Formal verification of business process configuration in the Cloud

Souha Boubaker

université PARIS-SACLAY

NNT : 2018SACLL002

Thèse de doctorat

TELECOM SudParis

Formal Verification of Business Process Configuration in the Cloud

Thèse de doctorat de l'Université Paris-Saclay
Préparée à TELECOM SudParis

École doctorale n°580
Sciences et technologies de l'information et de la communication

Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Évry, le 14/05/2018, par

Souha Boubaker

Composition du Jury :

M. Djamal Benslimane Professeur, Université Claude Bernard Lyon 1, France	Rapporteur
M. Yves Ledru Professeur, Université Grenoble Alpes, France	Rapporteur
Mme. Daniela Grigori Professeur, Université Paris Dauphine, France	Examinateur
M. Paul Gibson Maître de conférences HDR, TELECOM SudParis, France	Examinateur
M. Walid Gaaloul Professeur, TELECOM SudParis, France	Directeur de thèse
Mme. Amel Mammar Professeur, TELECOM SudParis, France	Co-encadrant
M. Mohamed Graiet Enseignant Chercheur HDR, ENSAI Rennes, France	Co-encadrant

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Vérification formelle de la configuration des processus métiers dans le Cloud

Mots clés : Gestion de processus métier, Vérification formelle, Allocation de ressources, Correction.

Résumé : Motivé par le besoin de la « Conception par Réutilisation », les modèles de processus configurables ont été proposés pour représenter de manière générique des modèles de processus similaires. Ils doivent être configurés en fonction des besoins d'une organisation en sélectionnant des options. Comme les modèles de processus configurables peuvent être larges et complexes, leur configuration sans assistance est sans doute une tâche difficile, longue et source d'erreurs.
De plus, les organisations adoptent de plus en plus des environnements Cloud pour déployer et exécuter leurs processus afin de bénéficier de ressources dynamiques à la demande. Néanmoins, en l'absence d'une description explicite et formelle de la perspective de ressources dans les processus métier existants, la correction de la gestion des ressources du Cloud ne peut pas être vérifiée.

Dans cette thèse, nous visons à (i) fournir de l'assistance et de l'aide à la configuration aux analystes avec des options correctes, et (ii) améliorer le support de la spécification et de la vérification des ressources Cloud dans les processus métier. Pour ce faire, nous proposons une approche formelle pour aider à la configuration étape par étape en considérant des contraintes structurelles et métier. Nous proposons ensuite une approche comportementale pour la vérification de la configuration tout en réduisant le problème bien connu de l'explosion d'espace d'état. Ce travail permet d'extraire les options de configuration sans blocage d'un seul coup. Enfin, nous proposons une spécification formelle pour le comportement d'allocation des ressources Cloud dans les modèles de processus métier. Cette spécification est utilisée pour valider et vérifier la cohérence de l'allocation des ressources Cloud en fonction des besoins des utilisateurs et des capacités des ressources.

Title : Formal verification of business process configuration in the Cloud

Keywords : Business Process Management, Formal verification, Resource allocation, Correctness.

Abstract: Motivated by the need for the "Design by Reuse", Configurable process models are proposed to represent in a generic manner similar process models. They need to be configured according to an organization needs by selecting design options. As the configurable process models may be large and complex, their configuration with no assistance is undoubtedly a difficult, time-consuming and error-prone task.
Moreover, organizations are increasingly adopting cloud environments for deploying and executing their processes to benefit from dynamically scalable resources on demand. Nevertheless, due to the lack of an explicit and formal description of the resource perspective in the existing business processes, the correctness of Cloud resources management cannot be verified.

In this thesis, we target to (i) provide guidance and assistance to the analysts in process model configuration with correct options, and to (ii) improve the support of Cloud resource specification and verification in business processes. To do so, we propose a formal approach for assisting the configuration step-by-step with respect to structural and business domain constraints. We thereafter propose a behavioral approach for configuration verification while reducing the well-known state space explosion problem. This work allows to extract configuration choices that satisfy the deadlock-freeness property at one time. Finally, we propose a formal specification for Cloud resource allocation behavior in business process models. This specification is used to formally validate and check the consistency of the Cloud resource allocation in process models according to user requirements and resource capabilities.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

To my family,
to my loving husband,
for their unconditional love and support.

Acknowledgment

First and foremost, I thank god for giving me strength, knowledge, ability and opportunity to undertake this research work and to continue and complete it satisfactorily. Without His blessings, this achievement would not have been possible.

I would like to thank all members of the jury. I thank Professor Djamal Benslimane and Professor Yves Ledru for accepting being my thesis reviewers and for their insightful comments and encouragement. I also thank Professor Daniela Grigori and Dr. Paul Gibson for accepting being my thesis examiners.

I would like to express my deepest thanks and gratitude to my supervisor Walid Gaaloul for his continuous support, patience, motivation, and immense knowledge. His precious guidance and fruitful ideas helped me in all the time of research and writing of my scientific papers as well as this manuscript. His valuable advice, enthusiasm and constant support during this thesis allowed me to acquire new understandings and extend my experiences. I deeply hope that we can continue our collaboration.

I am also grateful to my supervisor Amel Mammar for her encouragement, kindness and wide knowledge in formal specifications. Her precious advice and support have been invaluable. She deserves also special thanks for carefully reading through early drafts of this dissertation. A special gratitude is also due to my supervisor Mohamed Graiet for his great advices and his guidance. I am thankful for the opportunities he provided, and for having faith in me. Special thanks to Kais Klai for the beneficial collaboration we have had. I am deeply grateful for the great deal of time we spent discussing many technical details of our work together.

I also want to give special thanks to all the members of the computer science department of Telecom SudParis, especially Djamel Belaid and Brigitte Houassine for their kind help and assistance. I am indebted to many others for comments, discussions, criticism and encouraging enthusiasm: Nour, Rami, Emna, Rania, Hayet, Kunal, Nabila, Slim, Abderrahim.

Last but not least, I would like to say some words in French.

Je dédie cette thèse à la mémoire de mon grand-père Baba Hédi qui s'est éteint sans que je puisse lui dire au revoir. Je ne t'oublierai jamais, que le bon DIEU t'accueille dans son paradis.

Je dédie cette thèse à mes chers parents, Leila et Ouanes, symbole de la bonté par excellence et source de tendresse. Merci pour vos sacrifices, vos prières et votre amour inconditionnel. Votre bénédiction m'a permis de devenir ce que je suis aujourd'hui. J'espère avoir réussi à vous rendre fiers de moi. Je remercie également ma soeur Sana et mes deux fréres Seif et Salem, sans oublier ma belle petite niéce Alma. Je tiens

aussi à remercier ma belle-famille, Tata Assia, Tonton Fethi, mes belles-soeurs Hend et Hajer ainsi que son mari Oussema et le petit adorable Layth, qui m'ont toujours encouragé et soutenu avec grand amour.

Je dédie cette thèse aussi à mes meilleures amies Rania, ainsi que son mari Aymen et ma belle petite Lilia, Asma et Hend, pour leurs encouragements at support. Merci d'être toujours là, dans les meilleurs comme dans les pires moments.

Finalement, je réserve mes derniers remerciements à lui, Wael, mon meilleur ami, mon âme soeur et mon cher mari, pour son soutien, son grand amour et sa patience dans les moments difficiles qu'il a dû endurer pendant cette période de thèse.

A toute ma famille et à tous ceux qui m'aiment,

je vous aime tellement.

Souha Boubaker Rokbani

Abstract

In the last decade, growing attention has been paid to the emerging concept of Cloud Computing. Especially in Business Process Management (BPM) domain, business processes need to be deployed and executed at a high level of performance while reducing the development and maintenance cost. In a such multi-tenant environment, more and more companies are managing and executing similar processes across organizational boundaries. The use of *Configurable process models* came to represent and group business process that have similarities into a single generic process. The latter process will be shared among different companies, e.g., different branches, and adjusted during a configuration phase with respect to each specific company requirements. The obtained processes are called *variants*.

The design of process variants with respect to *configuration constraints* is becoming challenging. Since, the configurable elements may have complex interdependencies between their configuration choices, configuring process models is undoubtedly a difficult, time-consuming and error-prone task. Then, the analysts need assistance and guidance in order to correctly configuring process variants.

Furthermore, organizations are deploying part or all of their business process models in the Cloud to benefit from dynamically scalable and shared resources on demand. Nevertheless, due to the lack of an explicit and formal description of the resource perspective in the existing business processes, the correctness of Cloud resources management cannot be verified. In fact, the resource perspective in Business process models is not well addressed in the literature compared to other perspectives, such as the control flow. Also, the proposed approaches often target the human resources rather than Cloud ones.

In this thesis, we address the above shortcomings by proposing a formal approach for assisting designers in process model configuration step-by-step. We propose to verify the configurable process as well as the derived process variants correctness with respect to structural and business domain constraints provided by configuration guidelines. We thereafter propose a behavioral approach for configuration verification while reducing the well-known state space explosion problem. This work allows to extract configuration choices that satisfy the deadlock-freeness property at one time. Finally, we propose a formal specification for Cloud resource allocation policies in business process models. This specification is used to formally validate the consistency of Cloud resource allocation for process modeling at design time, and to analyze and check its correctness according to user requirements and resource capabilities.

Table of contents

1 Introduction **17**
- 1.1 Research Context . . . 17
- 1.2 Motivation and Problems Description . . . 22
  - 1.2.1 How to assist and verify business process configuration? . . . 22
  - 1.2.2 How to verify Cloud resource allocation in business process models? . . . 23
- 1.3 Objectives and Contributions . . . 25
- 1.4 Road Map . . . 26

2 Preliminaries **29**
- 2.1 Introduction . . . 29
- 2.2 Process Modeling Languages . . . 29
  - 2.2.1 Business Process Model Notation (BPMN) . . . 30
  - 2.2.2 Configurable BPMN (C-BPMN) . . . 31
- 2.3 Languages for Formal Process Representation . . . 32
  - 2.3.1 Petri Nets . . . 33
  - 2.3.2 The Event-B Method . . . 36
    - 2.3.2.1 Machines and Contexts . . . 36
    - 2.3.2.2 Refinement in Event-B . . . 37
    - 2.3.2.3 Verification and Validation of Event-B Models . . . 38
- 2.4 Conclusion . . . 40

3 State of The Art **43**
- 3.1 Introduction . . . 43
- 3.2 Verification of Business Process Models . . . 44
- 3.3 Support and Verification of Business Process Configuration . . . 45
  - 3.3.1 Configuration Support . . . 46
  - 3.3.2 Domain and Guidance Support . . . 50
  - 3.3.3 Correctness Support . . . 52
  - 3.3.4 Synthesis . . . 53
- 3.4 Formalization and Verification of the Resource Allocation Behavior in Business Processes . . . 55
  - 3.4.1 Resource Allocation in Business Processes . . . 55
  - 3.4.2 Cloud Resource Allocation in Business Processes . . . 57
  - 3.4.3 Verification of Resource Allocation Behavior . . . 57
  - 3.4.4 Synthesis . . . 58
- 3.5 Conclusion . . . 60

4 Assisting Correct Process Variant Design with Formal Guidance 61
4.1 Introduction . . . 62
4.2 Motivating Example . . . 63
4.3 Approach Overview . . . 66
4.4 Formal Specification of a Business Process Model . . . 67
4.4.1 Business Process Model Graph . . . 68
4.4.2 Business Process Modeling using Event-B . . . 68
4.5 Formal Specification of a Configurable Process Model . . . 69
4.6 Formal Specification of Configuration Steps . . . 72
4.6.1 Activity Configuration . . . 72
4.6.2 Connector Configuration . . . 73
4.7 Introduction of the Configuration Guidelines into the Model . . . 76
4.8 Verification and Validation . . . 78
4.8.1 Verification using Proofs . . . 78
4.8.2 Validation by Animation . . . 78
4.8.3 Case Study . . . 79
4.8.3.1 Objective . . . 80
4.8.3.2 Design, Data Collection and Execution . . . 80
4.8.3.3 Results Analysis and Findings . . . 81
4.8.3.4 Threats to Validity . . . 81
4.8.4 ATL Model Transformation: BPMN to Event-B . . . 81
4.9 Conclusion and Discussion . . . 84

5 Extracting Deadlock-free Process Variants using Symbolic Observation Graphs 87
5.1 Introduction . . . 87
5.2 Approach Overview . . . 89
5.3 Formal Model for Configurable Business Processes . . . 89
5.3.1 Business Process Petri Nets (BP2PN) . . . 90
5.3.2 Configurable Business Process Petri Nets (CBP2PN) . . . 93
5.4 The Symbolic Observation Graph (SOG) . . . 96
5.5 Extracting Deadlock-free Configurations using the SOG . . . 99
5.6 Validation and Experiments . . . 105
5.7 Discussion . . . 107
5.8 Conclusion . . . 109

6 Towards Correct Cloud Resource Allocation in Business Processes 111
6.1 Introduction . . . 112
6.2 Motivating example . . . 113
6.3 Approach Overview . . . 114
6.4 Cloud Computing Resources : OCCI Standard . . . 115
6.4.1 Cloud Resource Types . . . 116

6.4.2 Cloud Resource Properties 116
6.4.3 Abstract Definition of a Cloud Resource-based Process 117
6.5 Formal Specification of a Business Process Model 119
6.5.1 Modeling Control Flow using Event-B 119
6.5.2 Introducing Execution Instances: First Level of Refinement . . 121
6.6 Formal Specification of the Resource Perspective 124
6.6.1 Modeling Resource Allocation: Second and Third Levels of Refinement 125
6.6.1.1 Introducing Shareability Property 126
6.6.1.2 Introducing Substitution Dependency 127
6.6.1.3 Introducing Resource Instances 128
6.6.2 Introducing the Elasticity Property: Fourth Level of Refinement 130
6.7 Verification and Validation 132
6.7.1 Verification using Proofs 132
6.7.2 Validation by Animation 134
6.8 Proof of Concept: Integration of Cloud Resource Representation . . . 137
6.9 Conclusion 139

7 Conclusion and Future Work **141**
7.1 Fulfillment of Objectives 141
7.2 Future Research Directions 143

Appendices **147**

A List of publications **149**

B Event-B Symbols Summary **151**

List of Tables

2.1 Constraints for the configuration of connectors [1] 32

3.1 Evaluation of related configuration approaches 55
3.2 Evaluation of related resource-based approaches 59

4.1 The average time in minutes unit spent to derive variants either correct (C) or not (¬C), and either business-complaint (B) or not (¬B) 81

5.1 Connectors configuration constraints [1] having $x \in \{+, -\}$. 94
5.2 Deadlock-free extracted configurations for the *CBP2PN* in Figure 5.5 105
5.3 Checking deadlock-freeness on SOG vs RG 107

6.1 Cloud Resource Properties description 117
6.2 Proof statistics 132

B.1 Some Event-B symbols and their semantics 151

List of Figures

1.1 Configuration and individualization of a configurable process model . . 19
1.2 Cloud resource allocation to process model activities 20
1.3 Configuration and Resources in the Business Process Lifecycle (adapted from [2,3]). 21
1.4 Our configuration-related research problem 23
1.5 Our Resource-related research problem 25

2.1 The BPMN main elements 31
2.2 A possible process configuration 32
2.3 Possible choices of an activity configuration 33
2.4 An example of a Petri net 34
2.5 An example of a Workflow net 36
2.6 Event-B machine and context 37
2.7 Event-B event and refinement event 37
2.8 Prooving in Rodin 39

3.1 An example of configuring a C-YAWL connector: from an OR-split to an XOR-split [4] 47
3.2 An example of a CoSeNet with the corresponding Petri net [5] 47
3.3 An example of a C-EPC process model [1] 48
3.4 The provop approach of variant modeling [6] 49
3.5 An example of a questionnaire model [7] 50
3.6 An example of a configured feature model generated after configuration step [8] 51
3.7 An example of a configuration guidance model [9] 52
3.8 The five-steps approach for guaranteeing Provop soundness [10] . . . 53
3.9 The RALph approach for graphic resource assignments [11] 56

4.1 A configurable process model of a hotel and car reservation agency . . 63
4.2 A hotel reservation variant of the configurable process in Figure 4.1 . . 64
4.3 Examples of configuration mistakes 65
4.4 A hotel and car reservation process variant designing 66
4.5 Our approach overview 67
4.6 The connector *j2* configuration restriction using ProB 79
4.7 Structure of the BPMN to Event-B transformation tool 82
4.8 The Process2Project, Process2Context and Process2Machine rules . . 83
4.9 The getAxiom6() helper 84
4.10 The getAxiom7() helper 84
4.11 The getInitialisationEvent() helper 85
4.12 counter example 86

5.1 The SOG-based approach overview 90
5.2 A configurable hotel booking process model 90
5.3 The BP2PN connectors mapping to classical Petri Nets 91
5.4 Enabling and Firing examples 93
5.5 The CBP2PN of the configurable process in Figure 5.2 94
5.6 A possible variant of the CBP2PN in Figure 5.5 94
5.7 Markings graph example in case of a non-configurable and a configurable transition exhibiting non-determinism 95
5.8 The Symbolic Observation Graph (SOG) 97
5.9 The obtained reduced SOG of the CBP2PN in Figure 5.5 101
5.10 The case of a successor aggregate already treated 102
5.11 An example illustrating aggregates configurations lists when pushing and popping them from the stack 104

6.1 A process variant and its resources 113
6.2 Event-B model 115
6.3 Class diagram of OCCI Infrastructure types [12] 117
6.4 Examples of process fragments for the activation relations illustration 120
6.5 Activity instance life cycle 121
6.6 Business process execution events, Machine BPM1 125
6.7 Resource instances' states 128
6.8 Rodin interface for discharging Proof Obligations 135
6.9 Animating the case of non-shareable resource 136
6.10 Extract of the animation values of Machine BPM1 after adding activities instances 136
6.11 A screen-shot of the graphical interface for our Cloud resource-based process modeling in Signavio 138
6.12 Excerpt of the *xsd* file "semantics.xsd" of the BPMN 2.0 extension . . 138

7.1 Need for adaptability to change 144

Chapter 1

Introduction

Contents

1.1 Research Context 17
1.2 Motivation and Problems Description 22
1.2.1 How to assist and verify business process configuration? 22
1.2.2 How to verify Cloud resource allocation in business process models? 23
1.3 Objectives and Contributions 25
1.4 Road Map 26

1.1 Research Context

Since the beginning of the nineties, business processes (BPs) have gained increased significance for almost any business. In order to manage and improve their business processes, organizations are more and more aligning their information systems in a process-centered way. Along this trend, Process Aware Information Systems (PAISs) have emerged to better manage and execute operational processes involving people, applications, and/or information sources on the basis of *process models* [13]. Most notable examples of such systems are Workflow Management Systems (WfMSs) [14, 15] and Business Process Management Systems (BPMSs) [16–18].

Business process models are key instruments of Business Process Management (BPM) in such systems. They explicitly represent business processes in terms of their activities and the execution constraints between them [3]. While the essence still quite similar, a range of graphical notations have been proposed for business process modeling, such as Business Process Model and Notation (BPMN) [19], Event-driven Process Chain (EPC) [20], Yet Another Workflow Language (YAWL) [21], Unified Modeling Language (UML) [22], etc. In fact, process modeling is part of the initial phase of business process lifecycle in a PAIS, that is the process *design and analysis* (see Figure 1.3). Once the process model is designed, it needs to be analysed using validation, simulation and verification techniques. This step is crucial since errors detected in early design phases will spoil any re-design efforts that might follow. Next, the second phase is the process model *implementation*. BPs are automated

into operational/executable processes. Thirdly, BPs are *executed*, after deployment on a PAIS, according to the process model. Finally, in the process *diagnosis* stage, process executions are analyzed to identify possible improvements leading to process re-design.

During recent years, while some organizations still focus on the design and analysis of their internal activities, an increasing number of them are targeting to align their BPs across organizational boundaries for collaboration needs. Indeed, two or more autonomous organizations (maybe also subsidiaries of the same organization) carry out an organized group of joined activities to achieve a common business objective, the so-called Inter-Organizational Business Process (IOBP) [23–28]. Typically, collaborating business *partners* (often called *tenants*) are involved in one 'global' IOBP which serves as a contract between them. Each partner has its own 'local' private business process which is usually designed separately and in an ad-hoc manner. Since these corporations are designed in a rigid manner, then it would be inefficient for organizations to engage (re-)designing and modeling "from scratch" their process models without learning from each other's practice and experience. Moreover, BPs need to be *flexible* since organizations are continuously willing to align their processes with new requirements (e.g., law, regulation, technology, etc.). ***Configurable process models*** were proposed [1, 4, 6, 29–31] as a step toward enabling a process "*design by reuse*" while offering *flexibility*. Furthermore, corporations most often operate across organizational boundaries. Then, there is an increasing need to enhance their deployment in a flexible and controlled manner. One way of facilitating BPs deployment, is to use ***Cloud computing*** infrastructures [32, 33]. Such a multi-tenant environment supports the sharing of common processes as well as IT resources on demand. The verification of process configuration as well as the verification of Cloud resource allocation within a BP are the scope of this thesis work.

Configurable process models were proposed to represent and group business processes that have similarities while exhibiting some local variations. These variations are captured using *configurable elements* that allow for multiple design options. Such processes are designed in a generic and integrated manner in order to be shared among different companies or branches. For instance, considering a hotel reservation agency that has many branches in different cities and countries, all branches processes include activities related to hotel searching and selection. Not surprisingly, they may have many differences depending on specific needs and priorities. An example of such differences may be a deposit being required to confirm the reservation in some regions and not in others, or even the type of the deposit may differ among regions (cheque, cash or credit card). Then, these processes can be configured and adjusted according to each specific company requirements by simply selecting the desired options and deselecting the undesired ones. The obtained individualized processes are called *variants* and are produced with a minimal design efforts.

In order to set clear these configuration concepts, let us take the simple example in Figure 1.1. The configurable process model, on the left hand side of the figure,

is modeled using the Configurable BPMN (C-BPMN) notation. This notation may have two types of configurable nodes that are highlighted in a thick border: activities (represented with rectangles) and connectors (represented with diamonds). More details about BPMN and C-BPMN are given in Chapter 3. There are three types of connectors modeling the splits and joins in the model: OR (inclusive choice), XOR (exclusive choice) and AND (parallel flows). In case of ordinary processes, these connectors represent run-time choices. Whereas, a configurable element represent a design-time *configuration choice* that analysts should take according to each individual organization needs. Our example contains one configurable activity, *A*, and two configurable connectors, *s1* and *j1*. For example, the analyst may choose to exclude the activity *A*. As a configurable connector may be configured by removing one or more input/output branches, then, the analyst may block the first output path of *s1* and the corresponding input path of *j1*. Then, using the *individualization* phase, a configured process model (cf. *Variant 1*) that corresponds to these configuration choices, is derived in the original process modeling language (i.e., BPMN). Hence, in this derived model, excluded elements are removed and maintained configurable elements are replaced with ordinary ones. For instance, the second derived model *Variant 2* in Figure 1.1 is obtained, first, by maintaining the activity *A*. Second, since a configurable connector's type may be changed, *j1* is configured from OR to XOR and *s1* is maintained the same. As the obtained variants do not contain any configurable elements, they can be executed by a PAIS.

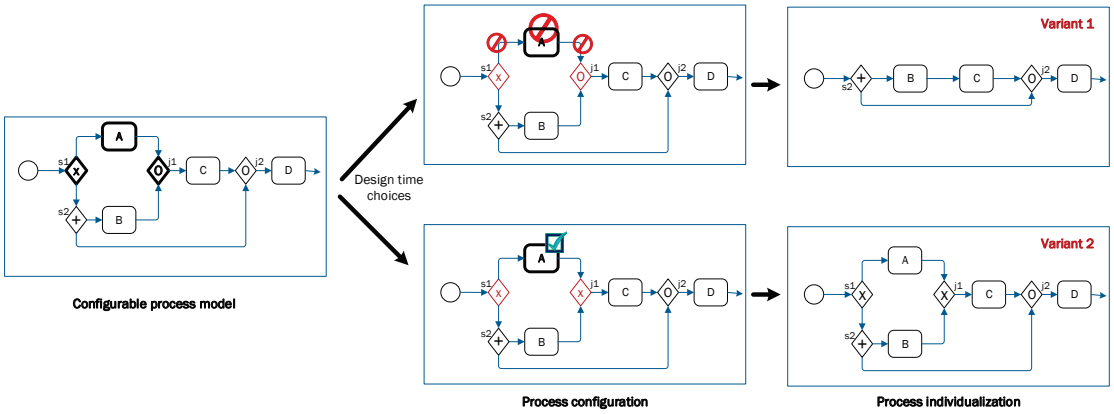


Figure 1.1: Configuration and individualization of a configurable process model

In this context, the *process design* phase in the BP lifecycle is replaced with the *configurable process design* phase. A number of configurable process modeling languages have been recently proposed such as Configurable BPMN (C-BPMN) (e.g. [6, 9]), Configurable EPC (C-EPC) (e.g. [1, 30]) and Configurable YAWL (C-YAWL) (e.g. [4]), that extend BPMN, EPC, and YAWL notations respectively with variable elements. Once the configurable process model is designed, the *diagnosis* phase may skip the phase of a new process model re-design (see Figure 1.3). The

process is instead directly adjusted by model users in the *configuration & individualisation* phase by applying configuration choices. As the configurable process may be very complex with a great number of configurable elements, interdependencies between the configuration choices may be very difficult to untangle. Hence, manually managing all the configurable elements of a process model one by one, leaving the designer the full responsibility for applying correct options, may be tedious and error-prone task. This implies that the configuration phase should also encompass an *analysis* step in order to verify and prevent errors in the derived process variants execution. Recent research work have proposed different approaches to facilitate the configuration of process models using, for example, configurable nodes [1,34], change operations [6], templates and rules [35], etc. Some approaches proposed to guide the configuration or/and to support domain-based constraints [7–9, 36]. Others attempted to ensure the process configuration correctness [10,37–39], but most of them still suffer form the exponential number of possible configurations.

In another side of such multi-tenant environment, in order to remain competitive, organizations are increasingly adopting PAIS on Cloud environments to benefit from rapid adaptability and flexibility in enabling access to IT resources on a "pay-as-you-go" basis. Hence, they are deploying part or all of their BPs variants in Cloud infrastructures with respect to customer requirements. Specifically, with such dynamic environment, process activities may be supplied with flexible and dynamic resources to rapidly and effectively respond to changing demands. These Cloud-delivered resources have the advantage of being *elastically scaled* on demand, and possibly *shared* by several activities. Let as take the example of Figure 1.2 where resources are assigned to the process activities. Suppose that activity A needs an elastic compute resource *r1* of 10 GB RAM to perform its task efficiently. This resource's initial offered capacity is 10 GB, but in case of changing needs of the activity, it may dynamically scale up or down during run-time. This is handled using horizontal (i.e., by supplying additional activity instances or removing them as necessary) or vertical (i.e., by increasing or decreasing offered resource capacity as necessary) elasticity techniques. Moreover, two activities, e.g., B and D, may be storing data in a shared storage resources *r3*.

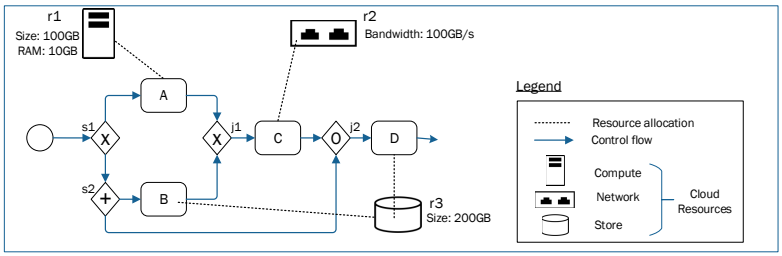


Figure 1.2: Cloud resource allocation to process model activities

Previous research in the field of resource allocation and management in the BPM

context mainly regarded human resources [40, 41]. Generally, extensions of business process models with the representation and the definition of human behavior were proposed [11, 42–45]. However, less attention has been paid to non-human resource allocation, especially Cloud ones. Hence, there is a clear lack of formal, unified and explicit description and representation of Cloud resources in the existing business process models. Consequently, the *correctness* and the *consistency* [46] of Cloud resources management in BPs cannot be verified. So, in the BPs design and analysis phases, the correctness criteria to be established should encompass not only the control flow but also the resource management. Thus, we extend these phases with *resources assignement* to process activities and *resources analysis* (see Figure 1.3).

In this thesis, in addition to *verifying the configuration aspect* of the control flow (i.e., process activities orchestration in a specific order) in process models, *we focus on enhancing the BP resource perspective* by specifically managing and analyzing the Cloud resource allocation issue. *"The resource perspective centers on the modeling of resources and their interaction with a process-aware information system (PAIS)"* [40]. Formal methods have proved their usefulness in the design of correct systems (e.g. Petri nets [47], Event calculus [48], LTL [49], Event-B [50]). They promote the use of mathematical foundations and formal logic to specify and reason about system properties.

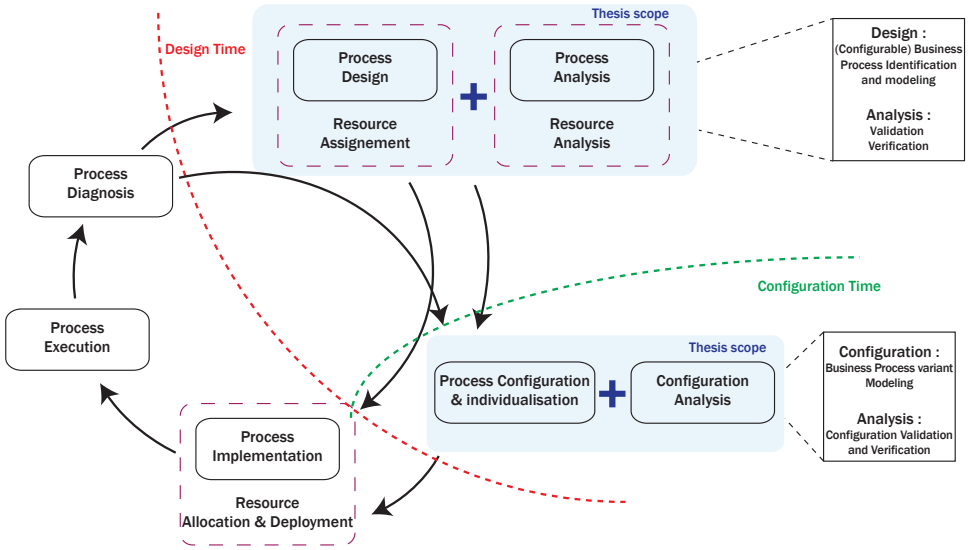


Figure 1.3: Configuration and Resources in the Business Process Lifecycle (adapted from [2, 3]).

To summarize, Figure 1.3 illustrates the cyclic structure of the process model lifecycle that consists of the different phases explained above (starting from the design phase). In this Figure, we highlight two phases that represent the scope of this

thesis: First, the *process configuration and individualization* phase is enhanced with an *analysis* phase in order to support the assistance and correctness verification of this configuration. Second, the *process design and analysis* phases are extended with the resource perspective in order to integrate the formal description of Cloud resources and to establish that the allocation behavior is correct.

1.2 Motivation and Problems Description

Our research work is motivated by the following two main issues: How to assist and verify business process configuration? and How to verify Cloud resource allocation in business process models?

1.2.1 How to assist and verify business process configuration?

The use of configurable process models still present challenges essentially related to the identification of the configuration steps with respect to different requirements and properties. Manual methods for configuring processes are undoubtedly tedious and error-prone. That is because, in case of complex processes, a very large number of configurable elements implies very complex inter-dependencies between their configuration alternatives. Therefore, analysts may easily be mistaken in their choices which undermine the correctness of the resulting variant.

We illustrate our research problem in Figure 1.4. During process configuration time, the process designer is wondering which configuration choices should he/she take. Once configured, he/she is not ensured that the derived process variant is correct with respect to different constraints: related to correctness and domain properties (i.e., rules allowing to comply with some business requirements). For example, a configurable activity may be removed from the model. In this case, the remaining process elements should be re-connected in order to maintain structural correctness of the model. When configuring a connector, removing an entire branch may lead to isolated activities. Even worse, the configuration of a join connector may lead to behavioral problems such as lack of synchronization and deadlock. For example, user should not configure a join connector to a synchronization if its corresponding split was already configured to an exclusive choice. This leads to a deadlock as only one branch is activated after the split, whereas the join needs the completion of all its incoming branches. These issues may be not easy to spot but have serious consequences on the process execution. Hence, there is a clear need to guide and assist the business analyst in order to correctly derive variants.

So far, a number of approaches have addressed the verification of the process configuration correctness. Some of them have discussed the strcutural correctness (e.g. [4, 6]). Others have addressed the behavioral correctness (e.g. [10, 37–39]). Traditionally, behavioral correctness verification often requires the reachability graph analysis (i.e. compute the state-space of process models). This means that, in case

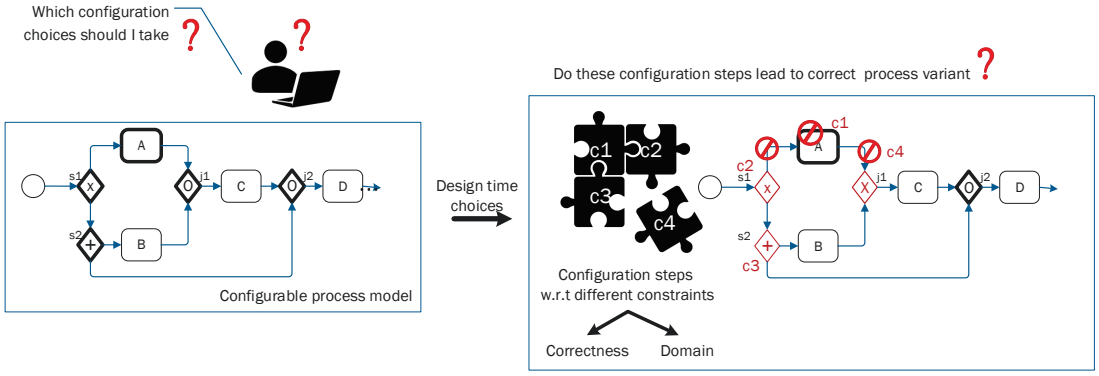


Figure 1.4: Our configuration-related research problem

of configurable processes, one needs to analyze the state-space of all possible configurations (e.g. using Woflan tool [51]). This may be too time-consuming and may lead to a state explosion problem since the number of states grows exponentially with the number of variation points.

In view of these issues, the process configuration options should be evaluated and automatically restrained with respect to all *configuration*, *correctness* and *domain* constraints. Then, the process user should be guided and assisted using these options in order to derive correct variants. Furthermore, as discussed in this section, behavioral verification of the process configuration often encounters the combinatorial explosion of state-space size challenge. To address our research problems, we need to answer the following sub-questions:

RQ1: How to identify configuration choices that satisfy designers and clients requirements?

RQ2: How to assist the designer in selecting the correct configuration choices?

RQ3: How to avoid the state-space explosion of the configuration verification issue?

1.2.2 How to verify Cloud resource allocation in business process models?

A PAIS running on a Cloud infrastructure offers the designers the access to a shared pool of IT resources delivered on demand and the ability to dynamically adjust their allocation in response to peaks in workload and according to their needs. This raises the introduction of the two important Cloud features: the shareability and the elasticity. These resources properties may depend on their Cloud providers and process tenant needs. Firstly, Cloud resources can be *elastically scaled* on demand to meet the real-time demand. For instance, in order to handle a growing amount of work, scaling

up a resource would be accomplished by resizing it. Secondly, different activities can cooperate to *share* the available resources in order to realize a specific task or to reduce the cost. Hence, Cloud resources may be shared and utilized at the same time by several tenants (in our case by several activities). Despite their wide-spread adoption in industry, the management and verification of Cloud resources in BPs still not yet mature enough. Mainly, considerable work [11, 42–45] have addressed the resource perspective in business process models by integrating human resource allocation, description, representation, interactions, roles, etc. Whereas, a formal description and representation of Cloud resources allocation as well as the assessment and verification of their properties behavior are still missing. Moreover, traditional process modeling standards such as BPMN do not support Cloud resource assignment and allocation parameters setting for process activities.

Establishing the correctness of Cloud resources behavior in BPs is becoming challenging. By doing so, designers may identify critical usage scenarios that might lead to emergent behaviors and could undermine the correctness of the process execution. Having a complex process model allocating many resources with different properties and dependencies, the designed process and the running process instances behavior can easily deviate from users' needs. For example, if an activity's needed capacity change at run-time, then its allocated resource should scale up to accommodate this need. In addition, the designer should respect consistency rules such as: a non-elastic resource should not be allocated to an activity if its offered capacity does not fit the needed capacity. These problems are illustrated in Figure 1.5, the process user is designing his/her process model with the necessary Cloud resources. The selected resources may have one of the following types: *store*, *compute* or *network*. Also, dependencies between resources can be captured. For instance, we consider the substitution dependency that allows to designate a substitute for a resource to perform the same work as another one in case of its unavailability. Once designed, the designer is wondering whether his/her process is correct in terms of resource allocation behavior as well as the defined properties and constrains.

Adopting formal methods and techniques to model Cloud-based process models and their resource allocation behavior can be very effective to validate and check resource constraints. As they provide a reliable mathematical basis that results in easily verifiable formal models, inconsistencies related to resource allocation and properties can be detected before deploying or even purchasing these resources from Cloud providers.

In this thesis, we aim to extend the resource perspective in business processes by considering Cloud resources. Hence, we need to formally describe and specify the Cloud resource behavior in BPs while considering different properties and constraints. For such purpose, we state the following research questions:

RQ4: How to formally specify and verify Cloud resource allocation behavior in BPs?

RQ5: How to integrate Cloud resources in BP models design?

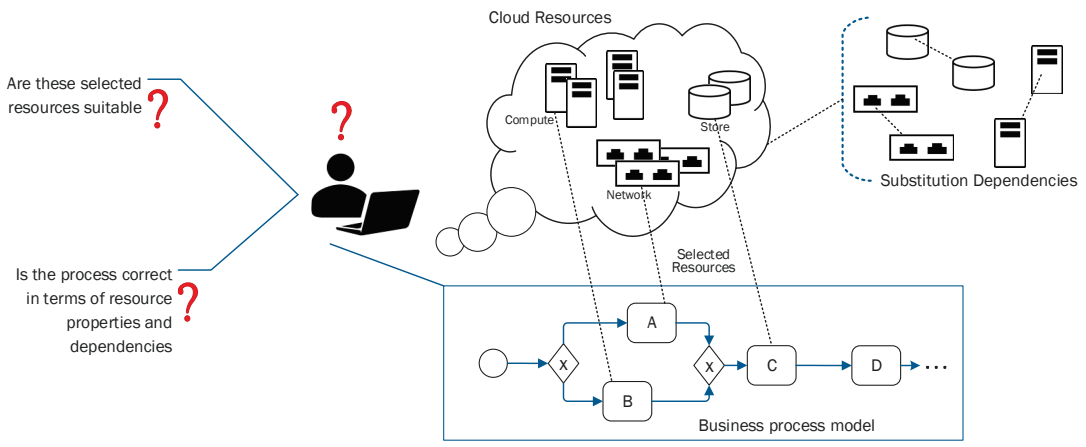


Figure 1.5: Our Resource-related research problem

1.3 Objectives and Contributions

In the light of the aforementioned shortcomings, the core objectives of this doctoral thesis are as follows:

- ***provide guidance and assistance to the analysts in process model configuration with correct options***. This objective is threefold: (1) formally model and specify the process model configuration, (2) identify correct configuration options with respect to different constraints and requirements, and (3) reduce the state space explosion problem related to process configuration verification.

- ***improve the support of Cloud resource specification and verification in BPs***. This objective is threefold: (1) precisely and formally model the Cloud resource perspective in BPs; (2) verify the Cloud resource allocation behavior in BPs; and (3) consider Cloud resources constraints and properties (i.e. elasticity and shareability).

This work strives to achieve the above objectives by proposing three contributions. ***The first contribution*** proposes an automated *stepwise approach* to assist the process models configuration in order to obtain correct and domain compliant variants. We introduce a formal specification of the configurable process model and the different *configuration steps* using the formal method *Event-B*. This specification includes different constraints related to the process variant structural correctness. Configuration guidelines referring to domain constraints are formally integrated as well. We verify and prove the correctness of our Event-B specification using formal proofs (Proof Obligations). These proofs allow to guarantee that the defined constraints are preserved by each configuration step and produce, thus, correct variants.

Moreover, we adopt the validation of our formal model by animating our specification using the plug-in ProB. Finally, this approach was automated by developing a transformation tool using the ATL model transformation language.

The second contribution aims at especially achieving the behavioral correctness that captures the dynamics of the executable configured process models. We propose an automated approach based on a *behavioral model* to assist the design of process variants. First, we define a formal model having precise and concise syntax and semantics for configurable process models using Petri nets. Then, based on this formal model, we use the Symbolic Observation Graph (SOG) [52, 53] in order to cope with the combinatorial explosion problem. The SOG is a symbolic representation formalism that allows to build *an abstraction of the reachability state graph* of the formally modeled system. In this work, we adapt and extend the existing definition and construction algorithm of the SOG graph in order to achieve such abstraction by: (i) observing the *configurable elements* of the process that label the SOG arcs, and (ii) hiding *non configurable elements* inside the SOG nodes. This reduced graph allows to extract all the possible configuration choices with respect to the deadlock-freeness property. Having these choices, the process user may select the combination that best meets his/her expectations and needs while being ensured that it leads to a correct process. Experiments have proven that the use of the SOG considerably reduce the state space size.

The third contribution proposes an *Event-B-based* approach to formally model and specify the resource perspective in BP models. The formal model allows for assigning Cloud resources to BPs activities and verifying the correctness of their allocation with respect to defined constraints and properties. In essence, we consider two Cloud properties: the elasticity and the shareability. First, we formalize the BP control flow perspective, then, we integrate the Cloud resources, their properties and execution constraints into the model using refinements. We use Event-B tools to prove and validate our specification by checking the defined properties and constraints at each process execution step. This refinement approach produces a *correct-by-construction* specification since we prove at each step the different properties of the system. Hence, our formal model guarantees that the process execution does not face failures and inconsistencies related to the resource sharing and elasticity properties w.r.t activities capacity and resources capabilities. Finally, with the aim to offer a tool that allows to assign Cloud resources to process activities, we developed an extension of the BPMN 2.0 notation in the Signavio modeling tool.

1.4 Road Map

This doctoral thesis is divided into seven chapters.

- **Chapter 2: Preliminaries** presents the necessary background information by introducing the process modeling languages as well as the formal descrip-

tions and notations used throughout the thesis. The business process modeling language BPMN, and the configurable business process modeling language C-BPMN, as well as the formal methods Event-B and Petri nets are introduced.

- **Chapter 3: State of The Art** positions our work, by reviewing existing literature on the verification of business process models. We also present existing work addressing the support and the verification of process configuration. The state of the art in the field of the formalization and the verification of the resource allocation behavior in BPs is also considered in this chapter.

- **Chapter 4: Assisting Correct Process Variant Design with Formal Guidance** describes our Event-B based approach that proposes the formal specification of the process configuration allowing to assist designers to configure, step-by-step, correct variants. The developed ATL transformation tool is also pointed out in this chapter.

- **Chapter 5: Extracting Deadlock-free Process Variants using Symbolic Observation Graphs** illustrates our SOG-based approach to assist the design of process variants by generating all the possible configuration choices with respect to the deadlock-freeness property. In this chapter, we evaluate our approach using experiments in order to prove the reduction of the state space size.

- **Chapter 6: Towards Correct Cloud Resource Allocation in Business Processes** presents our Event-B-based approach to formally model and specify the Cloud resource allocation behavior in BP models. We consider the elasticity and the shareability properties of Cloud resources in this formalization. A proof of concept is presented allowing to extend the BPMN 2.0 notation in the Signavio tool for process modeling.

- **Chapter 7: Conclusion and Future Work** concludes this thesis by summarizing the presented contributions and discussing potential future extensions.

Chapter 2

Preliminaries

Contents

2.1 Introduction 29
2.2 Process Modeling Languages 29
2.2.1 Business Process Model Notation (BPMN) 30
2.2.2 Configurable BPMN (C-BPMN) 31
2.3 Languages for Formal Process Representation 32
2.3.1 Petri Nets 33
2.3.2 The Event-B Method 36
2.3.2.1 Machines and Contexts 36
2.3.2.2 Refinement in Event-B 37
2.3.2.3 Verification and Validation of Event-B Models 38
2.4 Conclusion 40

2.1 Introduction

This chapter presents the basic preliminaries and background needed for the understanding of our contributions described in the remainder of this manuscript.

In Section 2.2, we present graph based (configurable) process modeling languages used to illustrate our research work. Then, Section 2.3 introduces the Event-B formal method and the Petri nets formalism that we used for the formal modeling of the both configuration aspect and resource perspective of business processes.

2.2 Process Modeling Languages

"Business processes are what companies do whenever they deliver a service or a product to customers" [54].

The business process modeling refers to creating *business process models* that describe the activities an organization has to perform to achieve a particular business goal, as well as their (i) execution order and constraints (*i.e., control flow*), (ii) required resources, e.g., humans or non-human/computer systems, (*i.e., resource flow*),

and (iii) processed data and information (*i.e., data flow*). Both perspectives: control flow and resource flow, are considered in the present thesis.

