpack shall setAddonsForNodejs synchronized with cleardb shall on.
- **buildpackClearDB6**: There is a HerokuBuildpack buildpack, a HerokuClearDB-MySQL cleardb, buildpack shall setAddonsForClojure synchronized with cleardb shall on.
- **buildpackClearDB7**: There is a HerokuBuildpack buildpack, a HerokuClearDB-MySQL cleardb, buildpack shall setAddonsForGradle synchronized with cleardb shall off.

- **buildpackClearDB8**: There is a HerokuBuildpack buildpack, a HerokuClearDB-MySQL cleardb, buildpack shall setAddonsForJvm synchronized with cleardb shall on.
- **buildpackClearDB9**: There is a HerokuBuildpack buildpack, a HerokuClearDB-MySQL cleardb, buildpack shall setAddonsForPhp synchronized with cleardb shall on.
- **buildpackClearDBm1**: There is a HerokuBuildpack buildpack, a HerokuClearDBMySQL cleardb, buildpack shall setAddonsForGo synchronized with cleardb shall on.
- **buildpackScout1**: There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout, buildpack shall setAddonsForJava synchronized with scout shall off.
- **buildpackScout2**: There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout, buildpack shall setAddonsForScala synchronized with scout shall off.
- **buildpackScout3**: There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout, buildpack shall setAddonsForPython synchronized with scout shall on.
- **buildpackScout4**: There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout, buildpack shall setAddonsForRuby synchronized with scout shall on.
- **buildpackScout5**: There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout, buildpack shall setAddonsForNodejs synchronized with scout shall off.
- **buildpackScout6**: There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout, buildpack shall setAddonsForClojure synchronized with scout shall off.
- **buildpackScout7**: There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout, buildpack shall setAddonsForGradle synchronized with scout shall off.
- **buildpackScout8**: There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout, buildpack shall setAddonsForJvm synchronized with scout shall off.
- **buildpackScout9**: There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout, buildpack shall setAddonsForPhp synchronized with scout shall on.
- **buildpackScoutm1**: There is a HerokuBuildpack buildpack, a HerokuScoutAPM scout, buildpack shall setAddonsForGo synchronized with scout shall off.
- **buildpackNewRelic1**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, buildpack shall setAddonsForJava synchronized with newrelic shall on.

- **buildpackNewRelic2**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, buildpack shall setAddonsForScala synchronized with newrelic shall off.

- **buildpackNewRelic3**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, buildpack shall setAddonsForPython synchronized with newrelic shall on.

- **buildpackNewRelic4**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, buildpack shall setAddonsForRuby synchronized with newrelic shall on.

- **buildpackNewRelic5**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, buildpack shall setAddonsForNodejs synchronized with newrelic shall on.

- **buildpackNewRelic6**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, buildpack shall setAddonsForClojure synchronized with newrelic shall off.

- **buildpackNewRelic7**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, buildpack shall setAddonsForGradle synchronized with newrelic shall off.

- **buildpackNewRelic8**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, buildpack shall setAddonsForJvm synchronized with newrelic shall on.

- **buildpackNewRelic9**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, buildpack shall setAddonsForPhp synchronized with newrelic shall on.

- **buildpackNewRelicm1**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, buildpack shall setAddonsForGo synchronized with newrelic shall off.

- **synthesisBuildpack1**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes setAddonsForJava and cleardb executes on and postgres executes on and scout executes off and newrelic executes on, buildpack shall setAddonsForJava.

- **synthesisBuildpack2**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes setAddonsForScala and cleardb executes on and postgres executes on and scout executes off and newrelic executes off, buildpack shall setAddonsForScala.

- **synthesisBuildpack3**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes setAddonsForPython and cleardb executes on and postgres executes on and scout executes on and newrelic executes on, buildpack shall setAddonsForPython.

- **synthesisBuildpack4**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes setAddonsForRuby and cleardb executes on and postgres executes on and scout executes on and newrelic executes on, buildpack shall setAddonsForRuby.

- **synthesisBuildpack5**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes setAddonsForNodejs and cleardb executes on and postgres executes on and scout executes off and newrelic executes on, buildpack shall setAddonsForNodejs.

- **synthesisBuildpack6**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes setAddonsForClojure and cleardb executes on and postgres executes on and scout executes off and newrelic executes off, buildpack shall setAddonsForClojure.

- **synthesisBuildpack7**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes setAddonsForGradle and cleardb executes off and postgres executes off and scout executes off and newrelic executes off, buildpack shall setAddonsForGradle.

- **synthesisBuildpack8**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes setAddonsForJvm and cleardb executes on and postgres executes on and scout executes off and newrelic executes on, buildpack shall setAddonsForJvm.

- **synthesisBuildpack9**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes setAddonsForPhp and cleardb executes on and postgres executes on and scout executes on and newrelic executes on, buildpack shall setAddonsForPhp.

- **synthesisBuildpackm1**: There is a HerokuBuildpack buildpack, a HerokuNewRelicAPM newrelic, a HerokuPostgres postgres, a HerokuScoutAPM scout, a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes setAddonsForGo and cleardb executes on and postgres executes on and scout executes off and newrelic executes off, buildpack shall setAddonsForGo.

- **buildpackdeployer1**: There is a HerokuBuildpack buildpack, a Deployer deployer, if buildpack executes setAddonsForJava, deployer shall setAddonsForJava.