To model and represent business processes, a range of graphical process modeling languages have been proposed such as BPMN [19], EPC [20], YAWL [21], UML activity diagram [22], etc. Without limiting the generality of our work, we select and use BPMN as input notation. BPMN is highly adopted by business analysts since it is considered as the internationally recognized industry standard notation for business process description.

A *configurable process model* is a model that captures multiple variants of a same business process in a grouped manner. In our work, we use the Configurable BPMN (C-BPMN) [9, 55, 56], an extension of BPMN, as a configurable process modeling notation. An overview of BPMN and C-BPMN concepts is provided in the following sections.

2.2.1 Business Process Model Notation (BPMN)

The Business Process Model Notation (BPMN)[1] was first released in 2004 by the Business Process Management Initiative (BPMI) [19]. The key objective of this graphical notation is to support business process management by stakeholders of different roles; e.g. IT architects, business analysts, process owners, etc. On the one hand, one can model highly detailed end to end business processes while supporting the modeled processes execution. On the other hand, as business stakeholders are notoriously averse to standards, BPMN offers an increasing business support and an easier modeling experience.

The BPMN elements can be grouped into four groups *Flow Objects*, *Connecting Objects*, *Swimlanes* and *Artifacts*. *Flow Objects* are the basic graphical elements that allow to define the behavior of the BPMN model. These elements are depicted in Figure. 2.1. An *activity* (also called a task) represents a unit of work that should be done. It is graphically represented as a rounded corner rectangle. *Events* include three types : *Start, Intermediate and End events*. Each type represents something that happens during the execution of the business process and is graphically represented as a circle. *Gateways* (also called connector) define the control flow divergence, i.e. split, and convergence, i.e. join. Gateway is graphically represented as a diamond including a marker that indicates its type. Each type of gateway defines a run-time decision:

- Exclusive OR ($\times$): Based on conditions, a decision routes the sequence flow to exactly one of the outgoing branches. Then, the merging waits for one incoming branch to terminate execution in order to trigger the outgoing branch.
- Inclusive OR ($\circ$): Based on conditions, one or more branches are activated. Then, the merging waits for all active incoming branches to terminate execution

[1] In its latest version, BPMN was renamed to "Business Process Modeling and Notation"

to trigger the outgoing branch.

- Parallel AND (+): When splitting, all outgoing branches are activated simultaneously. Then the merging waits for all incoming parallel branches to complete before triggering the outgoing flow.

Flow Objects are connected to each other using *Connecting Objects* that include: *Sequence Flow*, *Message Flow* and *Association*. A *Sequence Flow* shows the execution order of flow elements. *Message Flow* determines the message flowing between pools or flow elements in pools. An *Association* allows to associate *Data objects* to a flow or to connect them to an activity.

Artifacts are used to provide more details and information about the business process without affecting the sequence and message flows of the process. There are three main types of artifacts: *data object, group and annotation*. For example, *Data object* provides some data for an activity performance (capturing the data perspective). *Swimlanes* include: *Pools* and *Lanes*. *Pools* group a set of activities that have some common characteristic, e.g. a specific role, a process participant (capturing the resource perspective). A *lane* permits to divide a pool to group specific process steps.

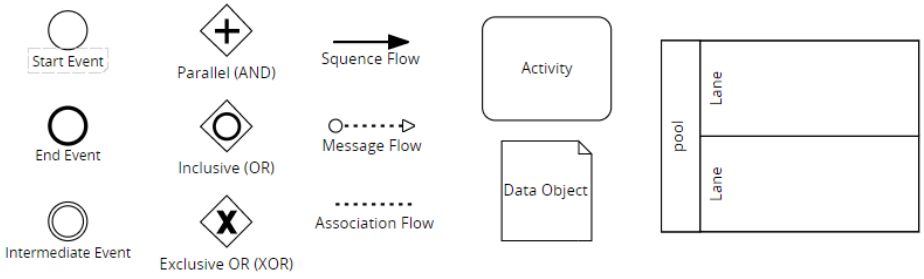


Figure 2.1: The BPMN main elements

2.2.2 Configurable BPMN (C-BPMN)

Configurable BPMN (C-BPMN) is an extension of BPMN to introduce the notion of variability in business process models. This variability is captured by restricting the process behavior through *configurable elements*. The non-configurable elements represent the commonalities in the configurable model. The configuration decision of a configurable element is made at design-time [1]. C-BPMN includes two configurable elements: *activities and connectors*, which are modeled with a thicker border.

A connector may be configurable to restrict its behavior by (i) changing its type (e.g. from OR to AND), or/and (ii) restricting its incoming or outgoing branches. A connector may change its type according to a set of configuration constraints [1] (see Table 2.1). Each row corresponds to the initial type that can be mapped to one

or more types in columns. For example, an *OR* type can be configured to any type while an *AND* remains unchangeable. Let us take the example of the configurable process example in Figure 2.2. The derived variant on the right hand side of the figure is obtained if, first, the analyst does not need both activities B and C. This refers to configuring *s1* to a sequence starting with A (i.e. the outgoing branch of *s3* starting with B is removed). Second, we suppose that the analyst needs the execution in parallel of the activities D and E. This refers to configuring *s2* (resp. *j2*) from OR-split (resp. OR-join) to AND-split (resp. AND-join) while maintaining the same outgoing (resp. ingoing) branches.

Table 2.1: Constraints for the configuration of connectors [1]

FROM-TO	OR	XOR	AND	seq
OR	√	√	√	√
XOR		√		√
AND			√	

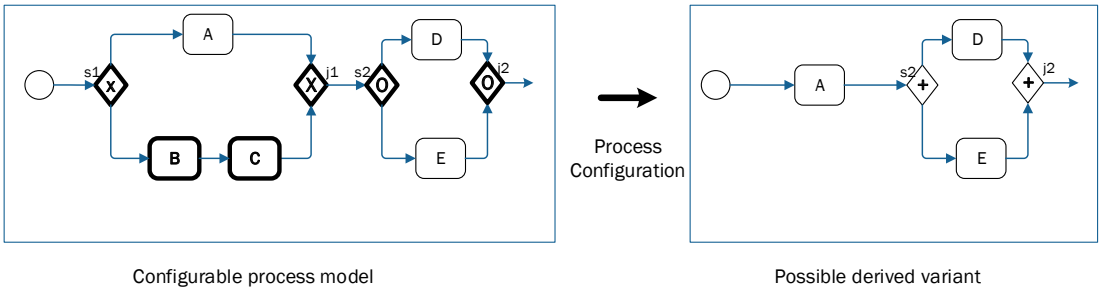


Figure 2.2: A possible process configuration

Also, an activity may be needed in a process variant and not in another depending on specific requirements. Hence, these activities should be configurable in order to be included (i.e. *configured to ON*) or excluded (i.e. *configured to OFF*), or optionally excluded (i.e. *configured to OPT*) from the model [1] (see Figure 2.3). The last configuration type refers to a run-time choice, whether to execute this activity or to skip it. This *OPT* configuration type is out of scope of this thesis. As we discussed about the variant in Figure 2.2, the analyst does not need the activities B and C. This also refers to configuring B and C to OFF (they are removed). Whereas, if he/she choose to maintain them, they will be configured to ON, hence, they will be kept in the resulting variant.

2.3 Languages for Formal Process Representation

Due to the lack of formal semantics of the process modeling languages, such as BPMN and C-BPMN, ambiguous interpretations remain possible and the verification and

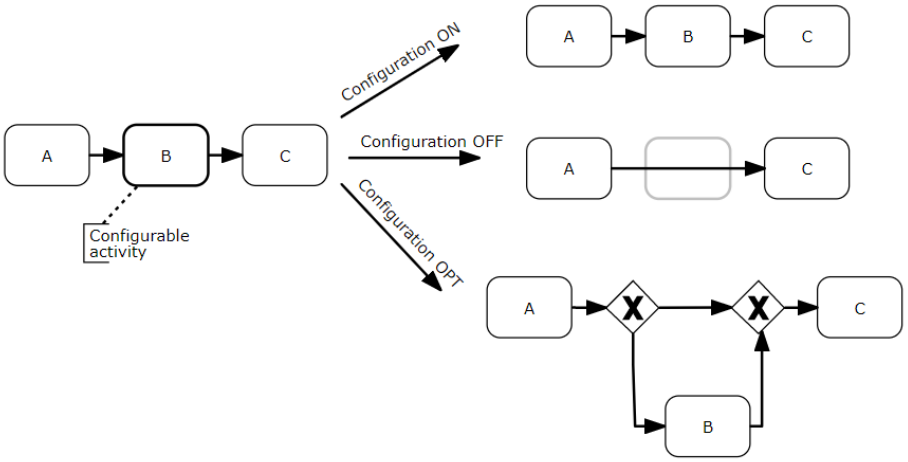


Figure 2.3: Possible choices of an activity configuration

validation of the designed process model is left to the testing and deployment phases. As a consequence, building correct, efficient and trustable process models becomes a major challenge. In this thesis, we use two formalisms that are Petri nets and Event-B.

2.3.1 Petri Nets

Petri nets are state-transition systems that offer a formal model for concurrent systems. Unlike most of the process modeling notations such as those discussed in Section 2.2, they include a mathematical definition of their execution semantics. Petri nets is a state-based technique that have a strong mathematical foundation and unambiguous semantics for modeling the behavior of a system.

In the following, firstly, we formally define the Petri net's syntax and semantics as well as some notations. Then, we define its subclass Workflow net.

Syntax: A Petri net is formally defined as follows.

Definition 2.3.1 (Petri Net)**.** *A Petri net is a tuple* $N = \langle P, T, F, W \rangle$ *s.t.:*

- P *is a finite set of places and* T *a finite set of transitions with* $(P \cup T) \neq \varnothing$ *and* $P \cap T = \varnothing$*,*
- *A flow relation* $F \subseteq (P \times T) \cup (T \times P)$*,*
- $W : F \to \mathbb{N}^+$ *is a mapping assigning a positive weight to arcs.*

A place p is called an input (resp. output) place of a transition t if there exists an arc from p to t (resp. from t to p). Hence, we define the following notations.

Notations:

- Each node $x \in P \cup T$ of the net has a pre-set and a post-set defined respectively as follows: ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$, and $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. This notation is extended to the sets of nodes X, s.t. $X \subseteq P \cup T$, as follows: ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$, and $X^\bullet = \bigcup_{x \in X} x^\bullet$.
- For a transition t, $W^-(t) \in \mathbb{N}^{|P|}$ (resp. $W^+(t) \in \mathbb{N}^{|P|}$) denotes the vector where, $\forall p \in P$, $W^-(t)(p) = W(p, t)$ (resp. $W^+(t)(p) = W(t, p)$).
- A marking of a Petri net N is a function $m : P \to \mathbb{N}$.

Semantics: Let m be a marking of a Petri net N and let $t \in T$ be a transition,

- **Enabling rule:** the transition t is said to be *enabled* by m, denoted by $m \xrightarrow{t}$, iff $W^-(t) \leq m$.
- **Enabling rule:** when t is enabled by m, its *firing* yields a new marking m', denoted by $m \xrightarrow{t} m'$, s.t. $m' = m - W^-(t) + W^+(t)$.

A Petri net transition t is enabled (i.e., may fire) once its input places are sufficiently filled. Firing this transition t in a marking m consumes $W^-(t)(p)$ tokens from each of its input places p, and produces $W^+(t)(p)$ tokens in each of its output places p.

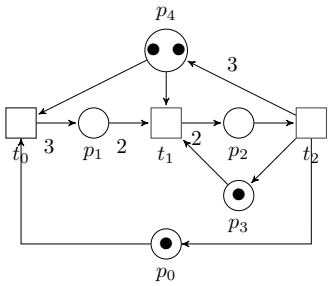


Figure 2.4: An example of a Petri net

An example of a Petri-net is illustrated in Figure 2.4. Places are represented by circles, transitions by rectangles, and the structure of the graph by arcs. The place p_0 has one token denoted by a black dot. The arc from place p_0 to transition t_0 denotes an ordinary arc, i.e., $W(p_0, t_0) = W^-(t_0)(p_0) = 1$, indicating that when firing t_0 it consumes one token from p_0 (i.e. p_0 becomes not marked). similarly, when firing t_0 it consumes also one token from p_4 (i.e. p_4 becomes marked with only one token). Whereas, the arc from transition t_0 to place p_1 has a weight 3, i.e., $W(t_0, p_1) = W^+(t_0)(p_1) = 3$ indicating that when firing t_0, it produces three tokens in p_1.

Notations:

- For a finite sequence $\sigma = t_1 \dots t_n$, $m_i \xrightarrow{\sigma} m_n$ denotes the fact that σ is enabled by m_i, and that its firing leads to m_n.

- Given a set of markings S, we denote by $Enable(S)$ the set of transitions enabled by elements of S.

- The set of markings reachable from a marking m in N is denoted by $R(N, m)$.

- Given the initial marking, denoted m_i, the reachability graph of a Petri net N, denoted by $G(N, m_i)$, is the graph where nodes are elements of $R(N, m_i)$ and an arc from m to m', labeled with t, exists iff $m \xrightarrow{t} m'$.

- The set of markings reachable from a marking m, by firing the transitions of a subset T' only is denoted by $Sat(m, T')$. By extension, given a set of markings S and a set of transitions T', $Sat(S, T') = \bigcup_{m \in S} Sat(m, T')$.

- For a marking m, $m \not\rightarrow$ denotes that m is a dead marking (i.e., there is no transition s.t. $m \xrightarrow{t}$ which means $Enable(\{m\}) = \varnothing$).

A number of sub-classes of Petri Nets have been defined in the literature, we specifically consider the Workflow (WF) nets [57, 58] which is tailored to express business process models and is usually used as an intermediary formalism in the modeling and the verification of the control flow of business processes. A transition is equivalent to a task in high level process modeling languages (e.g., activity in BPMN). The flow relation F is equivalent to the control-flow of a process model.

Definition 2.3.2 (WF-Nets)**.** *Let $WF = \langle P, T, F, W \rangle$ be a Petri net and F^* is the reflexive transitive closure of F. N is a Workflow net (WF-net) iff:*

- *there exists exactly one input place $i \in P$, s.t. $|{}^\bullet i| = 0$,*
- *there exists exactly one output place $o \in P$, s.t. $|o^\bullet| = 0$,*
- *each node is on a directed path from the input place to the output place, i.e. $\forall n \in P \cup T, (i, n) \in F^* and (n, o) \in F^*$.*

We note that the initial marking of WF where only i place is marked, is denoted by m_i, and the final marking where o is the sole place holding a token is denoted by m_f. Compared to classical Petri net, a WF-net has two particular places i and o representing the initial and the final states of the workflow respectively. Yet, the semantics of WF-nets remains as described above. An example of a WF-net is depicted by Figure 2.5.

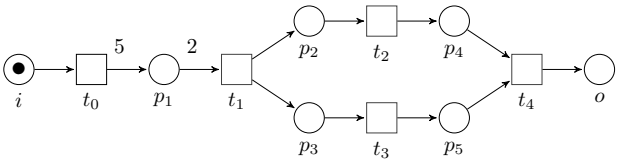


Figure 2.5: An example of a Workflow net

2.3.2 The Event-B Method

Event-B [50] is both a language and a method for formal specification and verification of secure systems. It has been proposed by J-R Abrial as the successor of the classic B Method [59]. Event-B has preserved the advantage and the simplicity of the B method while making improvements in several aspects, including the specification of reactive systems. Following the B Method, Event-B uses basic mathematical notations, first-order logic and set theory. It supports a large part of the development life cycle, from the specification/design phase to the implementation phase. This allows the early errors detection which prevents from execution errors and facilitates maintenance.

The complexity of a system is mastered thanks to the refinement concept allowing to gradually introduce the different parts that constitute the system starting from an abstract specification to a more concrete one. The abstract specification describes the fundamental properties of the system. Requirements and details are added incrementally through the refinement process.

One benefit of using Event-B is the supporting eclipse-based RODIN[2] platform [60] allowing to analyze, check types, and generating proof obligations of the model. For this aim, different external tools, e.g. Atelier B provers [61], animators, model-checkers like ProB [62], can be plugged on RODIN.

2.3.2.1 Machines and Contexts

An Event-B specification is made of two elements: *context* and *machine*. The machine is a fundamental component for the formal construction of a system in Event-B; it specifies its *dynamic part*. It includes elements such as variables V, invariants Inv, and events E that establish the system state change. The variables define the state of the system to be specified. The possible values that the variables hold are restricted using invariants written using first-order predicates. Invariants should remain valid in each state of the system. Thus, they should be valid in the initial state and after the execution of each event.

Machines often need static elements of the system such as constants C, sets S, and axioms A that specify their properties. These elements are included in a context that describes the *static part* of an Event-B specification. To have access to its elements,

[2]The Rodin Platform: `http://www.event-b.org/platform.html`

a context is seen by a machine (i.e. SEES *Context*). The structure of machines and contexts is depicted by Figure 2.6.

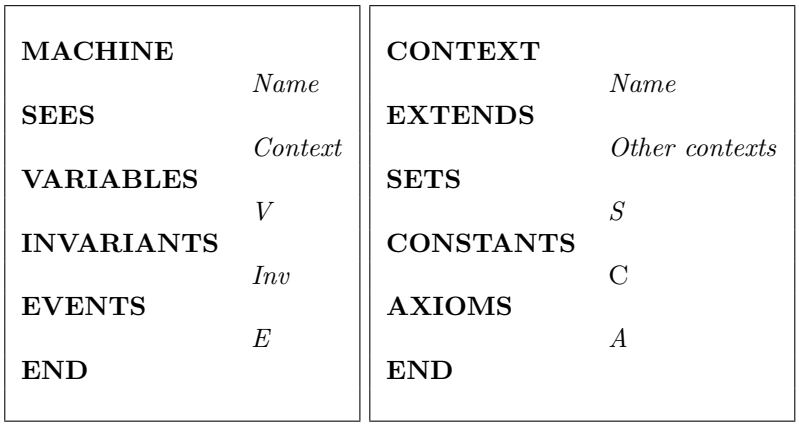


Figure 2.6: Event-B machine and context

An event can be executed if it is enabled, i.e. all the conditions G, named guards, prior to its execution hold. Among all enabled events, only one is executed. In this case, substitutions *Act*, called actions, are applied over variables. In this thesis, we restrict ourselves to the *becomes equal* substitution, denoted by $(x := e)$. Each event has the form depicted in Figure 2.7.

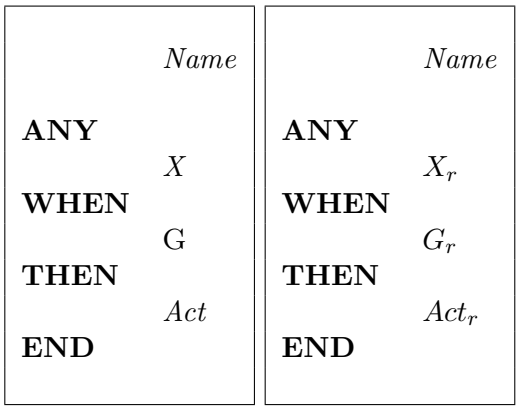


Figure 2.7: Event-B event and refinement event

2.3.2.2 Refinement in Event-B

Refinement is a process of enriching or modifying a model in order to augment the functionality being modeled, or/and explain how some purposes are achieved. Both Event-B elements *context* and *machine* can be refined. A context can be extended by

defining new sets S_r and/or constants C_r together with new axioms A_r. A machine is refined by adding new variables and/or replacing existing variables by new ones V_r that are typed with an additional invariant Inv_r. New events can also be introduced to implicitly refine a **skip** event (i.e. It does not modify the already existing variables). In this thesis, the refined events have the same form (see Figure 2.7).

The main advantages of the refinement process are as follows:

- Simplification of proof obligations (POs) that allow to check the model correctness
- The complexity of the development is apportioned between the abstraction levels

A stepwise refinement approach produces a *correct specification by construction* since we prove the different properties of the system at each step.

2.3.2.3 Verification and Validation of Event-B Models

This section describes two complementary techniques to ensure, respectively, the verification and the validation of an Event-B specification. In the verification step, we formally check the properties of the system, which are expressed in terms of invariants, using formal proofs (*proof obligations*). POs allows to prove that invariants (both the abstract and the concrete ones) hold in all system states: they hold initially; and each event preserves them. And then, we adopt the validation of our formal model by animating our specification using the plug-in ProB. This step allows to discover and observe the behavior of our specification.

Proof Obligations In order to demonstrate the model correctness, a collection of proof obligations (POs) is generated by the Rodin tool [60]. These POs ensure that invariants are preserved by each event. Therefore, for each event, we have to establish that:

$$\forall S, C, V, X.\ (A \wedge G \wedge Inv \Rightarrow [Act]Inv)$$

where $[Act]Inv$ gives the weakest precondition on the *before* state such that the execution of Act leads to an *after* state satisfying Inv.

To prove that a refinement is correct, we have to establish the following two proof obligations:

- *guard refinement*: the guard of the refined event should be stronger than the guard of the abstract one:

  $$\forall(S, C, S_r, C_r, V, V_r, X, X_r).\quad (A \wedge A_r \wedge Inv \wedge Inv_r \Rightarrow (G_r \Rightarrow G))$$

- *Simulation*: the effect of the refined action should be stronger than the effect of the abstract one:

$$\forall(S, C, S_r, C_r, V, V_r, X, X_r). \quad (A \wedge A_r \wedge Inv \wedge Inv_r \wedge [Act_r]Inv_r \Rightarrow [Act]Inv)$$

Formal definitions of all proof obligations are given in [50]. To discharge the different proof obligations, the Rodin[3] platform offers an automatic prover but also the possibility to plug additional external provers like the SMT and Atelier B provers that we use in this work. Both provers offer an automatic and an interactive options to discharge the proof obligations. Complex proof obligations could be discharged interactively using the proving perspective of Rodin shown in Figure 2.8.

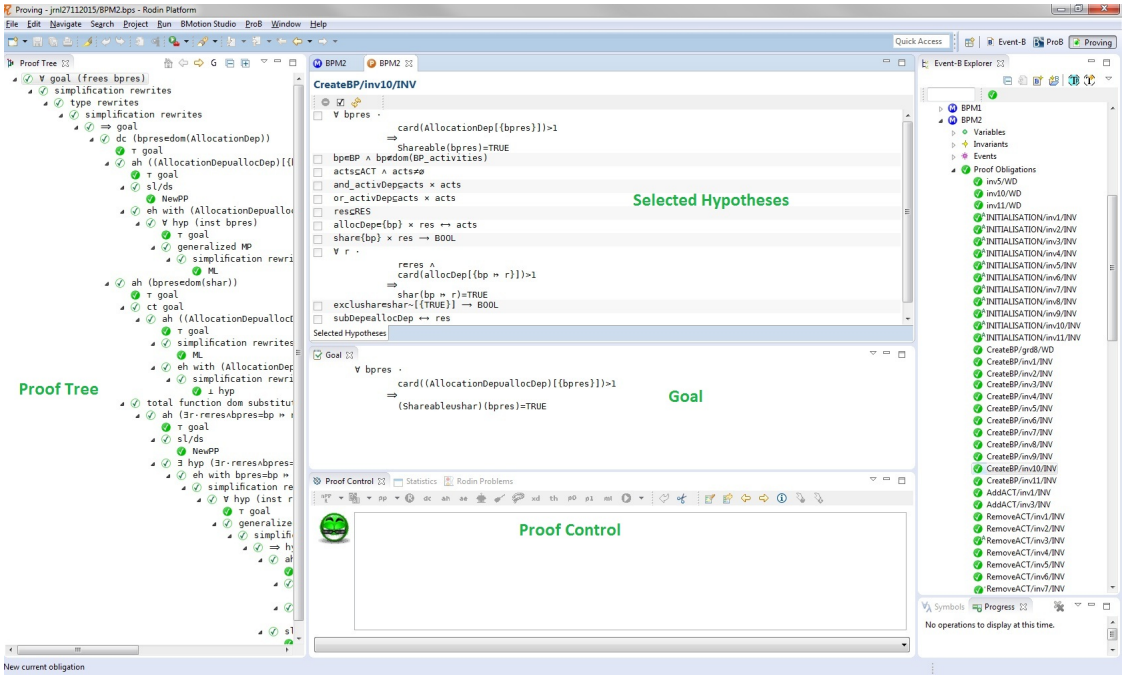


Figure 2.8: Prooving in Rodin

ProB (model checker/ animator) ProB [62] is an animator and explicit automatic model checker, originally developed for the verification and validation of software development based on the B language. Developed at the University of Düsseldorf starting from 2003, ProB [4] implements an automatic model checking technique to check LTL (linear temporal logic) [49] and CTL (Computational Tree Logic) [63] properties against a B specification. The core of ProB is written in a logical programming language called Prolog. Its purpose is to be a comprehensive tool in the area of formal verification methods. Its main functionalities can be summarized up as follows:

[3] `http://www.event-b.org/install.html`
[4] https://github.com/bendisposto/probparsers

1. ProB can find a sequence of operations that, starting from a valid initial state of the machine, moves the machine into a state that violates its invariant,

2. Giving a valid state, ProB can exhibit the operation that make the invariant violated,

3. ProB allows the animation of the B/EventB specification to permit the user to play different scenarios from a given starting state that satisfies the invariant. Through a graphical user interface implemented in Tcl/Tk, the animator provides the user with: (i) the current state, (2) the history of the operation executions that has led to the current state and (3) a list of all the enabled operations, along with proper argument instantiations. In this way, the user does not have to guess the right values for the operation arguments.

4. ProB supports the model checking of the LTL and CTL assertions.

In the domains of BPM and Service Oriented Computing, Event-B was especially used to address the web services and services composition verification by using proofs and refinements [64,65]. Specifically, properties related to the transactionnal behavior of languages like BPEL are checked [66–68]. For instance, in these models, the set of all basic services is defined in the CONTEXT, and available services are defined as its variable subset in the MACHINES. Dependencies such as sequence, compensation, abortion and cancellation are defined using relations in the INVARIANTS clause. The behavior of services and transitions is insured by *events*. Other few approaches proposed to verify business process models behavior [69,70]. For example, [69] propose a translation of the BPMN constructs and features to Event-B model. The control flow and data flow are considered. All possible processes and their instances, as well as activities and their possible instances are represented by sets in the CONTEXT. Sequence and data flows, are mapped to functions in the INVARIANT clause. Triggers, such as messages receiving, are captured by *events*.

In our thesis, we were inspired from these approaches especially in the specification of the control flow of a process model. Our work is proof-oriented and specifies a process model and all its properties (related to either configuration or resource perspective) to obtain an Event-B model. All the Event-B specifications presented in this manuscript have been verified within the RODIN platform. In Chapter 4, Event-B is used to formally specify and verify the configuration of a process model. Then, in Chapter 6, it is used to formally specify and verify the resource perspective in a BP while especially taking into consideration Cloud properties and constraints.

2.4 Conclusion

This chapter provided the background that helps position our contributions. In fact, it introduced two types of process modeling languages used in our work, namely

informal business process modeling languages and notations for formal process specifications. We presented the Business Process Modeling Notation (BPMN) as an example of business process modeling languages. We also introduced its extension with configurable elements, namely the Configurable BPMN (C-BPMN). Formal definitions and notations were provided for the Event-B language and the Workflow nets, which are a subclass of Petri nets. Throughout this manuscript, we use BPMN and C-BPMN whenever we discuss process models and configurable process models respectively. These notations serve as the starting points for building our approaches. We also use the illustrated formal languages for specifying and identifying the essence of process models in two main aspects: configuration and resource allocation, as well as for reasoning about the correctness of the process models.

Chapter 3

State of The Art

Contents

3.1 Introduction 43
3.2 Verification of Business Process Models 44
3.3 Support and Verification of Business Process Configuration . . 45
3.3.1 Configuration Support 46
3.3.2 Domain and Guidance Support 50
3.3.3 Correctness Support 52
3.3.4 Synthesis 53
3.4 Formalization and Verification of the Resource Allocation Behavior in Business Processes 55
3.4.1 Resource Allocation in Business Processes 55
3.4.2 Cloud Resource Allocation in Business Processes 57
3.4.3 Verification of Resource Allocation Behavior 57
3.4.4 Synthesis 58
3.5 Conclusion 60

3.1 Introduction

In this chapter, we survey the state of the art that allows to justify our problem statement and to have a clear position regarding the existing work.

In this these, we are interested in two relevant process modeling perspectives: the control flow, specifically the configuration aspect, and the resource perspective. Since some structural or behavioral anomalies may occur in both perspectives, e.g., the designed variant may be stuck during execution, or a resource may be allocated to process activity while not having a sufficient capacity. We propose to discuss the verification challenge in the BPM field. Specifically, we aim to address the verification of business process configuration in order to answer the question: How to assist and verify business process configuration? Then, we aim to examine the process resource allocation behavior in order to answer the question: How to verify Cloud resource allocation in business process models?

In the following, we firstly review the state of the art on the verification of business process models in Section 3.2. Then, in Section 3.3, we study the major existing approaches that support the process model configuration and the design of variants while discussing the supported properties and constraints. Afterwards, a broader view on the resource perspective in BPs is taken in Section 3.4, with a special look at the Cloud aspect and the allocation behavior verification and formalization. By the end on the last two sections, we give a comparison of the related approaches and we identify their shortcomings in order to motivate our research work.

3.2 Verification of Business Process Models

The process verification is the task of determining and checking, whether it exhibits erroneous behaviors. This refers to process correctness checking. Hence, the verification could be applied at design time in order to detect possible errors, and if so, the model should be modified before execution. The validation aims at checking whether the system actually behaves as expected. The later is context dependent and can only be applied with knowledge of the intended business process [71]. Since PAIS rely on process models for organization's work execution, careful verification and validation of process models at design time can greatly improve the reliability and efficiency of such systems. Therefore, there have been many efforts to achieve that by defining formal semantics of process models and applied various logics and formal methods. Since the most widely used process modeling languages do not have formal semantics, notably EPC [20], BPMN [19], UML activity diagram [22]. Therefore, these modeling techniques need to be mapped into formal models in order to be verified, e.g., [72–74].

At the beginning, most proposed approaches focused on simple languages, e.g., workflow graph model [75], which are even less expressive than classical Petri nets. After that, the use of *Petri net* formalism was widely investigated thanks to its understandable and graphical notation as well as well-defined semantics. The Workflow nets [57,76] (cf. Section 2.3.1) are its most notably sub-class, that were used as intermediate formalism to verify and analyze workflows/business processes, e.g., [77–85]. Other formal methods were used for verifying BP based on process algebra, such as *π-calculus* [86] (e.g., [87,88]) and *event calculus* [48] (e.g., [89]). Moreover, a number of *proof-based* approaches have been proposed to verify business processes [64,66–68,90]. In fact, verifying a formal specification using formal proofs is very important in order to guarantee the preservation of correctness criteria. Other work applied techniques for showing consistency of BPs using *model checking* [91, 92]. These techniques are used to automatically verify basic requirements of a BP such as the termination and reachability of states.

Using the different formalisms, proposals usually verify the *control flow* of a BP against a *correctness criterion*. A popular correctness property used in this context is the *soundness*. Firstly, a *structural soundness* [3] criterion has been defined in the context of Petri nets such that: (1) a process should have exactly one initial node

and one final node, and (2) each node in the process model is on a path from the initial node to the final one. Then, with the introduction of Workflow nets, the original (behavioral) *soundness* criterion [57,80,93,94] is proposed and then adapted for other modeling languages. A workflow is said to be *sound* if and only if, (i) when started, it can always complete processing *(option to complete)*, (ii) it terminates properly, i.e. no running tasks when the process ends *(proper completion)*, and (iii) there is no dead parts or activities, i.e., that will never be executed *(no dead transitions)*. So, in a sound workflow, anomalies such as *deadlock* and *livelock* are absent [95]. In order to decide the soundness of a given process, one can analyse and check its reachability graph that refers to explicitly representing the different states of a process instance. Other authors suggest weakening the soundness notion: e.g. the *relaxed soundness* [96] states that for each transition there should be at least one proper execution, which does not prohibit potential deadlocks and livelocks. Also, the notion of *weak soundness* [97] allows for dead transitions.

Nevertheless, the above mentioned approaches have mainly focused on verifying the *control flow* of process models. The verified processes do not support neither the configuration aspect nor the allocated resources. On the one hand, the resource perspective is not well defined in particular when business processes are deployed in Cloud environments (cf. Section 3.4). Hence, we seek to address the verification of this perspective. On the other hand, having a configurable process model, the presented proposals can be used to check every single configured process that can be derived from it. The configuration that generates an incorrect variant should be excluded from the set of possible configurations. However, this approach is costly and may be not feasible in case of configurable process models that yield a large number of process variants. Our aim is therefore to define an approach that allows to find correct configuration steps without computing all the possible configurations. We need to consider correctness properties and to use adequate formal methods. In the next section, we discuss the existing work on process configuration and on the verification of different properties and requirements.

3.3 Support and Verification of Business Process Configuration

The goal of configuring a process model is to customize and adapt an original model in order to better fit the user's specific needs and requirements. Several approaches have been proposed to model variability in configurable process models. We distinguish two types of variability: i) variability by restriction and ii) variability by extension [98]. The first type is used to restrict the behavior of the configurable process that should contain all possible behavior of the variants. The second type is used to add behavior to the configurable process which means that the later process contains the most common behavior and needs to be extended. Note also that it is possible to

combine the two types. In Section 3.3.1, we review the major approaches of variability regarding both types in order to motivate our choice to one of them.

An adaptation of a configurable process model can produce errors in terms of *structure* (e.g. disconnected nodes), *behavior* (e.g. deadlocks and livelocks), and *domain* (e.g. not satisfying a domain constraint). Therefore, the configuration decisions should not be taken freely and appropriate guidance should be provided. Hence, we present, in Section 3.3.2, approaches that support domain constraints and decision guidance. And in Section 3.3.3, we examine the correctness properties supported in the literature.

3.3.1 Configuration Support

In order to facilitate the design of configurable process models, a number of process modeling languages have been recently extended with variable elements, namely Configurable Event-driven Process Chain (C-EPC) (e.g. [4, 31, 34, 36, 99]), Configurable Business Process Model Notation (C-BPMN) (e.g. [6,9,55,100–102]) and Configurable Yet Another Workflow Language (C-YAWL) (e.g. [6, 102]).

Based on some of them, many approaches have been proposed to represent variation points in process models. We can classify them into five categories, based on their underlying variability mechanism: *hiding and blocking* of elements, *configurable nodes*, *annotations*, *fragment change*, and *templates and rules*.

Hiding and Blocking In [29,30], authors introduce a language-independent methodology to configure and restrict a workflow model by applying *hiding* and *blocking* operators on edges (i.e. transitions) of Labeled Transition Systems (LTSs). The blocking decision means that the transition will never be executed. Whereas, the hiding decision means that the transition will be skipped but the corresponding path still considered. This transition were called silent action. This approach was applied in [4] to suggest the extension of YAWL, namely C-YAWL, with the so-called ports as variation points while still using the blocking and hiding techniques for configuration. Each task has input port and output port representing respectively the join of arcs through which the task can be enabled, and the split of arcs that can be enabled after the task's completion. For instance, having the example of Figure 3.1, the OR-split connector is configured by activating the input port and only two the output ports having condition b and d (cf. green icons). Since each enabled output port refer to only a single flow, the split behavior is changed into an XOR-split behavior.

Another approach using blocking and hiding operations is introduced in [5]. Authors use *CoSeNets* (Configurable Service Nets) to represent configurable process models as a tree-like representation. Each node of a CoSeNet represents a process operator and each leaf represents a task. Hence, CoSeNet captures a block-structured process model, an example of such structure is depicted by Figure 3.2. The configuration of this process consists in hiding ad blocking special nodes that connect nodes

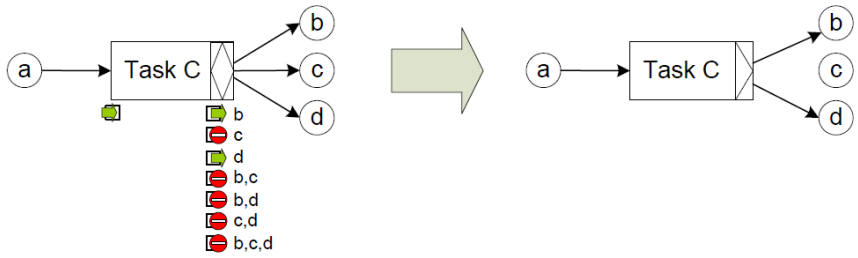


Figure 3.1: An example of configuring a C-YAWL connector: from an OR-split to an XOR-split [4]

and leafs. Because of its no cycles syntactic restrictions, this structure guarantee soundness by construction of the configurable process and derived variants. However, it may be not applicable in case of complex processes (e.g., processes with cycles)

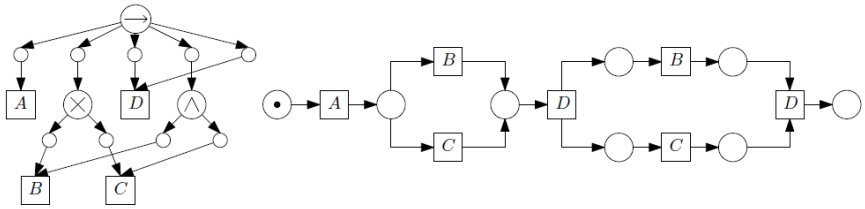


Figure 3.2: An example of a CoSeNet with the corresponding Petri net [5]

Configurable Nodes Rosemann et al. in [1, 34] introduce Configurable EPC (C-EPC) notation that is an extension of the EPC language. Configuration is achieved by restricting the behavior of the C-EPC in order to obtain an EPC process model. This is done thanks to configurable nodes and by assigning to each node one configuration choice or alternative. EPC notation has three main control-flow elements: event, function and gateway. Only active elements, i.e. functions and connectors, may be configurable in C-EPC notation. In addition, C-EPC introduce two new constructs: configuration requirements and guidelines. The role of the configuration requirements and guidelines is to assist the users in choosing the configuration choices. We can differentiate configuration requirements and guidelines by being hard and soft constraints respectively. But both constraints are expressed using logical predicates of the form if-then rules. An example of a C-EPC process model is illustrated in Figure 3.3. Configurable nodes are marked with a thicker border. There are three configurable functions, one configurable XOR connector, one configuration requirement and one configuration guideline.

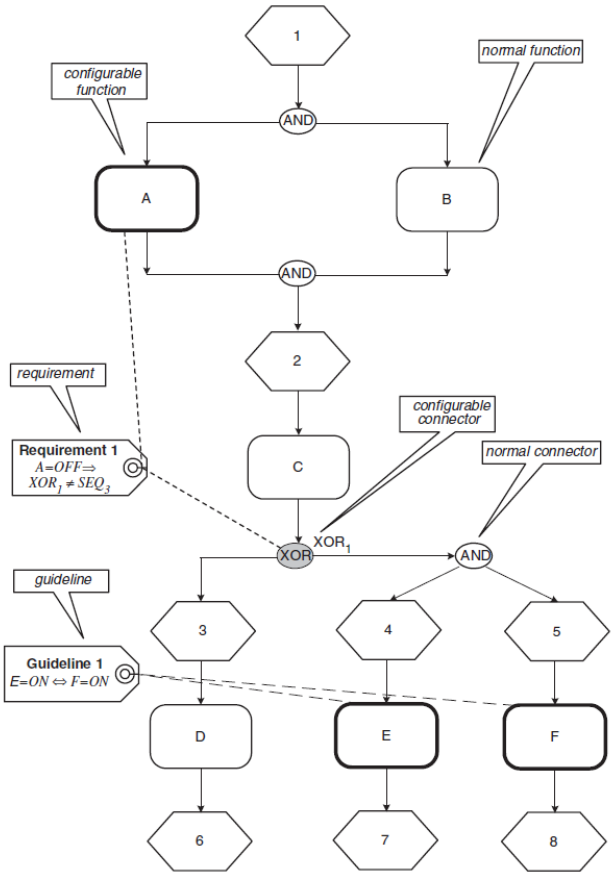


Figure 3.3: An example of a C-EPC process model [1]

Annotations The PESOA (Process Family Engineering in Service-Oriented Applications) project [100] defines so-called variant-rich process models as process models extended with stereotype annotations to identify variation points. For instance, the variable elements of a process model are marked as variation points using the stereotype $<< VarPoint >>$. These stereotypes are applied to both BPMN models and UML Activity Diagrams. In case of BPMN, annotations can only be applied to activities (as well as their connected data objects), While operators' variability is not considered. In addition, variant correctness issues were not considered as well.

Fragment change: Provop approach In [6], authors propose the *provop* (PROcess Variant by OPtions) approach to model process variants by applying a set of defined operations (i.e., a group of change operations INSERT/DELETE/MOVE fragment, MODIFY attribute) to a reference process model, namely a base process. These change operations (e.g. those that usually occur together) are grouped into

Options. Authors also define option constraints that are similar to the configuration guidelines and requirements proposed by [1]. Annotated adjustment points are the points where, by means of the later operations, either a restriction or an extension of the behavior of the base model can be made.

In Figure 3.4, an example of a base process model having two adjustment points is provided. Then, three possible variants are derived using specific three options. For instance, a behavior restriction of the original process is obtained in the first variant by applying DELETE operation, while a behavior extension is obtained in the second variant by applying the operation INSSERT process fragment.

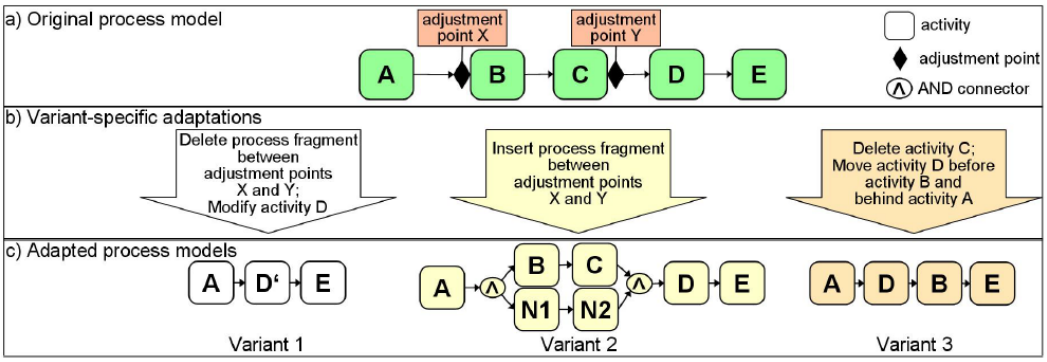


Figure 3.4: The provop approach of variant modeling [6]

Templates and Rules Akhil Kumar and Wen Yao [35] introduce the use of configuration rules in order to configure a generic process model, the so-called process template. A template is represented by a block-structured tree-like process model. The rules may restrict or extend the template's behavior via specific change operations (e.g., insert or delete a task). Authors provide a formal representation of these business rules and an individualization algorithm that derives the process variants, given the process template and the configuration rules. One advantage of this template structure, is that it is relatively simple and can facilitate process variant configuration. Also, it is proven that change operations cannot cause any syntactical nor behavioral issues in this structure. However, similar to [100] approach, the defined configuration rules can only be applied on workflow tasks, but not to connectors.

To represent our configurable process models, we choose to follow the approaches that use the *configurable nodes* for two reasons. Firstly, these approaches have a basic solid theoretical work on reference configurable process modeling in [1]. Secondly, in our work, we aim at starting from a most generic process that hold all the possible behavior and, while configuring the model, the designer do not need to add any information content to that original business process. Instead, the designer restrict the behavior of the model. Moreover, the modeling language BPMN that we use as a

starting point to each contribution was extended with configurable nodes to facilitate configuration.

3.3.2 Domain and Guidance Support

Questionnaires la Rosa et al. [7] propose a novel approach that provides guidance in configuring process models by means of a set of structured questionnaires. This configuration approach helps stockholders with no knowledge in the process modeling field by answering a set of questions expressed in natural language. Questions are defined by domain experts based on domain constraints, and answered by designers. The answers are then analyzed and used to configure one or more configuration points in a C-EPC model. This approach is supported by the Synergia[1] and Apromore[2] tools. Similarly, in [103], authors applied the questionnaire approach to derive executable YAWL models from C-YAWLs. Although this approach offers abstraction and guidance for the process models configuration, the consideration of correctness criteria is missing.

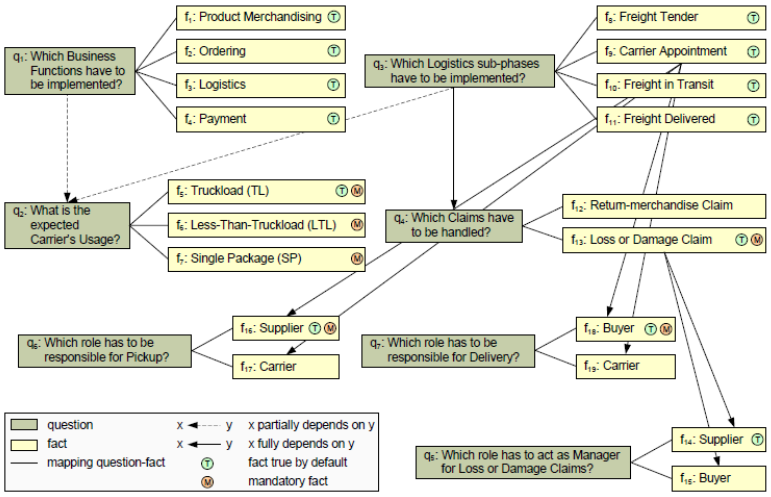


Figure 3.5: An example of a questionnaire model [7]

Feature Models Inspired from configuration management in Software Product Line Engineering (SPLE) [104], authors, in [8, 36, 105], support process models variability based on feature models [106]. In [105], a process model is considered as family of services, which are related via data dependencies. Each service can have any types of variation points that are represented with one or several feature models.

[1] `www.processconfiguration.com`

[2] `www.apromore.org`

Each feature represents a property of a specific domain and refers to one configuration alternative. Hence, a configuration is obtained by selecting the desired features. Families of workflow can be defined as compositions of feature models using proposed composition operators. However, only the configuration of an activity's inputs and outputs is considered. Hence, the control flow cannot be customized. Moreover, the approach does not provide guidance in configuring feature models. Also, the SPLE based approaches require that the domain analysts should be familiar with the feature models. An example of a feature model and a possible configuration is provided in Figure 3.6.

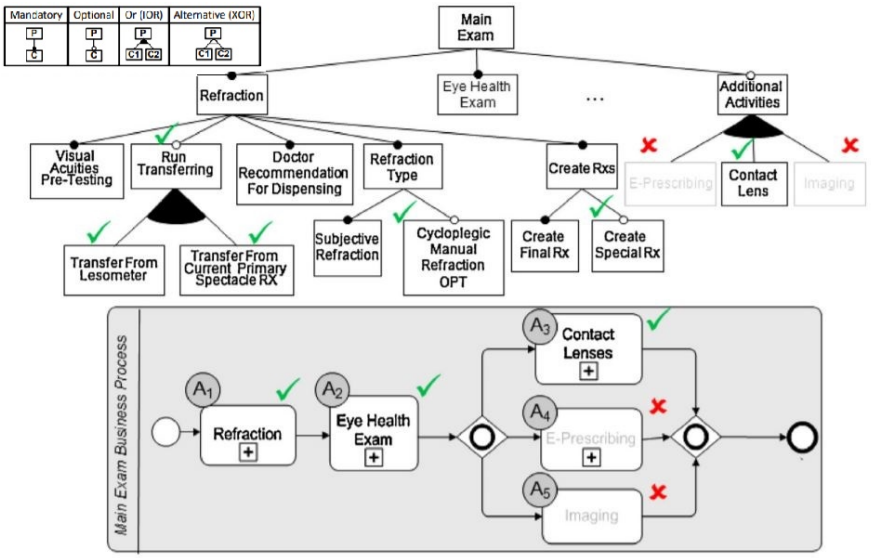


Figure 3.6: An example of a configured feature model generated after configuration step [8]

Although the above-mentioned two approaches consider relevant domain constraints, they require considerable manual effort and many steps to perform by a domain expert in order to create the domain model.

Configuration Guidelines Extracted from Process Repositories Assy et al. [9,107] attempted to address this issue by the use of configuration guidelines for assisting analysts in BPMN configuration. They propose to extract these configuration guidelines from business process repositories of existing configurations. This allows to learn from the users' experience in process configuration. They define a tree-like structure, so-called *configuration guidance model*, to represent the configuration options and the inclusion and exclusion dependencies between them in order to define an order of configuration choices. In Figure 3.7, an example of a configuration guidance model is extracted. It gives an hierarchical ordering of the configurable elements of

a process model based on the tree structure: the parent element is configured before the child element. However, this approach do not consider any correctness criterion. In fact, after applying the guidelines, it does not guarantee that the derived variants do not exhibit structural or behavioral issues.

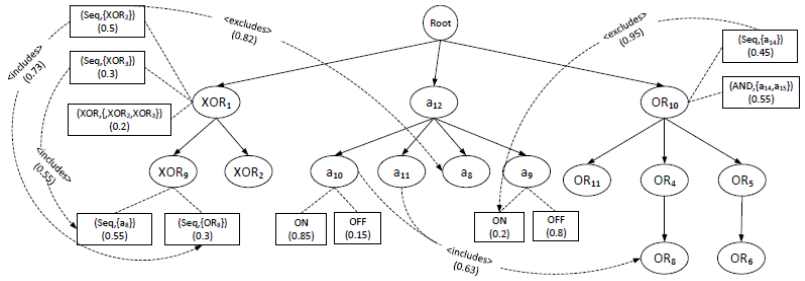


Figure 3.7: An example of a configuration guidance model [9]

3.3.3 Correctness Support

While configuration approaches allow for an easy adaptation of variants to individual needs, domain approaches attempted to provide further adaptation by considering domain constraints and guidelines. However, the correctness of the obtained variants is not necessarily provided. Since the inter-dependencies between configuration decisions may be very complex and of all kinds, the designers may easily be mistaken in their configuration choices which may result in critical errors. To address this problem, a number of approaches have attempted to reach correct process configuration either syntactically or behaviorally. Syntactical correctness is related to the derived process structure, for example, by avoiding disconnected nodes. Behavioral correctness is related to ensuring correct behavior of the variants, for example, by avoiding execution anomalies such as deadlocks and livelocks. Traditionally, the behavioral correctness related to process configuration can be handled by verifying every single possible configuration using existing work on verification of business processes and workflows (cf. Section 3.2). This raised the well known state space explosion problem. Also these methods are too time-consuming and may be a labor-intensive work especially in case of large and complex process models with an exponential number of possible configurations.

In [37, 38], Petri net was used to formalize and verify correctness and soundness properties of Configurable EPC (C-EPC) processes. They derive propositional logic constraints that guarantee the behavioral correctness of the configured model. However in these approaches, authors achieve correctness by checking constraints at each configuration step. Also, authors impose that the C-EPC process model should be syntactically correct. In [39], they focused on the behavioral correctness of the configured model and moved the checking up to design time. Thus, propose to find all fea-

sible configurations prior to execution. This approach is inspired from the "operating guidelines" used for partner synthesis [108]. Practically, the used synthesis algorithm use Open Petri net as a intermediate formalism and to represent the so-called *configuration interface*. This interface is constructed by adding, for each configurable task, a number of places, transitions and arcs to the original model. This not only adds complexity to the model, but also results in allowing potentially unreal behaviors. Based on this interface, an automaton is constructed to represent the configuration guidelines. Each path on this automaton corresponds to a feasible configuration. However, the considered correctness criterion in this work is the weak termination, which means that when deriving a variant holding unreachable (i.e. dead) transitions, the model will still be considered behaviorally correct. Finally, this technique was applied on C-YAWL and the configuration is built by hiding and blocking transitions.

Moreover, some approaches support correctness because of the imposed constraints on the structure of the configurable model. For instance, CoSeNets [5] achieve correctness because of their syntactic restrictions to avoid cycles. The same apply for approaches [8, 35, 36, 105] that are based on block-structures processes. However, this type of processes suffers from restrictions in terms of structure and could be not applicable in case of complex processes.

Regarding the *Provop* approach, authors in [10] discussed five steps, depicted in Figure 3.8, for ensuring soundness of derived variant models. This method does not require the base model to be sound. However, although they propose to reduce the number of variants to check using context information, they still suffer from the exponential number of the possible options permutations. So, this approach is not feasible in large processes and runs into the state space problem. Also, they propose to check soundness a posteriori. Hence, an incorrect variant is discarded at the end without any guidance to avoid it.

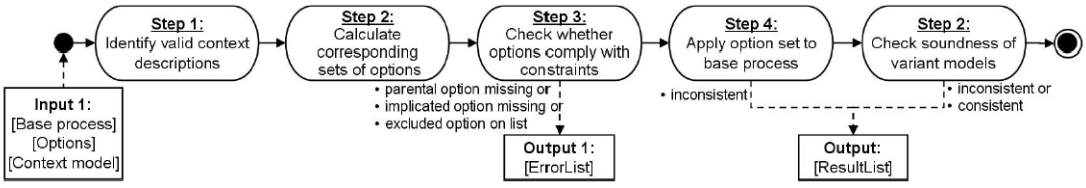


Figure 3.8: The five-steps approach for guaranteeing Provop soundness [10]

3.3.4 Synthesis

Table 3.1 provides a comparative overview of the presented configuration approaches in light of our evaluation criteria (inspired from [98]): (1) *Process Modeling Language*, (2) *Correctness Support*, (3) *Domain Support* (i.e., compliance with *domain*-specific configuration constraints), (4) *Guidance Support* (i.e., providing guidance to users when taking configuration decisions), and (5) *Formal Specification* (i.e., providing

rigorous description in terms of mathematical notations). We further decompose the *Correctness Support* column into two sub-criteria: (i) *structural* , and (ii) *behavioral* correctness. Note that, we used "+" to express that the corresponding criteria is fulfilled by the corresponding approach, "-" if it is not fulfilled, and "±" if it is partially fulfilled.

First, several approaches have been proposed to model variability and to facilitate the design of the configurable process models [4, 6, 9, 29–31, 34, 36, 55, 99–102]. The variability in configurable process models is handled by restricting or/and extension. We take the standpoint of the configuration by restriction which corresponds to preserving the desired behavior of the model, while removing the undesired one. We specifically use the *configurable nodes* approach since, as stated in Section 2.2, we picked the (C-)BPMN language for modeling (configurable) process models. This notation have two types of configurable nodes: activities or connectors. A configurable activity may be kept or excluded from the variant. The behavior of a configurable connector may be restricted either by changing its type or by restricting its outgoing or incoming sequence of nodes.

Then, on the one hand, some approaches proposed to guide the configuration or/and to support domain-based constraints [7–9, 36, 103, 105, 107]. Many of them require considerable *manual steps* from a domain expert to create the variant model. This may be an tedious task in case of large processes. While these approaches have not considered any correctness criteria, others attempted to verify and ensure the design of correct variant. We note that basic syntactical constraints were considered when defining configurable process model [1, 34]. Some approaches achieved correctness thanks to their restrictive block-structured processes [5, 8, 35, 36, 105]. Most importantly, other work attempted to ensure the behavioral correctness of the derived variants [4, 6, 10, 29, 30, 37–39]. Whereas, most of them still suffer form the exponential number of possible configurations. The most notably work addressing this issue is [39], however, the considered correctness criterion is the weak termination that permits unreachable nodes. Finally, even if the above proposals try to achieve configuration correctness, they nevertheless often lack the necessary guidance to become adaptable to a given domain and do not support the BPMN notation.

In our work, we propose to sustain the two aforementioned groups of approaches in order to offer the needed support for domain, guidance, and correctness. We propose to *guide and assist* the designer in deriving correct variants. Hence, we target to apply formal methods to verify the process configuration while respecting a set of constraints dealing with *correctness* and *domain*. For that aim, we propose two complementary approaches. The fist contribution in Chapter 4 deals with deriving structurally correct and domain-compliant variants. We also verify erroneous patterns that may affect the behavior of the process (which explains the ± symbol in the column behavioral correctness of Table 3.1). We use Event-B tools to perform an incremental verification by checking these constraints at each intermediate step of the configuration procedure. The second contribution in Chapter 5 addresses the

behavior verification of process configuration in order to derive deadlock-free variants. This work addresses the problem of the configurable models state-space explosion by using an abstraction of the reachability graph, namely the SOG.

Approaches \ Criteria	Process Modeling Language	Correctness Support		Domain Support	Guidance Support	Formal Specification
		structural	behavioral			
[4, 29, 30]	C-YAWL	+	±	-	-	+
[5]	CoSeNets	+	±	-	-	±
[1, 34]	C-EPC	+	-	+	-	+
[37, 38]	C-EPC	+	+	-	-	+
[39]	C-EPC	+	+	+	-	+
[100]	annotated BPMN	-	-	-	-	±
[6, 10]	any	+	+	-	-	±
[35]	Block-structured	+	+	-	±	±
[7, 103]	C-EPC/ C-YAWL	±	-	+	+	±
[8, 36, 105]	Block-structured	+	+	+	-	+
[9, 107]	C-BPMN	-	-	+	+	-
Chapter 4	C-BPMN	+	±	+	+	+
Chapter 5	C-BPMN	+	+	+	+	+

Table 3.1: Evaluation of related configuration approaches

3.4 Formalization and Verification of the Resource Allocation Behavior in Business Processes

Having discussed in the previous sections some aspects about the control flow perspective of business processes, now we focus on the resource perspective that concerns the management of human as well as non-human resources during the process lifecycle. Motivated by the need to achieving optimal process execution, efficient resource allocation in BPs is being increasingly explored. Actually, few works are handling the resource perspective in BPM and they mainly focused on human resources behavior and allocation (cf. Section 3.4.1). Whereas, the usage of Cloud resources to allow activities execution of such processes is becoming very challenging (cf. Section 3.4.2).

3.4.1 Resource Allocation in Business Processes

In [40], series of Workflow Resource Patterns were proposed to capture the various ways in which human resources are represented and executed in workfows. Patterns allow to assess the expressiveness of process models in a language-independent manner. The *creation patterns* are of specific interest to our work since they are related to resource selection and specify different ways of resource allocation to activities. For instance, *direct-allocation* (Pattern **R-DA**) and *capability-based allocation* (Pattern **R-CBA**) are two examples of allocation patterns. The direct-allocation provides the ability to specify at design time the resources that will execute a process activity. While the capability-based allocation provides the ability to allocate a resource to

an activity based on its specific capabilities compared to the corresponding activity specific requirements.

Stroppi et al. [42] developed an extension to BPMN models that enrich process models with human resources definition. This work also provides an extension of the BPMN 2.0 metamodel and comply with the assignment patterns defined by the workflow resource patterns. Afterwards, same authors [43] propose to identify resource perspective aspects and requirements in executable workflow specifications (in Workflow Management Systems (WfMSs)), and provide a supporting tool implementation.

Also, based on these resource patterns, few works focused on the human resource behavior management [11, 44]. Lately, Cabanillas et al. propose a complete graphical notation for assigning human resources to business process activities, the so-called RALph (Resource Assignment Language Graph). Formal semantics of this notation is obtained through its mapping to Resource Assignment Language (RAL) [45], a textual language (modeling language independent) for defining resource assignments in business processes. In the RALph notation, four types of *resource entities* are proposed (as depicted by Figure 3.9): *persons*, *roles*, *positions*, and *organizational units*. *Capability entities* are persons having specific capabilities. Resource assignments are expressed using connectors (i.e. same connectors linking activities in control flow). Resources dependencies as well as resource-activity dependencies were considered.

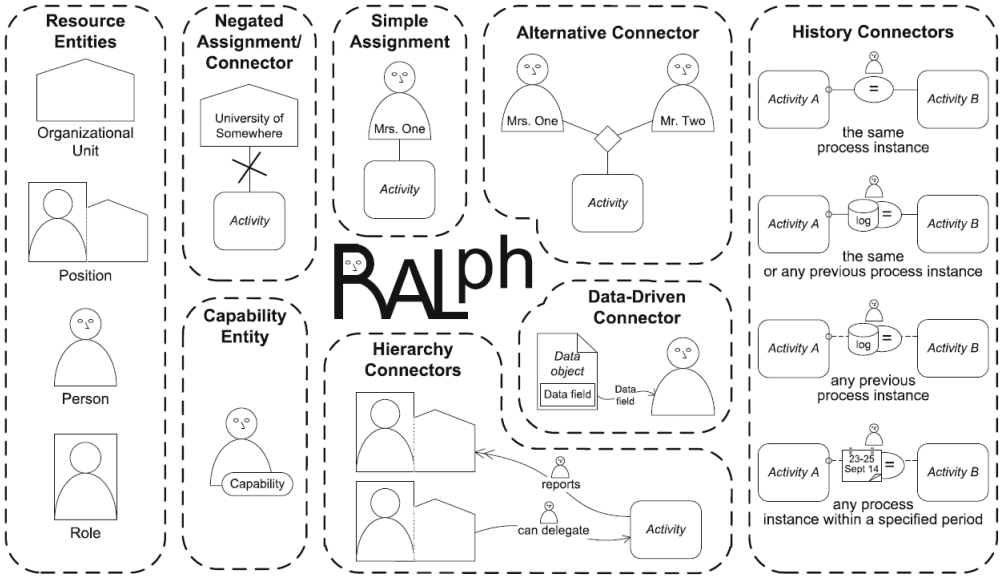


Figure 3.9: The RALph approach for graphic resource assignments [11]

In [109], a formal approach was developed for deriving an optimal work schedule while considering dependencies and resource conflicts between work items. Authors

used Answer Set Programming (ASP) for formal specification while taking into account human and non-human local resources.

As we can notice, the main focus in the literature regarding resource perspective is on human resources and their representation. However, they do not consider neither the representation of Cloud resources in BPs nor the verification of their allocation behavior.

3.4.2 Cloud Resource Allocation in Business Processes

Nowadays, there is a clear need from organizations to benefit from the Cloud Computing technologies in order to optimize their business processes. In such a multi-tenant environment, dynamically scalable and often virtualized resources on demand are offered. The benefits of Business Process Management BPM in a Cloud environment have been highlighted by different authors [110, 111]. In this section, we review some existing approaches that focused on the cloud resources allocation to business processes.

Existing researches usually focused on aspects like orchestration, scheduling and optimization. For instance, approaches in [112–116] were interested in workflow scheduling strategies and resource allocation algorithms that allow to use Cloud resources in an optimal way. The elasticity property of processes is considered either at an IaaS [112] or a Paas [113, 114] Cloud.

Moreover, some early research efforts adressed the issue of resource allocation optimization in business processes execution. For example, trying to minimize cost and to improve the execution performances, [117] propose a mechanism for resource allocation decision modeling based on Reinforcement Learning. The technique suggested in [118] predicts the execution path in order to estimate Cloud resource requirements prior to execution using process model metrics. Then, based on this prediction as well as pricing strategies, it allows an efficient allocation of the cloud resources while optimizing the leasing cost. Byun et al [119] propose an algorithm, named PBTS (Partitioned Balanced Time Scheduling), that aims to find the minimum number of computing resources for workflow execution under a user-specified deadline.

However, only non-functional behavior of the resource allocation is most often considered, in terms of response time and financial cost. The functional behavior correctness of resource allocation with respect to Cloud properties is still missing

3.4.3 Verification of Resource Allocation Behavior

As we have explained in Section 3.2, existing approaches tackling formal verification of business business processes have mainly considered the control flow perspective and transactional behavior of services, but have neglected the resource perspective. Few attempts and research studies based on formal methods are currently addressing the formal modeling and analysis of Cloud properties. For instance, authors, in [120], adopted a temporal logic called CLTLt(D) (Timed Constraint LTL) to formalize the

elastic behavior of Cloud-based systems. Authors formalize properties related to horizontal elasticity, resource management and quality of service QoS. Then, they propose to check whether these properties hold or not during execution of a cloud-based system. Authors in [121] used bigraphical reactive systems (BRS) for specifying both structural and behavioral aspects of elastic cloud-based systems. Then, they used Maude's model-checking invariants technique to simulate and verify the elasticity property by ensuring that the Cloud system scale up/down when needed. Gambi et al. [122] adopted the state transition systems to formalize Cloud-based systems while taking into account elasticity properties. This formalization is then used to automatically generate load test cases focusing on plastic behavior of elastic systems. Plasticity is verified by ensuring that, for each scaling up, there should correspond a scaling down. While, the aforementioned approaches considers the elasticity properties of resources or services in Cloud-based systems, they do not take into account the process perspective.

Authors in [123] defined a formal model based on Petri nets for horizontally elastic service-based business processes (SBPs) in the Cloud. Authors proposed to obtain an elastic SBP by composing its WF-net with each Petri net-based elastic controller of its services. The elasticity properties were formally characterized using Computational Tree Logic (CTL). Amziani et al. [124] defined a formal framework for the description and evaluation of service-based business processes elasticity and their strategies. Thereafter, they proposed to assess the correctness of the defined duplication and the consolidation mechanisms (i.e., the horizontal elasticity operators) by guaranteeing that the semantics of the SBP is preserved. An extension of this approach was proposed in [125] in order to support stateful SBP. In a similar vein, authors in [126], suggested an Event-B based approach for formal specification of correct elastic component-based applications. Horizontal elasticity mechanisms are expressed in terms of events. The verification of the preservation of the functional and non-functional properties is done using proof obligations and animation. Nevertheless, in the above mentioned approaches, only the horizontal elasticity that refers to replicating or removing instances of cloud services is taken into account.

Very recently, some approaches have attempted to formalize Cloud resource allocation problem in BPs, but considered other Cloud constraints rather than the elasticity and the shareability particularilly. For example, in [127], authors formalized temporal constraints for cloud resources allocation in cloud-based business processes. And in [128] authors formalized cloud resource allocation in the context of social business processes.

3.4.4 Synthesis

Table 3.2 summarizes the evaluation of the approaches presented above and align them with important criteria in the context of our work. Hence, we consider: (1) Process perspective, (2) Resource perspective, (3) Human resource, (4) Cloud resource, and

(5) the verified Cloud properties. As previously mentioned, we use "+" if the criteria is fulfilled and "-" otherwise.

Approaches \ Criteria	Process perspective	Resource perspective	Human Resource	Cloud Resource	Verified Cloud properties
[11, 40, 42–45]	+	+	+	-	-
[112–119, 127, 128]	+	+	-	+	-
[106, 120, 121]	-	-	-	+	Horizontal elasticity
[123–126]	+	-	-	+	Horizontal elasticity

Table 3.2: Evaluation of related resource-based approaches

We can observe that process perspective, resource perspective and cloud properties verification are only partially covered or not at all. The resource perspective is actually well addressed, however only focusing on human resources (line 1). Mainly, an extension of business process models with the representation and the definition of human behavior is proposed [11, 40, 42–45]. However, less attention has been paid to Cloud resource allocation in BPM field. In our work, we aim at supporting the Cloud resource allocation representation and verification in BP models.

Indeed, researches on Cloud resource management can be classified into two groupes: in the BPM context, or in the Cloud-based systems context. On the one hand, in the first group (line 2), approaches integrated Cloud resources into process models, however they were only interested either in allocation aspects such as orchestration, scheduling and optimization; or in only temporal and social constraints formalization [112–119, 127, 128]. They did not provide formal specification or representation for the Cloud resource elasticity behavior and no verification techniques are used. While in our work, we propose to formally specify resource allocation in BPs that integrates Cloud aspects. We also seek for checking Cloud resource properties and constraints.

On the other hand, in the second group (line 3), approaches basically focused on the verification of horizontal elasticity aspects of resources in Cloud based systems rather then processes [106,120,121]. Moreover, attempts to verify elasticity of process models in a Cloud context (line 4) were limited to horizontal elasticity at the service level without considering the vertical elasticity aspect [123–126]. By contrast, our work considers particularly the vertical elasticity of Cloud resources. In fact, we aim to check behavior of the resource allocation correctness with respect to two Cloud resource properties: (1) *the vertical elasticity* that consists of the ability to dynamically adjust process resources by scaling their capabilities up/down when needed (according to the process workload), and (2) *the shareability* of resources which consists of the ability to use resource by multiples activities.

In this work, we propose to use the Event-B formal method. Compared to the other formal languages, the strong point of Event-B is that it supports the incremental design and modeling of the process using the step-wise refinement concept. Instead of defining the whole system properties and functionalities at the same time (what

other approaches often do), Event-B allows to gradually introduce the different parts of the system starting from an abstract model to a more concrete one. Thus, at each specific abstraction level, a set of properties and rules is introduced. Then, these properties are maintained at each refined level. The consistency between the different refinement levels is obtained by formal proofs.

3.5 Conclusion

This chapter serves to present different approaches that are relevant to our work. First we reviewed the existing work on the verification of business process models. Then, we classified the existing approaches related to Business process configuration into three major categories: configuration-based approaches, domain and guidance-based approaches, and the ones dealing with ensuring variants correctness. We briefly introduced them and we provided their principles. Regarding the existing approaches related to the resource allocation behavior in business processes, we presented three groups: we started by the ones dealing with the resources perspective in BPM field, then we looked at specifically the Cloud resources integration in BPs, and finally we tackled the verification issue in this perspective.

Chapter 4

Assisting Correct Process Variant Design with Formal Guidance

Contents

4.1 Introduction 62
4.2 Motivating Example 63
4.3 Approach Overview 66
4.4 Formal Specification of a Business Process Model 67
4.4.1 Business Process Model Graph 68
4.4.2 Business Process Modeling using Event-B 68
4.5 Formal Specification of a Configurable Process Model 69
4.6 Formal Specification of Configuration Steps 72
4.6.1 Activity Configuration 72
4.6.2 Connector Configuration 73
4.7 Introduction of the Configuration Guidelines into the Model . 76
4.8 Verification and Validation 78
4.8.1 Verification using Proofs 78
4.8.2 Validation by Animation 78
4.8.3 Case Study 79
4.8.3.1 Objective 80
4.8.3.2 Design, Data Collection and Execution 80
4.8.3.3 Results Analysis and Findings 81
4.8.3.4 Threats to Validity 81
4.8.4 ATL Model Transformation: BPMN to Event-B 81
4.9 Conclusion and Discussion 84

4.1 Introduction

Depending on specific needs of an organization, a *configurable process model* need to be adapted and configured. As we explained in chapters 1 and 3, manually applying configuration choices is far from trivial, especially in case of large configurable process models involving complex inter-dependencies between their elements configuration choices. Such inter-dependencies may be difficult to unravel without guidance. In addition, manually considering business domain constraints may be also a tedious and error-prone task. In the light of these difficulties, the business analyst may be easily mistaken in selecting configuration choices leading to incorrect derived variants in terms of structure, behavior or domain. This chapter addresses the research questions: *RQ1: How to identify configuration choices that satisfy designers and clients requirements?* and *RQ2: How to assist the designer in selecting the correct configuration choices?*

To answer these questions, the contribution of this chapter is to provide guidance and assistance in the process configuration task with adequate possible choices at each step. Since formal methods have proven their benefits in performing mathematical analysis allowing to rigorously and precisely reason about a system. We propose a formal specification describing the process configuration task using Event-B. Moreover, we formally define different constraints and properties that each configuration step should preserve. These constraints and properties are related to (i) configuration, (ii) structural correctness; and (iii) domain-based configuration guidelines. Moreover, at each configuration step, our specification should consider previously selected choices. Hence, the approach targets to achieve the following objectives:

- *Objective 1*: analyze and check the correctness of a configurable process model;
- *Objective 2*: derive correct variants with respect to different constraints at each configuration step;
- *Objective 3*: integrate *Configuration guidelines* [1] (i.e., if-then rules based on specific domain or context) in our formal model to ensure that the obtained variants comply with their domain constraints.

Once obtained, our formal specification is verified using Event-B provers, e.g. AtelierB provers, that produce proof obligations allowing to ensure that these constraints are preserved by the configuration steps. Also, using animation, with ProB, we exhibit a significant witnesses that different scenarios can be played. Thus, we ensure, before the proof phase that can be long and complex, that constraints and properties are evaluated at each step and hence the resulting variant preserves them as well. So, Event-B tools allows to perform an incremental verification by checking different constraints at each intermediate step of the configuration procedure. Thus, we obtain a correct-by-construction model that assists the designers in their configuration task.

In the following, we start by giving an example to motivate our work in Section 4.2. Then, we present an overall overview of our approach in Section 4.3. We illustrate our approach first step that consists in the formalization of a regular process model in Section 4.4. Section 4.5 formally introduces configurable elements as well as corresponding constraints. Afterwards, configuration steps that represent our configuration model for extracting correct variants are formalized in Section 4.6. Finally, we formalize configuration guidelines taking into account domain constraints in Section 4.7. The verification and validation of this approach using the RODIN tool, as well as its evaluation using experiments on a case study are depicted in Section 4.8.

The work in this chapter was partially published in conference proceedings [129, 130].

4.2 Motivating Example

Let us introduce our motivating example used through the present chapter. A configurable process model for the hotel reservation and car rental agency is captured by Figure 4.1. The configurable process is modeled using the Configurable BPMN (C-BPMN). Without limiting the generality of our work, we use C-BPMN as notation for configurable process modeling, since BPMN is considered as the internationally recognized industry standard notation for business process description. It is also worth noting that our work can be easily adapted to other *graph-based* business process modeling notations such as C-EPC.

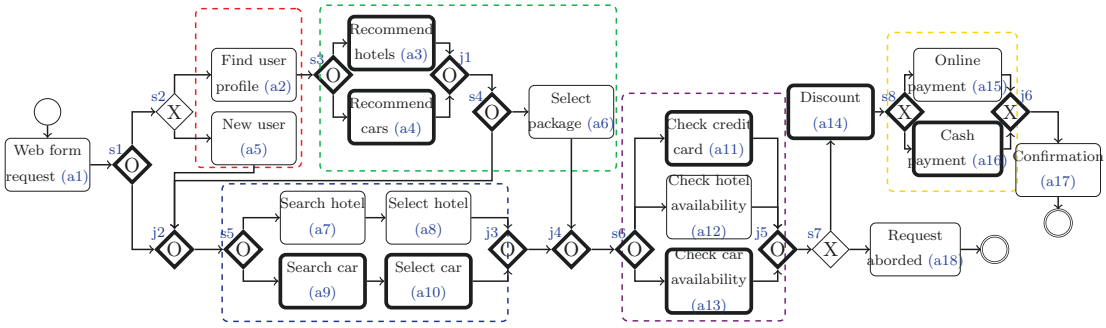


Figure 4.1: A configurable process model of a hotel and car reservation agency

In this process, the customer first submits a request through a web form (*a1*). Next, five main functionalities are proposed: (1) the user profile search or creation: the process fragment in the red dashed rectangle, (2) the recommendation of hotels or/and cars: the process fragment in the green dashed rectangle; (3) hotels or/and cars searching and selection: the process fragment in the blue dashed rectangle; (4) checking phase: the process fragment in the violet dashed rectangle; (5) discount offer: the process fragment in the black dashed rectangle; and, (6) payment: the process

fragment in the yellow dashed rectangle. Finally, an email of confirmation is sent to the customer using the activity *a17*.

As we have explained in Chapter 2, the C-BPMN notation includes two configurable elements: *activities* and *connectors*. In this example, 20 configurable elements (12 connectors and 8 activities) are highlighted with a thicker border. For instance, activities *a1* and *a18* are non-configurable, so they should be included in every configured variant. Whereas, the activity *a9* and the connector *s1* may vary from one process to another, as they are configurable. This configurable process will be shared between different users from different branches. Moreover, it will be configured according to their preferences and regulations.

In Figure 4.2, we give an example of a process variant of this configurable process modeled with BPMN 2.0 and used by a hotel reservation branch. Here, we suppose that this branch does not need the recommendation functionality for cars (i.e., activity *a4* is removed). But, it needs the execution of the search and selection for hotels only (i.e., the entire branch starting from *a7* is removed). Additionally, it needs a simultaneous execution of the hotel availability checking and the credit card checking tasks (i.e. this refers to modifying the connectors *s6* and *j7* types from OR to AND while removing activity *a10*). Finally the branch choose to adopt online payments only (i.e., activity *a16* is removed). As a result of such configuration choices, we obtain the individualized process of Figure 4.2.

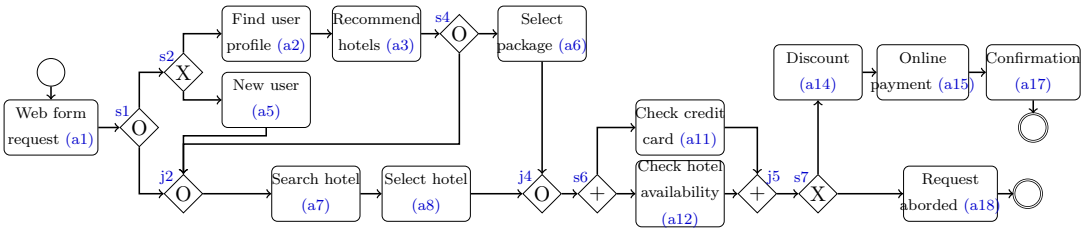


Figure 4.2: A hotel reservation variant of the configurable process in Figure 4.1

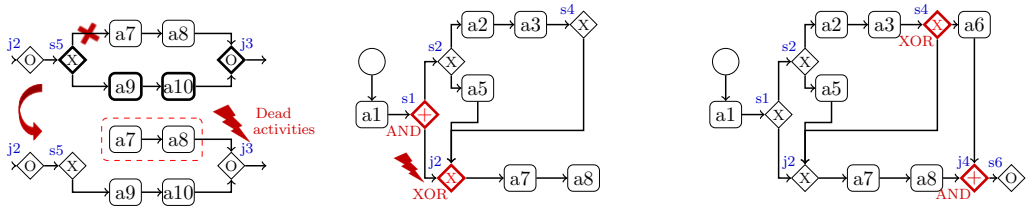
Configuration correctness checking In Figure 4.3a, *s5* has been configured to a sequence starting from *a9* (the edge between *s5* and *a7* disappears). Thus, the produced process is not sound, since activities *a7* and *a8* become unreachable from the initial event: they are dead as they can never be executed. In this chapter, we aim at preventing such configurations by formally ensuring that every connector configuration involving outgoing or incoming branches restriction is implicitly followed by a transformation phase allowing to remove the isolated activities from the resulting process. An isolated node is detected either if it is unreachable from the initial event, or it is not on a path leading to a final event.

Besides checking this structural property of the configurable process model, we aim also to prevent problems that may affect the soundness of the derived process

variants [95]. In the following, we illustrate two error patterns [95] that would happen during the process configuration resulting from mismatches between splits and joins: *deadlock* and *lack of synchronization*.

- In Figure 4.3b, the join operator *j2* has been configured to an XOR while the connector *s1* had been already configured to an AND-split. The two outgoing branches from the AND-split will be activated, however, the XOR-join needs the completion of exactly one of its incoming branches. This leads to multiple terminations of the process referred to as lack of synchronization problem.
- In Figure 4.3c, the connector *s4* has been configured to an XOR-split and the corresponding join *j4* to an AND-join. This implies a deadlock, as only one branch is activated after the XOR-split, whereas the AND-join needs the completion of all its incoming branches.

The prevention of these erroneous situations will be discussed in detail in Section 4.5.



(a) A configuration leading to *dead activities*

(b) A configuration leading to *lack of synchronization*

(c) A configuration leading to *deadlock*

Figure 4.3: Examples of configuration mistakes

Configuration guidelines To comply with specific domain business needs, the process analyst needs further guidelines to derive specific variants. *Configuration guidelines* provide recommendations and proposed best practices for a specific domain [1]. An example of such guidelines satisfied by the variant of Figure 4.2 is: *"if the hotel recommendation functionality is included (i.e. a3) in the derived variant, then the hotel searching functionality (i.e. a7) should be also included."* These guidelines are expressed in the form of logical *If-Then*-rules where the *if* and *then* parts contain configurations of different configurable elements. Such rules considerably increase the difficulty of manually applying configuration options. Hence, we will discuss the integration of these constraints into our configuration approach in Section 4.7.

Now, we assume that, in Figure 4.4, a process analyst is designing a "hotel and car reservation" process variant starting from the configurable process model. Here, when configuring a split connector, e.g. *s1*, or a join connector, e.g. *j4*, respectively; the designer should wonder about *(point (1)) the configured type*, *the number of output* or

input branches respectively; or *(point (2))* the *already configured connectors* in order to take them into account. Regarding configurable activities, e.g. *a3*, designer should choose whether to *include* or to *exclude* them from the resulting process variant *(point (3))*. The same principle applies for the rest of configurable elements. To answer these wonderings while configuring correctly, he/she should consider, at the same time, constraints related to configuration (e.g., an AND connector should not be configured to a sequence), as well as structure and domain constraints discussed above. Configuration steps with respect to all these constraints will be formally specified in Section 4.6. Note that, our contribution consists not only in proving the correctness of the process configuration steps but also in guiding analyst choices with respect to these constraints using the ProB animator (cf. Section 4.8.2).

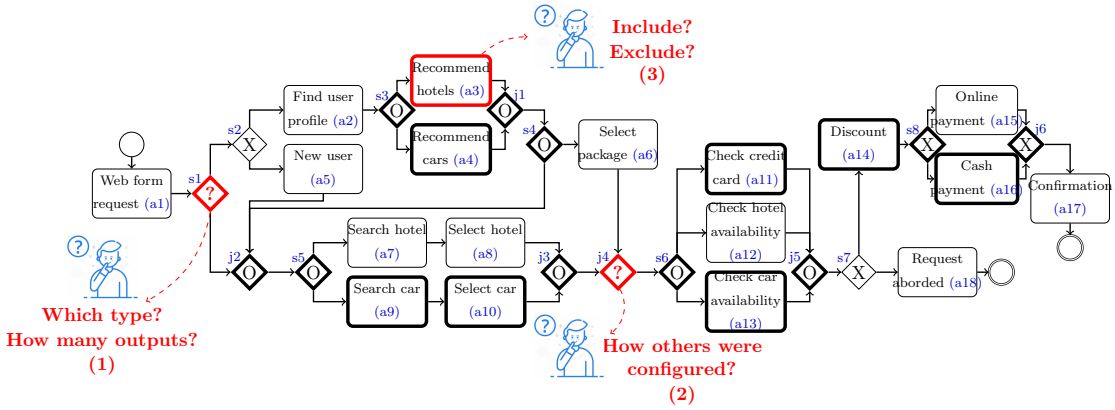


Figure 4.4: A hotel and car reservation process variant designing

4.3 Approach Overview

This section gives an overview of our contribution detailed in the next sections. Figure 4.5 depicts the configuration model that we propose in this chapter allowing to assist the designer in his/her decisions leading to correct process variants. Basically, using Event-B as a formal method, we defined a model of two abstraction levels: the first level introduces our model for process model configuration allowing to preserve correctness (machine M0). In this machine, we formally specify a process model (cf. Section 4.4) as well as configurable elements and the corresponding constraints (cf. Section 4.5). Configuration steps and their correctness are ensured by events and invariants (cf. Section 4.6). Next, configuration guidelines are formally integrated to our model in the second abstraction level as a model refinement of the first level (machine M1, cf. Section 4.7). Event-B defines proof obligations to guarantee that the invariants are preserved by all events (see Section 4.8.1).

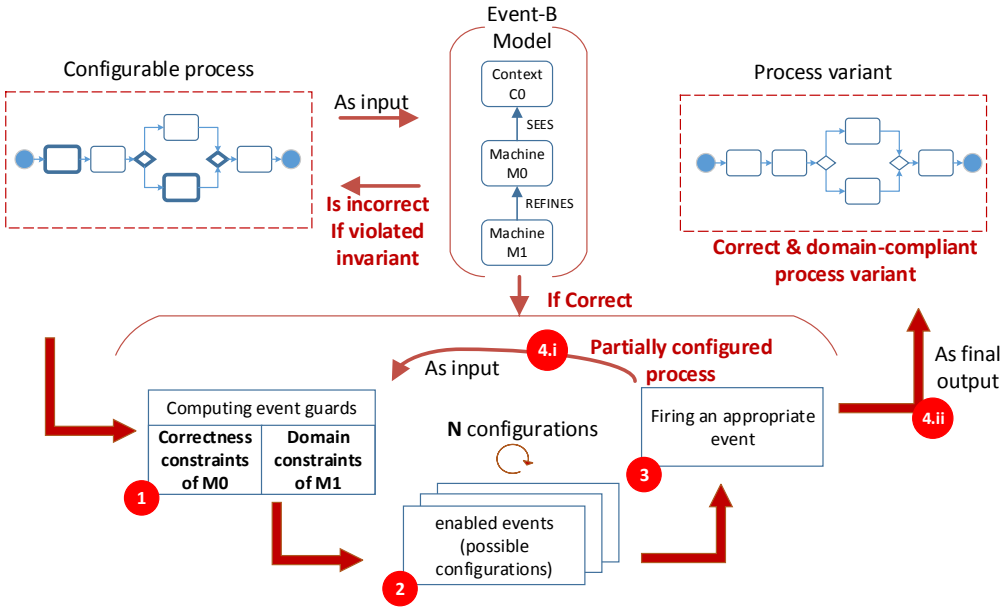


Figure 4.5: Our approach overview

First of all, on the top left side of the Figure 4.5, we have as input of our approach the Event-B representation of the configurable process model. Its correctness is verified with respect to the different properties expressed using invariants. The configuration can start only if all the invariants are verified. This allows to achieve our *objective 1* (defined in Section 4.1). Then, the analyst uses ProB animator [62] to perform configuration steps involving each element's configuration (cf. Section 4.8.2): firstly, the guards of each event are evaluated (step 1). These guards include both correctness and domain constraints. Moreover, events guards check for each connector's configuration, the type of the previously configured ones in order to prevent eventual mismatching. Then, only events whose guards are fulfilled are enabled (step 2). Thus, the configuration step can be applied (step 3). The set of potential configuration options is updated after each step. These steps are repeated (step 4.i) while there are configurable elements. As a result, the analyst derives a correct and domain-compliant process variant (step 4.ii), satisfying our *objectives 2* and *3*.

4.4 Formal Specification of a Business Process Model

The control-flow (or process) perspective describes activities and their execution ordering through different constructors, which permit flow of execution control [47]. In this subsection, we introduce the first step of our formalization which consists of two abstraction levels in order to consider this process perspective.

4.4.1 Business Process Model Graph

A business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal [3]. More formally, a business process model is composed of a number of activities or tasks which are connected to form a directed graph. Thus, we present an abstract formal definition of a business process model as follows.

Definition 4.4.1 (Business process model)**.** *A business process model is a directed graph* $Bp = < NODES, Initial, Final, SEQ, CON\_Type >$ *where:*

- $NODES$ *is the set of nodes that may be either activities* ACT*, split connectors* CON_S*, or join connectors* CON_J*;*
- $Initial \in ACT$ *in the unique initial activity;*
- $Final \subset ACT$ *in the non-empty set of final activities;*
- $SEQ : NODES \longleftrightarrow NODES$ *is edges connecting two nodes and is defined as the control flow relationship between them;*
- $CON\_Type : CON\_S \cup CON\_J \longrightarrow TYPES$ *is a function that assigns for each connector, either a split or a join, a type in* $\{OR, XOR, AND\}$

In the following subsection, we use this definition in the Event-B modeling of a business process.

4.4.2 Business Process Modeling using Event-B

We start the formal modeling of our configuration approach using Event-B by introducing the first level of our specification which holds processes, activities and their relationships. The semantics of the Event-B mathematical symbols used throughout this chapter are illustrated in Appendix B.

Firstly, we present the context *C0* depicted by Listing 1 which holds the following finite sets: (i) *BPS* (*axm1*), which defines the set of possible processes, (ii) *NODES* (*axm2*), which contains three values denoting types of nodes (*axm3*): activities (i.e. *ACTS*), split connectors (i.e. *CON_S*), and join connectors (i.e. *CON_J*), and (iii) *TYPES*, which defines three types of connectors (*axm4*): *OR*, *XOR* and *AND*. In order to initialize our model, we use constants to represent the different elements of these sets.

Listing 1: Context C0's constants and sets

```
CONTEXT C0
SETS BPS NODES TYPES
CONSTANTS ACTS CON_S CON_J AND XOR OR
                {a1} {a2} {a3} ... {s1} {s2} {s3} ...
AXIOMS
axm1 : finite(BPS)
axm2 : finite(NODES)
axm3 : partition(NODES, ACTS, CON_S, CON_J)
axm4 : partition(TYPES, {AND}, {XOR}, {OR})
axm5 : partition(ACTS, {a1}, {a2}, {a3}, ...)
axm6 : partition(CON_S, {s1}, {s2}, {s3}, ...)
       ...
```

Afterwards, we define the machine *M0* which sees the context *C0* described above. The variables of *M0* and their typing invariants are given in Listing 2. We define a variable *BP* to store the created processes. To map each process to its nodes, we introduce the relation *BP_Nodes* from *BP* to *NODES* (*Inv2*). We define the start and the end events as activities using respectively the total function *Initial* (*Inv3*) and the relation *Final* (*Inv4*); since we assume that a process has exactly one initial activity but may have several final ones. In BPMN, each connector, either a split or a join, has a type. This is modeled using the total function *CON_Type* (*Inv5*). The execution ordering between the different nodes is modeled using the total function *SEQ* (*Inv6*).

Listing 2: Machine M0's variables and typing invariants

```
MACHINE M0
SEES C0
VARIABLES BP BP_Nodes Initial Final SEQ CON_Type
INVARIANTS
Inv1 : BP ⊆ BPS
Inv2 : BP_Nodes ∈ BP ↔ NODES
Inv3 : Initial ∈ BP → ACTS
Inv4 : Final ∈ BP ↔ ACTS ∧ dom(Final) = BP
Inv5 : CON_Type ∈ BP_Nodes ▷ (CON_S ∪ CON_J) → TYPES
Inv6 : SEQ ∈ BP → (NODES ↔ NODES)
```

4.5 Formal Specification of a Configurable Process Model

To take the configurable nodes into account, we define a total function *Configurable_Nodes* (*Inv22*, Listing. 3) returning a *Boolean* value to state whether a given node is configurable or not in each process in which it appears.