- **buildpackdeployer2**: There is a HerokuBuildpack buildpack, a Deployer deployer, if buildpack executes setAddonsForScala, deployer shall setAddonsForScala.

- **buildpackdeployer3**: There is a HerokuBuildpack buildpack, a Deployer deployer, if buildpack executes setAddonsForPython, deployer shall setAddonsForPython.

- **buildpackdeployer4**: There is a HerokuBuildpack buildpack, a Deployer deployer, if buildpack executes setAddonsForRuby, deployer shall setAddonsForRuby.

- **buildpackdeployer5**: There is a HerokuBuildpack buildpack, a Deployer deployer, if buildpack executes setAddonsForNodejs, deployer shall setAddonsForNodejs.

- **buildpackdeployer6**: There is a HerokuBuildpack buildpack, a Deployer deployer, if buildpack executes setAddonsForClojure, deployer shall setAddonsForClojure.

- **buildpackdeployer7**: There is a HerokuBuildpack buildpack, a Deployer deployer, if buildpack executes setAddonsForGradle, deployer shall setAddonsForGradle.

- **buildpackdeployer8**: There is a HerokuBuildpack buildpack, a Deployer deployer, if buildpack executes setAddonsForJvm, deployer shall setAddonsForJvm.

- **buildpackdeployer9**: There is a HerokuBuildpack buildpack, a Deployer deployer, if buildpack executes setAddonsForPhp, deployer shall setAddonsForPhp.

- **buildpackdeployerm1**: There is a HerokuBuildpack buildpack, a Deployer deployer, if buildpack executes setAddonsForGo, deployer shall setAddonsForGo.

- **deployFreeAddon1**: There is a HerokuPostgres postgres, a Deployer deployer, if postgres executes sendAddonResponse, deployer shall receiveAddonResponse.

- **deployFreeAddon2**: There is a HerokuPostgres postgres, a Deployer deployer, if deployer executes addHerokuPostgres, postgres shall sub1.
- **deployFreeAddon3**: There is a HerokuClearDBMySQL cleardb, a Deployer deployer, if deployer executes addClearDBMySQL, cleardb shall sub1.
- **deployFreeAddon4**: There is a HerokuScoutAPM scout, a Deployer deployer, if deployer executes addScoutAPM, scout shall sub1.
- **deployFreeAddon5**: There is a HerokuNewRelicAPM newrelic, a Deployer deployer, if deployer executes addNewRelicAPM, newrelic shall sub1.
- **resetSetup1**: There is a Deployer deployer, a HerokuDynoType dynotype, if deployer executes resetAll, dynotype shall reset.
- **resetSetup2**: There is a Deployer deployer, a HerokuRegion region, if deployer executes resetAll, region shall USreset.
- **resetSetup3**: There is a Deployer deployer, a HerokuRegion region, if deployer executes resetAll, region shall EUreset.
- **resetSetup4**: There is a Deployer deployer, a HerokuBuildpack buildpack, if deployer executes resetAll, buildpack shall removeJava.
- **resetSetup5**: There is a Deployer deployer, a HerokuBuildpack buildpack, if deployer executes resetAll, buildpack shall removeScala.
- **resetSetup6**: There is a Deployer deployer, a HerokuBuildpack buildpack, if deployer executes resetAll, buildpack shall removeJvm.
- **resetSetup7**: There is a Deployer deployer, a HerokuBuildpack buildpack, if deployer executes resetAll, buildpack shall removePython.
- **resetSetup8**: There is a Deployer deployer, a HerokuBuildpack buildpack, if deployer executes resetAll, buildpack shall removeRuby.
- **resetSetup9**: There is a Deployer deployer, a HerokuBuildpack buildpack, if deployer executes resetAll, buildpack shall removeNodejs.
- **resetSetupm1**: There is a Deployer deployer, a HerokuBuildpack buildpack, if deployer executes resetAll, buildpack shall removeClojure.
- **resetSetupm2**: There is a Deployer deployer, a HerokuBuildpack buildpack, if deployer executes resetAll, buildpack shall removeGradle.

- **resetSetupm3**: There is a Deployer deployer, a HerokuBuildpack buildpack, if deployer executes resetAll, buildpack shall removePhp.
- **resetSetupm4**: There is a Deployer deployer, a HerokuBuildpack buildpack, if deployer executes resetAll, buildpack shall removeGo.
- **resetAddon1**: There is a Deployer deployer, a HerokuPostgres postgres, if deployer executes resetAll, postgres shall reset.
- **resetAddon2**: There is a Deployer deployer, a HerokuPostgres postgres, if deployer executes resetAll, postgres shall off.
- **resetAddon3**: There is a Deployer deployer, a HerokuClearDBMySQL cleardb, if deployer executes resetAll, cleardb shall reset.
- **resetAddon4**: There is a Deployer deployer, a HerokuClearDBMySQL cleardb, if deployer executes resetAll, cleardb shall off.
- **resetAddon5**: There is a Deployer deployer, a HerokuScoutAPM scout, if deployer executes resetAll, scout shall reset.
- **resetAddon6**: There is a Deployer deployer, a HerokuScoutAPM scout, if deployer executes resetAll, scout shall off.
- **resetAddon7**: There is a Deployer deployer, a HerokuNewRelicAPM newrelic, if deployer executes resetAll, newrelic shall reset.
- **resetAddon8**: There is a Deployer deployer, a HerokuNewRelicAPM newrelic, if deployer executes resetAll, newrelic shall off.