We assume that a configurable process could be changed by applying a sequence of operations on it (e.g. excluding an activity or changing a connector type). So, at each configuration step, we define the process change using the partial function

Listing 3: Configuration invariants

```
Inv22 : Configurable_Nodes ∈ BP_Nodes → BOOL
Inv23 : Is_Configuration_Of ∈ BP ↠ BP
Inv24 : Is_Configuration_Of = Is_Configuration_OFFAct ∪ Is_Configuration_ONAct ∪
                              Is_Configuration_OR_S ∪ Is_Configuration_OR_J ∪
                              ... ∪ Is_Configuration_ToSeq
```

Is_Configuration_Of (*Inv23*, Listing. 3). For instance, a couple (*bp2*, *bp1*) belongs to *Is_Configuration_Of* if and only if at least one potential configuration has been applied on *bp1* in order to obtain a configured process *bp2*. *Inv24* asserts that these configurations could affect either (i) *activity configuration*, by excluding (i.e. *Is_Configuration_OFFAct*) or including it (i.e. *Is_Configuration_ONAct*), or (ii) *connector configuration*, by restricting splits outgowing branches (i.e. *Is_Configuration_OR_S*, *Is_Configuration_XOR_S* and *Is_Configuration_AND_S*), by restricting joins incoming branches (i.e. *Is_Configuration_OR_J*, *Is_Configuration_ XOR_J* and *Is_Configuration_AND_J*) or by keeping only one branch (i.e. *Is_Configuration_ToSeq*). These configuration types are further detailed in the following sections.

Structural Constraints In order to ensure consistent and structurally correct process control flow, we define a set of constraints to be respected. We illustrate some of them in Listing 4:

- Except the initial and the final nodes, each activity have exactly one incoming (*Inv11*[1]) and one outgoing arc (*Inv12*[2]).
- A split connector has exactly one incoming (*Inv13*) and at least two outgoings arcs (*Inv14*).
- A join connector has exactly one outgoing arc (*Inv15*), and at least two incomings arcs (*Inv16*).

Listing 4: Structural constraints invariants

```
....
Inv11 : ∀bp.(bp ∈ BP ⇒ (ACTS ◁ SEQ(bp)) ∈ ACTS ∩ BP_Nodes[{bp}] \ Final[{bp}]
                                              → BP_Nodes[{bp}] \ Initial[{bp}]
Inv12 : ∀bp.(bp ∈ BP ⇒ (SEQ(bp) ▷ ACTS) ∼∈ ACTS ∩ BP_Nodes[{bp}] \ Initial[{bp}]
                                              → BP_Nodes[{bp}] \ Final[{bp}]
Inv13 : ∀bp.(bp ∈ BP ⇒ CON_S ◁ (SEQ(bp) ∼) ∈ CON_S ∩ BP_Nodes[{bp}] ⤖ NODES)
Inv14 : ∀bp, nd.(bp ∈ BP ∧ nd ∈ CON_S ∧ bp ↦ nd ∈ BP_Nodes ⇒ card(SEQ(bp)[{nd}]) ≥ 2)
Inv15 : ∀bp.(bp ∈ BP ⇒ CON_J ◁ SEQ(bp) ∈ CON_J ∩ BP_Nodes[{bp}] ⤖ NODES)
Inv16 : ∀bp, nd.(bp ∈ BP ∧ nd ∈ CON_J ∧ bp ↦ nd ∈ BP_Nodes ⇒ card(SEQ(bp) ∼ [{nd}]) ≥ 2)
....
```

[1] $A \lhd f$ denotes a domain restriction: $A \lhd f = \{x \mapsto y | x \mapsto y \in f \wedge x \in A\}$

[2] The inverse of a function f, (f^{-1}), is denoted in Event-B as $(f \sim)$.

A process is considered to be *structurally sound* [3] if it fulfills the following two conditions:

- all nodes of the process can be activated, i.e. every node can be reached from the initial activity, as depicted by *Inv20* in Listing. 5 where cls^3 is the transitive closure of the relation $SEQ(bp)$; and
- for each activity in the process, there is at least one possible path leading from this activity to a final activity, i.e. the termination is always possible. This condition is captured by *Inv21* of Listing. 5.

Listing 5: Soundness constraints invariants

$$\texttt{Inv20} : \forall bp, node.(bp \mapsto node \in BP\_Nodes \wedge node \neq Initial(bp) \Rightarrow node \in (cls(SEQ(bp)))[\{Initial(bp)\}])$$
$$\texttt{Inv21} : \forall bp, node.(bp \mapsto node \in BP\_Nodes \wedge node \notin Final[\{bp\}] \Rightarrow (cls(SEQ(bp)))[\{node\}] \cap Final[\{bp\}] \neq \varnothing)$$

Erroneous patterns The configuration of a business process model may affect the soundness by two types of potential errors: *lack of synchronization* and *deadlocks* [95]. These situations result from a mismatch between splits and joins. To formally prevent these situations during configuration procedure, we defined six invariants: three for the splits and three for the joins. These invariants should be preserved by all the events that we define in the following subsections to capture the configuration operations.

An example of the lack of synchronization situation is captured by joining with an XOR operator, a control-flow that was split by an AND operator (cf. Figure 4.3b). The two outgoing branches from the AND-split will be activated, however, the XOR-join needs the completion of exactly one of its incoming branches. Thanks to *Inv25* (Listing. 6), this situation leading to an improper termination is not allowed in our model. Specifically, having a AND-split operator *ops* (line 2), for each couple of outgoing nodes *n1* and *n2* (line 3), the *first common node opj* (lines 4 to 6) should be an AND (line 7) or a not yet configured OR connector that should be eventually

Listing 6: Synchronization invariant

$$\begin{array}{ll}
1 & \texttt{Inv25} : \forall bp, ops, n1, n2.(bp \mapsto ops \in BP_Nodes \rhd CON_S \\
2 & \wedge \quad CON_Type(bp \mapsto ops) = AND \\
3 & \wedge \quad n1 \in SEQ(bp)[\{ops\}] \ \wedge \ n2 \in SEQ(bp)[\{ops\}] \ \wedge \ n1 \neq n2 \\
4 & \Rightarrow \ (\forall opj.opj \in (\cup t.t \in ((cls(SEQ(bp)))[\{n1\}] \cup \{n1\}) \\
5 & \quad \cap \ ((cls(SEQ(bp)))[\{n2\}] \cup \{n2\}) \ \wedge \ SEQ(bp) \sim [\{t\}] \cap (((cls(SEQ(bp)))[\{n1\}] \cup \{n1\}) \\
6 & \quad \cap \ ((cls(SEQ(bp)))[\{n2\}] \cup \{n2\})) = \varnothing \mid \{t\}) \\
7 & \Rightarrow \ (CON_Type(bp \mapsto opj) = AND \\
8 & \vee \quad (CON_Type(bp \mapsto opj) = OR \ \wedge \ Configurable_Nodes(bp \mapsto opj) = TRUE)) \ ))
\end{array}$$

[3] $cls(r)$ denotes the closure of the relation r defined, for each relation $(r \in S \leftrightarrow S)$, by:
(1) $cls(r) = \bigcup_{i=1..\infty} r^i$; (2) $r^1 = r$; and (3) for each $n >= 2 r^n = (r; r^{n-1})$
The transitive closure formulations were expressed as machine theorems.

configured as an AND (line 8). Note that, Having two nodes *n1* and *n2*, the first common node is the first node which belongs to the transitive closure of both nodes *n1* and *n2*

Similar invariants are defined to ensure a deadlock-free control flow which is an important criterion for soundness. We aim to guarantee the absence of situations where a node can never be activated (cf. Figure 4.3a). We model such situations using *Inv26* in Listing. 7. This invariant asserts that an OR-split or an XOR-split should not be followed by an AND-join. Basically, having a split type different from AND (lines 2 and 3), we check that, for each couple of outgoing nodes *n1* and *n2* (line 4), the first common node (lines 5 to 7) is not an AND-join (line 8).

Listing 7: Deadlock-freeness invariant

```
1 Inv26 : ∀bp, ops, n1, n2.(bp ↦ ops ∈ BP_Nodes ▷ CON_S
2   ∧     (CON_Type(bp ↦ ops) = XOR
3   ∨     (CON_Type(bp ↦ ops) = OR ∧ Configurable_Nodes(bp ↦ ops) = FALSE))
4   ∧     n1 ∈ SEQ(bp)[{ops}]  ∧  n2 ∈ SEQ(bp)[{ops}]  ∧  n1 ≠ n2
5   ⇒  (∀opj.opj ∈  (∪t.t ∈ ((cls(SEQ(bp)))[{n1}] ∪ {n1})
6      ∩  ((cls(SEQ(bp)))[{n2}] ∪ {n2})  ∧  SEQ(bp) ~ [{t}] ∩ (((cls(SEQ(bp)))[{n1}] ∪ {n1})
7      ∩  ((cls(SEQ(bp)))[{n2}] ∪ {n2})) = ∅ | {t}))
8   ⇒  CON_Type(bp ↦ opj) ≠ AND  ))
```

In the following section, we tackle the configuration procedure. Hence, we define configuration operations to apply on a business process as well as a set of configuration constraints to be respected.

4.6 Formal Specification of Configuration Steps

In this section, we introduce the formalization of process elements configuration: *activity configuration*, and *connector configuration*. In this formalization, each configuration step is performed by a single event. In order to derive correct variants, we define a set of constraints using invariants and event guards. Then, we prove that each event preserves them, which implies that the erroneous situations presented above will be avoided, namely the deadlock and the lack of synchronization.

4.6.1 Activity Configuration

A configurable activity could be included or excluded in a process variant according to the process analyst choice. To define this activity configuration, two events and two invariants defining configuration constraints are introduced.

With regard to invariants, they allow to define the configuration constrains before and after variables change by the defined events. For instance, we take the example of excluding an activity, the invariant *inv27* (cf. Listing 8) insures that for each couple *(bp2, bp1)* belonging to *Is_Configuration_OFFAct* (line 2), such that *bp2* process results from *bp1*. Then, there exists an activity *act*; such that: (i) *act* should be

configurable activity belonging to *bp1* (lines 4 to 5); (ii) *act* is the only difference between *bp1* nodes and *bp2* nodes (line 6); (iii) *act* and its dependencies are removed from *bp1* and a new dependency linking its predecessor and its successor is created in *bp2* (line 7); (iv) all activities other than *act* retain the same configuration (lines 8 to 9); (v) all connectors retain the same type (lines 12 to 13); and (vi) *initial* and *final* activities remain the same (lines 10 to 11). Likewise, a second invariant is defined to constrain the activity preservation after its configuration.

Listing 8: OFF Activity Configuration invariant

```
1  Inv27 : ∀bp1, bp2. (bp1 ∈ BP ∧ bp2 ∈ BP ∧
2                      bp2 ↦ bp1 ∈ Is_Configuration_OFFAct
3  ⇒
4  ∃ act.(act ∈ ACTS ∧ act ∈ BP_Nodes[{bp1}]
5      ∧ bp1 ↦ act ↦ TRUE ∈ Configurable_Nodes
6      ∧ BP_Nodes[{bp1}] \ BP_Nodes[{bp2}] = {act}
7      ∧ SEQ(bp2) = (({act} ⩤ SEQ(bp1)) ⩥ {act}) ∪ ((SEQ(bp1)) ∼ [{act}] × ((SEQ(bp1))[{act}])) )
8      ∧ (∀ actx. actx ∈ ran({bp1} ⩤ dom(Configurable_Nodes)) \ {act}
9                      ⇒ Configurable_Nodes(bp2 ↦ actx) = Configurable_Nodes(bp1 ↦ actx) )
10         ∧ Initial(bp2) = Initial(bp1)
11         ∧ Final[{bp2}] = Final[{bp1}]
12         ∧ ∀con. (con ∈ BP_Nodes[{bp1}] ∩ (CON_S ∪ CON_J) ⇒
13               CON_Type(bp1 ↦ con) = CON_Type(bp2 ↦ con)) )
```

With regard to events, activity configuration is performed through either: (i) *ConfigureACTON* event which keeps the activity; or (ii) *ConfigureACTOFF* event which excludes it. We present in Listing. 9 the *ConfigureACTOFF* event. Based on a configurable process *bp1*, a configured process *bp2* is a result of excluding an activity *act*. As guard, in addition to typing ones (*grd1* and *grd2*), *act* must be configurable (*grd3*). This event allows *bp2* to inherit from *bp1*: (i) its nodes whilst removing *act* (*act2*), (ii) its initial and final activities (*act3* and *act4*), (iii) all its nodes relations (i.e. SEQ(bp1)) while removing *act* dependencies and creating a new one connecting *act* successor and predecessor (*act5*), (iv) its configurable nodes (*act6*), and (v) the types of its connectors (*act7*). Finally, we define *bp2* as a configuration of *bp1* whilst excluding *act* (*act8*). Similarly, the event *ConfigureACTON* allows to maintain the same process by keeping the configurable activity. The only change applied on the resulting process consists in making the activity as non configurable.

4.6.2 Connector Configuration

A connector configuration represents a decision point that is of relevance during the process configuration life cycle. Each decision has to consider the following requirements: (1) the configuration constraints for each type of connector (e.g. an AND could not be configured to an XOR), (2) only configurable nodes can be removed in order to avoid unreachable nodes, and (3) the connectors types matching checking in order to prevent erroneous situations.

Listing 9: Excluding activity event

```
ConfigureACTOFF ≙   ANY    bp1   bp2   act
WHERE
grd1: bp1 ∈ BP ∧ act ∈ ACTS ∧ bp1 ↦ act ∈ BP_Nodes
grd2: bp2 ∈ BPS \ BP
grd3: Configurable_Nodes(bp1 ↦ act) = TRUE
  ...
THEN
act1: BP := BP ∪ {bp2}
act2: BP_Nodes := BP_Nodes  ∪  ({bp2} × (BP_Nodes[{bp1}] \ {act}))
act3: Initial(bp2) := Initial(bp1)
act4: Final := Final ∪ ({bp2} × (Final[{bp1}]))
act5: SEQ(bp2) := (({act} ⩤ SEQ(bp1)) ⩥ {act})  ∪((SEQ(bp1)) ∼ [{act}] × SEQ(bp1)[{act}])
act6: Configurable_Nodes := Configurable_Nodes  ∪  (⨆  node.node ∈ BP_Nodes[{bp1}] \ {act}
                  | {bp2 ↦ node ↦ Configurable_Nodes(bp1 ↦ node)})
act7: CON_Type := CON_Type  ∪  (⨆  con.con ∈ BP_Nodes[{bp1}] ∩ (CON_S ∪ CON_J)
                  | {bp2 ↦ con ↦ CON_Type(bp1 ↦ con)})
act8: Is_Configuration_OFFAct := Is_Configuration_OFFAct  ∪  {bp2 ↦ bp1}
```

Concretely, in order to obtain a well-structured configured process, an invariant for each configuration choice should be respected. For instance, in case of OR-split configuration, the invariant *Inv29* (see Listing.10) ensures that for each couple *(bp2, bp1)* belonging to *Is_Configuration_OR_S* (line 1) there exists an OR split operator *ops* (line 3) which belongs to both *bp1* and *bp2* nodes (line 4) such that *ops* is configurable in *bp1* (line 5) and non configurable in *bp2* (line 6). Also other connectors in the processes do not change type (lines 7-8). According to this configuration, some nodes (line 9) as well as their dependencies (line 10) may be removed when restricting *ops* outgoings. No additional dependencies are created as well. Finally, the *initial* and *final* activities always remain the same (lines 11 to 12).

Listing 10: Or Split configuration invariant

```
1  Inv29  :  ∀bp1, bp2.(bp1 ∈ BP ∧ bp2 ∈ BP  ∧  bp2 ↦ bp1 ∈ Is_Configuration_OR_S
2  ⇒  (
3      ∃  ops.(ops ∈ CON_S  ∧  bp1 ↦ ops ↦ OR ∈ CON_Type
4      ∧  ops ∈ BP_Nodes[{bp1}] ∧ ops ∈ BP_Nodes[{bp2}]
5      ∧  bp1 ↦ ops ↦ TRUE ∈ Configurable_Nodes
6      ∧  bp2 ↦ ops ↦ FALSE ∈ Configurable_Nodes
7      ∧  ∀con.(con ∈ BP_Nodes[{bp1}] ∩ (CON_S ∪ CON_J) \ {ops}
8                                  ⇒  CON_Type(bp1 ↦ con) = CON_Type(bp2 ↦ con)))
9      ∧  BP_Nodes[{bp2}] ⊆ BP_Nodes[{bp1}]
10     ∧  SEQ(bp2) ⊆ SEQ(bp1)
11     ∧  Initial(bp2) = Initial(bp1)
12     ∧  Final[{bp2}] = Final[{bp1}] )  )
```

These configuration choices are insured by two events (either split or join) for each connector type. A first set of events model the split configuration: *ConfigureORSplit*, *ConfigureXORSplit* and *ConfigureANDSplit*. A second set of events model the join configuration: *ConfigureORJoin*, *ConfigureXORJoin* and *ConfigureANDJoin*.

For instance, Listing.11 illustrates the event *ConfigureORSplit*. This event allows

configuring a configurable split connector *ops* (*grd3*) from OR type (*grd4*) to any type *to* (*grd5*) (as OR could be configured to any type, see line 1 of Table 2.1). This event enables also to preserve the branches starting by nodes in *nodes*. Obviously, the number of remaining branches should be greater than two (*grd6*). However, each branch can be removed only if all its nodes are configurable (*grd8*). Furthermore, our model avoids connectors types mismatching by considering corresponding join connectors. For example, using *grd11*, for each two outgoing branches *n1* and *n2*, if the corresponding join is an AND, then the split should be configured to an AND as well.

Listing 11: Or Split configuration event

```
ConfigureORSplit ≙  ANY  bp1  bp2  ops  nodes
      to  deletedNodes  subgraph  conToSeq
WHERE
grd1: bp1 ∈ BP ∧ ops ∈ CON_S ∧ bp1 ↦ ops ∈ BP_Nodes
grd2: bp2 ∈ BPS \ BP
grd3: Configurable_Nodes(bp1 ↦ ops) = TRUE
grd4: CON_Type(bp1 ↦ ops) = OR
grd5: to ∈ TYPES
grd6: nodes ⊆ SEQ(bp1)[{ops}] ∧ card(nodes) ≥ 2
grd7: deletedNodes = (∪zz. zz ∈ BP_Nodes[bp1] \ {Initial(bp1)} ∧
        zz ∉ (cls(SEQ(bp1) \ (ops × (SEQ(bp1)[{ops}] \ nodes))) [{Initial(bp1)}]) | {zz})
grd8: deletedNodes ⊆ Configurable_Nodes ∽ [{TRUE}][{bp1}]
grd9: subgraph = (deletedNodes ⩤ (SEQ(bp1)\ ({ops} × (SEQ(bp1)[{ops}] \ nodes)))) ⩥deletedNodes
grd10:conToSeq = (∪x.x ∈ CON_S ∧ card(subgraph[{x}]) = 1 | {x})
        ∪ (∪x.x ∧ CON_J ∧ card(subgraph ∼ [{x}]) = 1 | {x})
grd11: ∃n1, n2.( n1 ∈ nodes ∧ n2 ∈ nodes ∧ n1 ≠ n2
  ∧ ( (∃opj.opj ∈ (∪t.t ∈ ((cls(SEQ(bp1)))[{n1}] ∪ {n1})
        ∩((cls(SEQ(bp1)))[{n2}] ∪ {n2})
        ∧ SEQ(bp1) ∼ [{t}] ∩ (((cls(SEQ(bp1)))[{n1}] ∪ {n1})∩
            ((cls(SEQ(bp1)))[{n2}] ∪ {n2})) = ∅ | {t})
        ∧ CON_Type(bp1 ↦ opj) = AND ))
                ⇒        to = AND
...
THEN
...
```

Similarly, the join operator configuration may depend on one or more splits. Thus, the corresponding split of each two ingoing branches is taken into account by checking its configured or not yet configured type using guards. For instance, in the event dealing with configuring OR-join, namely *ConfigureORJoin*, we add the *grd9* in Listing.12 to verify that: If there exists two distinct nodes *n1* and *n2* belonging to *opj* incomings (line 1), and having the first common previous node (lines 2-3) configured to an XOR type (line 4), then *opj* should not be configured to an AND (line 5). Hence, this condition guarantees a deadlock-free configuration of this join operator. Similarly, we add other guards to avoid the lack of synchronization situation.

Listing 12: Or join guard

```
1 grd9 : ∃n1, n2.( n1 ∈ nodes ∧ n2 ∈ nodes ∧ n1 ≠ n2
2   ∧ ( (∃ops. ops ∈
    (⊔t. t ∈ ((cls((SEQ(bp1)) ∼))[{n1}] ∪ {n1}) ∩ ((cls((SEQ(bp1)) ∼))[{n2}] ∪ {n2})
```

```
3        ∧ SEQ(bp1)[{t}] ∩ (((cls((SEQ(bp1)) ∼))[{n1}] ∪ {n1}) ∩
     ((cls((SEQ(bp1)) ∼))[{n2}] ∪ {n2})) = ∅ | {t})
4        ∧ CON_Type(bp1 ↦ ops) = XOR ))
5                ⇒          to ≠ AND
```

Besides the configuration of a connector from one type to another, it is possible to configure it to a sequence by keeping a single branch. This is defined using the events *ConfigureCONSToSeq* (for a split connector) and *ConfigureCONJToSeq* (for a join connector). As mentioned previously, only OR and XOR types could be configured to a sequence. This constraints is ensured by a guard (*grd6*: $CON\_Type(bp1 \mapsto ops) \neq AND$). This event checks also whether the corresponding join (resp. split) should be mapped into a sequence or not. As a result, the branch to retain is linked to the predecessor and the successor of the deleted operators.

4.7 Introduction of the Configuration Guidelines into the Model

Process providers may define specific business domain constraints for their process configurations. Thus, *configuration guidelines* are introduced to depict relevant inter-dependencies between the configuration decisions in order to be inline with domain constraints and best practices. Such guidelines are expressed via logical expressions of the form ***If-Then*-rules**. Both the *if* and *then* parts contain statements about binding configurable nodes to concrete values [1]. As examples of such rules:

$$\textbf{\textit{if}} \quad a9 = OFF \quad \textit{and} \quad s5 = Seq(a7) \quad \textbf{\textit{then}} \quad a14 = OFF \tag{4.1}$$

$$\textbf{\textit{if}} \quad a3 = ON \quad \textbf{\textit{then}} \quad a7 = ON \tag{4.2}$$

$$\textbf{\textit{if}} \quad s3 = Seq(a3) \quad \textit{and} \quad s5 = Seq(a7) \quad \textbf{\textit{then}} \quad s6 = OR(a11, a12) \tag{4.3}$$

This means that: (4.1) if the car searching and selection functionalities are excluded in a given variant, then the discount activity is excluded too; (4.2) if the hotel recommendation functionality is included in the derived variant, then the hotel searching functionality should be also included; and (4.3) if the car searching, selection and also the car recommendation functionalities are excluded, then the checking for car availability should be ignored as well.

Note that, the *if-part* may contain many conditions and the *then-part* contains only one statement or consequent. If the conditions are *true* (i.e., the configuration choices were applied in a previous step), then the consequent statement should be respected (i.e., only this configuration choice should be applied). For example, the rule (4.2) have one condition (i.e., $a3 = ON$) and one consequent configuration that should follow this condition (i.e. $a7$ should be set ON). Whereas, rules (4.1) and

(4.3) have two condition statements (e.g., $a9 = OFF$ *and* $s5 = Seq(a7)$) that need to be applied in order to require the application of the consequent configuration.

In order to integrate these domain constraints in our model, we define a second abstraction level, namely machine *M1*, that refines the machine *M0*. In fact, we define for each type of *then-part* statement one invariant. In other words, we define one invariant for each element configuration that represent a consequent to other configuration choices. We give three examples of such invariants in Listing.13. For instance, the relation *ConfigurationG_ACT* (*inv1*) defines a guideline that leads to a specific configuration (either *ON* or *OFF*) of a specific activity. Thus, the guideline may have five different conditions: an activity configuration (line 1), a split or join configuration to a type (line 2), and a split or join configuration to a sequence (line 3). Similarly, *ConfigurationG_CONS* (resp. *ConfigurationG_CONJ*) defines the guideline for a split (resp. a join) connector configuration to a specific type and specific outgoing (resp. ingoing) branches. And $ConfigurationG\_CONS\_SEQ$ represent the guideline for recommending a sequence configuration after the application of some conditions.

Listing 13: Guidelines invariants

```
1  Inv1 : ConfigurationG_ACT ∈ P(ACTS × CONF) ×
2     P(CON_S × TYPES×P1(NODES)) × P(CON_J × TYPES×P1(NODES)) ×
3     P(CON_S × NODES) × P(CON_J × NODES) ⟷ ACTS × CONF
4  Inv2 : ConfigurationG_CONS ∈ P(ACTS × CONF) ×
5     P(CON_S × TYPES×P1(NODES)) × P(CON_J × TYPES×P1(NODES)) ×
6     P(CON_S × NODES) × P(CON_J × NODES) ⟷ CON_S × TYPES× P1(NODES)
7  Inv3 : ConfigurationG_CONS_SEQ ∈ P(ACTS × CONF) ×
8     P(CON_S × TYPES×P1(NODES)) × P(CON_J × TYPES×P1(NODES)) ×
9     P(CON_S × NODES) × P(CON_J × NODES) ⟷ CON_S× NODES
        ...
```

As each configuration step must fulfill the configuration guidelines, we refined our abstract events by adding one guard for each guideline. For instance, considering the first example (4.1) above, we have $\{a9 \mapsto OFF\} \mapsto \varnothing \mapsto \varnothing \mapsto \{s5 \mapsto a7\} \mapsto \varnothing \mapsto (a14 \mapsto OFF) \in$ *ConfigurationG_ACT*. Thus, we have two conditions consisting of $\{a9 \mapsto OFF\}$ and $\{s5 \mapsto a7\}$ that if satisfied, *a14* should be mapped to *OFF*. Hence, we added a guard in the event *ConfigureACTON* to ensure that in order to set an activity to *ON* at least one condition is not satisfied in a guideline leading to the configuration of this activity to *OFF*. In this particular case, *a14* can be set to *ON* if *a9* and *s5* have been both configured and *a9* has been set to *ON* or *s5* has not been configured as a sequence of *a7*. Reciprocally, the configuration *a14* to *ON* is not allowed if at least one of *a9* and *s5* is not configured yet or both have been configured according to the guideline.

4.8 Verification and Validation

4.8.1 Verification using Proofs

In order to demonstrate that the formal specification of configurable process models is correct, a the number of generated proof obligations (POs) should be discharged. Using the Rodin tool [60], our model generated 358 proof obligations; most of them (272 POs $\simeq$ 76%) were automatically discharged; more complex ones (86 POs $\simeq$ 24%) required the interaction with the provers to help them find the right rules to apply but also to define additional rules that may lack in the rule base of the prover. These POs ensure that the invariants which model the different constraints on the configurable business processes and the derived variants, are always satisfied (i.e. they hold initially; and each event preserves them). For each event of the form (**WHEN** G **THEN** Act) with G and Act representing the guard and the action respectively, the following proof obligation is generated to verify that the execution of the action Act under the guard G permits to preserve the invariant [50]: $(Inv \wedge G) \Rightarrow [Act]Inv$.

An example of the proofs, we have established, concerns the event *Configure-ACTOFF* correctness with respect to the invariant $inv20$: we have to prove that even if an activity *act* is removed (set to OFF), it remains possible to reach each node from the initial one. This holds since we have added a control from linking the predecessor of *act* to its successor. To discharge this proof that refers to the closure of a relation, we have added the rule defining the closure of the union of two relation s and r:

$$r \in t \leftrightarrow t \wedge s \in t \leftrightarrow t \Rightarrow cls(r \cup s) = cls(r)((id(t) \cup cls(r)); s)+; (id(t) \cup cls(r))$$

4.8.2 Validation by Animation

Now, based on a correct model, we validate our Event-B specification by animation and model checking using the ProB plugin [62]. Concretely, we play and observe different scenarios and check the behavior of our model by showing at each step the values of each variable, which events are enabled or not.

For instance, we illustrate the animation of the scenario captured by the Figure 4.3b in Section 4.2 as follows. After initializing the model using our motivating configurable process (cf. Figure 4.1 in Section 4.2), i.e. after triggering the first event: *Initialization*, all invariants should be respected to ensure the correctness of the configurable process model. This verification is ensured by the ProB animation view as depicted in the bottom side (1) of the Figure 4.6. Next, we process our scenario by triggering enabled events, and at each configuration step, we observe that invariants are always re-established:

i) we trigger the*ConfigureORSplit* event to configure the split operator *s1* from OR to an AND ($to = AND$) while maintaining the same branches,

ii) *s3* and *j1* are configured (using *ConfigureToSeq* event) to a sequence starting from *a3* (*a3* is set to *ON* as well),

iii) the activity *a7* is set to *ON* (using *ConfigureACTON* event). Since *a3* is included in the previous step, the mapping of *a7* to *OFF* is not allowed in accordance with the guideline *(2)* defined in Section 4.7,

iv) *s5* and *j3* are also configured to a sequence starting from *a7*; only this branch should be preserved, since the second branch nodes are configurable. Next,

iv) when configuring the join operator *j2*, the only allowed alternative is to fire the event *ConfigureORJoin* with the connector type parameter AND (see Figure 4.6).

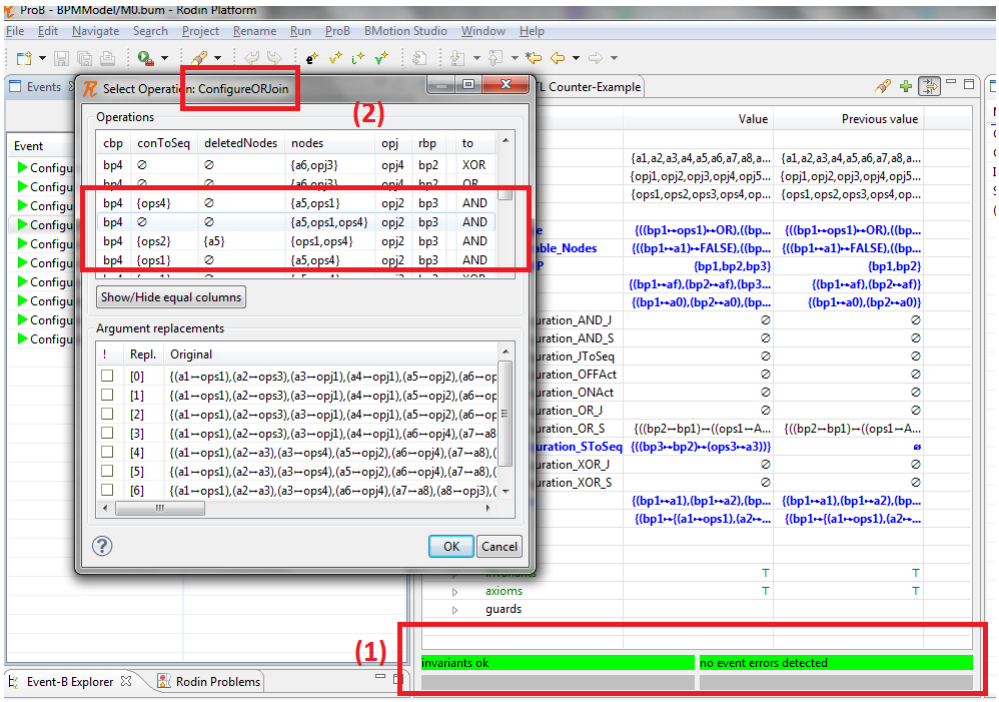


Figure 4.6: The connector *j2* configuration restriction using ProB

By formally defining the correct configurations, we guaranteed that the resulted variant have not improper termination caused by the lack of synchronization.

4.8.3 Case Study

In order to evaluate the practical usefulness and identify the opportunities of using our approach, we conducted a case study with a group of business process experts and analysts. In the following, we examine its objectives, analyze and discuss its practical experience in conducting business process model configuration.

4.8.3.1 Objective

The main goal of our work is to evaluate how our approach helps and guides analysts in generating correct and domain-compliant process configuration. Therefore, we define the following research question: How can our approach assist process analyst in applying correct configuration steps?

To answer this question, we formulate three hypotheses that our approach allows:

(H1) to save time and facilitate the identification of the configuration steps;

(H2) to guarantee a correct process model at each configuration step; and

(H3) to derive domain-compliant process variants based on the configuration guidelines.

4.8.3.2 Design, Data Collection and Execution

Our case study is a real configurable supervision process adopted by Orange, a french telecom industrial partner. Different variants of this process are used by Orange affiliates in different cities and countries according to their specific needs. Based on 28 variants, a set of configuration guidelines was generated by an automated approach and validated by a domain expert [9].

With a population of 9 participants that are familiar with process configuration, we targeted experiments to derive a set of different variants using the considered configurable process model. With this purpose, we divided the population into three groups of three people each. After a workshop organized to explain the basics needed in this study, the first group (G_1) is asked to manually derive a maximum of process variants without any guidance. Then, the second group (G_2) is also asked to manually derive process variants, but, while providing them with the generated configuration guidelines rules. Whereas, the third group (G_3) is provided with the complete Event-B model (installed under the RODIN tool) and asked to generate process variants with respect to the allowed configuration choices by the model checking. So, participants of latter group can apply only configuration steps that are allowed by our model. As mentioned in the previous sections, this model includes both correctness and domain constraints. However, the first two groups take the burden of verifying the correctness of their choices.

The resulted process variants are then collected for comparison. In order to answer the identified research question and confirm its hypotheses, we evaluated the results according to two parameters: (1) the time needed to derive process variants for the different groups, (2) the number of errors for the identified correctness and domain constraints.

4.8.3.3 Results Analysis and Findings

Regarding the time needed to derive variants, the group G_1 took in average 16 minutes and the group G_2 took in average 14 minutes, whereas the group G_3 took only 5 minutes. Table 4.1 shows the distribution of the time according to the correctness and the business criteria. Through this table, we notice that the more participants of G_1 and G_2 take time in deriving variants the less correctness errors are detected. This can explain that participants are making a special effort. Also, it is clear that the first two groups took much more time in deriving correct and domain-compliant variants than the group G_3. It is worth noting that all derived variants by G_3 contain neither structural nor behavioral correctness errors. No domain errors are detected as well. Moreover, the participants of group G_3 affirmed that the ProB model checker is quite straightforward to use and it assisted them in defining appropriate configuration steps. They easily followed the enabled events to make their choices which helped them to be compliant not only to correctness constraints but also to domain recommendations. As a result, we concluded that our approach allows (1) to save time and to assist users in defining their configuration choices, which supports the hypothesis *H1*; and (2) to respect correctness and domain constraints, supporting *H2* and *H3*.

Table 4.1: The average time in minutes unit spent to derive variants either correct (C) or not (¬C), and either business-complaint (B) or not (¬B)

Group \ Variant	C & B	C & ¬B	¬C & B	¬C & ¬B
G_1	23	17	15	8
G_2	17	×	11	×
G_3	5	×	×	×

4.8.3.4 Threats to Validity

First, the small number of the collected process variants, used to generate our configuration guidelines, can be considered as a threat of validity. However, in this study we have chosen 28 variants that are relevant and depict various business needs. Secondly, one case study has been only conducted by 9 participants. We believe that a larger group of participants with varied backgrounds need to be used to highlight the validity and reliability of the experiments results. We leave this to future work.

4.8.4 ATL Model Transformation: BPMN to Event-B

In this section, we describe our Model-to-Model transformation that we developed to map the BPMN process models into an Event-B specification using the ATL model transformation language. A model transformation is the automatic creation of target models from source models. Figure 4.7 presents the structure of the tool that we implemented.

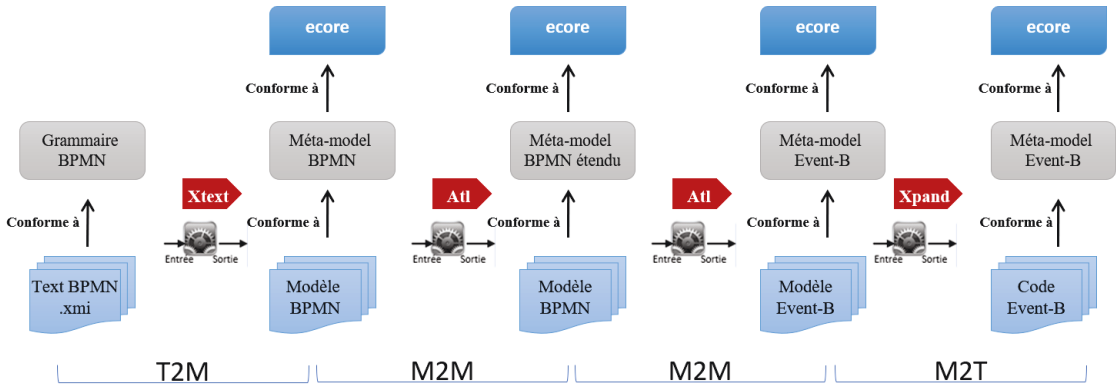


Figure 4.7: Structure of the BPMN to Event-B transformation tool

This tool takes as input an XML file that contains a textual representation of a BPMN model to which a number of transformations are applied in order to create the code for the corresponding Event-B model. The complete process comprises four transformations: (1) one Text To Model transformation (T2M), (2) two Model To Model transformations (M2M), and (3) one Model To Text transformation (M2T).

The first transformation consists in injecting the input XML file via Xtext. We started by defining the BPMN grammar and, using Xtext, were able to generate a lexical analyser and a meta-model to which that grammar conforms. This meta-model is the one used for this T2M transformation. In fact, this first step consists in transforming the textual representation of the input BPMN model contained in the input XML file into an XMI model conforming to the previously generated meta-model.

The second transformation is an endogenous one that takes as input the output model of the first transformation. It is implemented using ATL. The target meta-model to which conforms the resulting XMI model of this transformation is an extended version of the BPMN meta-model (BPMN20.ecore) as defined and used by the Eclipse BPMN2 Modeler. It is an extended version because we had to integrate elements specific to configurable Business processes as the original meta-model does not support such concepts. In terms of correspondences between the elements of the input and output models, this transformation brings no change. Nevertheless, it was necessary to use a number of ATL helpers to guarantee the accuracy of the resulting model.

The third transformation is at the core of the whole transformation process. It is an exogenous ATL transformation that, taking as input the output of the previous transformation, generates an XMI model that conforms to the Event-B meta-model (EVENTB.ecore). This meta-model is the same one used by the Rodin platform. The transition between the two models pertaining to two different languages is achieved by applying a set of transformation rules and helpers that associate BPMN elements

to Event-B elements.

The last transformation treats the extraction phase using Xpand. The goal of this step is to transform the Event-B XMI model resulting from the third transformation into a textual Event-B code. Consequently, the execution of this transformation generates an Event-B project, including both a context and a machine, that can be directly imported and used by the Rodin platform.

Transformation rules and helpers We choose to give some examples of transformation rules and helpers from the third transformation as it is the one that bears the most important correspondences and closes the gap between the BPMN and Event-B languages. Figure 4.8 shows the three main transformation rules which are responsible for the creation of the whole Event-B project including the context and the machine. These rules use multiple other transformation rules and helpers in order to create all the required Event-B elements. For example, the helpers used for the creation of the axioms axm6 and axm7 of the Event-B model's context can be seen in figures 4.9 and 4.10 respectively. Similarly, the helper responsible for the creation of the initialisation event in model's machine is given in figure 4.11. We note that we use helpers for the creation of the actions 11 to 17 only since the rest of the actions (1 to 10) are invariable, that is to say, the do not depend on the input BPMN model.

```
rule Process2Project {
    from in1 : BPMN!Process
    to out1 : EVENTB!Project(
        name <- in1.name,
        components <- thisModule.Process2Context(in1),
        components <- thisModule.Process2Machine(in1, out1.components.first()))
}

lazy rule Process2Context {
    from in1 : BPMN!Process
    to out1 : EVENTB!Context(
        name <- in1.name.concat('C'),
        sets <- in1.getDefaultCarrierSets(),
        constants <- in1.getDefaultConstants(),
        constants <- thisModule.Process2Constant(in1),
        constants <- in1.flowElements -> collect (fe | if fe.oclIsTypeOf(BPMN!Task) then thisModule.Task2Constant(fe) else OclUndefined endif),
        constants <- in1.flowElements -> collect (fe | if fe.oclIsKindOf(BPMN!Gateway) then thisModule.Gateway2Constant(fe) else OclUndefined endif),
        constants <- in1.flowElements -> collect (fe | if fe.oclIsTypeOf(BPMN!StartEvent) then thisModule.StartEvent2Constant(fe) else OclUndefined endif),
        constants <- in1.flowElements -> collect (fe | if fe.oclIsTypeOf(BPMN!EndEvent) then thisModule.EndEvent2Constant(fe) else OclUndefined endif),
        axioms <- in1.getDefaultAxioms(),
        axioms <- in1.getAxiom6(),
        axioms <- in1.getAxiom7(),
        axioms <- in1.getAxiom8(),
        axioms <- in1.getAxiom9())
}

lazy rule Process2Machine {
    from in1 : BPMN!Process, in2 : EVENTB!Context
    to out1 : EVENTB!Machine(
        name <- in1.name.concat('M'),
        sees <- in2,
        variables <- in1.getDefaultVariables(),
        invariants <- in1.getDefaultInvariants(),
        events <- in1.getInitialisationEvent(),
        events <- in1.getDefaultEvents())
}
```

Figure 4.8: The Process2Project, Process2Context and Process2Machine rules

Threats to validity It is worth mentioning that, in order to validate this transformation work, we have developed the reverse-engineering of the transformation process.

```
helper context BPMN!Process def : getAxiom6() : EVENTB!Axiom =
    let seq : Sequence(BPMN!FlowElement) = self.getTasks().union(self.getEndEvents().append(self.getStartEvent()))
    in thisModule.String2Axiom(Sequence{'axm6', 'partition(ACTS,'+ seq -> iterate (fe; str : String = '' | str+'{'+fe.name+'}' +
    if seq.size()=seq.indexOf(fe) then '' else ',' endif)+')'});
```

Figure 4.9: The getAxiom6() helper

```
helper context BPMN!Process def : getAxiom7() : EVENTB!Axiom =
    let seq : Sequence(BPMN!Gateway) = self.getDivergingGateways()
    in thisModule.String2Axiom(Sequence{'axm7', 'partition(CON_S,'+seq -> iterate (gw; str : String = '' | str+'{'+gw.name+'}' +
    if seq.size()=seq.indexOf(gw) then '' else ',' endif)+')'});
```

Figure 4.10: The getAxiom7() helper

In fact, we have implemented the reverse transformations to prove that, starting from the resulting Event-B code, we can regenerate the initial textual representation of the used BPMN model.

Although we were able to assess the conformity of the reverse-engineered BPMN model to the one initially used by the first transformation process by comparing their textual representations, we had trouble in the visualization of the former. In fact, this problem stems from the inability of the Signavio tool to edit models that had not initially been created by the said tool due to some required graphical data in the XML file. However, we were able to visualize reverse-engineered BPMN models that did not include any configurable element using the Eclipse BPMN2 Modeler.

4.9 Conclusion and Discussion

Our main contribution in this work is the formal specification of process model configuration allowing to assist designers to configure, step-by-step, correct variants. We answered two research questions raised in Chapter 1 as follows:

RQ1: How to identify configuration choices that satisfy designers and clients requirements? We proposed a formal Event-B based approach to derive correct variants from well-defined configurable processes. We reached our goal in considering different constraints related to: (i) configuration (e.g., an XOR may be configured only by restricting its ingoing/outgoing branches), (ii) structural correctness (e.g., no dead activities), (iii) erroneous patterns (e.g., no mismatching between connectors configurations that may result in behavioral problems) and (iv) domain compliance (the configuration of an element effects another to satisfy client expectations). The defined constraints and properties for a correct configuration are expressed in terms of mathematical predicates (i.e., invariants). The configuration steps are modeled using events. The events are constrained by means of specified enabling conditions (i.e., guards). We used the Event-B tools to prove the correctness of our specification by checking the defined properties by discharging Proof Obligations. At each configuration step, the events are evaluated and only configurations leading to variants satisfying invariants can be

```
212⊖helper context BPMN!Process def : getInitialisationEvent() : EVENTB!Event=
213    thisModule.String2Event(Sequence{'INITIALISATION',
214                        Sequence{},
215                        Sequence{},
216                        Sequence{},
217                        Sequence{Sequence{'act1','Is_Configuration_OFFAct&#8788;&#8709;'},
218                                Sequence{'act2','Is_Configuration_ONAct&#8788;&#8709;'},
219                                Sequence{'act3','Is_Configuration_OR_S&#8788;&#8709;'},
220                                Sequence{'act4','Is_Configuration_OR_J&#8788;&#8709;'},
221                                Sequence{'act5','Is_Configuration_XOR_S&#8788;&#8709;'},
222                                Sequence{'act6','Is_Configuration_XOR_J&#8788;&#8709;'},
223                                Sequence{'act7','Is_Configuration_AND_S&#8788;&#8709;'},
224                                Sequence{'act8','Is_Configuration_AND_J&#8788;&#8709;'},
225                                Sequence{'act9','Is_Configuration_SToSeq&#8788;&#8709;'},
226                                Sequence{'act10','Is_Configuration_JToSeq&#8788;&#8709;'},
227                                self.getAct11(),
228                                self.getAct12(),
229                                self.getAct13(),
230                                self.getAct14(),
231                                self.getAct15(),
232                                self.getAct16(),
233                                self.getAct17()}});
```

Figure 4.11: The getInitialisationEvent() helper

applied. Animation tools are used to validate this assumption.

RQ2: How to assist the designer in selecting the correct configuration choices? Our proposed approach allows to guide the analyst by providing at each step the potential configuration choices. Concretely, this is achieved by using animation interface of the ProB tool. In fact, analyst is provided with our specification installed on the RODIN tool, and then he/she is able to generate process variants with respect to the allowed steps by the model checking. Hence, he/she should follow the enabled events to make their configuration choices. Our case study showed that the use of our approach was easy and saved users considerable time in identifying the configuration steps that are compliant not only to correctness constraints but also to domain recommendations.

Note that in order to automate our approach, we have developed a transformation tool to map BPMN process models into an Event-B specification.

The major benefit of this approach is the incremental verification procedure that allows the checking of the defined properties at each step of the development using generated Proof Obligations. POs are theorems that must be proved, in order to ensure that the developed specification is correct and consistent. Compared to other methods, Event-B has the advantage of rigorous reasoning tools assisting the verification task by modeling, proving and validating a system.

Although this approach has proven its effectiveness in assisting process configuration, the considered structural properties may be not sufficient to decide *soundness*. For instance, on the right hand side of the Figure 4.12, a structurally correct process variant is captured. In fact, each activity is on a path from the initial activity *a1* to the final one *a8* (i.e., structurally sound). Also, suppose that, as depicted by the figure, one has configured *s1* to *AND-split* and both *s2* and *s3* to *XOR-split*, then, when configuring *j1* and *j2* one can notice that our model allows their configuration

to *AND-join*. Hence, *s2* and *s3* are not their corresponding splits since there is no mismatching between them. The first common node of *s2* outgoings, for example, is neither *j1* nor *j2*, it is, instead, *j3*. Hence, the model is considered correct once *j3* is configured to *XOR-join*. Indeed, during execution, it is possible to activate *a3* and *a5*, which allows the activation of *a7* after their termination. However, this proper termination may be not preserved in all possible process instances; e.g. in cases where *a2* and *a5* are executed, neither *a6* nor *a7* could be enabled, which causes a deadlock. Thus, this process instance cannot terminate properly. In order to detect such problems, one should verify the behavior of the process instances. This implies the need for the analysis of the reachability graph of the BP. This graph refers to the representation of the different states that a process instance can take [3]. For this, formal execution semantics of the process modeling language (BPMN in our case) have to be taken into account. Since Event-B could hardly represent such dynamic semantics, we propose to adopt the more adapted Petri-net formalism. In the next chapter, we tackle the behavior verification of the process configuration relying on a Petri-net-based model and the SOG abstraction model. The SOG method [52, 53] will help to reduce the size of the reachability graph that may exponentially increase with the number of configurable elements.

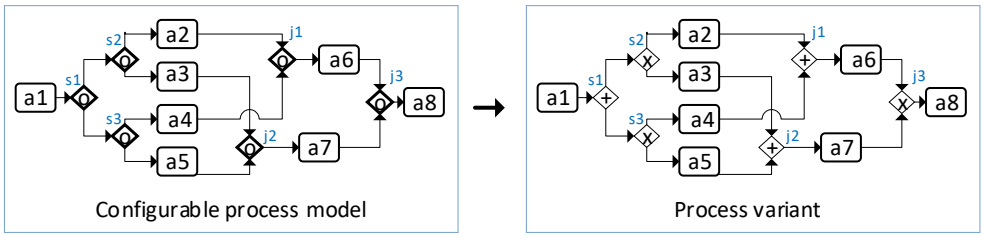


Figure 4.12: counter example

Chapter 5

Extracting Deadlock-free Process Variants using Symbolic Observation Graphs

Contents

5.1 Introduction 87
5.2 Approach Overview 89
5.3 Formal Model for Configurable Business Processes 89
5.3.1 Business Process Petri Nets (BP2PN) 90
5.3.2 Configurable Business Process Petri Nets (CBP2PN) 93
5.4 The Symbolic Observation Graph (SOG) 96
5.5 Extracting Deadlock-free Configurations using the SOG 99
5.6 Validation and Experiments 105
5.7 Discussion 107
5.8 Conclusion 109

5.1 Introduction

As we have discussed, process models configuration is notoriously difficult and error prone. Hence, the assistance and the verification of the configuration has become a must. In the Chapter 4, we presented an approach to assist the design of business process variants step-by-step during the configuration time. However, we especially focused on the designed configurable model structure. In fact, the verified constraints are expressed in terms of *invariants* specifying the variant structure that should be respected in order to prevent errors. For example, to avoid deadlocks, we cannot configure a join connector to a synchronization while its corresponding split was already configured to an exclusive choice. Whereas, structural verification may be not sufficient since some behavior anomalies may not be detected. As the Event-B method is not adequate to verify the dynamics of the possible executions of a process

variant, in this chapter, we choose to adopt new formal method (based on Petri-nets). Nevertheless, in the context of process configuration, checking the behavioral correctness of every single possible variant is obviously very time consuming and even unfeasible in case of large real-life models. This typically refers to solving the problem of an exponential number of states. So far, some approaches have attempted to verify behavioral correctness but have faced this exponential number of states (e.g. [10]). Very few have addressed the configuration behavior verification while trying to reduce this state-space explosion problem (e.g. [37, 39]) but still suffer from the exponential complexity of generating their reachability graph. This work aims to address this problem while verifying one of the most important behavioral correctness properties a process execution should hold, that is, the deadlock-freeness. Hence, in this chapter we aim at addressing three research questions: *RQ1: How to identify configuration choices that satisfy designers and clients requirements?*, *RQ2: How to assist the designer in selecting the correct configuration choices?* and *RQ3: How to avoid the state-space explosion of the configuration verification issue?*

In order to remedy the raised problems, we propose to use the Symbolic Observation Graph (SOG for short) [52, 53] to *verify* and *abstract* the representation of a configurable process model. The SOG is a versatile symbolic representation formalism that allows to build *an abstraction of the reachability state graph* of a formally modeled system (e.g. using Petri net). In our work, we start by adapting Petri-net formalism in order to formally represent a configurable process (cf. Section 5.3). Note that our work does not rely on specific Petri net properties but can be applied to any formal model as soon as states and transitions relations are well defined. Depending on the property or aspect we are interested in, the SOG abstraction is built over a particular set of defined observed elements, namely *Obs*. In our case, we are interested in the verification of process configurations, so the SOG is based on the set of its configurable elements.

This abstraction offers a two-fold advantage: (1) the analysis and the verification of the corresponding configurable process can be reduced to the analysis of its abstraction, and (2) the set of combinations of elements configurations that result in deadlock-free variants are obtained prior to configuration time. Once found, these combinations are used to *assist the business analyst in deriving deadlock-free variants*.

In the following, we start by giving an overall overview of our approach in Section 5.2. Then, the new Petri net-based models for business process models and for configurable process models as well as their semantics are defined in Section 5.3. In Section 5.4, we define the Symbolic Observation Graph associated with the defined configurable formal model. Afterwards, we illustrate our approach based on the SOG construction algorithm in Section 5.5. With the aim to prove the reduction of the configurable model state-space size, we conduct experiments in Section 5.6.

The work presented in this chapter was published in a conference proceedings [131].

5.2 Approach Overview

The SOG-based approach that we define to generate deadlock-free process variants consists of:

(1) defining a formal model for the configurable process model having concise and not ambiguous syntax and semantics, that we called *CBP2PN*. This semantics takes into account configurable connectors. For this aim, we rely on Petri nets, specifically the WF nets sub-class that we explained in Section 2.3.1. This will allow us to map an input C-BPMN process to a formal model having well-defined semantics (see Section 5.3).

(2) adapting and extending not only the SOG definition but also the construction algorithm of the SOG graph. In this extension three main points are considered: (i) *observe* and highlight configurable connectors that label the graph *arcs*; (ii) *hide* non-configurable elements' states in *aggregates*, i.e., the SOG nodes (see Section 5.4); and (iii) *restrict* the graph nodes to the ones belonging to paths leading to deadlock-free configurations (see Section 5.5). As a result, we obtain a reduced SOG graph that groups the behavior of all correct configurations. The set of correct configurations combinations is also extracted. We note that the extraction of correct configurations is performed on-the-fly during the SOG construction w.r.t the above defined points. Hence, using a developed tool that implements this algorithm with respect to the semantics defines in the first step, we obtain the set of correct configurations leading to deadlock-free variants.

(3) this set of correct configurations is finally supplied to the business analyst in order to derive deadlock-free variants, with no need to verify correctness at each intermediate configuration step.

Figure 5.1 illustrates the followed milestones, using our approach, in order to obtain a deadlock-free variant starting from, as depicted on the left-hand side of the figure, a configurable business process model in C-BPMN notation. In this chapter, we use the simplified example in Figure 5.2 for the illustration of our approach.

5.3 Formal Model for Configurable Business Processes

In order to obtain an abstract formal definition of a business process model, we formally map a process in BPMN notation to *Petri nets*, specifically into a new model called Business Process Petri Nets (*BP2PN*). It is an enriched version of a WF-net with new transitions representing the process connectors and having a specific type. Then, we extend the *BP2PN* to take into account configurable connectors, leading to a new model, namely the Configurable Business Process Petri Nets (*CBP2PN*). Authors in [74] have established a mapping from well-formed BPMN models to Petri nets. In

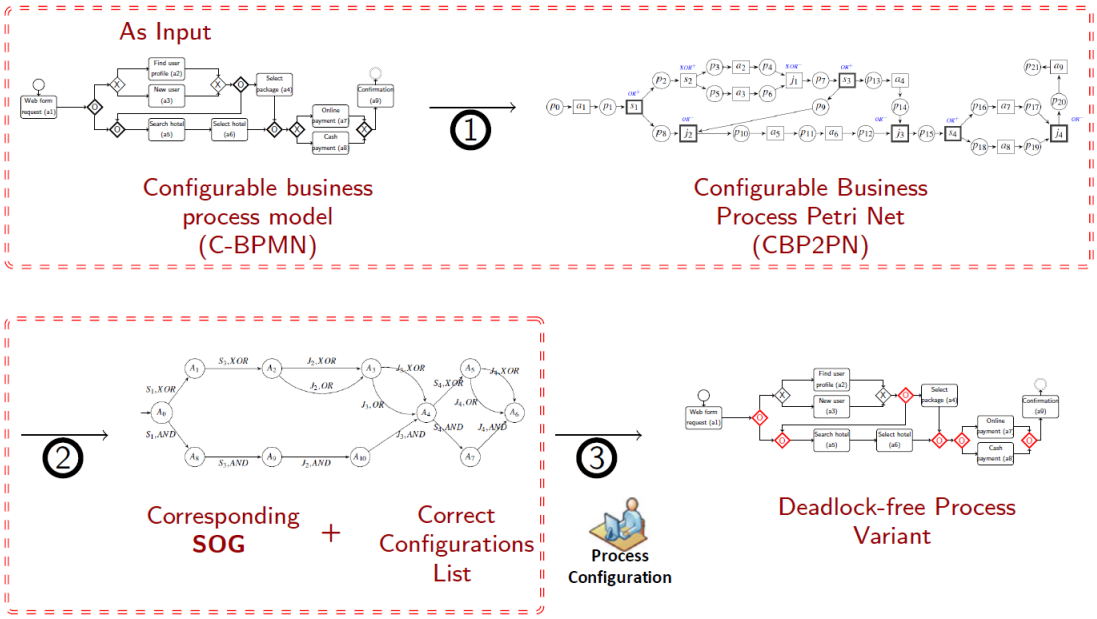


Figure 5.1: The SOG-based approach overview

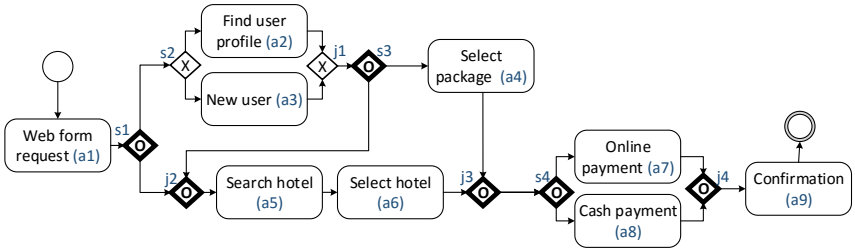


Figure 5.2: A configurable hotel booking process model

this work, we extend this mapping by preserving connectors blocks as transitions allowing to define configurable transitions.

5.3.1 Business Process Petri Nets (BP2PN)

A Business Process Petri Net is formally defined in Definition 5.3.1. Note that, we use the definitions and notations earlier presented for Petri nets and Workflow nets in Section 2.3.1.

Definition 5.3.1 (BP2PN). *A BP2PN is a tuple* $\mathcal{B} = \langle P, T \cup OP, F, W, O \rangle$ *where:*

- $\langle P, T \cup OP, F, W \rangle$ *is a WF-Net,*
- $F \subseteq (P \times T \cup OP) \cup (T \cup OP \times P)$ *is the flow relation,*

- $O : OP \rightarrow \{OR^-, OR^+, XOR^-, XOR^+, AND^-, AND^+\}$ *is a mapping that assigns a type to each operator/connector.*

BP2PN is a *Workflow net* such that, the set of places P corresponds to the set of conditions determining the enabling of a task or a connector; and the set of transitions $T \cup OP$ corresponds to the set of tasks T and connectors OP. These nodes are interconnected through a set of arcs (using F). Each connector must either be a join (the $-$ right exponent) or a split (the $+$ exponent) while having a type: OR, XOR or AND. In Figure 5.3 we represent the mapping of BP2PN connectors (both splits and joins) to standard Petri nets places, transitions and arcs.

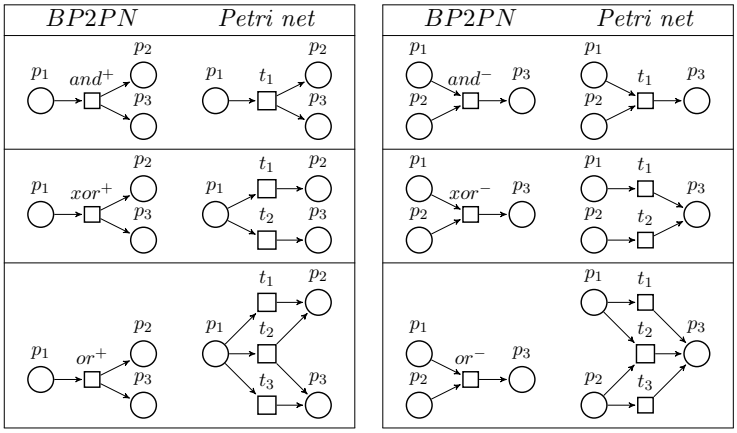


Figure 5.3: The BP2PN connectors mapping to classical Petri Nets

Semantics: In the previous notation, we retain the connectors blocks and we define new execution semantics inspired from the original semantics of Petri nets.

Given a marking m of a *BP2PN*, $\mathcal{B}$, the fireability and the firing of any transition in $T \cup \{t \in OP \mid O(t) \in \{AND^-, AND^+\}\}$ follows the original semantics of Petri nets presented in Section 2.3.1. However, a transition $t \in OP$ s.t. $O(t) \in \{OR^-, OR^+, XOR^-, XOR^+\}$ follows a new semantics:

Let m be a marking of $\mathcal{B}$, and t be a transition of OP, i.e., $t \in OP$. We denote by $m \xrightarrow{t}$ the fact that t is enabled by m, and we denote by $m \xrightarrow{t} m'$ the fact that m' is reached by firing t from m. Then the enabling rule as well as the firing of the transition t at the marking m leading to a marking m' are defined as follows.

- if $t \in OP \wedge O(t) = OR^-$
  - **Enabling rule:** m enables t iff $\exists S \subseteq {}^\bullet t$ s.t. $m_{|S} \geq W^-(t)_{|S}$
  - **Firing rule:** when m enables t, the firing of t from m leads to a marking m' iff $m' = m - W^-(t)_{|S} + W^+(t)$ where S is the biggest subset of ${}^\bullet t$ satisfying $m_{|S} \geq W^-(t)_{|S}$.

- if $t \in OP \wedge O(t) = XOR^{-}$
  - **Enabling rule:** m enables t iff $\exists p \in {}^{\bullet}t$ s.t. $m(p) \geq W^{-}(t)(p) \wedge \forall q \in {}^{\bullet}t,\ m(q) < W^{-}(t)(q)$
  - **Firing rule:** when m enables t, the firing of t from m leads to a marking m' iff $m' = m - W^{-}(t))_{|\{p\}} + W^{+}(t)$ where p is the sole place satisfying the firability condition.
- if $t \in OP \wedge O(t) = OR^{+}$ **(resp.** $O(t) = XOR^{+}$**)**
  - **Firing rule:** when m enables t, the firing of t from m leads to a marking m' iff $\exists S \subseteq t^{\bullet}$ **(resp.** $\exists p \in t^{\bullet}$**)** s.t. $m' = m - W^{-}(t) + W^{+}(t)_{|S}$ **(resp.** $m' = m - W^{-}(t) + W^{+}(t)_{|\{p\}}$**)**.

  Note that only the firing of transitions t s.t. $O(t) \in \{OR^{+}, XOR^{+}\}$ is defined because the enabling rule follows the original semantics.

Figure 5.4 illustrates the enabling and the firing of each connector type of BP2PN connectors, either a split or a join. For example, the enabling of a split connector, of any type, needs the input places to be sufficiently marked, same as standard Petri net transition. Similarly, the firing of an AND^{+} connector follows this semantics and produces tokens in every output place.

However, we adapted these rules in case of OR and XOR connectors. For instance, we emphasize that the semantics of a join transition OR^{-} is inline with the well defined *Pattern 8* in [132] (*Multi merge*), that expressly allows the firing of the join as soon as its condition is satisfied (without synchronizing the different flows), e.g., having the OR^{-} of Figure 5.4 with two inputs p_1 and p_2, one of the following markings enables the transition: m_1 (only p_1 is marked), m_2 (only p_2 is marked), or m_{1_2} (both places are marked). We note that, we are currently working on an extension of this work to consider the semantics of the *Pattern 7* (*Synchronizing Merge*). This semantics expressly impose that, first there is at least one token in at least one of its incoming branches, then it should be checked that for an incoming branch having no token, it is not possible for a token to reach this flow [74, 133].

It is worth mentioning also that the new semantics of OR^{+} and XOR^{+} leads to *non-deterministic firing*. For instance, having the split transition OR^{+} of the Figure 5.4 with two output places p_2 and p_3, its firing leads to 3 possible reachable markings: m_2 (only p_2 is marked), m_3 (only p_3 is marked), or m_{2_3} (both places are marked).

Since we are particularly interested in this chapter in verifying the deadlock-freeness property, we introduce the definition of this property in case of *BP2PN* as follows.

Definition 5.3.2 (Deadlock-free BP2PN)**.** *Let $\mathcal{B} = \langle P, T \cup OP, F, W, O\rangle$ be a BP2PN and m_i, m_f be the initial (i.e. only i is marked) and final (i.e. only o is marked) markings respectively. $\mathcal{B}$ is said to be deadlock-free iff $\nexists m \in (R(N, m_i) \setminus \{m_f\})$ s.t. $m \not\rightarrow$.*

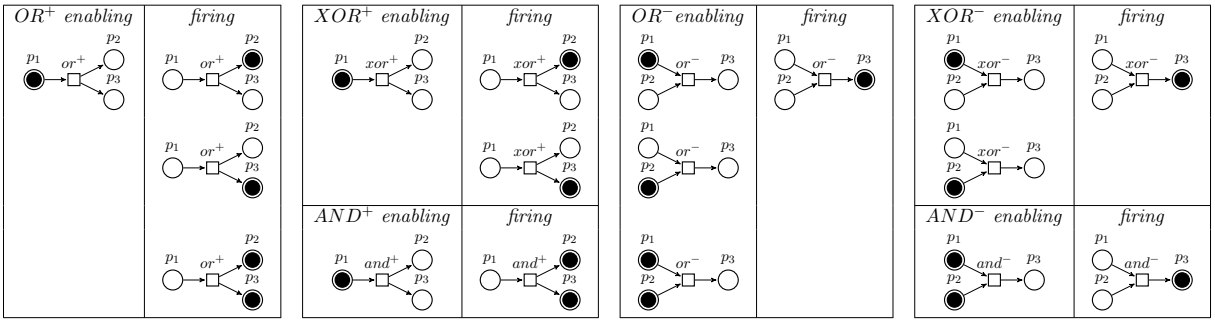


Figure 5.4: Enabling and Firing examples

Hence, according to this definition, a BP2PN, $\mathcal{B}$, is *deadlock-free* iff there is no dead marking m reachable from the initial marking m_i.

5.3.2 Configurable Business Process Petri Nets (CBP2PN)

Definition 5.3.3 (CBP2PN)**.** *A CBP2PN is a tuple* $\mathcal{CB} = \langle P, T \cup OP, F, W, O, C \rangle$ *where:*

- $\langle P, T \cup OP, F, W, O \rangle$ *is a BP2PN;*
- $C : OP \rightarrow \{true, false\}$ *is a function stating whether a connector is configurable or not. For instance, configurable connectors are any* $t \in OP$ *s.t.* $C(t) = true$*).*

Back to our example, the *C-BPMN* process model of Figure 5.2 is mapped onto *CBP2PN* in Figure 5.5. In this notation, according to Definition 5.3.3, activities and connectors are modeled by transitions and their ordering is modeled by places connecting the different transitions. Configurable transitions are also highlighted with a thick border. This example includes 6 configurable transitions: s_1, s_3, s_4, j_2, j_3 and j_4.

We denote by OP^c the set of configurable operators s.t. $OP^c = \{o \in OP \mid C(o) = true\}$. A configurable operator $c^c \in OP^c$ includes a generic behavior which is restricted using the configuration phase. It is configured by changing its type (e.g. from OR to AND) w.r.t. the set of configuration constraints [1] defined in Section 2.2.2. Here, we recall these constraints in Table 5.1 while adopting the notations of the *BP2PN* connectors in Definition 5.3.1. Each row corresponds to a configurable connector that can be configured to one or more of the connectors in columns. Thus, these constraints allow to specify which regular connector's type may be used in the derived process variant. For example, a configurable *OR* can be configured to any connector's type, while a configurable *AND* can only be configured to an *AND*. As can be noticed, we omit the sequence configuration column in this table, as in this work we are only interested in the configuration of connectors by changing their types. As

an extension to this work, we are currently working on the connectors configuration by restricting output or input branches, as well as the activity configuration (i.e. to ON or OFF).

In the following, we define a configuration of a connector $t^c \in OP^c$ by $Conf(t^c) \in \{OR^-, OR^+, XOR^-, XOR^+, AND^-, AND^+\}$ and the set of all possible configurations of t^c by $AllConf(t^c)$.

Table 5.1: Connectors configuration constraints [1] having $x \in \{+, -\}$.

FROM-TO	OR^x	XOR^x	AND^x
OR^x	√	√	√
XOR^x		√	
AND^x			√

Note that, when configuring all configurable connectors of a *CBP2PN*, we obtain a *BP2PN*, as a configurable connector is changed into regular connector after configuration. One possible configuration of the process net of Figure 5.5 is depicted by Figure 5.6. This variant is derived by selecting the following configuration choices: (i) s_1, s_3 and s_4 are configured to regular XOR^+, (ii) j_2 is configured to a regular AND^-; and (iii) j_3 and j_4 are configured to regular XOR^-. As mentioned previously, the resulting process is a *BP2PN* since it does not contain any configurable transitions.

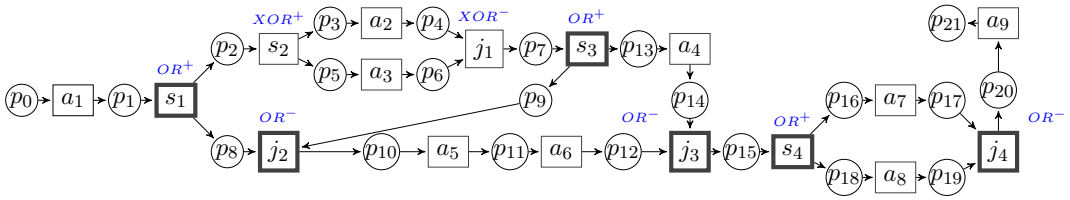


Figure 5.5: The CBP2PN of the configurable process in Figure 5.2

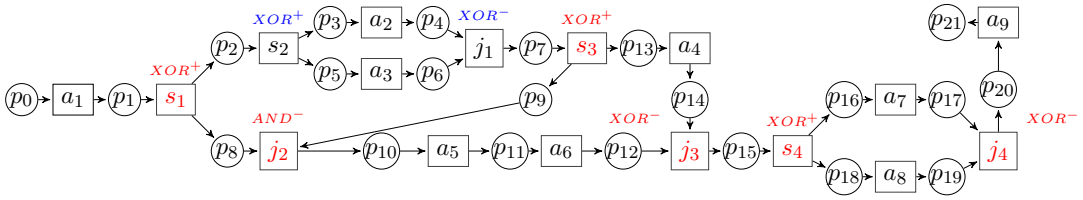


Figure 5.6: A possible variant of the CBP2PN in Figure 5.5

Semantics: The semantics of *CBP2PN* is described in the following, on the one hand, by inheriting the dynamics of *BP2PN* for non configurable connectors, on the other hand, by adding new semantics for configurable ones. This semantics is

defined such that any reachable marking by any possible instance of a configuration is represented. In the following, we consider a configurable transition as the union of all its possible configurations. That way, we can define its enabling and firing rules as if it is the union of all executable configured transitions. Since a configuration of AND^-, AND^+, XOR^- and XOR^+ do not change type, its semantics remains the same as previously defined. Regarding configurable OR^- and OR^+ transitions, the fireability and the firing rules follow the new semantics as follows.

Let m be a marking and t^c be a transition of OP^c, s.t. $O(t^c) \in \{OR^-, OR^+\}$:

- **Enabling rule:** m enables t^c, denoted by $m \xrightarrow{t^c}$ iff $\exists x \in AllConf(t^c)$ s.t. $m \xrightarrow{x}$
- **Firing rule:** when m enables t^c, for some configuration x of t^c, the firing of t^c from m, under configuration x, leads to a marking m', denoted by $m \xrightarrow{t^c,x} m'$ iff $m \xrightarrow{x} m'$

Using this semantics, the reachability marking graph associated with a *CBP2PN* covers the behavior of all the possible configurations. For instance, having the *CBP2PN* of Figure 5.5, the configurable transition $s1$ could be configured either to: (i) AND^+, with all of its output places marked, (ii) XOR^+, with only one of the output places marked, or (iii) OR^+ with one or more output places marked (see Figure 5.7).

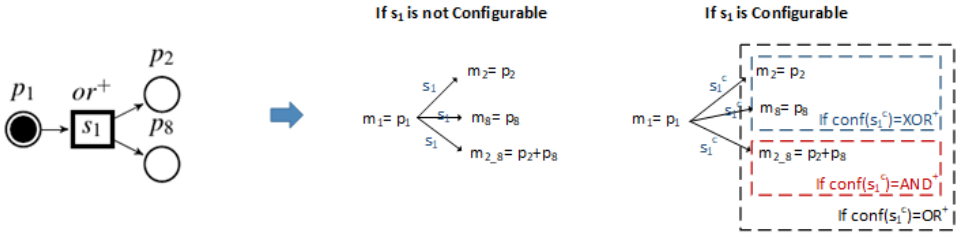


Figure 5.7: Markings graph example in case of a non-configurable and a configurable transition exhibiting non-determinism

Definition 5.3.4 (Deadlock-free CBP2PN)**.** *Let $\mathcal{CB}$ be a CBP2PN. $\mathcal{CB}$ is said to be deadlock-free if at least one deadlock-free BP2PN could be configured from it.*

Our *CBP2PN* of Figure 5.5 is considered to be deadlock-free since one can configure at least one correct variant by choosing XOR type as configuration choice for all its configurable connectors (the correctness of such a variant is proven in Section 5.5). However, incorrect variants could be derived from this process as well. For instance, one can choose the alternatives presented earlier of Figure 5.6 that leads to a deadlock caused by an exclusive choice XOR^+ (i.e. s_1) followed by a synchronizing join AND^- (i.e. j_2). In this situation, in order to be enabled, the transition AND^- will be waiting for both places p_8 and p_9 to be marked, however only one could be

marked. So, the resulting variant could never terminate properly and the corresponding reachability graph contains a dead marking (i.e., from which no transition could be enabled).

In this work, we aim to check the behavior correctness of all possible configurations of a configurable model *CBP2PN*. This refers to verifying the reachability graph that covers them all. Obviously, this leads to the well known state space explosion problem. In order to reduce the underlying problem, we propose to use the Symbolic Observation Graph (SOG). In the following section, we formally define the SOG graph. The SOG-based abstraction technique was introduced for model checking of concurrent systems [52] and then applied on the verification of inter-enterprise business processes [91].

5.4 The Symbolic Observation Graph (SOG)

The *Symbolic Observation Graph* was initially introduced in [52] as an abstraction of the *reachability state graph* of concurrent systems. An event-based verification was applied on formula of $LTL \setminus X$ (LTL minus the next operator). Then, the SOG was extended to its state-based form in [53].

In this section, we define the SOG based on our formal model introduced in Section 5.3, namely the *CBP2PN*, and then we will see in the next sections how to use them in our verification approach.

Depending on a specific property we are interested in, the *SOG* is built over a particular set of *observed elements*, denoted by *Obs*. In our work, we are interested in observing the configuration behavior, that is why we define *Obs* as the set of *configurable transitions*. since only configurable connectors are considered in this work , given a *CBP2PN*, we define *Obs* as follows: $Obs = OP^c$. Whereas, the other transitions belongs to the set of unobserved ones, denoted by *UnObs*, this means that $UnObs = (T \cup OP) \setminus Obs$.

In such a way, as illustrated in Figure 5.8, we construct *the Symbolic Observation Graph (SOG)* as a graph where each node is a set of states linked by unobserved transitions, and each arc linking two nodes is labeled by an observed transition. Nodes of the SOG are called *aggregates* and are represented and managed efficiently using Binary Decision Diagrams (BDDs). As a result, by highlighting configurable transitions, the SOG represents the global behavior of a process configuration in only one reduced graph.

In the following, we present the adaption of the SOG to our process configuration issue. So, firstly we formally define the new aggregation criterion in Definition 5.4.1 such that: (1) the states belonging to the same *aggregate* must be be linked by non configurable transitions, and (2) the firing of a configurable transition by a state in an *aggregate* must lead to another *aggregate*. Hence, *an aggregate* is defined as follows.

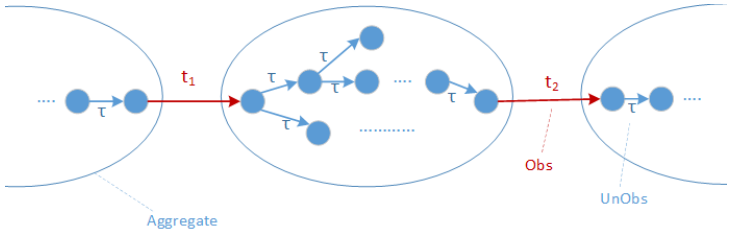


Figure 5.8: The Symbolic Observation Graph (SOG)

Definition 5.4.1 (Aggregate)**.** *Let $N = \langle P, T \cup OP, F, W, O, C \rangle$ be a CBP2PN having m_i and m_f as initial and final markings respectively. An aggregate A of N w.r.t. Obs is a triplet $\langle S, d, f \rangle$ s.t.:*

- *$S \subseteq R(N, m_i)$ is a set of reachable markings, where $\forall s \in S$:*
  - *$(\exists (s', u) \in R(N, m_i) \times UnObs \mid s \xrightarrow{u} s') \Leftrightarrow s' \in S$;*
  - *$(\exists (s', o) \in R(N, m_i) \times Obs \mid s \xrightarrow{o} s') \wedge (\nexists (s'', u) \in S \times UnObs) \mid s'' \xrightarrow{u} s') \Leftrightarrow s' \notin S$.*
- *$d \in \{true, false\}$; $d = true$ iff S contains a dead state.*
- *$f \in \{true, false\}$; $f = true$ iff S contains a final state (i.e. $m_f \in S$).*

In addition to the d and f attributes of an aggregate, the above definition specifies the states that must belong to an aggregate (the aggregation criterium) and those that must be excluded: (1) For any state s in the aggregate, any state s' being reachable from s by the occurrence of an unobserved transition, belongs necessarily to the same aggregate. (2) For any state s in the aggregate, any state s' which is reachable from s by the occurrence of an observed transition is necessarily outside the aggregate, unless s' is reachable from a state s'' in the aggregate by an unobserved transition. In the following, we denote by $A.S$, $A.d$ and $A.f$, the attributes of an aggregate A.

Before providing the definition of the SOG associated with a *CBP2PN*, let us introduce the operations *Sat* and *Out* as follows.

- $Sat(S, UnObs)$ calculates the closure of a set of markings S by the set of unobserved transitions $UnObs$. In other words, it returns the set of reachable markings from any marking in S, by unobserved transitions. It is formally defined as follows:

$$Sat(S, UnObs) = \{s' \in R(N, m_i) \mid \exists s \in S \ \wedge \ \exists t \in UnObs, \ s \xrightarrow{t} s'\}$$

- $Out(a, t)$ returns, for an aggregate a and an observed transition t, the set of states that are outside of a and reachable from some state in a by firing t. It is formally defined as follows:

$$Out(a, t) = \{s' \in R(N, m_i) \mid \exists s \in a.S, \ s \xrightarrow{t} s'\}$$

Definition 5.4.2 (Deterministic SOG)**.** *Let $N = \langle P, T \cup OP, F, W, O, C\rangle$ be a CBP2PN having m_i and m_f as initial and final markings respectively. The Deterministic Symbolic Observation Graph (SOG) associated with N is a graph $\mathcal{G} = \langle A, Obs, \rightarrow, A_0, \Omega\rangle$ where:*

(1) A is a non empty finite set of aggregates satisfying :

- $\forall a \in A,\ \forall t \in Obs,\ Out(a,t) \neq \varnothing \implies \exists a' \in A \text{ s.t. } a' = Sat(Out(a,t), UnObs)$

(2) $\rightarrow \subseteq A \times Obs \times A$ is the transition relation where:

- $((a,t,a') \in \rightarrow') \Leftrightarrow ((t \in Obs) \wedge Out(a,t) \neq \varnothing \wedge a' = Sat(Out(a,t), UnObs))$

(3) A_0 is the initial aggregate s.t. $A_0.S = Sat(m_i, UnObs)$.

(4) $\Omega = \{a \in A \mid m_f \in a.S\}$.

The nodes of the symbolic observation graph are aggregates (1). The finite set of aggregates A of a SOG is defined in a complete manner so that the necessary aggregates are represented. Point (2) defines the transitions relation: there exists an arc, labeled with an observed transition t, from a to a' iff a' is obtained by saturation (by applying Sat) on the set of reached states (obtained using $Out(a,t)$) by the firing of t from $a.S$. The last two points of Definition 5.4.2 characterize the initial aggregate A_0 and the set of final aggregates Ω (i.e. aggregates containing the final marking) respectively.

Starting from the initial marking, the original SOG construction algorithm introduced in [52] follows a classical depth-first search based traversal of the built aggregates. Each aggregate is built by a transitive closure (using Sat) application on unobserved transitions. The successor a' of an aggregate a is built by, first, firing an observed transition from states of a, then by adding all the reachable states by unobserved transition.

At this stage, the correctness of the *SOG* can be formally characterized as follows.

Definition 5.4.3 (Correct SOG)**.** *Let $N = \langle P, T \cup OP, F, W, O, C\rangle$ be a CBP2PN. Let $\mathcal{G} = \langle A, Obs, \rightarrow, A_0, \Omega\rangle$ the SOG associated with N.*

*$\mathcal{G}$ is correct **iff** there exists a configuration c of N ($c = \{\langle t, Conf(t)\rangle : t \in OP^c\}$) s.t. for every path $\pi = A_0 \xrightarrow{t_1, conf(t_1)} A_1 \ldots A_{n-1} \xrightarrow{t_n, conf(t_n)} A_n$, with $A_n \in \Omega$; if $\{\langle t_i, Conf(t_i)\rangle : 0 \leq i \leq n\} = c$ then $\forall 0 \leq i \leq n, A_i.d = false$.*

Based on Definition 5.3.4, characterizing a deadlock-free *CBP2PN*, and Definition 5.4.3, characterizing a correct *SOG* associated with a *CBP2PN*, the following result naturally links these two characterizations.

Proposition 5.4.1. *Let $N = \langle P, T \cup OP, F, W, O, C\rangle$ be a CBP2PN. Let $\mathcal{G} = \langle A, Obs, \rightarrow, A_0, \Omega\rangle$ the SOG associated with N. Then, N is deadlock-free **iff** $\mathcal{G}$ is correct.*

Proof. Let N be a *CBP2PN* and $\mathcal{G}$ its corresponding *SOG*. First, according to Definition 5.4.3, if $\mathcal{G}$ is correct then there exists a configuration c s.t. for every path π in the *SOG* having $\pi = A_0 \xrightarrow{t_1,conf(t_1)} A_1 \dots A_{n-1} \xrightarrow{t_n,conf(t_n)} A_n$, with $A_n \in \Omega$; if it's configurations set $\{\langle t_i, Conf(t_i)\rangle : 0 \leq i \leq n\}$ is equal to c, then all aggregates are deadlock-free, i.e. $A_i.d = false, 0 \leq i \leq n$. Since the *SOG* preserves by construction all possible configurations of N, then each path from the initial to the final aggregate represents one configuration allowing to derive one variant. Hence, there exist at least a deadlock-free variant of N. Consequently, according to Definition 5.3.4, N is correct. □

In the following, we propose to adapt the original SOG construction algorithm [52], associated with a *CBP2PN*, in three ways. First, by adopting the new semantics. Second, the deadlock-freeness property is checked on the fly, such that any aggregate containing a deadlock state is not inserted in the graph and so are all the underlying paths. Finally, the set of correct configurations is extracted on-the-fly.

5.5 Extracting Deadlock-free Configurations using the SOG

In this section, we present the core contribution of this work: The construction algorithm of the SOG associated with a *CBP2PN*. Campared to the original SOG construction algorithm [52], Algorithm 5.10 allows to reduce the SOG, by **removing**, on-the-fly, the paths involved in *incorrect* configurations, and by **saving**, within the initial aggregate the *correct* configurations.

Used data: To reach this goal, we firstly define the different data used in this algorithm. As input, a *CBP2PN*, namely N, the set of the observed transitions *Obs* as well as the initial and the final markings m_i, m_f are introduced.

Then, we add two new attributes to an aggregate $a \in A$:

- ***c*** is the set of correct (possibly partial) configurations, starting from the aggregate a and leading to a final aggregate;
- ***nc*** is the set of incorrect (possibly partial) configurations, starting from the aggregate a and leading to a dead one (i.e. including a dead state).

Once the *SOG* is built, the set of correct configurations will be saved within the *initial* aggregate.

In addition to this set of configurations C, the algorithm also returns as output the SOG graph $\mathcal{G}$, containing aggregates (in A set), and edges (in $\rightarrow$ relation).

Another fundamental data used in the SOG construction is the *stack st*, containing a couple of the to-be-treated aggregates associated with the set of its fireable observed transitions F_{obs}.

Algorithm 1 Deadlock-free Symbolic Observation Graph

Require: $N\langle P, T \cup OP, F, W, O, C\rangle$, Obs, m_i, m_f
Ensure: $\mathcal{G}\langle A, Obs, \rightarrow, A_0, \Omega\rangle$, C

1: Vertices $A{=}\varnothing$; vertex a, a'; {Aggregates}
2: Vertices $C{=}\varnothing$; {Correct configurations}
3: set S, S', $UnObs = (T \cup OP) \setminus Obs$, F_{obs}, F'_{obs};
4: stack st; Edges E= $\varnothing$;
5: $S = Sat(\{m_i\}, UnObs)$; {first Aggregate}
6: $a.S = S$;
7: $a.d = DetectDead(a.S)$;
8: $a.f = IsFinal(a)$;
9: $F_{obs} = fireableObs(a)$; {fireable observed transitions of a}
10: $st.Push(\langle a, F_{obs}\rangle)$;
11: **while** $st \neq \varnothing$ **do**
12: $\langle a, F_{obs}\rangle = st.Top()$;
13: **if** $(F_{obs} \neq \varnothing)$ **then**
14: $t = F_{obs}.next()$;
15: $S' = Out(a.S, t)$
16: **if** $(S' \neq \varnothing)$ **then**
17: $S' = Sat(S', UnObs)$;
18: $a'.S = S'$;
19: $a'.d = DetectDead(a'.S)$;
20: $a'.f = IsFinal(a')$;
21: **if** $(\neg a'.d)$ **then** {there is no dead state in a'}
22: **if** $(\nexists x \in A$ s.t. $x == a')$ **then** {a' found for the first time}
23: $F'_{obs} = fireableObs(a')$;
24: $st.Push(\langle a', F'_{obs}\rangle)$;
25: **else** {a' is an existing aggregate}
26: free a';
27: Let a' be the already existing aggregate;
28: $UpdateC(a, a', t)$;
29: $UpdateNC(a, a', t)$;
30: **end if**
31: $\rightarrow = \rightarrow \cup\{a, \langle t, Conf(t)\rangle, a'\}$;
32: **else** {there is a dead state in a'}
33: $a.nc = a.nc \cup \{\langle t, Conf(t)\rangle\}$;
34: $recRemoveAggregate(a, t)$
35: **end if**
36: **end if**
37: $a.c = a.c \cup \{\langle t, Conf(t)\rangle\}$;
38: $CompareCorrect(a)$;
39: $st.Pop()$;
40: $A = A \cup \{a\}$;
41: **if** $(m_i \in a.S)$ **then**
42: $C = a.c$;
43: **end if**
44: **end if**
45: **end while**

Algorithm steps: In the following, we go through Algorithm 5.10 to explain the main steps. We use for illustration our running example and the corresponding resultant reduced SOG in Figure 5.9. Note that the main novelties of this algorithm w.r.t. the algorithm of [52], are underlined.

The first step of Algorithm 5.10 (lines $5-10$) allows to build the initial aggregate A_0 by applying the saturation on the initial marking and to push it onto the stack. Then, the main loop (lines $11-45$) processes the set of to-be-treated aggregates as follows: First, a stack item (i.e., $\langle a, F_{obs}\rangle$) holding the current aggregate a and its F_{obs}, is picked (line 12). Then, the corresponding current observed transition t belonging to F_{obs}, if any, is selected (line 14), and the successor a' of the current aggregate a by that transition t is calculated (lines $15-20$). Note that, the calculation of the reachable states by using the operations Sat and Out respects the semantics defined in Subsection 5.3.2. This includes the computation of the dead (line 19) and final (line 20) attributes, i.e., $a'.d$ and $a'.f$ respectively (cf. Definition 5.4.1), of the obtained successor aggregate. The function $DetectDead$ checks if there exists a dead state $s \in a'.S$ from which neither transition could be enabled (i.e. $Enable(s) = \varnothing$).

If the latter checked aggregate is deadlock-free (line 21), and if it has not already been explored (does not exist in the aggregates set A) (line 22), then it is pushed onto the stack with its set of fireable observed transitions F_{obs} (lines $23-24$).

For instance, following the path at the top of Figure 5.9, the new aggregates: A_0 until the final one A_6 are consecutively pushed onto the stack one after the other. Since A_6 is a final aggregate (i.e., does not enable any observable transition), the corresponding configuration is considered correct (line 37) and it is popped from the stack (line 39), and then we start the loop again by picking A_5 to consider its remaining observed transitions (in this case the transition $\langle j4, OR\rangle$ leads again to A_6). After that, since $\langle j4, AND\rangle$ can not be fired from A_5, we pick A_4, and so on.

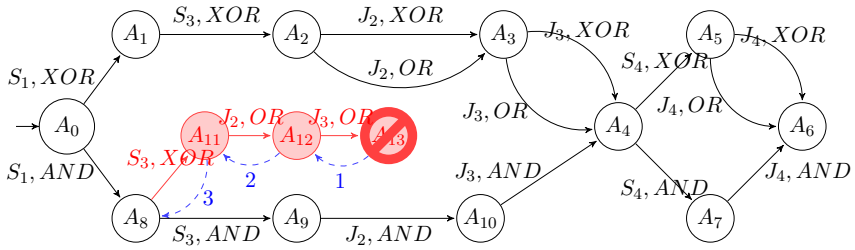


Figure 5.9: The obtained reduced SOG of the CBP2PN in Figure 5.5

If the newly built successor aggregate a' has been already treated and added to aggregates set (lines $25-30$). Then, as illustrated in Figure 5.10, the current aggregate a inherits from a' its correct and incorrect configurations, c and nc respectively. In the figure, inside an aggregate, the green list represent the correct configurations and the red one represent the incorrect configurations. Obviously, we add to these

configurations the transition linking a to a'. In our Algorithm, this is ensured by the functions *UpdateC* and *UpdateNC* (lines 28 – 29).

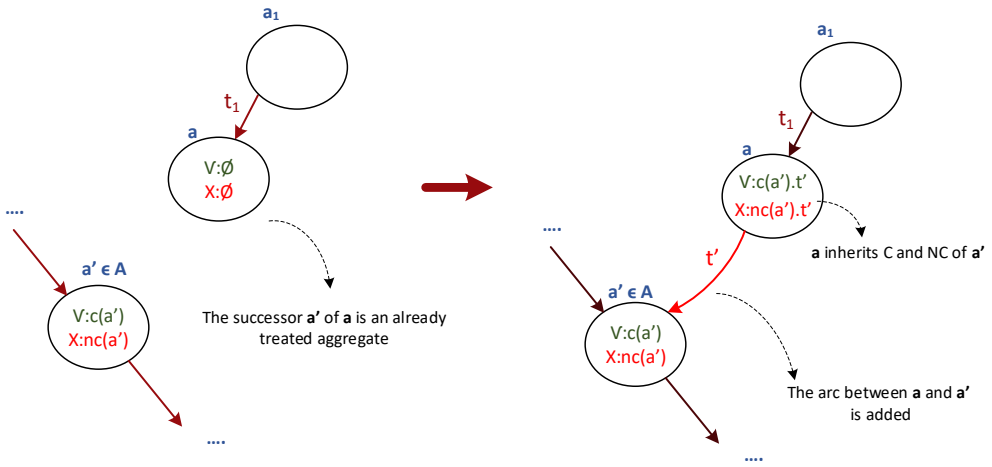


Figure 5.10: The case of a successor aggregate already treated

The function *UpdateC* also verifies that, starting from the same aggregate a, a correct configuration do not include an existing (or to-be-treated) incorrect one, as in this case it leads to a deadlock in a different transitions' firing order. Regarding our SOG in Figure 5.9, consider the aggregate A_{10} obtained through A_8 and A_9, the firing of $\langle j3, AND\rangle$ leads to the existing aggregate A_4. As A_4 was already dealt with earlier through the path on top of the graph, this means that 3 correct *partial* configurations are added to this aggregate, namely $\{\langle s4, XOR\rangle,\langle j4, XOR\rangle\}$, $\{\langle s4, XOR\rangle, \langle j4, OR\rangle\}$ and $\{\langle s4, AND\rangle,\langle j4, AND\rangle\}$. Hence, A_{10} inherits these configurations while being concatenated to the current fired transition $\langle j3, AND\rangle$.

Regarding an aggregate a' holding a dead state, firstly, the corresponding fired observed transition is concatenated to the incorrect configurations of its predecessor a (line 33). Obviously, a' is not pushed onto the stack and no edge is created. Then, we recursively verify its predecessors starting from a using the function *recRemoveAggregate*(a, t) (line 34). Using this function, each predecessor aggregate is removed only if the states enabling the current one becomes dead (i.e. there is no other enabled transition from that state). In this case, its successors are also recursively eliminated in case they do not have other predecessors. As an example, the red path in Figure 5.9 refers to firing $\langle s_1, AND\rangle$, $\langle s_3, XOR\rangle$ then $\langle j_2, OR\rangle$. According to our semantics, $\langle j_3, OR\rangle$ may be fired by 4 possible markings in the aggregate A_{12}, namely m_{12} (i.e., the place *p12* marked), m_{10_14} (i.e., both places *p10* and *p14* are marked), m_{11_14} and m_{12_14}. However, in case of firing by either m_{10_14} or m_{11_14}, the obtained aggregate will allow a second firing of the same transition (i.e. using the remaining token in p_{10} or p_{11}). This leads to a final state holding two tokens, which is a dead state in our approach. Hence, according to Algorithm 5.10 the obtained

aggregate is eliminated as well as its predecessors A_{12} and A_{11} (following the blue dashed line). And yet, since it enables $\langle S_3, AND \rangle$, A_{10} is not deleted. Similarly, going backwards to A_0 after entirely processing A_8 and A_9, we obtain the complete correct configurations $13 - 15$ depicted in Table 5.2.

To better explain the evolution of the sets of correct and incorrect configurations in the aggregates, we illustrate the steps of pushing and popping aggregates to/from the stack using an example in Figure 5.11. First of all, starting from the calculated initial aggregate a_0, aggregates are consecutively pushed into the stack (step 1). Initially, the set of correct configurations as well as incorrect ones are empty (represented in the figure using green and red lists inside the aggregates). Then, when the final aggregate a_7 (holding the final marking) is found, it is popped from the stack (since no more transitions may occur) and the configuration $c6$ is added to the correct configurations of its predecessor a_6 (step 2). Next, a_6 is picked to inspect if there remains fireable observed transitions, which is not the case. So, it is also popped and the set of correct configurations of a_5 is updated by adding $c5$. Here, a_5 still have a fireable transition that is $c7$. Hence, we push the aggregates a_8 and a_9 and we find out that the latter holds a dead state. So, it is popped from the stack (step 6), as well as a_8, however in this case the set of incorrect configurations is updated at each step. Thus, a_5 includes two partial configurations one correct and another one incorrect (step 7). Similarly, we check the remaining transitions and so on. This way, correct and incorrect configurations are computed backwards starting from the final aggregate to the initial one. Hence, the initial aggregate will hold the complete sets of configurations.

It is worth noting that before popping an aggregate from the stack and storing it in the graph (lines $39 - 40$), a final check is carried out on its correct configurations by the function *CompareCorrect* (line 38). Actually, many observed transitions may be fired from the same aggregate, so some of the corresponding correct configurations may refer to the same one. Hence, a correct sequence is preserved if, for every first fired observed transition *op*, (i) it is fireable by the states that have fired another sequence starting by *op* (i.e. different configurations), or (ii) if their common operators have the same configured type (i.e. the same configurations but in a different order). Otherwise, the sequence is considered as incorrect and is eliminated.

Finally, the set of correct configurations is obtained from the initial aggregate, the last one popped from the stack. As a result, each path of the obtained SOG starting from the initial aggregate and leading to a final aggregate, represents one possible configuration and belongs to the set of configurations C. In this case, this configuration leads to a deadlock-free *BP2PN*. Note that, different paths could represent a configuration (e.g. two concurrent configurable connectors).

Usage: The reduced SOG of our example contains 8 nodes and 10 arcs, and all correct configurations are summarized in Table 5.2. Hence, the analyst may be helped on-the-fly during the configuration process by confronting his/her configurations with

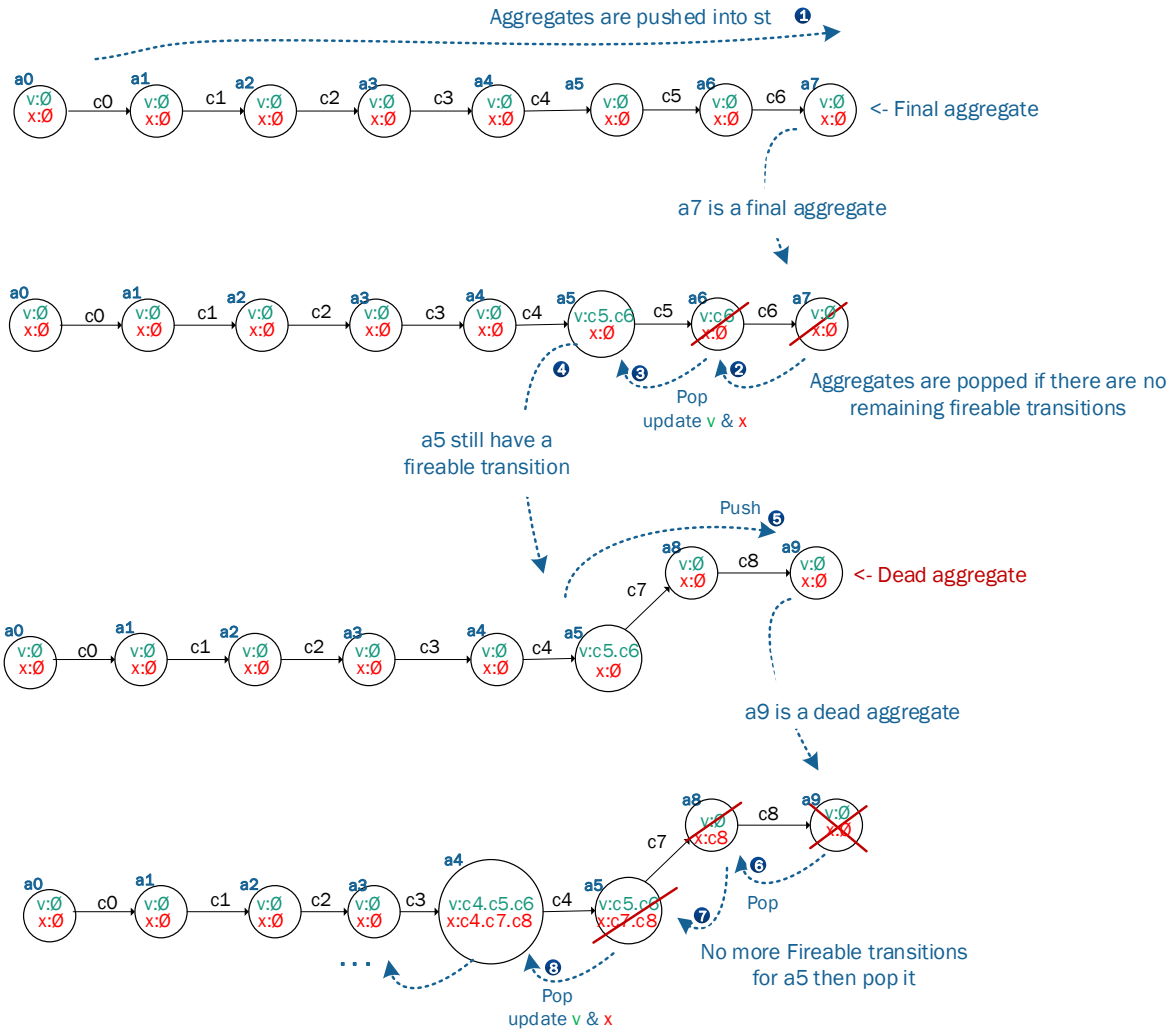


Figure 5.11: An example illustrating aggregates configurations lists when pushing and popping them from the stack

the correct configurations in this table.

For instance, we can evaluate the correctness of the *BP2PN* variant discussed in Section 5.3.2. After applying $\langle s_1, XOR \rangle$, the control-flow is either propagated through the place p_2 or p_8. In this case, it is clear that the connector j_2 (i.e. after applying $\langle j_2, AND \rangle$) could never be enabled, which causes a deadlock. Relying on Table 5.2, we can notice that there is no configuration starting with $\{\langle s_1, XOR \rangle, \langle j_2, AND \rangle\}$.

Using the SOG, the state space is greatly reduced in three fashions: (i) only configurable transitions are observed, and the remaining transitions are hided in aggregates; (ii) the graph is deterministic since it groups, for each configuration, all

reachable markings in one aggregate; and (iii) the different process variants share common markings in one common SOG graph, instead of constructing graphs as much as the number of possible configurations. In the following section, we conduct experiments to demonstrate such mitigation of the state explosion problem as well as the feasibility of our approach.

Table 5.2: Deadlock-free extracted configurations for the *CBP2PN* in Figure 5.5

	S1	***S2***	***J2***	***J3***	***S4***	***J3***
1	XOR^+	XOR^+	XOR^-	XOR^-	XOR^+	XOR^+
2	XOR^+	XOR^+	XOR^-	XOR^-	XOR^+	OR^+
3	XOR^+	XOR^+	XOR^-	XOR^-	AND^+	AND^+
4	XOR^+	XOR^+	XOR^-	OR^-	XOR^+	XOR^+
5	XOR^+	XOR^+	XOR^-	OR^-	XOR^+	OR^+
6	XOR^+	XOR^+	XOR^-	OR^-	AND^+	AND^+
7	XOR^+	XOR^+	OR^-	XOR^-	XOR^+	XOR^+
8	XOR^+	XOR^+	OR^-	XOR^-	XOR^+	OR^+
9	XOR^+	XOR^+	OR^-	XOR^-	AND^+	AND^+
10	XOR^+	XOR^+	OR^-	OR^-	XOR^+	XOR^+
11	XOR^+	XOR^+	OR^-	OR^-	XOR^+	OR^+
12	XOR^+	XOR^+	OR^-	OR^-	AND^+	AND^+
13	AND^+	AND^+	AND^-	AND^-	XOR^+	XOR^+
14	AND^+	AND^+	AND^-	AND^-	XOR^+	OR^+
15	AND^+	AND^+	AND^-	AND^-	AND^-	AND^-

5.6 Validation and Experiments

To prove its feasibility, we have implemented and deployed our approach as an extension of an existing tool that initially computes the SOG of a petri-net model w.r.t. a set of observed transitions. As explained previously, this extension takes into account the new semantics presented in this chapter for *CBP2PN* models. It also allows to symbolically detect on-the-fly deadlocks within aggregates and to reduce the SOG accordingly.

The developed tool takes as input a GrML (Graph Markup Language) file [134], an XML file describing the *CBP2PN* model (i.e. transitions, operators annotated as configurable, and arcs) and returns the reduced SOG and the correct configurations. The translation of CBP2PN model into GrML file is done as follows. Each model element is associated with a node tag in this file. These tags are differentiated using the value of the attribute *"nodeType"*: (1) *transition* have three defined at-

tributes: "name", "configurable" (specifying configurable transitions using a Boolean value), "operator" (defining an operator type: ORPLUS, ORMOINS, ANDPLUS, ANDMOINS, ORMOINS, XORMOINS, or *null* in case of a regular transition), and "observed" (specifying observed transitions using a Boolean value); (2) *"place"* have two defined attributes: "name" and "marking" (defining the marking at the initial state); and (3) *"arc"* have the attribute "valuation" that defines its weight. Finally, we define the attribute *"finalMarking"* defining the final place and its possible number of tokens.

In order to evaluate its performances and to demonstrate the opportunities offered by our approach, we performed experiments to show (i) the reduction of the space explosion problem and (ii) the impact of the input model structure on the size of the obtained SOG. Firstly, we propose to explore the size of the constructed SOG using our tool against a naive approach, where each variant of a *CBP2PN* is built and analyzed separately. Secondly, we propose to analyse the impact of the variation of the structure complexity and the number of observed transitions of a *CBP2PN*, on the size of the corresponding SOG. Taking our running example model (cf. Figure 5.5), this variation leads to 86 different process models. We basically evaluate the structure complexity using the well known metric CFC (Control Flow Complexity) [135] which is defined as:

$$CFC = \sum_{c \in AND^+} 1 + \sum_{c \in XOR^+} |c^\bullet| + \sum_{c \in OR^+} (2^{|^\bullet c|} - 1)$$

Table 5.3 contains three multi-columns. The first one varies the considered parameters of the *CBP2PN* model (i.e. CFC and observed transitions (Obs)) and gives the number of possible configurations for each variation. Then, the size of the obtained SOG is evaluated in terms of number of correct configurations (Nb correct confs), aggregates (A), edges (E) and execution time. This graph is finally compared against the naive approach. However, since the naive approach is very fastidious, we built only the reachability graphs corresponding to the correct configurations. The three first columns give the average number if states, arcs and execution time over these correct configurations. The last column, gives the worst execution time in case all the configuration have been analyzed to extract correct ones. The construction of the reachability graph hase been performed with our SOG-based tool as well, by observing all the transitions of the model (in this case, the SOG coincides with the reachability graph).

In this evaluation, as we can observe from the Table 5.3, we took into account three levels of complexity (depending on the number of OR^+). The higher the value of CFC, the more complex is a process's configuration, since the number of possible configurations increases with the number of configurable OR connectors. For example, the CFC value 21 regards the process with only OR connectors, we can observe that the number of possible configurations as well as the extracted correct ones are relatively high compared to those having CFC 10. Moreover, the more transitions are

Table 5.3: Checking deadlock-freeness on SOG vs RG

CBP2PN			*SOG*				*Naive approach (RG)*			
CFC (avg)	Obs	Nb possible confs(avg)	Nb correct confs(avg)	A(avg)	E(avg)	Exec time(sec)	Sates (sum)	Arcs (sum)	Exec time correct(sec)	Overall Exec time(sec)
21	6	729	15	13	26	1.580	283.50	331.95	0.051	2.478
	5	243	5.66	8.66	16	0.693	104.14	133.57	0.017	0.729
$3OR^+$	4	81	2.33	5.66	8.66	0.353	42.17	49.62	0.007	0.243
	3	27	1	4	4	0.044	18	21	0.003	0.070
15.5	6	243	11.33	11	21	0.093	208.47	243.25	0.037	0.802
	5	81	5	7.77	13.77	0.051	93	106.30	0.017	0.267
$2OR^+$	4	57.85	3.66	6.09	10.33	0.030	66.72	77.81	0.012	0.191
	3	22.50	2	4.33	5.83	0.018	36.20	42.20	0.006	0.068
10	6	81	8	9.50	17.50	0.015	144	168	0.024	0.243
	5	54	4	7	11.83	0.010	72	84	0.014	0.184
$1OR^+$	4	18	4.25	5.75	9.87	0.008	76.71	89.46	0.014	0.058
	3	13.24	2.58	4.23	6.29	0.006	46.44	54.18	0.008	0.040

observed, the less reduced is the SOG comparing to the reachability graph.

Comparing to the naive approach, the obtained results in Table 5.3 show that the SOG is always significantly smaller in terms of number of states and arcs. For example, in case of a model having 6 configurable operators with OR type (i.e. the first row), we can observe that the obtained SOG includes only 13 aggregates and 26 arcs which is very reduced comparing to the size of the original graph of 729 possible configurations. Indeed, after applying a naive approach on only correct configurations (i.e. extracted from the SOG), the obtained graph has almost 283 states and 331 arcs resulting from the sum of 15 reachability graphs. Consequently, our work not only helps finding correct configurations but also further minimize the memory usage and the computing time, since only one reduced graph is constructed. To ensure the reproducibility of our experiments, please refer to our web page[1].

5.7 Discussion

As discussed in this thesis, the verification of process configuration appears as a core challenge to avoid the derived variants execution problems. To deal with this challenge, we proposed to improve this verification task by using two complementary contributions based on formal methods. The two approaches have some commonalities

[1] `http://www-inf.it-sudparis.eu/SIMBAD/tools/SOGImplementation`

and some differences to be noted:

- *The verified properties:* in the approach presented in Chapter 4, we consider our process model as a graph and we reason about correctness while essentially taking into account its structure. In fact, we verify structural properties such as, an activity should be on a path from the initial node to the final one. Also, we check erroneous patterns that my affect soundness property and thus affect the behavior of the model. This verification is actually done based on the structure of the graph. For example, we check that there is not a mismatching between split and join configurations that may cause a deadlock or a lack of synchronization. For instance, when joining by an AND connector a control flow that was previously split by an XOR connector, this implies a deadlock. However, structural correctness may be not sufficient. Some behavioral problems may be not easy to detect by exploring the structure of the model and need, instead, the analysis of the process instance states (cf. Section 4.9). For that aim, we proposed our second contribution in order to remedy this shortcoming by verifying the behavior of the process executions. The verification is achieved using well-defined semantics describing the dynamics of the process variants executions. This work focus on the deadlock-freeness property but can be easily adapted in order to obtain *sound* [57, 93] process variants. Another type of considered properties is the domain constraints. We check that the configuration of a process variant comply with some domain requirements specified by business analysts. These constrains are not yet integrated in the SOG-based contribution. We leave this for future work.

- *The problem of state space explosion:* In the first contribution, the calculation of possible configurations of elements is done one after the other. This means that the state space of configurations is reduced after each configuration application. Also, in case of a split (resp. join) configuration, the calculation considers only the corresponding joins (resp. splits) that comes after (resp. before) this connector. We do believe that our proof-based Event-B specification implicitly do not suffer from the state explosion problem. The main contribution of second SOG-based approach is actually the significant redaction of space state size. In fact, this is achieved by the compact and aggregated representation of all the possible configurations reachability graphs in one reduced SOG. The aggregation criterion is the connectors configuration. The experimental evaluation then goes on to examine the consequences of the structure and the number of configurable elements in the input process on the state space size.

- *The considered configurations constraints:* In the Event-B-based model, we considered all elements configurations, i.e. activities and connectors. However, in the second approach, we firstly dealt with the configuration of connectors, then we are currently working on integrating other configuration constraints, such as

removing activities. This is done by extending the presented semantics while preserving the same SOG definition and algorithm. Also, new semantics of OR-join connectors needs to be integrated. It is worth noting that Event-B based approach have the advantage of easily integrating new properties and constraints to the specification.

5.8 Conclusion

Our main contribution in this work is the SOG-based configuration model allowing to extract deadlock-free variants at design time while reducing the state explosion problem. We answered three research questions raised in the thesis problematic (Section 1.2) as follows:

RQ1: How to identify configuration choices that satisfy designers and clients requirements? In order to extract the complete list of the configuration choices that satisfy the considered property, i.e., the deadlock-freeness, we proposed a new SOG-based configuration approach. Basically, with the aim to verify behavioral issues, we firstly defined formal execution semantics of the configurable process model using Petri net (since the C-BPMN do not have formal semantics). Based on this formal model, we proposed an extension of the SOG definition such that nodes hold non configurable elements states, and each arc is labeled with a configurable element and its possible configuration. Then, the SOG-construction algorithm is extended in order to eliminate dead nodes on-the-fly. As a result, we obtained a graph including deadlock-free nodes, and thus, every path from the initial node to the final one represent a correct configuration. Our approach was validated by developing an extension to the SOG existing tool. This tool takes into consideration the new semantics of our model and the modifications made to the SOG definition and algorithm.

RQ2: How to assist the designer in selecting the correct configuration choices? As we mentioned, this approach generates all correct configuration choices. So, this configurations list is supplied to the process analyst with the C-BPMN configurable process. Hence, he/she may choose to apply the configuration that better satisfies his/her needs and preferences while being insured that the derived variant is correct. Consequently, no need to check correctness at each intermediate configuration step.

RQ3: How to avoid the space-state explosion of the configuration verification issue? The major advantage of this approach is the considerable reduction of the state space size. This achieved thanks to the compact and aggregated representation of all the possible configurations reachability graphs provided by the SOG. The experiments we conducted using our developed tool prove that our approach have addressed this problem.

To build toward our main goal regarding process configuration verification: *provide guidance and assistance to the analysts in process model configuration with correct options*, we proposed two complementary contributions using two different formalisms. In the next chapter, we target to address our second goal towards *improving the support of Cloud resource specification and verification in BPs*.

Chapter 6

Towards Correct Cloud Resource Allocation in Business Processes

Contents

6.1 Introduction 112
6.2 Motivating example 113
6.3 Approach Overview 114
6.4 Cloud Computing Resources : OCCI Standard 115
6.4.1 Cloud Resource Types 116
6.4.2 Cloud Resource Properties 116
6.4.3 Abstract Definition of a Cloud Resource-based Process 117
6.5 Formal Specification of a Business Process Model 119
6.5.1 Modeling Control Flow using Event-B 119
6.5.2 Introducing Execution Instances: First Level of Refinement 121
6.6 Formal Specification of the Resource Perspective 124
6.6.1 Modeling Resource Allocation: Second and Third Levels of Refinement 125
6.6.1.1 Introducing Shareability Property 126
6.6.1.2 Introducing Substitution Dependency 127
6.6.1.3 Introducing Resource Instances 128
6.6.2 Introducing the Elasticity Property: Fourth Level of Refinement . 130
6.7 Verification and Validation 132
6.7.1 Verification using Proofs 132
6.7.2 Validation by Animation 134
6.8 Proof of Concept: Integration of Cloud Resource Representation 137
6.9 Conclusion 139

6.1 Introduction

Nowadays, a growing number of companies are using Cloud Computing to optimize their business processes by using dynamically scalable and often virtualized resources on demand. Nevertheless, due to the lack of an explicit and formal description of the resource perspective in the existing business processes, Cloud resource allocation cannot be efficiently and correctly managed. The aim of this chapter is to offer a formal definition of the resource perspective in BPs as a step towards ensuring a correct and consistent Cloud resource allocation in business process modeling. For this purpose, we intend to answer the following research questions: *RQ4: How to formally specify and verify the Cloud resource allocation in BPs?* and *RQ5: How to integrate Cloud resources in BP models design?*

Concretely, we propose a formal specification based on the Event-B method for the resource perspective in BP models. This specification is used to formally validate the consistency of Cloud resource allocation for process modeling at design time, and to analyze and check its correctness according to user requirements and resource capabilities. In this work, we are specifically interested in formalizing and verifying Cloud resources properties (i.e., elasticity and shareability) and dependencies (i.e., allocation and substitution).

More practically, we propose to use the step-wise refinement technique by structuring the development into a chain of machines linked by refinement relations. This refinement approach produces a *correct-by-construction* specification since we prove at each step the different properties of the system. Regarding the proving and verification tasks, we use Event-B tools, first, to generate proof obligations that guarantee the constraints preservation. The process execution steps are ensured by *events*, and the different constraints are expressed in terms of *invariants*. Prior to the proof activity, that can be long and complex, we use the ProB animator to play some scenarios and gain some insurance about the correctness of the Event-B specification.

The remainder of this chapter is organized as follows. We start by giving a motivating example to illustrate our approach in Section 6.2. We present the overview of our formal specification in Section 6.3. The Cloud resource types and properties are pointed out in conformity with the OCCI standard, and a formal definition of a required resource and the Cloud resource-based business process model are described in Section 6.4. Section 6.5 illustrates our formal specification of the control flow perspective in BPs. Section 6.6 tackles the formalization of Cloud resource allocation in business processes. The verification and the validation of our Event-B specification are presented in Section 6.7. Finally, we present our proof of concept to integrate resource description into a process modeling tool in Section 6.8.

The content of this chapter was published in conferences proceedings [136, 137] and peerreviewed journal [138].

6.2 Motivating example

In the following, we present our motivating example of the process model in Figure 6.1. Cloud resources are assigned to different activities. Since traditional process modeling standards does not support resource perspective, we have added the Cloud resource representation to *Signavio Process Editor* [1] which is an existing web-based platform for business process modeling (see Section 6.8). Three Cloud resources types are taken into account: *storage*, *network*, and *compute*. In addition to the control flow relation between activities depicted by arrows (thereafter will be named *activation dependency*), we consider two other dependencies: (i) dependency between an activity and a resource, named *allocation dependency* and (ii) dependency between two resources, named *substitution dependency*. The latter dependency means that, in case of its absence or unavailability, a resource would most likely name a substitute to do the same work on its behalf. These dependencies are depicted by dotted arcs.

Concretely, the execution of the activity *a1* is performed in a virtual machine (i.e. resource *Compute1*) with 4 GB of RAM and 100 GB of disk. Moreover, both *a7* and *a8* activities are sharing a database service (i.e. resource *Store1*) hosted in the Cloud with 1 GB of storage size. *a8* and *a12* activities are hosted in the same virtual machine *Compute2* with 8 GB of RAM and 100 GB of disk. Besides, *a15* is hosted in a private local machine and needs to communicate with a virtual machine via a virtual networking Cloud resource (i.e. resource *Network1*). Finally, the activity *a6* stores its data in a local database, (i.e. resource *Store2*) with 1 TB of storage size.

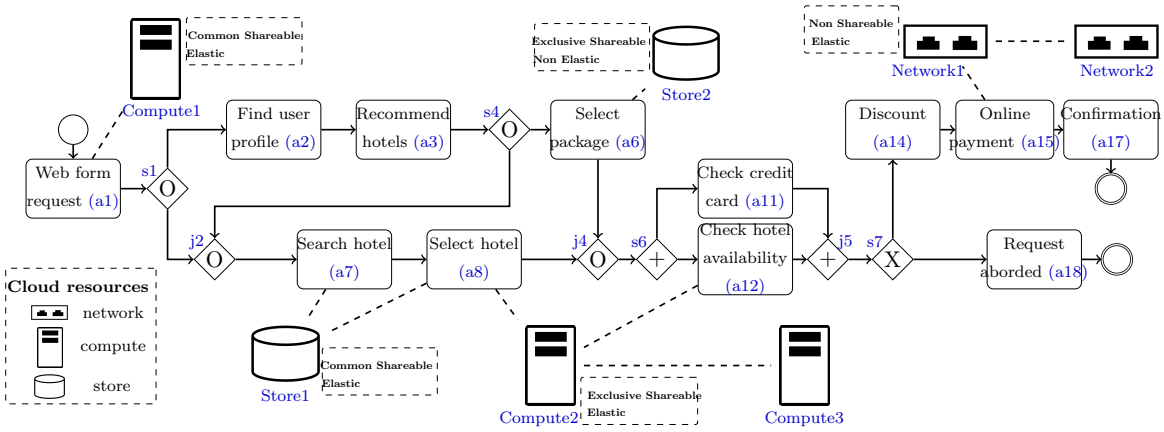


Figure 6.1: A process variant and its resources

Furthermore, we used different types of resources coming from different Cloud providers (especially Microsoft Azure and Google). Thus, the process tenant can freely purchase and configure the resources it uses to meet its requirements. The used

[1] https://code.google.com/p/signavio-core-components/

resources can be classified into two groups. Elastic resources are Cloud resources having specific set of policies agreed through an SLA between the Cloud provider and the process tenant. In our example, *Compute1*, *Store1*, *Compute2* and *Network1* are configured to be elastic resources. In order to handle concurrent requests, the elastic resource instance has the ability to change its capacity to accommodate the workload. Hence, the Cloud consumer pays for what it consumes. Non-elastic resources are classical on-premises resources having fixed capacity (i.e. cannot be changed at runtime). In our example, *Store2* is non-elastic, therefore its capacity can not exceed 1 GB. Also, these resources may be shareable or non-shareable. In our example, (1) *Compute1* and *Store1* are two resources *commonly shareable*, which means that two or more activity instances can be executed at the same time. (2) *Compute2* and *Store2* are two resources *exclusively shareable*, which means that two or more activity instances can use them but not at the same time. Finally, (3) *Network1* is non-shareable resource, which means that it can be used by only one activity instance and is released after activity instance's completion.

Moreover, dependencies between resources can be captured. In this work, we consider the substitution dependency. For instance, the computing resource *compute3* can substitute the computing resource *compute2* if this latter becomes no more available. Obviously, *Compute3* should provide the same properties and policies. The same shall apply to the networking resource *network2* which substitutes the networking resource *network1*.

As we can notice, the model is complex and contains many properties and dependencies, so the designed process and the running process instances behavior can easily deviate from users' needs. Basically, the designer should respect several model consistency rules. For instance, an *exclusive shareable* resource (e.g. *Compute2*) cannot be consumed by more than one activity instance at the same time. Also, a non-elastic resource, for example *Store2*, cannot be allocated by an activity instance of *TP* while its available capacity does not fit the needed capacity. So, in order to validate and check these resource constraints, we propose to apply formal techniques to avoid, on the one hand, structural inconsistencies before deploying or even purchasing these resources from Cloud providers, and, on the other hand, behavioral inconsistencies which may occur during runtime.

6.3 Approach Overview

Let us recall that an Event-B specification is structured around *machines* and *contexts*. A key concept in this work is the use of the stepwise refinement. This concept consists in progressively making an abstract specification more precise through incremental steps. Figure 6.2 depicts the formalization architecture of our Event-B model which is composed of five abstraction levels. Each level is a refinement of the previous one and adds specific constraints and requirements towards the formalization of the Cloud resource allocation behavior.

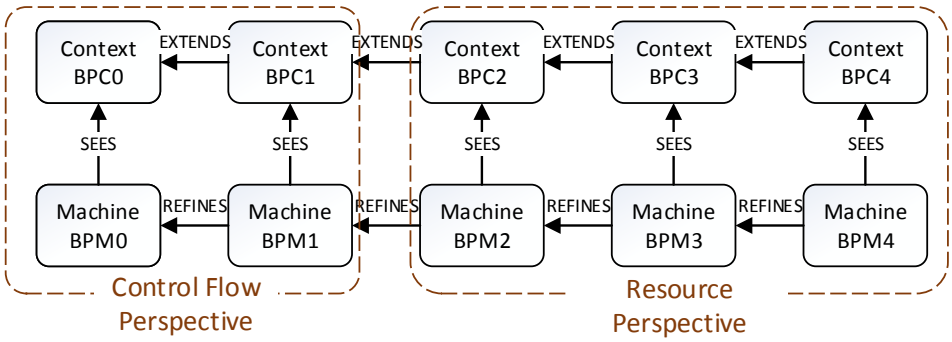


Figure 6.2: Event-B model

(1) The machine *BPM0* is the most abstract one and it starts by the the modeling of the control flow perspective. It introduces the different variables and their invariants allowing to model the coordination between the process activities. This machine *sees* a context *BPC0* that defines some sets and constants (see Section 6.5.1).

(2) The machine *BPM1* refines *BPM0* and introduces the process and the activity execution instances in order to consider the runtime requirements. This machine *sees* a context *BPC1* which *extends* the context *BPC0* (see Section 6.5.2).

(3) The machine *BPM2* refines *BPM1* by adding the allocated resources to a process activities as well as the substitute resources. In this refinement level, the shareability property of a Cloud resource is pointed out. This machine *sees* the context *BPC2* which *extends BPC1* (see Section 6.6.1).

(4) The machine *BPM3* refines *BPM2* and further details related to resource instances are added. This machine *sees* the context *BPC3* which *extends BPC2* (see Section 6.6.1.3).

(5) The machine *BPM4* refines *BPM3* and introduces the elasticity property of a Cloud resource. The elasticity mechanisms are modeled using events in this machine. This machine *sees* the context *BPC4* which *extends BPC3* (see Section 6.6.2).

The following sections describe these abstraction levels in detail. But before, let as discuss Cloud resources types and properties in the next section.

6.4 Cloud Computing Resources : OCCI Standard

The emerging Cloud Computing paradigm offers a pool of shared resources between applications at three different layers: at the top layer *Software-as-a-Service (SaaS)*,

consumers can remotely access software applications via web based interfaces; the middle layer *Platform-as-a-Service (PaaS)* provides an operational platform allowing customers to manage, develop and execute their applications; and at the bottom layer Infrastructure-as-a-Service (IaaS), consumers may access to highly automated and scalable resources delivered as a service via the Internet. These resources that may be compute/servers, cloud storage and networking capability are needed to power or support users applications. IaaS provides the highest level of flexibility and management control regarding offered resources.

In this work, we focus on the IaaS layer, and assume that offered resources at this level are *compute*, *network*, and *storage*. Cloud resources may be modeled with different existing standards such as, TOSCA [139] (Topology and Orchestration Specification for cloud Applications), OCCI [140] (Open Cloud Computing Interface), and CIMI [141] (Cloud Infrastructure Management Interface). In order to describe our cloud resources, we use OCCI that is a set of open standards and specifications that was initially developed for IaaS cloud offerings. In the following, we present concepts and properties that we use in the reminder of this chapter to characterize relevant behavior of Cloud resource-based process models. This section is divided into three subsections dealing with: resource types, properties, and formal definition of resource and Cloud resource-based process model.

6.4.1 Cloud Resource Types

The Cloud Computing delivers three important types of Infrastructure as a Service (IaaS) resources on demand. As can be seen in the OCCI infrastructure [12] class diagram of Figure 6.3, three classes inherit from the core basic *Resource* class that was defined in OCCI core Model [142]. Hence, a *Resource* is specialized into:

a) *Compute* represent processing resources that are a collection of Physical Machines (PMs), each comprised of one or more processors, memory, network interface and local I/O, which together provide the computational capacity of a Cloud environment [143] (e.g., a virtual machine).

b) *Network* represent networking entities that may be needed to interconnect these PMs with a high-bandwidth network (e.g., a virtual switch).

c) *Storage* represent data persistent storage services.

6.4.2 Cloud Resource Properties

The above presented resources may be classified by considering two relevant Cloud Computing properties: (i) the *resource elasticity*, and (ii) the *resource sharing*. In this work, we focus on the vertical elasticity property of Cloud resources which refers to adding or reducing resource's capacity to an activity.

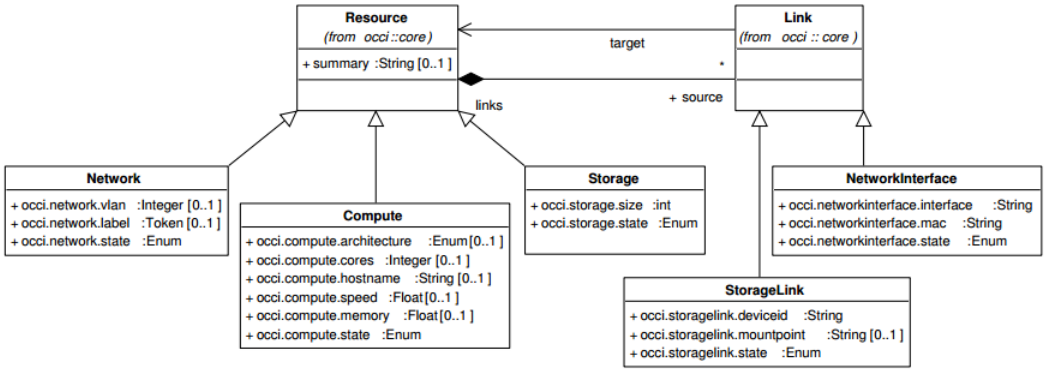


Figure 6.3: Class diagram of OCCI Infrastructure types [12]

According to Table 6.1, a Cloud resource may be either elastic or non elastic, and either shareable or non shareable. Also, the definition of a shareable resource is refined into two properties: exclusive shareable or common shareable.

Table 6.1: Cloud Resource Properties description

Type		**Description**
Elastic		A resource is *elastic*, if it can be resized up or down by changing its capacity at runtime.
Non Elastic		A resource is *non elastic*, if its capacity is fixed and can not be modified at runtime.
Shareable	*Exclusive*	A resource is *exclusive shareable* if its instances can be allocated and used by different activities' instances *but not* at the same time.
	Common	A resource is *common shareable* if its instances can be allocated and consumed by several activities' instances at the same time.
Non Shareable		A resource is *Non shareable* if it can be allocated to only one activity instance.

6.4.3 Abstract Definition of a Cloud Resource-based Process

In addition to activities ordering captured by the control flow perspective, resources may be needed for activities execution such as users, machines, services, etc. As discussed earlier, we focus on non-human ones, particularly Cloud resources. Moreover, several relationships could be captured. In light of this, we present an abstract formal definition of a business process model taking into account the Cloud resource properties and relationships (see Definition 6.4.1):

Definition 6.4.1 (Cloud Resource-based process model)**.** *A business process model is a tuple* $Bp = < ACT, RES, E, D_a, D_r, C_a >$ *where:*

- ACT *is the set of activities;*
- RES *is the set of used resources;*
- $E : ACT \leftrightarrow ACT$ *is a control flow relationship between two activities. In this work we define two types of relationships: AND and OR* ***activation dependencies****;*
- $D_a : RES \leftrightarrow ACT$ *is a relationship between a resource and an activity. In this work we define the relationship:* ***the allocation dependency****;*
- $D_r : RES \leftrightarrow RES$ *is a relationship between two resources. In this work we define the relationship:* ***the Substitution dependency****;*
- $C_a : D_a \rightarrow \mathbb{N}$ *is the needed resource capacity for each activity execution.*

More Formally, we define a required resource for a specific task as a set of four elements. In fact, a resource (1) has a type (i.e. Storage, Network or Compute), (2) may be *shareable* or not, (3) may be *exclusive* or *common shareable*, (3) may be *elastic* or not, and (4) has a capacity. We formally define a resource as follows (see Definition 6.4.2):

Definition 6.4.2 (Resource)**.** *A required resource* $r \in RES$ *(i.e.* RES *is the set of available resources) is defined as a tuple* $r =< T, Shareable, ExclusiveShareable, Elastic, C_r >$ *where:*

- $T : RES \rightarrow \{Storage, Network, Compute\}$ *is a function that assigns for each resource* $r \in RES$ *a type. In case of Cloud resources we consider three types: Storage, Network and Compute.*
- $Shareable : RES \rightarrow BOOL$ *is a function that assigns for each resource* $r \in RES$ *the value TRUE if it is Shareable, and FALSE if it is not.*
- $ExclusiveShareable : Shareable^{-1}[\{TRUE\}] \rightarrow BOOL$ *is a function that assigns for each shareable resource the value TRUE if it is exclusive shareable, and FALSE if it is common shareable.*
- $Elastic : RES \rightarrow BOOL$ *is a function that assigns for each resource* $r \in RES$ *the value TRUE if it is Elastic, and FALSE if it is not.*
- $C_r : RES \rightarrow \mathbb{N}$*is a function that assigns for each resource* $r \in RES$ *a capacity. In case of non-elastic resources this capacity is fixed (as a constant value) and does not change, and in case of elastic ones it is variable.*

In Sections 6.5 and 6.6, we describe our Event-B formal model based on Definitions 6.4.1 and 6.4.2 .

6.5 Formal Specification of a Business Process Model

The control-flow (or process) perspective describes activities and their execution ordering through different constructors, which permit flow of execution control [47]. In this section, we introduce the first step of our formalization which consists of two levels of abstraction considering this process perspective. Firstly, in Section 6.5.1, we present the first abstraction level of our formal specification that addresses the control flow of process models. Secondly, in Section 6.5.2, we formalize the process execution dynamics behavior relying on the activity instance lifecycle. Please note that, the semantics of the Event-B mathematical symbols used throughout this chapter are illustrated in Appendix B.

6.5.1 Modeling Control Flow using Event-B

In the abstract model, we introduce the first level of our specification which holds processes, activities and their relationships.

Firstly, in the first context *BPC0* illustrated in Listing 14, we use a finite set BP (axm1) to define the set of possible processes, and a finite set ACT (axm2) to define the set of possible process activities.

Listing 14: BPC0's sets and axioms

```
CONTEXT  BPC0
SETS     BP  ACT
AXIOMS
axm1 : finite(BP)
axm2 : finite(ACT)
```

Then, we define the initial machine *BPM0* which sees the context *BPC0* described above. Listing 15 shows the variables and the invariants of *BPM0*. To map each process to its activities, we introduce the variable *BP_activities* (Inv1). To model the order in which the different activities will be performed in the process, we add two variables *AND_ActivationDep* (Inv2) and *OR_ActivationDep* (Inv4). These activation dependencies specify which activity must finish execution to activate a given activity.

- *AND_ActivationDep* (inv2) states which activities instances have to finish their executions to activate another activity.

  For instance, having $BP0$ the process fragment of our motivating example in the Figure 6.4a, $BP0 \mapsto \{a17 \mapsto a15, a17 \mapsto a16\} \in AND\_ActivationDep$ means that in order to activate an instance of the activity *a17*, an instance of each activity among *a15* and *a16* must finish execution. Moreover, $BP0 \mapsto \{a15 \mapsto a14, a16 \mapsto a14\} \in AND\_ActivationDep$ means that in order to activate instances of both activities *a15* and *a16*, the activity instance of *a14* should finish execution. This dependency is used also in case of a sequence between two activities;

- *OR_ActivationDep* (inv4) states that we have two or more alternative paths, i.e. one or more of the alternative paths or activities may be chosen. For example, having similarly the process fragment $BP0$ of Figure 6.4b, $BP0 \mapsto \{a17 \mapsto a15, a17 \mapsto a16\} \in OR\_ActivationDep$ means that in order to activate an instance the activity *a17*, one instance of either the activity *a15* or the activity *a16* must finish its execution. It is worth noting that, in this work, *OR_ActivationDep* may represent the inclusive OR or the exclusive OR.

Invariants *Inv3* and *Inv5* guarantee that two activities related with an activation dependency relationship belong to the same process.

Listing 15: BPM0's variables and invariants

```
MACHINE  BPM0
SEES  BPC0
VARIABLES  BP_activities  AND_ActivationDep  OR_ActivationDep
INVARIANTS
Inv1 :  BP_activities ∈ BP ⟷ ACT
Inv2 :  AND_ActivationDep ∈ BP ⟶ (ACT ⟷ ACT)
Inv3 :  ∀p.(p ∈ BP ⇒ AND_ActivationDep(p) ⊆  BP_activities[{p}] × BP_activities[{p}])
Inv4 :  OR_ActivationDep ∈ BP ⟶ (ACT ⟷ ACT)
Inv5 :  ∀p.(p ∈ BP ⇒ OR_ActivationDep(p) ⊆  BP_activities[{p}] × BP_activities[{p}])
```

(a) *AND_ActivationDep* relation (b) *OR_ActivationDep* relation

Figure 6.4: Examples of process fragments for the activation relations illustration

In the machine *BPM0*, we define our first event, as shown in Listing. 16. The event *CreateBP* allows to construct a new process model *bp* by adding its activities *acts* and relationships between activities, i.e. activation dependencies *and_activDep* and *or_activDep*.

Listing 16: Business process creation event, Machine BPM0

```
CreateBP  ≙
ANY   bp   acts   and_activDep   or_activDep
WHERE
grd1:   bp ∈ BP ∧ bp ∉ dom(BP_activities)
grd2:   acts ⊆ ACT ∧ acts ≠ ∅
grd3:   and_activDep ⊆ acts × acts
grd4:   or_activDep ⊆ acts × acts
THEN
act1:   BP_activities := BP_activities ∪ ({bp} × acts)
act2:   AND_ActivationDep(bp) := and_activDep
act3:   OR_ActivationDep(bp) := or_activDep
```

In order to manage the addition or deletion of activities and their relations in an existing process (i.e. already created using the previous event), we add the following events: (1) the events *AddACT* and *RemoveACT* for respectively adding and removing activities, (2) the events *AddAND_Dep* and *RemoveAND_Dep* for respectively adding and removing an AND activation dependency, and (3) the events *AddOR_Dep* and *RemoveOR_Dep* for respectively adding and removing an OR activation dependency.

6.5.2 Introducing Execution Instances: First Level of Refinement

An execution instance of a workflow model is called a *case* or a *process instance*. In this step, we specify the process instances behavior. For this aim, we start by introducing the context *BPC1* that extends the first one, i.e., *BPC0*. As illustrated in the Listing. 17, we define the set of possible processes' instances as a carrier set *BP_INSTANCES*. This context is *seen* by a new machine *BPM1* captured by Listing. 18. Of course, to make possible the use of its variable and the triggering of its events, this machine refines the machine *BPM0* described above (i.e., using REFINES in Event-B). In *BPM1*, we define a variable *BP_Instances* to store all created processes' instances. Then, it is obvious that *BP_Instances* is a subset of *BP_INSTANCES* (Inv1). In the same way, we define the set of all activities' instances, as a carrier set *ACT_INSTANCES* in *BPC1*, and the created activities' instances, as the subset *ACT_Instances* (Inv3).

Each *activity instance* during its lifetime goes through different states. Figure 6.5 depicts an activity instance life cycle inspired by the Workflow Management Coalition (WfMC) [144]. After its creation, the activity instance moves to the state *Initiated*. During this state a resource may be allocated to this activity instance. Then, the state becomes *Running* when a work item is created and assigned to the activity instance for processing. An activity instance may be *canceled* while being either *Initiated* or *Running*. A successful execution toggles between *Running* state and *Completed* state. Whereas an unsuccessful execution moves from the state *Running* to the state *Failed*. These states are defined as distinct subsets of the set *ACT_STATES* in *axm2* of Listing. 17.

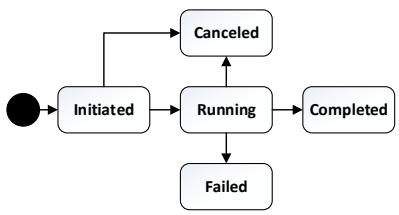


Figure 6.5: Activity instance life cycle

Listing 17: BPC1's sets and axioms

```
CONTEXT   BPC1
EXTENDS   BPC0
SETS      BP_INSTANCES  ACT_INSTANCES  ACT_STATES
CONSTANTS    initiated  running  failed  canceled  completed
AXIOMS
axm1 : finite(ACT_INSTANCES)
axm2 : partition(ACT_STATES, {initiated}, {running}, {failed}, {canceled}, {completed})
```

We define a total function *ACT_Instances_State*, which gives the current state of each activity instance (*Inv5*). The variable *BP_Instances_Type* (*Inv2*) (resp. *ACT_Instances_Type* (*Inv4*)) defines the process (resp. the activity) to which an instance belongs. Also, we introduce the variable *ACT_Instances_BP_Instances* to define the process instance to which belongs an activity instance (*Inv6*).

Listing 18: The first refinement machine, Machine BPM1

```
MACHINE BPM1  REFINES BPM0
SEES BPC1
VARIABLES  BP_Instances  BP_Instances_Type
ACT_Instances  ACT_Instances_Type  ACT_Instances_State
ACT_Instances_BP_Instances
INVARIANTS
Inv1:  BP_Instances ⊆ BP_INSTANCES
Inv2:  BP_Instances_Type ∈ BP_Instances ⟶ BP
Inv3:  ACT_Instances ⊆ ACT_INSTANCES
Inv4:  ACT_Instances_Type ∈ ACT_Instances ⟶ ACT
Inv5:  ACT_Instances_State ∈ ACT_Instances → ACT_STATES
Inv6:  ACT_Instances_BP_Instances ∈ ACT_Instances ⟶ BP_Instances
...... //here we omit other detailed invariants
```

Once we defined the different constants, variables and typing constraints related to process instances, we focus now on formally modeling the behavior of an activity instance. We define a set of events that will serve to modify the state of an activity instance w.r.t. Figure 6.5. For that aim, we defined an event for each transition from an activity instance state to another. We introduce the following events.

- *Process instance creation*: First of all, the *AddBpInst* event is defined. Its occurrence allows the creation of a new process instance *bpi* of a process *bp*;
- *Activity instance creation*: The *AddACTInst* event allows the creation of an activity instance *ai*, having initially the state *initiated*, in the process instance *bpi*;
- *Activity instance execution*: The *RunACTInst* event allows the execution of an activity instance *ai* by changing its state to *running*;

- *Activity instance termination*: The *CompleteACTInst* event permits an activity instance *ai* to complete execution successfully, by moving its state from *running* to *completed*;
- *Activity instance cancellation*: The *CancelACTInst* event interrupts the activity instance execution and cancel it, so its state passes from *running* to *canceled*;
- *Activity instance failure*: The *FailACTInst* event represents the case of any failure of the activity instance performance, then its state moves from *running* to *failed*.

Let as give more details about some of the above mentioned events. For instance, Listing. 19[2] depicts the *AddACTInst* event that allows the creation of the new instance *ai* of a process activity *ac*. Then, *ac* should belong to the process of *bpi* (grd2), and the state of the created activity instance should be initialized to *initiated* (*act4*).

Listing 19: Activity instance adding event, Machine BPM1

```
AddACTInst ≙
ANY   bpi   ac   ai
WHERE
grd1:  bpi ∈ BP_Instances
grd2:  ac ∈ ACT ∧ ac ∈ BP_activities[{BP_Instances_Type(bpi)}]
grd3:  ai ∈ ACT_INSTANCES ∧ ai ∉ ACT_Instances
grd4:  ACT_Instances_BP_Instances ∼ [{bpi}]  ∩  ACT_Instances_Type ∼ [{ac}] = ∅
THEN
act1:  ACT_Instances := ACT_Instances ∪ {ai}
act2:  ACT_Instances_Type(ai) := ac
act3:  ACT_Instances_BP_Instances(ai) := bpi
act4:  ACT_Instances_State(ai) := initiated
AND
```

As shown in Listing. 20, the event *RunACTInst* activates an activity instance *ai* by changing its state from *initiated* to *running* (*grd2* and *act1* respectively). As we discussed earlier, the AND and OR activation dependencies define the succession relationships between two activities. Thus, the guard *grd3* expresses the fact that the activity instance *ai*, as a successor to a set of activities instances, may be activated only after all their executions completion (AND activation condition). A similar guard is added to express OR activation condition.

Listing 20: Activity instance execution event, Machine BPM1

```
RunACTInst ≙
ANY   ai
WHERE
grd1:  ai ∈ ACT_Instances
grd2:  ACT_Instances_State(ai) = initiated
grd3:  ∀ac. (ac ∈ ACT  ∧  ACT_Instances_Type(ai) ↦ ac
         ∈    AND_ActivationDep(BP_Instances_Type(  ACT_Instances_BP_Instances(ai)))
```

[2]The inverse of a function f, (f^{-1}), is denoted in Event-B as ($f \sim$).

$\Rightarrow$ **card(** $ACT\_Instances\_BP\_Instances \sim [\{ ACT\_Instances\_BP\_Instances(ai)\}]$
$\cap$ $ACT\_Instances\_Type \sim [\{ACT\_Instances\_Type(ai)\}]$
$\cap$ $ACT\_Instances\_State \sim [\{running\}]$ **)**
$<$ **card(** $ACT\_Instances\_BP\_Instances \sim [\{ ACT\_Instances\_BP\_Instances(ai)\}]$
$\cap$ $ACT\_Instances\_Type \sim [\{ac\}] \cap ACT\_Instances\_State \sim [\{completed\}])$ **)**

...

THEN
act1: $ACT\_Instances\_State(ai) := running$
AND

Regarding the event *CompleteACTInst*, the Listing. 21 indicates that the state of an activity instance *ai* passes from the state *running* to the state *completed* (grd2 and act1) in order to complete successfully.

Listing 21: Activity instance completion event, Machine BPM1

CompleteACTInst $\triangleq$
ANY ai
WHERE
grd1: $ai \in ACT\_Instances$
grd2: $ACT\_Instances\_State(ai) = running$
THEN
act1: $ACT\_Instances\_State(ai) := completed$
AND

Figure 6.6 specifies the sequencing between these *BPM1* machine's events. The diagram explicitly illustrates that the consequence of the *CreateBP* event occurrence may allow, at this refined level, the triggering of *AddBpInst* event followed by *AddACTInst* event, followed by either *CancellACTInst* or *RunACTInst* events (having the activity instance state *initiated*). Then, after triggering the event *RunACTInst*, the activity instance state becomes *running* which allows it to be completed successfully, using the event *CompleteACTInst*, or unsuccessfully, using *CancelACTInst* or *FailACTInst* events.

So far, at this level, we have not yet considered the resources that may be used by the activity instances during their lifecycle described above. The resource perspective is the subject of the next section.

6.6 Formal Specification of the Resource Perspective

In this section, we define our last three abstraction levels taking into consideration the resource perspective and the different resource properties. We try to formally specify the resource allocation in our model relying on some patterns defined by N. Russel et al. in [40].

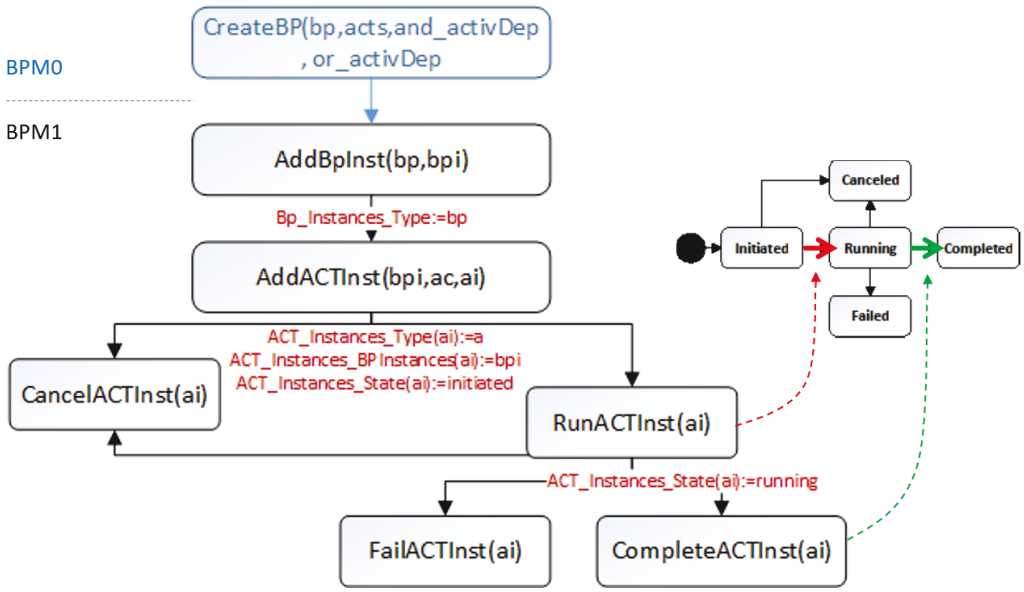


Figure 6.6: Business process execution events, Machine BPM1

6.6.1 Modeling Resource Allocation: Second and Third Levels of Refinement

At this stage, we extend the context *BPC1* by adding resources into our model (see Listing. 22). For this aim, we introduce a new finite set named *RES* (*axm1*) in the context *BPC2* to represent all available resources. Then, as seen in Section 6.4.1, a resource has three main types RESType: *compute*, *storage*, *network* (*axm2*). We define a constant function *RES_Type* mapping each resource to its type (*axm3*). We also introduce the relation *BP_Resources* to the second refinement machine *BPM2* (see Listing. 23), in order to map each process to its resources (*Inv1*). An available resource can be added to (resp. removed from) a process using the event *AddRES* (resp. *RemoveRES*).

Listing 22: Context BPC2

```
Context BPC2 EXTENDS BPC1
SETS RES RESType
CONSTANS compute storage network RES_Type
AXIOMS
axm1: finite(RES)
axm2: partition(RESType, {compute}, {storage}, {network})
axm3: RES_Type ∈ RES ⟶ RESType
```

In our work, we specify the identity of the required resource responsible for executing an activity at design time, which is in conformity with the pattern *Direct Allocation (WRP-01)* defined by N. Russel et al. in [40]. Thus, we formally define

the allocation dependency between an activity and a resource using a new variable *AllocationDep* (Inv2, Listing. 23). This latter denotes, for each process, the relation of a possible allocation between a resource and an activity. We can add (resp. remove) an allocation dependency using the event *AddAllocDep* (resp. the event *RemoveAllocDep*).

Listing 23: Allocation dependency invariants, Machine BPM2

```
Inv1: BP_Resources ∈ dom(BP_activities) ↔ RES
Inv2: AllocationDep ∈ BP_Resources ↔ ACT
Inv3: ∀bp, res. (bp ↦ res ∈ dom(AllocationDep) ⇒ AllocationDep[{bp ↦ res}] ⊆ BP_activities[{bp}])
```

For instance, taking back our motivating process model in Figure 6.1, we define an allocation dependency between *a1* and *Compute1* as follows: $BP0 \mapsto Compute1 \mapsto a1 \in AllocationDep$; which means that, in the process *BP0*, an instance of the activity *a1* needs an instance of the resource *Compute1* to complete its execution. Of course, this implies that *a1* should belong to *BP0* (i.e. $BP0 \mapsto a1 \in BP\_activities$) as constrained by the invariant *Inv3* in Listing. 23.

6.6.1.1 Introducing Shareability Property

As we have discussed in Section 6.4.2, a Cloud resource may be shareable or non-shareable. Also, a resource may be shareable in a given process and non-shareable in another. So, we add a variable *Shareable* defined as a total function (cf. *Inv4*, Listing 24) that, for each couple (process, its resource), defines whether the resource is shareable or not (using a boolean value). Invariant *Inv5* specifies that only shareable resources may have several allocation dependencies (i.e., $card(AllocationDep[\{bpres\}]) > 1$).

Besides, in order to further refine the shareability property, we define a total function *ExclusiveShareable* (*Inv6*, Listing 24) having as input all shareable resources of a specific process and returns (i) TRUE if the resource is *exclusive shareable*, or (ii) FALSE if it is *common shareable*. Then, we define two events *AddRES* and *RemoveRES* that allows to respectively add and remove a resource to/from a process. In the event *AddRES*, we add two parameters to specify whether the added resource is shareable, non-shareable, exclusive shareable or common shareable. The shareability property is also managed using two events: to make a resource shareable (resp. non-shareable) we use the event *MakeRESShareable* (resp. *MakeRESNonShareable*).

Listing 24: Shareability property invariants, Machine BPM2

```
Inv4: Shareable ∈ dom(AllocationDep) → BOOL
Inv5: ∀bpres. (card(AllocationDep[{bpres}]) > 1  ⇒  Shareable(bpres) = TRUE)
Inv6:  ExclusiveShareable ∈ Shareable ∼ [{TRUE}] → BOOL
```

6.6.1.2 Introducing Substitution Dependency

The substitution dependency captures the possibility to replace a resource by another to perform some work in case of its unavailability or absence. For this aim, we introduce the relationship *SubstitutionDep* (cf. *Inv7*, Listing. 25) which maps each allocation dependency, linking a process and its resource to an activity, to a substitute resource. Obviously, the substitute resource should belong to the considered process (*inv8*). In addition, it should not have an allocation dependency with the considered activity (*Inv9*), in order to avoid redundancy, which also implies an *irreflexive* relation (i.e. a resource should not substitute itself). Furthermore, a substitute resource inherits all the privileges/properties of the resource it substitutes for. So, they must have the same resource type and the same shareable property's value (*Inv10*). Also, in case of a shareable resource, they should have the same exclusive shareable property's value (*Inv11*). For instance, in our motivating example, we need to substitute the resource *compute2* by the resource having the same type (i.e. a computing resource), namely *compute3*. Hence, *compute3* must have the same properties as *compute2* (i.e. shareable and exclusive).

Listing 25: Substitution dependency invariants, Machine BPM2

```
Inv7:  SubstitutionDep ∈ AllocationDep ↔ RES
Inv8:  ∀bp, ac, re1, re2.(bp ↦ re1 ↦ ac ↦ re2 ∈ SubstitutionDep  ⇒  re2 ∈ BP_Resources[{bp}]
Inv9:  ∀bp, ac, re1, re2.(bp ↦ re1 ↦ ac ↦ re2 ∈ SubstitutionDep
                  ⇒  bp ↦ re2 ↦ ac ∉ AllocationDep)
Inv10: ∀bp, ac, re1, re2.(bp ↦ re1 ↦ ac ↦ re2 ∈ SubstitutionDep
    ⇒  RES_Type(re1) = RES_Type(re2)  ∧  Shareable(bp ↦ re1) = Shareable(bp ↦ re2))
Inv11: ∀bp, ac, re1, re2.(bp ↦ re1 ↦ ac ↦ re2 ∈ SubstitutionDep
    ∧  bp ↦ re1 ∈ Shareable ∼ [{TRUE}]
                  ⇒ ExclusiveShareable(bp ↦ re1) = ExclusiveShareable(bp ↦ re2))
```

It is worth noting that, at each step, some of the previously presented events are refined to take into account the new properties and requirements. For instance, as can be seen in Listing 26, additional parameters are added in the refinement event *CreateBP*, namely *res* (set of resources), *allocDep* (allocation dependencies between *bp*, *res* and *acts*), *shar* (shareable or non-shareable *res*), *exclushar* (exclusive or common shareable *res*) and *subDep* (resources in *res* having substitution dependencies). Thus, corresponding guards are needed.

Listing 26: Business process creation event, Machine BPM2

```
CreateBP ≙ REFINES  CreateBP
ANY
      bp  acts  and_activDep  or_activDep  res  allocDep  shar  exclushar  subDep
WHERE
// see Listing 16
grd5:  res ⊆ RES
grd6:  allocDep ∈ {bp} × res ↔ acts
grd7:  shar ∈ {bp} × res → BOOL
grd8:  ∀r. (r ∈ res ∧ card(allocDep[{bp ↦ r}]) > 1  ⇒  shar(bp ↦ r) = TRUE)
grd9:  exclushar ∈ shar ∼ [{TRUE}] → BOOL
```

```
grd10: subDep ∈ allocDep ↔ res
THEN
// see Listing 16
act5:  AllocationDep := AllocationDep ∪ allocDep
act6:  Shareable := Shareable ∪ shar
act7:  BP_Resources := BP_Resources ∪ ({bp} × res)
act8:  ExclusiveShareable := ExclusiveShareable ∪ exclushar
act9:  SubstitutionDep := SubstitutionDep ∪ subDep
```

6.6.1.3 Introducing Resource Instances

In the third level of refinement, we introduce the resource instances. First, in the context *BPC3*, we add a new set *RES_INSTANCES* for all available resources instances. Afterwards, we add the set of created resources instances $RES\_Instances$ in the refinement machine *BPM3* (cf. *inv1*, Listing 27). Besides, we map each resource instance to its resource type using the function $RES\_Instances\_Type$ (*inv2*) and we map each resource instance to the process instance to which it belongs using the function $RES\_Instances\_BP\_Instances$ (*inv3*).

Listing 27: Resource instances invariants, Machine BPM3

```
Inv1: RES_Instances ⊆ RES_INSTANCES
Inv2: RES_Instances_Type ∈ RES_Instances ⟶ RES
Inv3: RES_Instances_BP_Instances ∈ RES_Instances ⟶ BP_Instances
```

It is worth pointing out that each resource instance goes through different states during its lifetime in a business process execution. Figure 6.7 illustrates resource instance's states and transitions. This life cycle is made independently of the Cloud resource type. After its creation, a resource is in the *Inactive* state. When it is allocated to an activity instance it moves to the state *Allocated*. Once the activity instance starts its execution, the resource moves to the state *Consumed*. While being in consumption, a Cloud resource could be *resized* (more details are given in Section 6.6.2). After the activity instance completion, the resource becomes *Inactive*, for a future reallocation, if it is shareable, otherwise it is *Released* (i.e. withdrawn).

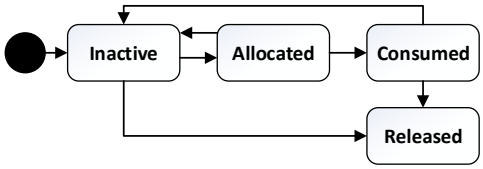


Figure 6.7: Resource instances' states

Formally, we define the set of resources' instances belonging to one of the following states: *Inactive*, *Allocated*, *Consumed* (see Listing. 28). Obviously, when a resource instance is *released*, it will no longer be existing (i.e. it becomes not in

RES_Instances). *Allocated* and *Consumed* are two relations specifying a resource instance's states according to an activity instance (*Inv6* and *Inv7*, Listing. 28). For example, having the resource *Compute1 commonly shared* between two instances of *a1*, namely $a1_1$ and $a1_2$, an instance of *Compute1* could be allocated to $a1_1$ and consumed by $a1_2$ at the same time. So here, after completion of $a1_2$, the resource state could not become *Inactive* until the completion of $a1_1$.

Listing 28: Resource states invariants, Machine BPM3

```
Inv5:  Inactive ⊆ RES_Instances
Inv6:  Allocated ∈ RES_Instances \ Inactive  ↔  ACT_Instances_State ~ [{initiated}]
Inv7:  Consumed ∈ RES_Instance \ Inactive  ↔  ACT_Instances_State ~ [{running}]
... //here we omit other detailed invariants
Inv11:  ∀ri, bpi.(ri ↦ bpi ∈ RES_Instances_BP_Instances
    ⇒  BP_Instances_Type(bpi) ↦ RES_Instances_Type(ri)  ∈ dom(AllocationDep))
Inv12:  ∀bpi, r.(bpi ∈ BP_Instances ∧ r ∈ RES  ∧  BP_Instances_Type(bpi) ↦ r ∈ dom(AllocationDep)
         ⇒  card(RES_Instances_BP_Instances ~ [{bpi}]  ∩  RES_Instances_Type ~ [{r}]) ⩽ 1)
Inv13:  Allocated ∩ Consumed = ∅
```

Other invariants are added to this machine to ensure the consistency of our model. For instance, the invariant *Inv11* specifies that a resource instance cannot be created for a process instance whose process does not have an allocation dependency with the corresponding resource. Also, the invariant *Inv12* ensures that, for each process instance, only one resource instance of a given resource could be created. This constraint is also ensured by the guard *grd3* of the event *AddRESInst* (see Listing. 29) which allows to add a resource instance *ri* of a resource *r* to a process instance *bpi*. Obviously, a resource instance cannot be allocated and consumed by the same activity instance at the same time (*inv13*).

Listing 29: Add resource instance event, Machine BPM3

```
AddRESInst  ≙
ANY   bpi   r   ri
WHERE
grd1:  r ∈ BP_Resources[{BP_Instances_Type(bpi)}]
grd2:  ri ∈ RES_INSTANCES ∧ ri ∉ RES_Instances
grd3:  RES_Instances_BP_Instances ~ [{bpi}]∩ RES_Instances_Type ~ [{r}] = ∅
THEN
act1:  RES_Instances := RES_Instances ∪ {ri}
act2:  RES_Instances_Type(ri) := r
act3:  RES_Instances_BP_Instances(ri) := bpi
act4:  Inactive := Inactive ∪ {ri}
```

To model the resource allocation of a created resource instance *ri* to an activity instance *ai*, we introduce the event *AllocateRESInst* in Listing. 30. This event is guarded by five conditions: (i) *ai*'s current state is *initiated* (*grd2*), (ii) the resource instance *ri* is not released (*grd3*), (iii) the activity to which *ai* belongs and the resource to which *ri* belongs have an allocation dependency between them (*grd4*), (iv) *ri* is *Inactive* or its resource type is *common shareable* (*grd5*), and (v) *ri* and *ai* belong to

the same process instance (*grd6*). As a result, the resource instance *ri* is allocated to the activity instance *ai* (*act2*).

Listing 30: Allocate resource instance event, Machine BPM3

```
AllocateRESInst ≙
ANY    ai    ri
WHERE
grd1:  ai ∈ ACT_Instances
grd2:  ACT_Instances_State(ai) = initiated
grd3:  ri ∈ RES_Instances ∧ ri ↦ ai ∉ Allocated
grd4:  BP_Instances_Type(ACT_Instances_BP_Instances(ai)) ↦ RES_Instances_Type(ri)
                  ∈  AllocationDep ∼ [{ACT_Instances_Type(ai)}]
grd5:  ri ∈ Inactive ∨  BP_Instances_Type(
     ACT_Instances_BP_Instances(ai)) ↦ RES_Instances_Type(ri)
                  ∈  ExclusiveShareable ∼ [{FALSE}]
grd6   RES_Instances_BP_Instances(ri) = ACT_Instances_BP_Instances(ai)
THEN
act1:  Inactive := Inactive \ {ri}
act2:  Allocated := Allocated ∪ {ri ↦ ai}
```

After the completion, failure or cancellation of an activity instance, the introduced resource instance should be either *released* (in case of *non-shareable* resources), or returned to *Inactive* state waiting to be reallocated (if only the current activity instance is using it). This is modeled using the event *FreeRESInst*.

6.6.2 Introducing the Elasticity Property: Fourth Level of Refinement

In this section, we outline the elasticity property of Cloud resources in the final refinement machine *BPM4*. For this aim, we add a variable *Elastic* (*Inv1*, Listing. 31) returning *TRUE* if the resource is *elastic* and *FALSE* otherwise. Obviously, we refine the event *AddRES* by adding a parameter to specify if the added resource is elastic or not.

Moreover, the allocation relationship between activities and resources in our model is based on specific capabilities that they possess. This corresponds to the pattern *Capability-based Allocation (WRP-08)* defined by N. Russel et al. in [40]. This pattern supports the allocation by the matching of specific activities requirements with the capabilities of resources. In our case, we take into consideration the capacity of resources (whether elastic on not) and the required capacity for each activity. Therefore, we first introduce the constant *RES_Capacity* (in the context *BPC4*) to define the initial offered capacity of a resource and the variable *RESInstance_Capacity* (*Inv2*) to define a resource instance capacity which may vary in case of elastic resources. Then, we define for each specific allocation dependency between an activity and a non-elastic resource, the required/needed capacity for a correct performance using the function ACT_RES_Needs (*Inv3* and *Inv4*, Listing. 31)[3]. However, in case of elastic resources,

[3] $A \lhd f$ denotes a domain restriction: $A \lhd f = \{x \mapsto y | x \mapsto y \in f \wedge x \in A\}$

these capacity needs may increase or decrease at runtime depending on the dynamics of the received requests. So, we define the function $ACTInstance\_RES\_Needs$ (*Inv5*) to specify the activity instance need while consuming a resource instance. The event *SetElasticNeed* is specified to manage this capacity variation.

Listing 31: Machine BPM4's invariants

```
Inv1: Elastic ∈ BP_Resources → BOOL
Inv2: RESInstance_Capacity ∈ RES_Instances → N1
Inv3: ACT_RES_Needs ∈ AllocationDep ⇸ N1
Inv4: (Elastic ∼ [{FALSE}] ◁ AllocationDep) ⊆ dom(ACT_RES_Needs)
Inv5: ACTInstance_RES_Needs ∈ Consumed ⇸ N1
Inv6: ∀bp, r.(r ∈ RES ∧ bp ∈ BP ∧ bp ↦ r ∈ dom(AllocationDep) ∧ Elastic(bp ↦ r) = FALSE
        ⇒ SUM(({bp ↦ r} × BP_activities[{bp}]) ◁ ACT_RES_Needs) ⩽ RES_Capacity(r))
Inv7: ∀ai, ri.(ai ∈ ACT_Instances ∧ ri ∈ RES_Instances∧
    ACT_Instances_State(ai) = completed ∧ ri ↦ ai ∈ Consumed
        ∧ ri ↦ ai ∈ dom(ACTInstance_RES_Needs) ⇒
            ACTInstance_RES_Needs(ri ↦ ai) ⩽ RESInstance_Capacity(ri))
```

As we have seen, the non-elastic resource's capacity cannot be changed. So, in order to ensure a correct allocation dependency, the *sum* (using a new *SUM* operator) of all needed capacities of all the allocation dependencies of a non-elastic resource must be lower or equal to its offered capacity (*Inv6*, Listing. 31). Moreover, at run-time, an activity instance should not complete execution until having the needed capacity of its consumed resource instance (*Inv7*, Listing. 31). Also, we have added this constraint as a guard in the refined event *CompleteACTInst*.

We have defined the *SUM* operator to be able to sum several needed capacities (*axm1*, Listing. 32). More precisely, this operator allows to sum the values of the hash table: (AllocationDep, capacity value) pairs. This specific operator is not already defined in Event-B. Therefore, we have defined it as a *Theory* using the axioms of the Listing. 32.

Listing 32: The SUM operator's axioms

```
axm1: SUM ∈ (BP × RES × ACT ⇸ N1) → N
axm2: SUM(∅) = 0
axm3: ∀x, y. (x ∈ BP × RES × AC ∧ y ∈ N1 ⇒ SUM({x ↦ y}) = y) ⇒ SUM({x ↦ y}) = y)
axm4: ∀s, t. (s ∈ (BP × RES × ACT) ⇸ N1 ∧ t ∈ (BP × RES × ACT); ⇸N1
                                    ⇒ SUM(s ∪ t) = SUM(s) + SUM(t))
```

As mentioned earlier, an activity instance may require more/less capacity at runtime, which is due to changes in demand or workloads in a Cloud environment. Our model must be able to react to these dynamic changes. This is handled by using two events, *ResizeUpRESInst* and *ResizeDownRESInst* that respectively increases and decreases the capacity of a resource instance according to the activities instances needs. For instance, at runtime, when the activity instance demand increases and becomes greater than the resource instance capacity (*grd3*, Listing. 33), the elastic resource instance's capacity can be modified using the event *ResizeUpRESInst* to have

a value greater or equal to the activity instance need (*grd4*). Otherwise, the activity instance could not complete its execution.

Listing 33: Elasticity event, Machine BPM4

```
ResizeUpRESInst  ≙
ANY    ri   val
WHERE
grd1:  ri ∈ RES_Instances \ Inactive ∧ val ∈ N1
grd2:  ri ∈ RES_Instances_Type ~ [(Elastic ~ [{TRUE}])[BP]]
grd3:  SUM(({ri} × Consumed[{ri}]) ◁ ACTInstance_RES_Needs)  > RESInstance_Capacity(ri)
grd4:  SUM(({ri} × Consumed[{ri}]) ◁ ACTInstance_RES_Needs)  ⩽ val
THEN
act1:  RESInstance_Capacity(ri) := val
AND
```

6.7 Verification and Validation

6.7.1 Verification using Proofs

The proof statistics, given in Fig. 6.2, show that 338 proof obligations were generated by the Rodin platform. 257 proof obligations (76%) were automatically discharged while others, which are more complex ones, require the interaction with the provers to help them find the right rules to apply.

Machines/Contexts	Total POs	Automatic	Interactive
BPC0	0	0	0
BPC1	0	0	0
BPC2	0	0	0
BPC3	0	0	0
BPC4	0	0	0
BPM0	34	22	12
BPM1	61	46	15
BPM2	52	38	14
BPM3	112	89	23
BPM4	79	62	17
Overall	338	257 (76%)	81 (24%)

Table 6.2: Proof statistics

As an example, when modeling the second refinement machine *BMP2*, in the refined event *CreateBP* (see Listing 26) we have started by adding two new parameters: the parameter *allocDep*, to define allocation dependency ($grd6 : allocDep \in \{bp\} \times RES \leftrightarrow acts$); and the parameter *shar*, to define the corresponding shareable resources ($grd7 : shar \in dom(allocDep) \rightarrow BOOL$). This refinement gives rise to four new proof obligations (one for each of inv2, inv3, inv4 and inv5 of Listing. 23). The PO CreateBP/inv5/INV was not automatically discharged. Using the proof

obligation explorer we can inspect this unproved PO and see that it has a goal as follows:

$$\forall bpres. \quad card((AllocationDep \cup allocDep)[\{bpres\}]) > 1 \implies (Shareable \cup shar)(bpres) = TRUE \quad (6.1)$$

Clearly this cannot be proven because *Inv5* depicted in Listing. 23 has not yet been considered by the prover. Moreover, this proof needs several steps to be discharged. Let $bpres0$ be a couple of a process and a resource, which satisfies (6.1).

We have to demonstrate that:

$$(Shareable \cup shar)(bpres0) = TRUE \quad (6.2)$$

under the hypothesis:

$$card((AllocationDep \cup allocDep)[\{bpres0\}]) > 1 \quad (6.3)$$

To do so, we proceed by case-based reasoning by distinguishing two cases:

1. $\underline{bpres0 \in dom(AllocationDep)}$: which means that the process is already existing, so we have necessarily:

   $$bpres0 \in dom(Shareable)$$

   hence, the goal (6.2) is written into [4]:

   $$Shareable(bpres0) = TRUE \quad (6.4)$$

   and the hypothesis (6.3) is written into:

   $$card(AllocationDep[\{bpres0\}]) > 1 \quad (6.5)$$

   thus, the instanciation of the invariant *Inv5* of listing 23 by $bpres0$ gives:

   $$card(AllocationDep[\{bpres0\}]) > 1 \Rightarrow Shareable(bpres0) = TRUE \quad (6.6)$$

   So, the goal (6.4) is accomplished using (6.5) + (6.6) + *Modus ponens*[5] rule.

[4] $(F \cup G)(x) = G(x)$ if $x \in dom(G)$

[5] The modus ponens rule: If $(P \wedge P \Rightarrow Q)$ then Q

2. $\underline{bpres0 \in \{bp\} \times res}$: which means that there exists a resource $r1 \in res$ such that:

$$bpre0 = bp \mapsto r1 \tag{6.7}$$

hence, having (6.7), the goal (6.2) is written into:

$$shar(bp \mapsto r1) = TRUE \tag{6.8}$$

and the hypothesis (6.3) is written into:

$$card(allocDep[\{bp \mapsto r1\}]) > 1 \tag{6.9}$$

Moreover, the instanciatiation of the *grd8*:

$$grd8 : \forall r.(r \in res \wedge card(allocDep[\{bp \mapsto r\}]) > 1 \Rightarrow shar(bp \mapsto r) = TRUE)$$

by $r1$ gives:

$$card(allocDep[\{bp \mapsto r1\}]) > 1 \Rightarrow shar(bp \mapsto r1) = TRUE \tag{6.10}$$

Consequently, (6.10) + (6.9) + the *modus ponens* rule prove (6.8)

Thus, the PO (6.1) is proven. The proof tree in Figure 6.8 shows the proving process of this PO.

Then, we used ProB to verify several dynamic properties that cannot be specified as invariants since they refer to several states of the system taken at different moments. Such properties have been specified using LTL [80]. For instance, we have verified that after its creation, an activity instance is in the state *initiated* using an *LTL* formula involving the next operator:

$$\mathrm{G}(x \notin ACT\_Instances \Rightarrow \mathrm{X}(x \in ACT\_Instances \Rightarrow ACT\_Instances\_State(x) = initiated))$$

6.7.2 Validation by Animation

Using *animation*, we have played and observed different scenarios and have checked the behavior of our model. In this work, we use the ProB plugin for this animation. The process of animating an Event-B model involves three steps: (1) we first give values to the constants and carrier sets in the context (in our case we used the context *BPC4*), and (2) we start the animation by firing the INITIALISATION event to set the system in its initial state; then, (3) we proceed the steps of our scenario, and for each step: (i) the animator computes all guards of all events, enables the ones with true guards, and shows parameters which make these guards true; then, (ii) we

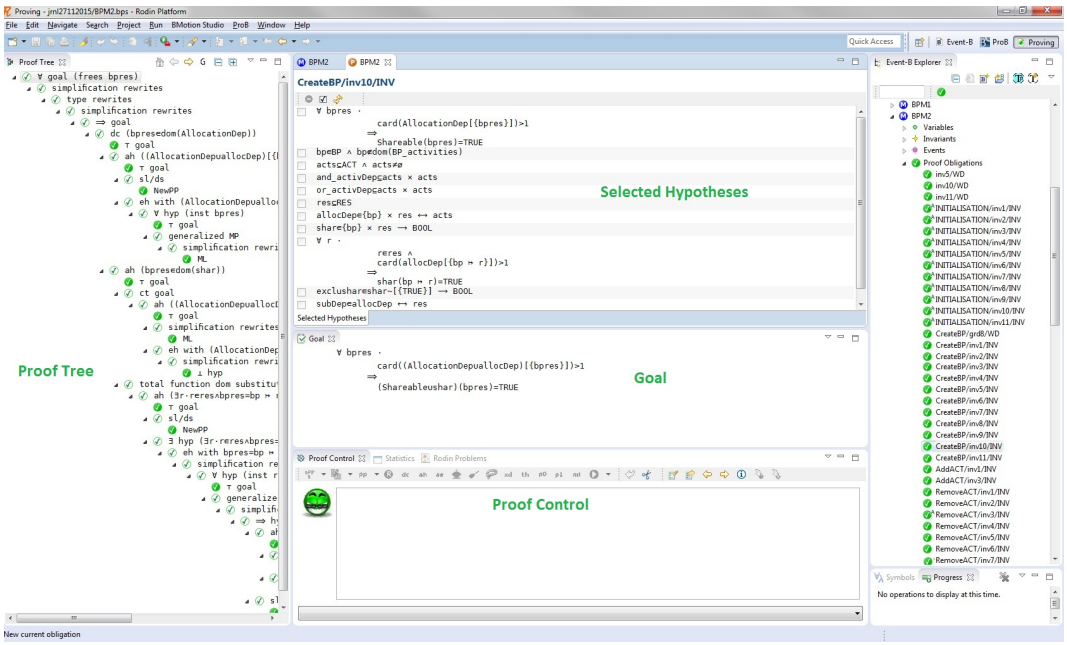


Figure 6.8: Rodin interface for discharging Proof Obligations

pick one value, if any, which allows to fire the enabled event and, in consequence, the substitutions are computed. Finally, (iii) the animator checks if the invariants still hold.

For instance, we animated the complete behavior of an activity instance from its creation until its completion while verifying the different states in which it may move. We have also verified the allocation of resources to activities and checked how an elastic resource instance adapts its capacity when capacity needs increase/decrease.

In order to highlight the different steps of a process execution taking into account all dependencies described above (i.e. activation, allocation and substitution) as well as resource properties (i.e. shareable and elastic resources), we have successfully applied the animation of ProB on our final level of refinement model using our case study of Figure 6.1 as follows.

1. We create the business process *BP0*, its activities and their activation dependencies, and its resources corresponding to our case study. We also specify if the added resources are: (i) shareable or not, (ii) exclusive or common shareable and (iii) elastic or not. For example *Compute1* is common shareable and elastic.

2. We add the allocation dependencies, e.g. between *a1* and *Compute1*, between *a7* and *Store1*, etc. Note that after adding an allocation dependency between *Network1* and *a15*, we cannot add another allocation dependency with this resource since it is not shareable. As can be seen in the screen shot of Figure

6.9, the second allocation of the resource *network1* is not proposed in the list of choices. Also, we specify the activity needed capacity in case of non-elastic resources.

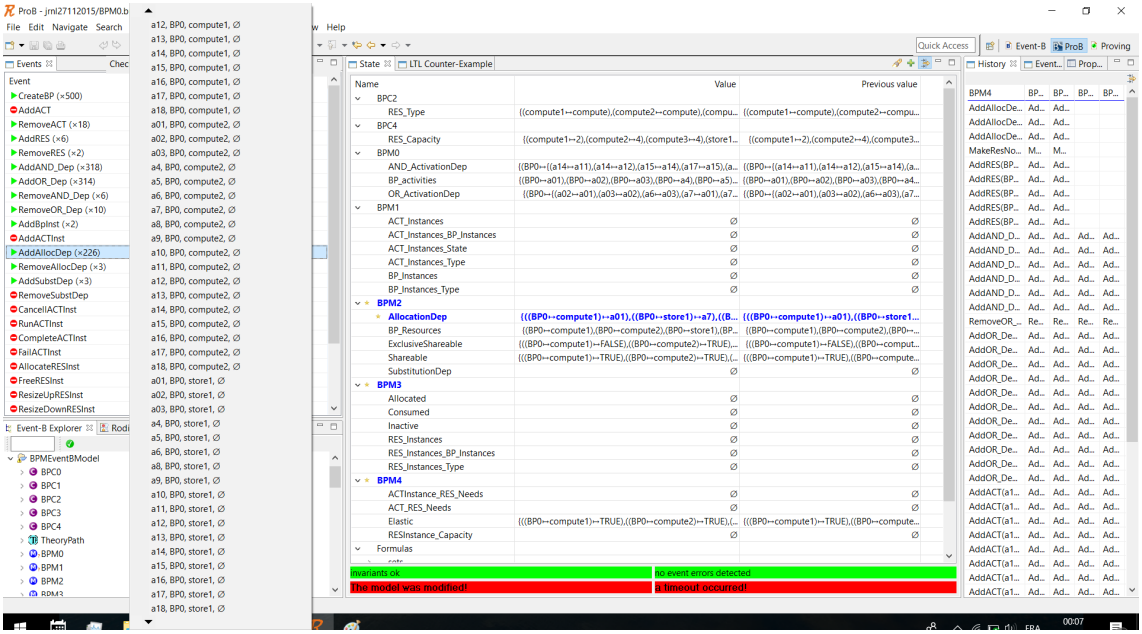
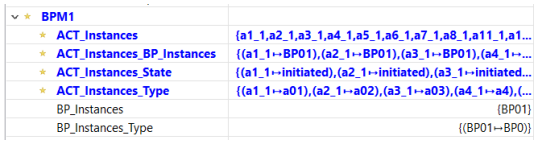


Figure 6.9: Animating the case of non-shareable resource

3. We add two substitution dependencies using the event *AddSubstDep*: between *Compute2* and *Compute3*, and between *network1* and *network2*;

4. We create an instance of *BP0*, named *BP01* using *AddBpInst*.

5. We create an instance of each activity belonging to *BPO*: *a1*, *a2*, *a3*, etc., named *a1_1*, *a2_1*, *a3_1*, etc. As expected the state of each instance is *initiated*, cf. Figure 6.10.



BPM1	
ACT_Instances	{a1_1,a2_1,a3_1,a4_1,a5_1,a6_1,a7_1,a8_1,a11_1,a1...
ACT_Instances_BP_Instances	{(a1_1↦BP01),(a2_1↦BP01),(a3_1↦BP01),(a4_1↦...
ACT_Instances_State	{(a1_1↦initiated),(a2_1↦initiated),(a3_1↦initiated...
ACT_Instances_Type	{(a1_1↦a01),(a2_1↦a02),(a3_1↦a03),(a4_1↦a4),(...
BP_Instances	{BP01}
BP_Instances_Type	{(BP01↦BP0)}

Figure 6.10: Extract of the animation values of Machine BPM1 after adding activities instances

6. In order to execute these activities which mainly use resources, we need to add an instance of each resource: *Compute1*, *Store1*, etc., named *Compute11*,

Store11, etc.

7. Then, we allocate each resource instance to its corresponding activity instance. For example, *AllocateRESInst(a1_1,Compute11)*, to do so, an allocation dependency was necessarily added in a previous step between *a1* and *Compute1*.

   Here, when having an allocation dependency, e.g. $BP0 \mapsto Compute2 \mapsto a8 \in AllocationDep$, and $Compute3$ could substitute $Compute2$, i.e. $BP0 \mapsto Compute2 \mapsto a8 \mapsto Compute3 \in SubstitutionDep$, we have the choice to allocate an instance of *a8* to either an instance of *Compute2* or an instance of *Compute3*.

8. Now, we can run *a1_1*. As some resources are elastic, their instances can be resized up or down. For example, when having *Compute11*'s capacity equal to 2 and *a1_1*'s needed capacity equal to 3, then we must resize up *Compute11* (i.e. by applying *ResizeUpRESInst(Compute11,3)*). Each activity instance could not complete its execution until its resource instance capacity covers its need. Also, we could not execute *a2_1* before having executed *a1_1*, due to the AND activation dependency between *a2* and *a1*, etc.

6.8 Proof of Concept: Integration of Cloud Resource Representation

As a proof of concept, we have extended the *Signavio Process Editor*[6]. *Signavio* is an open source web application for modeling business processes in BPMN that supports its latest version of BPMN 2.0. Since this application does not support Cloud resources representation and management, we have extended the BPMN 2.0 with the considered Cloud resource types, i.e., storage, network, and compute; and integrated their representation in the modeling interface.

A screen-shot of the graphical interface is depicted by Figure 6.11. As shown, graphical elements that represent each type of resource are highlited in the red square. The designer may drag and drop the needed resource and link it to the desired activity using an association. Also, he/she can specify the attributes of each designed resource such as cores, speed, hostname, etc. This resource representation comply with the Open Cloud Computing Interface (OCCI) standard [145]. We have also considered the substitution relationship between two resources. So, substitute resources may also be assigned to other resources using an association. Hence, this web application extension offers the designer the ability to assign Cloud resources to process activities and to define the different dependencies between them[7].

[6] http://www.signavio.com/

[7] Please refer to our web page for source code: `http://www-inf.it-sudparis.eu/SIMBAD/tools/BPMEventBModel`

It is worth noting that we have extended the BPMN XSD (XML Schema Definition) file that describes the structure of a BPMN Model (i.e., it defines the different elements and attributes). We have added an element tag for each resource type. Figure 6.12 illustrate one of these elements dedicated for the *compute* resource as well as the different defined attributes As an output, this extension allows to generate the BPMN XML file including the resources tags.

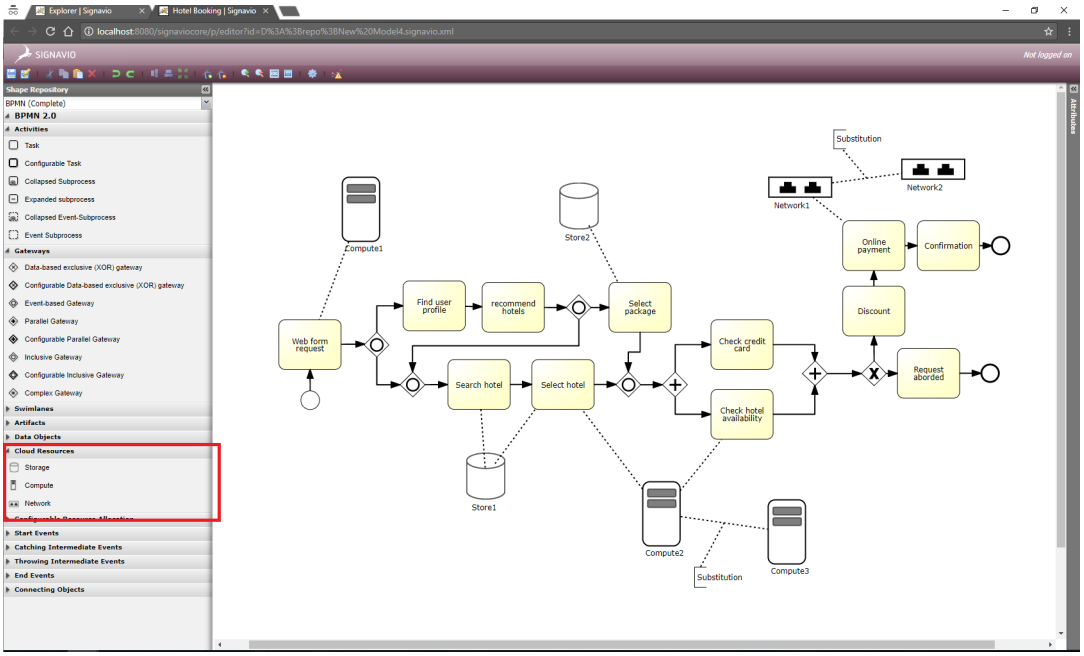


Figure 6.11: A screen-shot of the graphical interface for our Cloud resource-based process modeling in Signavio

```
<xsd:element name="compute" substitutionGroup="rootElement" type="tCompute"/>
▼<xsd:complexType name="tCompute">
 ▼<xsd:complexContent>
  ▼<xsd:extension base="tRootElement">
     <xsd:attribute name="architecture" type="xsd:string"/>
     <xsd:attribute name="cores" type="xsd:integer"/>
     <xsd:attribute name="hostname" type="xsd:string"/>
     <xsd:attribute name="speed" type="xsd:string"/>
     <xsd:attribute name="memory" type="xsd:string"/>
     <xsd:attribute name="cstate" type="xsd:string"/>
   </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 6.12: Excerpt of the *xsd* file "semantics.xsd" of the BPMN 2.0 extension

6.9 Conclusion

Our contribution in this work is the formal specification of the Cloud resource allocation in business processes while considering Cloud resources constraints and properties. We attempted to respond to our defined research questions as follows:

RQ4: How to formally specify and verify Cloud resource allocation behavior in BPs? we introduced an Event-B based model that formally specify the resource allocation behavior in a business process within a Cloud environment. Different properties were formalized. For instance, resource sharing has been regarded as a relevant property of Cloud computing. We also considered vertical elasticity property which refers to adding or reducing resources capacity to an activity. However we haven't yet considered horizontal elasticity property which refers to adding additional activity instances or removing them as necessary as well as some local properties such as safety and liveness properties. We leave these limitations for the future work. This approach allows to check different Cloud resource properties and constraints while considering both the design and the runtime requirements. The correctness and the consistency of our approach are checked by discharging proof obligations and by animating the specification using the ProB plugin.

RQ5: How to integrate Cloud resources in BP models design? We developed an extension to the Signavio modeling tool as a proof of concept to integrate Cloud resources in process modeling. Hence, we extended the latest version of BPMN 2.0 with the definition of the three types of IaaS Cloud resources. Then, we added their representation in the Signavio interface as well as their attributes and dependencies.

As we can remark, the Event-B specification we have built makes a separation between the control flow and the resource perspective modeling. In fact, we started by the more abstract model that specify processes, activities and their ordering, and then we added progressively: (i) activities instances behavior with respect to their lifecycle, (ii) resources and their instances behavior as well, and (iii) the different considered properties and dependencies. The advantage of such a separation is twofold. New control flow or resource requirements or properties can be integrated to the model without altering the existing Event-B specification. In addition, the verification and validation phase may be applied on specific parts of the development. So, depending on the designers qualifications, we may assign to them specific parts checking rather then the complete model.

Furthermore, in the near future, we target to consider Cloud resource allocation in our behavioral verification approach. Hence, we aim at defining new formal execution semantics based on Petri nets that takes into consideration resource instances and orchestration between available ones.

Chapter 7

Conclusion and Future Work

Configurable process models allow a systematic reuse of business processes in a flexible way. Recently, they are increasingly adopted thanks to their integrated representation of the common and variable parts of a family of processes. These processes are typically adjusted according to the organization specific needs and preferences. In another side, more and more organizations are adopting PAIS on cloud environments to benefit from a large pool of shared Cloud resources. These resources may be assigned to process activities to accommodate dynamic demands. This thesis strives to address two main research questions: *How to assist and verify business process configuration?* and *How to verify Cloud resource allocation in business process models?*.

The first problem stems from the lack of assistance in the process configuration task in order to obtain correct variants. Indeed, when considering complex configurable processes with potentially large number of configurable elements, this task becomes quiet difficult and error-prone. Also, the exponential number of possible variants quickly leads to the combinatorial explosion of the state space.

The second problem is justified by the lack of a formal and explicit description of Cloud resources in BPs while considering Cloud properties. Indeed, previous work in the field of resource perspective in BPs mainly addressed the human resources. The Cloud resources allocation, properties and interactions has not been considered yet.

In this manuscript, we presented in details three contributions to respond to the mentioned problems. In this final chapter, we summarize our work in Section 7.1 and present our future research directions in Section 7.2.

7.1 Fulfillment of Objectives

The first aim of the research presented in this thesis is to provide guidance and assistance to the analysts in process model configuration with correct options. This manuscript presents two approaches that contributed to this goal.

In the first contribution, we propose a formal Event-B based approach to derive correct variants step-by-step. Such a configuration approach guides the analyst by providing at each step the potential configuration choices. For each configurable element (i.e., activity and connector), we formalized configuration choices using events.

Then, we defined a set of constraints using *invariants* specifying the variant structure that should be respected in order to prevent errors. These constraints are related to structure (e.g., every node should be reachable from the initial activity), soundness (no deadlocks and lack of synchronization situations), but also domain requirements provided by *configuration guidelines*. The verification of the preservation of all the defined constraints and requirements was done using proof obligations generated by the Rodin tool. This tool also supports the validation of our specification using animation. This approach has been automated through the use of a model transformation language, ATL. Such a tool allows to map a C-BPMN process model into the corresponding Event-B specification. Finally, our conducted case study showed the practical usefulness of our approach as well as the facility in identifying the configuration choices.

In the second contribution, we are interested in the definition of deadlock-freeness variants while reducing the state space explosion problem. In this work, the verified correctness criterion of the obtained variant is the deadlock-freeness. Traditionally, such behavioral property verification can be handled by verifying the behavior of all possible configurations. This means that the reachability state graph of all possible variants need to be explored leading to the state explosion problem. An effective solution to this issue is proposed in this work by abstractly representing all possible configurations in a reduced SOG graph. To this end, we adapt the original SOG definition and construction algorithm based on observed configurable elements as follows. First, the SOG abstraction is defined such that the observed configurable connectors label the graph arcs and the non-configurable elements are hidden in the graph nodes. Since the SOG should be associated with a formal model, we use Petri nets as a pivot formalism to represent a C-BPMN process by defining the corresponding syntax and semantics. Then, relying on this semantics, we extend the SOG construction algorithm in order to check the deadlock-freeness property on-the-fly. In fact, aggregates are constructed in a depth-first search style such that any aggregate containing a deadlock state is not inserted in the graph and so are all the underlying paths. As a result, we obtain a reduced graph as well as a set of correct configurations. Then, this set will serve to support analysts during configuration. Our approach was implemented as an extension to an existing tool to implement the proposed algorithm. Preliminary experiments show that our approach outperforms naive approaches in terms of size of the explored configurable models.

The third contribution of this thesis aims at improving the support of Cloud resource specification and verification in BPs. To reach this goal, we introduced an Event-B based model that formally specifies and verifies the resource allocation behavior in a business process deployed in a Cloud infrastructure. In fact, We have applied a correct-by-construction refinement technique in order to formally model and reason about a process model from both perspectives: control flow and resource flow. We started by the more abstract model that specifies process models, activities and their ordering, and then we added progressively details to introduce (i) activities

instances behavior with respect to their lifecycle, (ii) resources allocation to activities, (iii) resources instances behavior with respect to their lifecycle as well, and (iv) the different considered properties and dependencies. In this approach, we considered the *resource sharing* that has been regarded as a relevant property of Cloud computing allowing multiple activities to allocate the same resource simultaneously. We also considered the *resource vertical elasticity* that refers to adding or reducing resources capacity to an activity. The correctness and the consistency of our approach were checked by discharging proof obligations and by animating the specification using the ProB plugin. Finally, as a proof of concept, we integrated the representation of Cloud resources according to the OCCI standard in the BPMN modeling language definition.

7.2 Future Research Directions

Our work opens several research perspectives to accomplish in short and middle terms. At first, we intend to enrich our work with additional properties, constraints, perspectives, etc. This would provide higher expressiveness to our research. Then, we plan to study the adaptability to change in process configuration. Finally, we intend to entirely automate our configuration approaches.

More expressiveness. We are currently working on extending our *second contribution* presented in Chapter 5. Firstly, we are considering configurable processes with cycles and synchronizing OR-joins. Regarding cycles, we propose to adapt our algorithm in order to consider them while building the SOG aggregates. As we have explained in Section 5.5, the construction algorithm is performed, first, by pushing the initial aggregate into a stack, then, by incrementally pushing *new* aggregates linked with observed transitions. These aggregates are popped from the stack and added to the graph once entirely checked. Hence, we aim to check the case of a found successor aggregate that is already in the stack (previously pushed and not yet popped). In this case, the found aggregate is both successor and predecessor of an aggregate since it is not entirely treated and may still have other observed transitions to fire. Regarding OR-joins, we propose to adapt our proposed semantics to consider the *Synchronizing Merge* captured by the *Pattern 7* in [132]. New semantics should expressly impose that, first there is at least one token in at least one of its incoming branches, then it should be checked that for an incoming branch having no token, it is not possible for a token to reach this flow [74,133]. Secondly, we are working on respecting additional configuration constraints: activity configuration and connectors configuration by restricting output or input branches. This needs to be done by adapting our semantics in order to consider not only skipping one activity in case of its configuration to OFF, but also by removing an entire branch that is an output (resp. input) of a split (resp. join). Then, we aim at adapting the SOG construction algorithm in order to integrate other correctness constraints, e.g., soundness.

Furthermore, we intend to enrich our configuration approaches with other important perspectives such as the resource and the data perspectives. Regarding resource perspective, in the work of Hachicha et al. [146], configurable operators for resource configuration were proposed in order to support the resource variability. Through configuration, these operators allow designers to easily derive variants by selecting the desired resources. However, such new dependencies between activities and resources were not formally specified and verified. As future work, we aim at adding a formal specification and verification phase that may assist and recommend resource configuration. After that, we target to take into account the analyst specified QoS constraints as well as the Cloud resource properties in order to derive adequate process variants. Moreover, we plan to identify the impact of the configuration of a multi-perspective configurable process on the process variants correctness and performance.

Our last approach might not cover every need for every organization in terms of resources. Indeed, our Event-B model could be extended to consider (i) additional resource types, e.g., PaaS and SaaS resources, (ii) QoS constraints and temporal requirements, (iii) additional dependencies between resources, e.g., delegation and peering, (iv) allowed actions, e.g., create, edit and delete. Also, we aim to consider *horizontal elasticity property* which refers to adding additional activity instances or removing them as necessary.

More adaptability. In real-life process models, any changes to the process environment also lead to changes and variation in the process model: either if it is the configurable process model or even its derived process variants. Indeed, organization are continuously willing to align their processes with new requirements (e.g., new law, regulation, technology, etc.). As our ultimate goal is to prevent organizations from redesigning their processes "from scratch", then, the adaptation to change of these processes need to be considered with minimal effort. Hence, we plan to specify techniques to manage the change of a configurable process model and adapt its process variants to match the new requirements, and vise versa (cf. Figure 7.1). These techniques need to maintain the overall consistency between a configurable process model and its derived variants.

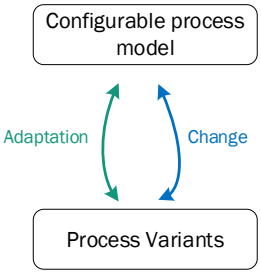


Figure 7.1: Need for adaptability to change

Tool support. As can be noticed, an automatic support of the configuration restriction and analysis step in a process modeling tool is missing in the current implementation of our work. Using a previous extension in our research team [107], the Signavio process modeling tool was adapted to support configurable elements, and thus, it allows the modeling of a configurable process. However, the configuration of this process is not considered yet. Hence, we are currently working on a

proof-of-concept of the SOG-based approach that consists of an extension of this web application. This extension considers the configuration aspect and, most importantly, restricts the configuration with only correct options. The set of correct configurations are extracted from our developed SOG-based tool based on the modeled configurable process in the Signavio interface. Using this tool, we target to conduct experiments with large process models data set to show the effectiveness of our work.

Now this is not the end.
It is not even the beginning of the end.
But it is, perhaps,
the end of the beginning.

Winston Churchill

Appendices

Appendix A

List of publications

1. Mohamed Graiet, Amel Mammar, Souha Boubaker, Walid Gaaloul, *Towards Correct Cloud Resource Allocation in Business Processes*, IEEE Transactions on Services Computing 10(1): 23-36 (2017)

2. Souha Boubaker, Kais Klai, Katia Schmitz, Mohamed Graiet, Walid Gaaloul, *Deadlock-Freeness Verification of Business Process Configuration Using SOG*, - 15th International Conference Service-Oriented Computing, ICSOC 2017, Malaga, Spain, November 13-16, 96-112.

3. Souha Boubaker, Amel Mammar, Mohamed Graiet, Walid Gaaloul, *Formal Verification of Cloud Resource Allocation in Business Processes Using Event-B*, 30th IEEE International Conference on Advanced Information Networking and Applications, AINA 2016, Crans-Montana, Switzerland, 23-25 March, 746-753

4. Souha Boubaker, Amel Mammar, Mohamed Graiet, Walid Gaaloul, *A Formal Guidance Approach for Correct Process Configuration*, Service-Oriented Computing - 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 483-498.

5. Souha Boubaker, Amel Mammar, Mohamed Graiet, Walid Gaaloul, *An Event-B Based Approach for Ensuring Correct Configurable Business Processes*, IEEE International Conference on Web Services, ICWS 2016, San Francisco, CA, USA, June 27 - July 2, 460-467

6. Souha Boubaker, Walid Gaaloul, Mohamed Graiet, Nejib Ben Hadj-Alouane, *Event-B Based Approach for Verifying Cloud Resource Allocation in Business Process*, IEEE International Conference on Services Computing, SCC 2015, New York City, NY, USA, June 27 - July 2, 538-545.

Appendix B

Event-B Symbols Summary

Table B.1 gives the semantics of some mathematical symbols used in this manuscript where:

- A and B denote any sets of elements,
- A_1 and B_1 denote any subsets of A and B respectively,
- S denotes any set expression.

Symbol	Semantics
Relation($R \in A \leftrightarrow B$)	$R \subseteq \{a \mapsto b \cdot a \in A \wedge b \in B\}$
Inverse($R\sim$)	$R\sim = \{b \mapsto a \cdot a \mapsto b \in R\}$
Image($R[A_1]$)	$R[A_1] = \{b_1 \cdot (b_1 \in B \wedge \exists a_1 \cdot (a_1 \in A_1 \wedge a_1 \mapsto b_1 \in R))\}$
Domain($dom(R)$)	$dom(R) = \{a_1 \cdot (a_1 \in A \wedge \exists b_1 \cdot (b_1 \in B \wedge a_1 \mapsto b_1 \in R))\}$
Range($ran(R)$)	$ran(R) = \{b_1 \cdot (b_1 \in B \wedge \exists a_1 \cdot (a_1 \in A \wedge a_1 \mapsto b_1 \in R))\}$
Domain restriction($A_1 \lhd R$)	$A_1 \lhd R = \{a \mapsto b \cdot (a \mapsto b \in R \wedge a \in A_1)\}$
Range restriction($R \rhd B_1$)	$R \rhd B_1 = \{a \mapsto b \cdot (a \mapsto b \in R \wedge b \in B_1)\}$
Domain subtraction ($A_1 \lhd\!\!\!- R$)	$A_1 \lhd\!\!\!- R = \{a \mapsto b \cdot (a \mapsto b \in R \wedge a \notin A_1)\}$
Range subtraction ($R \rhd\!\!\!- B_1$)	$R \rhd\!\!\!- B_1 = \{a \mapsto b \cdot (a \mapsto b \in R \wedge b \notin B_1)\}$
Partial function($f \in A \pfun B$)	$f \in A \pfun B$ $\Leftrightarrow$ $f \in A \leftrightarrow B \wedge \forall a \cdot (a \in A \Rightarrow card(f[\{a\}]) \leq 1)$
Total function($f \in A \to B$)	$f \in A \to B$ $\Leftrightarrow$ $f \in A \pfun B \wedge dom(f) = A$
Surjective function($f \in A \twoheadrightarrow B$)	$f \in A \twoheadrightarrow B$ $\Leftrightarrow$ $f \in A \pfun B \wedge ran(f) = B$
Injective function($f \in A \rightarrowtail B$)	$f \in A \rightarrowtail B$ $\Leftrightarrow$ $f \in A \pfun B \wedge f\sim \in B \pfun A$
Quantified union($\bigcup x \cdot (P \mid S)$)	if $\forall x.(P \Rightarrow S \subseteq T)$ then $\bigcup x \cdot (P \mid E) = \{y \cdot y \in T \wedge \exists z \cdot (z \in T \wedge P \wedge y \in S)\}$

Table B.1: Some Event-B symbols and their semantics

Bibliography

[1] M. Rosemann and W.M.P Van Der Aalst. A configurable reference modelling language. *Information Systems*, 32(1):1 – 23, 2007.

[2] F. Gottschalk. *Configurable Process Models*. Phd thesis, Eindhoven University of Technology, December 2009.

[3] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[4] Florian Gottschalk, Wil M. P. van der Aalst, Monique H. Jansen-Vullers, and Marcello La Rosa. Configurable workflow models. *International Journal of Cooperative Information Systems*, 17(02):177–221, 2008.

[5] Dennis M. M. Schunselaar, Eric Verbeek, Wil M. P. van der Aalst, and Hajo A. Reijers. *Business Information Systems: 15th International Conference, BIS, Proceedings*, chapter Creating Sound and Reversible Configurable Process Models Using CoSeNets, pages 24–35. 2012.

[6] Alena Hallerbach, Thomas Bauer, and Manfred Reichert. Capturing variability in business process models: The provop approach. *J. Softw. Maint. Evol.*, 22(6-7):519–546, 2010.

[7] Marcello La Rosa, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Questionnaire-based variability modeling for system configuration. *Software & Systems Modeling*, 8(2):251–274, 2008.

[8] Mohsen Asadi, Bardia Mohabbati, Gerd Gröner, and Dragan Gasevic. Development and validation of customized process models. *Journal of Systems and Software*, 96:73 – 92, 2014.

[9] Nour Assy and Walid Gaaloul. *Business Process Management: 13th International Conference, BPM*, chapter Extracting Configuration Guidance Models from Business Process Repositories, pages 198–206. 2015.

[10] Alena Hallerbach, Thomas Bauer, and Manfred Reichert. Guaranteeing soundness of configurable process variants in provop. In *IEEE Conference on Commerce and Enterprise Computing, CEC*, pages 98–105, 2009.

[11] Cristina Cabanillas, David Knuplesch, Manuel Resinas, Manfred Reichert, Jan Mendling, and Antonio Ruiz-Cortés. *RALph: A Graphical Notation for Resource Assignments in Business Processes*, pages 53–68. Springer International Publishing, Cham, 2015.

[12] Open cloud computing interface - infrastructure (june 2011). `http://ogf.org/documents/GFD.184.pdf`.

[13] Marlon Dumas, Wil M. van der Aalst, and Arthur H. ter Hofstede. *Process-aware Information Systems: Bridging People and Software Through Process Technology*. John Wiley & Sons, Inc., New York, NY, USA, 2005.

[14] Wil M. P. van der Aalst and Kees M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[15] Andreas Oberweis. *Person-to-Application Processes: Workflow Management*, pages 21–36. John Wiley & Sons, Inc., 2005.

[16] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. Business process management: A survey. In *Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings*, pages 1–12, 2003.

[17] Howard Smith and Peter Fingar. *Business process management: the third wave*. Springer, 2003.

[18] Mathias Weske. *Business Process Management - Concepts, Languages, Architectures, 2nd Edition*. Springer, 2012.

[19] OMG. Business process model and notation (bpmn) 2.0. `http://www.omg.org/spec/BPMN/2.0/`.

[20] August-Wilhelm Scheer. *ARIS - vom Geschäftsprozess zum Anwendungssystem*. Springer, 4., durchges. aufl. edition, 2002.

[21] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: Yet another workflow language. *Inf. Syst.*, 30(4):245–275, June 2005.

[22] v2.3 OMG. Unified modelling language. http://www.omg.org/spec/uml/2.3/.

[23] Wil M. P. van der Aalst and Mathias Weske. The p2p approach to interorganizational workflows. In Klaus R. Dittrich, Andreas Geppert, and Moira C. Norrie, editors, *Advanced Information Systems Engineering*, pages 140–156, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[24] Roger W.H. Bons, Ronald M. Lee, and René W. Wagenaar. Designing trustworthy interorganizational trade procedures for open electronic commerce. *International Journal of Electronic Commerce*, 2(3):61–83, 1998.

[25] M. P. Van Der Aalst. Modeling and analyzing interorganizational workflows. In *Proceedings 1998 International Conference on Application of Concurrency to System Design*, pages 262–272, Mar 1998.

[26] Robert Engel, Worarat Krathu, Marco Zapletal, Christian Pichler, R. P. Jagadeesh Chandra Bose, Wil van der Aalst, Hannes Werthner, and Christian Huemer. Analyzing inter-organizational business processes. *Information Systems and e-Business Management*, 14(3):577–612, Aug 2016.

[27] Wil M.P. van der Aalst, Niels Lohmann, Peter Massuthe, Christian Stahl, and Karsten Wolf. Multiparty contracts: Agreeing and implementing interorganizational processes. *The Computer Journal*, 53(1):90–106, 2010.

[28] Wil van der Aalst. Loosely coupled interorganizational workflows: Modeling and analyzing workflows crossing organizational boundaries. *Inf. Manage.*, 37(2):67–75, March 2000.

[29] Florian Gottschalk, Wil M. P. van der Aalst, and Monique H. Jansen-Vullers. *Configurable Process Models — A Foundational Approach*, pages 59–77. Physica-Verlag HD, Heidelberg, 2007.

[30] Wil M. P. van der Aalst, Alexander Dreiling, Florian Gottschalk, Michael Rosemann, and Monique H. Jansen-Vullers. Configurable process models as a basis for reference modeling. In Christoph J. Bussler and Armin Haller, editors, *Business Process Management Workshops*, pages 512–518, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[31] Alexander Dreiling and Michael Rosemann. Configurable process models as a basis for reference modeling - position paper. 03 2018.

[32] W. M. P. van der Aalst. Business process configuration in the cloud: How to support and analyze multi-tenant processes? In *2011 IEEE Ninth European Conference on Web Services*, pages 3–10, Sept 2011.

[33] Wil M. P. van der Aalst. Configurable services in the cloud: Supporting variability while enabling cross-organizational process mining. In Robert Meersman, Tharam Dillon, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2010*, pages 8–25, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[34] Jan Recker, Michael Rosemann, Wil van der Aalst, and Jan Mendling. *On the Syntax of Reference Model Configuration – Transforming the C-EPC into Lawful EPC Models*, pages 497–511. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[35] Akhil Kumar and Wen Yao. Design and management of flexible process variants using templates and rules. *Computers in Industry*, 63(2):112 – 130, 2012.

[36] Gerd Gröner, Marko Bošković, Fernando Silva Parreiras, and Dragan Gašević. Modeling and validation of business process families. *Information Systems*, 38(5):709 – 726, 2013.

[37] Wil M. P. van der Aalst, Marlon Dumas, Florian Gottschalk, Arthur H. M. ter Hofstede, Marcello La Rosa, and Jan Mendling. Preserving correctness during business process model configuration. *Formal Aspects of Computing*, 22(3-4):459–482, 2008.

[38] Wil M. P. van der Aalst, Marlon Dumas, Florian Gottschalk, Arthur H. M. ter Hofstede, Marcello La Rosa, and Jan Mendling. *Fundamental Approaches to Software Engineering, FASE*, chapter Correctness-Preserving Configuration of Business Process Models, pages 46–61. 2008.

[39] Wil M. P. van der Aalst, Niels Lohmann, and Marcello La Rosa. Ensuring correctness during process configuration via partner synthesis. *Information Systems*, 37(6):574 – 592, 2012.

[40] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and David Edmond. Workflow resource patterns: Identification, representation and tool support. pages 216–232, 2005.

[41] L.J.R Stroppi, O. Chiotti, and P.D. Villarreal. Extended resource perspective support for BPMN and BPEL. In *Proceedings of the XV Iberoamerican Conference on Software Engineering*, pages 56–69, 2012.

[42] L. J. R. Stroppi, O. Chiotti, and P. D. Villarreal. A BPMN 2.0 Extension to Define the Resource Perspective of Business Process Models. In *XIV Congreso Iberoamericano en Software Engineering (CIbSE 2011)*, 2011.

[43] Luis Jesús Ramón Stroppi, Omar Chiotti, and Pablo David Villarreal. Defining the resource perspective in the development of processes-aware information systems. *Information and Software Technology*, 59:86 – 108, 2015.

[44] Cristina Cabanillas, Alex Norta, Manuel Resinas, Jan Mendling, and Antonio Ruiz-Cortés. *Towards Process-Aware Cross-Organizational Human Resource Management*, pages 79–93. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[45] Cristina Cabanillas, Manuel Resinas, Adela del Río-Ortega, and Antonio Ruiz-Cortés. Specification and automated design-time analysis of the business process human resource perspective. *Information Systems*, 52:55 – 82, 2015. Special Issue on Selected Papers from SISAP 2013.

[46] D. Zowghi and V. Gervasi. On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology*, 45(14):993 – 1009, 2003.

[47] B. Kiepuszewski, A.H.M. Ter Hofstede, and W.M.P. Van Der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2002.

[48] R. Kowalski and M.Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, January 1986.

[49] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57. IEEE Computer Society, 1977.

[50] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[51] H M. W. (Eric) Verbeek, Twan Basten, and Wil M. P. Aalst. Diagnosing workflow processes using woflan. 44:246–279, 01 2001.

[52] Serge Haddad, Jean-Michel Ilié, and Kais Klai. Design and evaluation of a symbolic and abstraction-based model checker. In Farn Wang, editor, *Automated Technology for Verification and Analysis*, pages 196–210, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[53] K. Klai and L. Petrucci. Modular construction of the symbolic observation graph. In *2008 8th International Conference on Application of Concurrency to System Design*, pages 88–97, June 2008.

[54] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013.

[55] Monika Weidmann, Falko Kötter, Maximilien Kintz, Daniel Schleicher, Ralph Mietzner, and Frank Leymann. Adaptive Business Process Modeling in the Internet of Services (ABIS). In Proceedings of the Sixth International Conference on Internet, Web Applications, and Services (ICIW) 2011, editors, *Adaptive Business Process Modeling in the Internet of Services (ABIS)*, pages 29–34. Xpert Publishing Services, March 2011.

[56] PhD thesis.

[57] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[58] Arthur ter Hofstede and Wil van der Aalst. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In Jensen, editor, *Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools.*, pages 1–17, Aarhus, Denmark, 2002. Department of Computer Science, University of Aarhus.

[59] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[60] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *STTT*, 12(6):447–466, 2010.

[61] Atelier b tool homepage. `http://www.atelierb.eu/en/atelier-b-tools/`.

[62] M. Leuschel and M. Butler. Prob: an automated analysis toolset for the b method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[63] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, 1982.

[64] I. Ait-Sadoune and Y. Ait-Ameur. A proof based approach for modelling and verifying web services compositions. In *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 1–10, June 2009.

[65] Idir Ait-Sadoune and Yamine Ait-Ameur. *Stepwise Design of BPEL Web Services Compositions: An Event_B Refinement Based Approach*, pages 51–68. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[66] Mohamed Graiet, Imed Abbassi, Lazhar Hamel, Mohamed Tahar Bhiri, Mourad Kmimech, and Walid Gaaloul. Event-b based approach for verifying dynamic composite service transactional behavior. In *IEEE 20th International Conference on Web Services*, pages 251–259, 2013.

[67] Imed Abbassi, Mohamed Graiet, Walid Gaaloul, and Nejib Ben Hadj-Alouane. A formal approach for enforcing transactional requirements in web service compositions. In *IEEE International Conference on Services Computing*, pages 637–644, 2014.

[68] G. Babin, Y. A. Ameur, and M. Pantel. Formal verification of runtime compensation of web service compositions: A refinement and proof based proposal with event-b. In *IEEE International Conference on Services Computing (SCC)*, pages 98–105, June 2015.

[69] Jeremy W. Bryans and Wei Wei. Formal analysis of bpmn models using event-b. In Stefan Kowalewski and Marco Roveri, editors, *Formal Methods for Industrial Critical Systems*, pages 33–49, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[70] Iman Poernomo and Timur Umarov. A mapping from normative requirements to event-b to facilitate verified data-centric business process management. In Tomasz Szmuc, Marcin Szpyrka, and Jaroslav Zendulka, editors, *Advances in Software Engineering Techniques*, pages 136–149, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[71] Wil M. P. van der Aalst. Challenges in business process management: Verification of business processing using petri nets. *Bulletin of the EATCS*, 80:174–199, 2003.

[72] B.F. van Dongen, M.H. Jansen-Vullers, H.M.W. Verbeek, and W.M.P. van der Aalst. Verification of the sap reference models using epc reduction, state-space analysis, and invariants. *Computers in Industry*, 58(6):578 – 601, 2007.

[73] B. f. V. Dongen, J. Mendling, and W. m. p. V. Der Aalst. Structural patterns for soundness of business process models. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, pages 116–128, Oct 2006.

[74] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software Technology*, 50(12):1281 – 1294, 2008.

[75] Wasim Sadiq and Maria E. Orlowska. On correctness issues in conceptual modeling of workflows, 1997.

[76] W. M. P. van der Aalst. Verification of workflow nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets 1997*, pages 407–426, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[77] M.T. Wynn, H.M.W. Verbeek, W.M. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Business process verification - finally a reality! *Business Process Management Journal*, 15(1):74–92, 2009.

[78] Boudewijn F. van Dongen, Wil M. P. van der Aalst, and H. M. W. Verbeek. Verification of epcs: Using reduction rules and petri nets. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering*, pages 372–386, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[79] H.M.W. Verbeek, M.T. Wynn, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Reduction rules for reset/inhibitor nets. *Journal of Computer and System Sciences*, 76(2):125 – 143, 2010.

[80] W.M.P van der Aalst. *Business Process Management: Models, Techniques, and Empirical Studies*, chapter Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques, pages 161–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[81] Wil M.P. van der Aalst and Arthur H.M. ter Hofstede. Verification of workflow task structures: A petri-net-baset approach. *Information Systems*, 25(1):43 – 69, 2000.

[82] M.T. Wynn, H.M.W. Verbeek, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Soundness-preserving reduction rules for reset workflow nets. *Information Sciences*, 179(6):769 – 790, 2009.

[83] Julio Clempner. An analytical method for well-formed workflow/petri net verification of classical soundness. *Int. J. Appl. Math. Comput. Sci.*, 24(4):931–939, December 2014.

[84] Julio Clempner. Verifying soundness of business processes: A decision process petri nets approach. *Expert Systems with Applications*, 41(11):5030 – 5040, 2014.

[85] Y. He, G. Liu, D. Xiang, J. Sun, C. Yan, and C. Jiang. Verifying the correctness of workflow systems based on workflow net with data constraints. *IEEE Access*, 6:11412–11423, 2018.

[86] Robin Milner. *Communicating and Mobile Systems: The &Pgr;-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[87] Frank Puhlmann and Mathias Weske. Using the π-calculus for formalizing workflow patterns. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, pages 153–168, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[88] Frank Puhlmann. Soundness verification of business processes specified in the pi-calculus. In *Proceedings of the 2007 OTM Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, OTM'07, pages 6–23, Berlin, Heidelberg, 2007. Springer-Verlag.

[89] W. Gaaloul, S. Bhiri, and M. Rouached. Event-based design and runtime verification of composite service transactional behavior. *IEEE T. Services Computing*, 3(1):32–45, 2010.

[90] S. Wang, J. Wan, and X. Yang. Describing, verifying and developing web service using the b-method. In *International Conference on Next Generation Web Services Practices*, pages 11–16, Sept 2006.

[91] Kais Klai, Samir Tata, and Jörg Desel. Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes. In Umeshwar Dayal, Johann Eder, Jana Koehler, and Hajo A. Reijers, editors, *Business Process Management*, pages 294–309, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[92] J. Koehler, G. Tirenni, and S. Kumaran. From business process model to consistent implementation: a case for formal verification methods. In *Proceedings. Sixth International Enterprise Distributed Object Computing*, pages 96–106, 2002.

[93] Wil M. P. van der Aalst, Kees M. van Hee, Arthur H. M. ter Hofstede, Natalia Sidorova, H. M. W. Verbeek, Marc Voorhoeve, and Moe Thandar Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2010.

[94] Dirk Fahland, Cédric Favre, Barbara Jobstmann, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. *Business Process Management: 7th International Conference, BPM. Proceedings*, chapter Instantaneous Soundness Checking of Industrial Business Process Models, pages 278–293. 2009.

[95] B.F. Van Dongen, J. Mendling, and W.M.P Van Der Aalst. Structural patterns for soundness of business process models. In *Enterprise Distributed Object Computing Conference*, pages 116–128, 2006.

[96] Juliane Dehnert and Peter Rittgen. Relaxed soundness of business processes. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, CAiSE '01, pages 157–170, London, UK, UK, 2001. Springer-Verlag.

[97] Axel Martens. Analyzing web service based business processes. In *Proceedings of the 8th International Conference, Held As Part of the Joint European Conference on Theory and Practice of Software Conference on Fundamental Approaches to Software Engineering*, FASE'05, pages 19–33, Berlin, Heidelberg, 2005. Springer-Verlag.

[98] Marcello La Rosa, Wil M. P. Van Der Aalst, Marlon Dumas, and Fredrik P. Milani. Business process variability modeling: A survey. *ACM Comput. Surv.*, 50(1):2:1–2:45, March 2017.

[99] Marcello La Rosa, Marlon Dumas, Arthur H.M. ter Hofstede, and Jan Mendling. Configurable multi-perspective business process models. *Information Systems*, 36(2):313 – 340, 2011. Special Issue: Semantic Integration of Data, Multimedia, and Services.

[100] Arnd Schnieders and Frank Puhlmann. Variability mechanisms in e-business process families. In *Business Information Systems, 9th International Conference on Business Information Systems, BIS 2006, May 31 - June 2, 2006, Klagenfurt, Austria*, pages 583–601, 2006.

[101] I. Rychkova and S. Nurcan. Towards adaptability and control for knowledge-intensive business processes: Declarative configurable process specifications. In *2011 44th Hawaii International Conference on System Sciences*, pages 1–10, Jan 2011.

[102] Frank Puhlmann. Variability mechanisms for process models. 01 2005.

[103] Marcello La Rosa, Florian Gottschalk, Marlon Dumas, and Wil M. P. van der Aalst. *Linking Domain Models and Process Models for Reference Model Configuration*, pages 417–430. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[104] Paul C. Clements. Managing variability for software product lines: Working with variability mechanisms. In *10th International Software Product Line Conference (SPLC'06)*, pages 207–208, Aug 2006.

[105] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Managing Variability in Workflow with Feature Model Composition Operators. In *International Conference on Software Composition 2010*, LNCS, page 16, Malaga, Spain, July 2010. springer. accepté à International Conference on Software Composition 2010.

[106] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[107] Nour Assy. *Automated support of the variability in configurable process models. (Automatiser le support de la variabilité dans les modèles de processus configurables)*. PhD thesis, University of Paris-Saclay, France, 2015.

[108] Karsten Wolf. *Does My Service Have Partners?*, pages 152–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[109] Giray Havur, Cristina Cabanillas, Jan Mendling, and Axel Polleres. *Resource Allocation with Dependencies in Business Process Management Systems*, pages 3–19. Springer International Publishing, Cham, 2016.

[110] E. F. Duipmans, L. F. Pires, and L. O. B. d. Silva. Towards a bpm cloud architecture with data and activity distribution. In *2012 IEEE 16th International Enterprise Distributed Object Computing Conference Workshops*, pages 165–171, Sept 2012.

[111] Stefan Schulte, Christian Janiesch, Srikumar Venugopal, Ingo Weber, and Philipp Hoenisch. Elastic business process management: State of the art and open challenges for BPM in the cloud. *CoRR*, abs/1409.5715, 2014.

[112] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick H.J. Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems*, 29(1):158 – 169, 2013. Including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures.

[113] Philipp Hoenisch, Stefan Schulte, Schahram Dustdar, and Srikumar Venugopal. Self-adaptive resource allocation for elastic process execution. In *IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 220–227, 2013.

[114] Philipp Hoenisch, Stefan Schulte, and Schahram Dustdar. Workflow scheduling and resource allocation for cloud-based execution of elastic processes. In *Proceedings of the 6th International Conference on Service-Oriented Computing and Applications*, pages 1–8, 2013.

[115] Kahina Bessai, Samir Youcef, Ammar Oulamara, Claude Godart, and Selmin Nurcan. Scheduling strategies for business process applications in cloud environments. *Int. J. Grid High Perform. Comput.*, 5(4):65–78, October 2013.

[116] P. Varalakshmi, Aravindh Ramaswamy, Aswath Balasubramanian, and Palaniappan Vijaykumar. *An Optimal Workflow Based Scheduling and Resource Allocation in Cloud*, pages 411–420. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[117] Zhengxing Huang, Wil M. P. van der Aalst, Xudong Lu, and Huilong Duan. Reinforcement learning based resource allocation in business process management. *Data Knowl. Eng.*, 70(1):127–145, 2011.

[118] T. Mastelic, W. Fdhila, I. Brandic, and S. Rinderle-Ma. Predicting resource allocation and costs for business processes in the cloud. In *IEEE World Congress on Services*, pages 47–54, June 2015.

[119] Eun-Kyu Byun, Yang-Suk Kee, Jin-Soo Kim, and Seungryoul Maeng. Cost optimized provisioning of elastic resources for application workflows. *Future Gener. Comput. Syst.*, 27(8):1011–1026, October 2011.

[120] Marcello M. Bersani, Domenico Bianculli, Schahram Dustdar, Alessio Gambi, Carlo Ghezzi, and Srdan Krstic. Towards the formalization of properties of cloud-based elastic systems. In *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, PESOS, pages 38–47, 2014.

[121] H. Sahli, F. Belala, and C. Bouanaka. A brs-based approach to model and verify cloud systems elasticity. *Procedia Computer Science*, 68:29 – 41, 2015.

[122] Alessio Gambi, Antonio Filieri, and Schahram Dustdar. Iterative test suites refinement for elastic computing systems. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 635–638, New York, NY, USA, 2013. ACM.

[123] Kais Klai and Samir Tata. Formal modeling of elastic service-based business processes. In *Services Computing (SCC), 2013 IEEE International Conference on*, pages 424–431, June 2013.

[124] Mourad Amziani, Tarek Melliti, and Samir Tata. Formal modeling and evaluation of service-based business process elasticity in the cloud. In *IEEE 22nd International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 284–291, 2013.

[125] Mourad Amziani, Tarek Melliti, and Samir Tata. Formal modeling and evaluation of stateful service-based business process elasticity in the cloud. In *On the Move to Meaningful Internet Systems: OTM 2013 Conferences - Confederated International Conferences: CoopIS, DOA-Trusted Cloud, and ODBASE 2013, Graz, Austria, September 9-13, 2013. Proceedings*, pages 21–38, 2013.

[126] Mohamed Graiet, Lazhar Hamel, Amel Mammar, and Samir Tata. A verification and deployment approach for elastic component-based applications. *Formal Aspects of Computing*, 29(6):987–1011, Nov 2017.

[127] Rania Ben Halima, Slim Kallel, Kais Klai, Walid Gaaloul, and Mohamed Jmaiel. Formal verification of time-aware cloud resource allocation in business process. In Christophe Debruyne, Herve Panetto, Robert Meersman, Tharam Dillon, eva Kühn, Declan O'Sullivan, and Claudio Agostino Ardagna, editors, *On the Move to Meaningful Internet Systems: OTM 2016 Conferences*, pages 400–417, Cham, 2016. Springer International Publishing.

[128] Emna Hachicha, Walid Gaaloul, and Zakaria Maamar. Social-based semantic framework for cloud resource management in business processes. In *SCC*, pages 443–450. IEEE Computer Society, 2016.

[129] Souha Boubaker, Amel Mammar, Mohamed Graiet, and Walid Gaaloul. An event-b based approach for ensuring correct configurable business processes. In *ICWS*, pages 460–467. IEEE Computer Society, 2016.

[130] Souha Boubaker, Amel Mammar, Mohamed Graiet, and Walid Gaaloul. A formal guidance approach for correct process configuration. In *ICSOC*, volume 9936 of *Lecture Notes in Computer Science*, pages 483–498. Springer, 2016.

[131] Souha Boubaker, Kais Klai, Katia Schmitz, Mohamed Graiet, and Walid Gaaloul. Deadlock-freeness verification of business process configuration using SOG. In *ICSOC*, volume 10601 of *Lecture Notes in Computer Science*, pages 96–112. Springer, 2017.

[132] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, Jul 2003.

[133] Marlon Dumas, Alexander Grosskopf, Thomas Hettel, and Moe Wynn. Semantics of standard process models with or-joins. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, pages 41–58, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[134] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. *GraphML Progress Report Structural Layer Proposal*, pages 501–512. 2002.

[135] Jorge Cardoso. Business process control-flow complexity: Metric, evaluation, and validation. 5:49–76, 04 2008.

[136] Souha Boubaker, Amel Mammar, Mohamed Graiet, and Walid Gaaloul. Formal verification of cloud resource allocation in business processes using event-b. In *AINA*, pages 746–753. IEEE Computer Society, 2016.

[137] Souha Boubaker, Walid Gaaloul, Mohamed Graiet, and Nejib Ben Hadj-Alouane. Event-b based approach for verifying cloud resource allocation in business process. In *SCC*, pages 538–545. IEEE Computer Society, 2015.

[138] Mohamed Graiet, Amel Mammar, Souha Boubaker, and Walid Gaaloul. Towards correct cloud resource allocation in business processes. *IEEE Transactions on Services Computing*, 10(1):23–36, Jan 2017.

[139] Topology and orchestration specification for cloud applications (tosca). `https://www.oasis-open.org/committees/tosca/`.

[140] Open cloud computing interface (occi). `http://occi-wg.org/`.

[141] Cloud infrastructure management interface (cimi). `https://www.dmtf.org/standards/cloud`.

[142] Open cloud computing interface - core (june 2011). `http://ogf.org/documents/GFD.183.pdf`.

[143] Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and System Management*, 2013.

[144] D. Hollingsworth and UK. Hampshire. Workflow management coalition the workflow reference model. *Workflow Management Coalition*, 68:26, 1993.

[145] A. Edmonds, T. Metsch, A. Papaspyrou, and A. Richardson. Toward an open cloud standard. *Internet Computing, IEEE*, 16(4):15–25, 2012.

[146] Emna Hachicha, Nour Assy, Walid Gaaloul, and Jan Mendling. A configurable resource allocation for multi-tenant process development in the cloud. In *CAiSE*, volume 9694 of *Lecture Notes in Computer Science*, pages 558–574. Springer, 2016.